

# ABC Library Reference



*SoftVelocity*

**COPYRIGHT 1994-2003 SoftVelocity Incorporated. All rights reserved.**

**This publication is protected by copyright and all rights are reserved by SoftVelocity Incorporated. It may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from SoftVelocity Incorporated.**

This publication supports Clarion. It is possible that it may contain technical or typographical errors. SoftVelocity Incorporated provides this publication "as is," without warranty of any kind, either expressed or implied.

**SoftVelocity Incorporated**

2769 East Atlantic Blvd.  
Pompano Beach, Florida 33062  
(954) 785-4555  
[www.softvelocity.com](http://www.softvelocity.com)

**Trademark Acknowledgements:**

SoftVelocity is a trademark of SoftVelocity Incorporated.

Clarion™ is a trademark of SoftVelocity Incorporated.

Btrieve® is a registered trademark of Pervasive Software.

Microsoft®, Windows®, and Visual Basic® are registered trademarks of Microsoft Corporation.

All other products and company names are trademarks of their respective owners.

Printed in the United States of America (1103)

## Contents:

<b>Foreword</b>	<b>45</b>
Welcome .....	45
Documentation Conventions.....	46
Keyboard Conventions .....	46
Other Conventions.....	46
<b>ABC Library Overview</b>	<b>47</b>
About This Book.....	47
Application Builder Class (ABC) Library .....	47
Class Libraries Generally .....	47
Application Builder Classes—The ABCs of Rapid Application Development .....	47
ABC Library and the ABC Templates.....	51
ABC Coding Conventions.....	53
Method Names .....	53
Where to Initialize & Kill Objects .....	55
Return Values.....	55
PRIVATE (undocumented) Items.....	56
PROTECTED, VIRTUAL, DERIVED, and PROC Attributes .....	57
Documentation Conventions .....	58
Reference Item and Syntax Diagram .....	58
Property (short description of intended use) .....	58
Method (short description of what the method does) .....	58
Conceptual Example .....	60
<b>ASCIIFileClass</b>	<b>61</b>
ASCIIFileClass Overview.....	61
AsciiFileClass Properties .....	64
ASCIIFile (the ASCII file) .....	64
ErrorMgr (ErrorClass object) .....	64
OpenMode (file access/sharing mode).....	64
AsciiFileClass Methods.....	65
ASCIIFileClass Functional Organization--Expected Use .....	65
FormatLine (a virtual to format text) .....	66
GetDOSFilename (let end user select file) .....	67
GetFilename (return the filename) .....	68
GetLastLineNo (return last line number) .....	69
GetLine (return line of text) .....	70
GetPercentile (convert file position to percentage:ASCIIFileClass) .....	71
Init (initialize the ASCIIFileClass object) .....	72
Kill (shut down the ASCIIFileClass object) .....	73
Reset (reset the ASCIIFileClass object) .....	74
SetLine (a virtual to position the file) .....	75
SetPercentile (set file to relative position) .....	76

ValidateLine (a virtual to implement a filter) .....	77
<b>ASCIIPrintClass</b>	<b>79</b>
ASCIIPrintClass Overview .....	79
AsciiPrintClass Properties.....	82
FileMgr (AsciiFileClass object:AsciiPrintClass) .....	82
PrintPreview (print preview switch) .....	82
Translator (TranslatorClass object:AsciiPrintClass) .....	82
AsciiPrintClass Methods .....	83
Ask (solicit print specifications) .....	83
Init (initialize the ASCIIPrintClass object) .....	84
PrintLines (print or preview specified lines) .....	85
<b>ASCIISearchClass</b>	<b>87</b>
ASCIISearchClass Overview .....	87
AsciiSearchClass Properties.....	90
Find (search constraints) .....	90
FileMgr (AsciiFileClass object:AsciiSearchClass) .....	91
LineCounter (current line number) .....	91
Translator (TranslatorClass object:ASCIISearchClass) .....	91
AsciiSearchClass Methods .....	92
Ask (solicit search specifications) .....	92
Init (initialize the ASCIISearchClass object) .....	93
Next (find next line containing search text) .....	94
Setup (set search constraints) .....	95
<b>ASCIIViewerClass</b>	<b>97</b>
ASCIIViewerClass Overview .....	97
AsciiViewerClass Properties.....	100
Popup (PopupClass object) .....	100
Printer (ASCIIPrintClass object) .....	100
Searcher (ASCIISearchClass object) .....	101
TopLine (first line currently displayed).....	101
AsciiViewerClass Methods.....	102
AsciiViewerClass Functional Organization--Expected Use .....	102
AddItem (program the AsciiViewer object) .....	104
AskGotoLine (go to user specified line) .....	105
DisplayPage (display new page) .....	106
Init (initialize the ASCIIViewerClass object) .....	107
Kill (shut down the ASCIIViewerClass object) .....	109
PageDown (scroll down one page) .....	111
PageUp (scroll up one page) .....	112
Reset (reset the ASCIIViewerClass object) .....	113
SetLine (position to specific line) .....	114
SetLineRelative (move <i>n</i> lines) .....	115
SetTranslator (set run-time translator:ASCIIViewerClass) .....	116

TakeEvent (process ACCEPT loop event:ASCIIViewerClass) .....	117
<b>BreakManagerClass</b>	<b>119</b>
BreakManagerClass - Overview.....	119
BreakManagerClass - Concepts .....	119
BreakManagerClass - Relationship to Other Application Builder Classes.....	119
BreakManagerClass - ABC Template Implementation .....	119
BreakManagerClass - Source Files.....	120
BreakManagerClass - Conceptual Example .....	120
BreakManagerClass - Properties.....	123
BreakManagerClass - Methods .....	123
AddBreak (add a Break).....	123
AddLevel (add a Level to the BreakId Break) .....	124
AddResetField (add a reset field to the last Level added) .....	125
AddTotal (add a total field to the last Level added).....	126
Construct (automatic initialization of the BreakManager object).....	127
Destruct (automatic destroy of the Breakmanager object).....	127
Init (initialize the BreakManager object) .....	127
TakeEnd (Break closed) .....	128
TakeStart (Break opened) .....	129
UpdateTotal (Calculate Break totaling) .....	130
<b>BrowseEIPManagerClass</b>	<b>131</b>
BrowseEIPManagerClass--Overview .....	131
BrowseEIPManagerClass Concepts.....	132
BrowseEIPManagerClass--Relationship to Other Application Builder Classes .....	133
BrowseEIPManagerClass--ABC Template Implementation .....	133
BrowseEIPManagerClass Source Files .....	133
BrowseEIPManagerClass--Conceptual Example .....	134
BrowseEIPManagerClass Properties .....	137
BrowseEIPManagerClass Properties .....	137
BC (browse class) .....	137
BrowseEIPManagerClass Methods .....	138
BrowseEIPManagerClass--Functional Organization--Expected Use .....	138
ClearColumn (reset column property values) .....	139
Init (initialize the BrowseEIPManagerClass object) .....	140
Kill (shut down the BrowseEIPManagerClass object) .....	141
TakeCompleted (process completion of edit) .....	142
TakeNewSelection (reset edit-in-place column) .....	143
<b>BrowseClass</b>	<b>145</b>
BrowseClass Overview .....	145
BrowseClass Properties.....	150
BrowseClass Properties .....	150
ActiveInvisible (obscured browse list action) .....	150
AllowUnfilled (display filled list) .....	151

ArrowAction (edit-in-place action on arrow key) .....	152
AskProcedure (update procedure) .....	153
ChangeControl (browse change/update control number) .....	153
CurrentChoice (current LIST control entry number) .....	154
DeleteAction (edit-in-place action on delete key) .....	154
DeleteControl (browse delete control number) .....	154
EditList (list of edit-in-place controls) .....	155
EIP (edit-in-place manager) .....	155
EnterAction (edit-in-place action on enter key) .....	156
FocusLossAction (edit-in-place action on lose focus) .....	157
HasThumb (vertical scroll bar flag) .....	158
HideSelect (hide select button) .....	158
ILC(reference to IListControl interface) .....	158
InsertControl (browse insert control number) .....	159
ListQueue (browse data queue reference) .....	159
Loaded (browse queue loaded flag) .....	159
Popup (browse popup menu reference) .....	160
PrevChoice (prior LIST control entry number) .....	160
PrintControl (browse print control number) .....	160
PrintProcedure (print procedure).....	161
Processors (reference to ProcessorQueue) .....	161
Query (reference to QueryClass) .....	161
QueryControl (browse query control number).....	162
QueryShared (share query with multiple sorts).....	162
QuickScan (buffered reads flag) .....	163
RetainRow (highlight bar refresh behavior) .....	164
SelectControl (browse select control number) .....	164
Selecting (select mode only flag) .....	165
SelectWholeRecord (select entire record flag) .....	165
Sort (browse sort information) .....	166
StartAtCurrent (initial browse position) .....	167
TabAction (edit-in-place action on tab key) .....	168
Toolbar (browse Toolbar object) .....	169
ToolbarItem (browse ToolbarTarget object) .....	170
ToolControl (browse toolbox control number).....	171
ViewControl (view button) .....	171
Window (WindowManager object) .....	171
BrowseClass Methods .....	172
BrowseClass Methods.....	172
BrowseClass Functional Organization--Expected Use .....	172
AddEditControl (specify custom edit-in-place class) .....	174
AddField (specify a FILE/QUEUE field pair) .....	175
AddItem(program the BrowseClass object) .....	176
AddLocator (specify a locator) .....	176
AddResetField (set a field to monitor for changes) .....	177
AddSortOrder (specify a browse sort order) .....	178
AddToolbarTarget (set the browse toolbar) .....	179

ApplyRange (refresh browse based on resets and range limits) .....	180
Ask (update selected browse item) .....	181
AskRecord (edit-in-place selected browse item) .....	183
Fetch (get a page of browse items) .....	184
Init(initialize the BrowseClass object) .....	185
InitSort (initialize locator values) .....	186
Kill (shut down the BrowseClass object) .....	187
Next (get the next browse item) .....	188
NotifyUpdateError (throw error on update).....	189
PostNewSelection (post an EVENT:NewSelection to the browse list) .....	190
Previous (get the previous browse item) .....	191
Records (return the number of browse queue items) .....	192
ResetFields(reinitialize FieldPairsClass) .....	192
ResetFromAsk (reset browse after update) .....	193
ResetFromBuffer (fill queue starting from record buffer) .....	195
ResetFromFile (fill queue starting from file POSITION) .....	196
ResetFromView (reset browse from current result set) .....	197
ResetQueue (fill or refill queue) .....	198
ResetResets (copy the Reset fields) .....	199
ResetSort (apply sort order to browse) .....	200
ScrollEnd (scroll to first or last item) .....	201
ScrollOne (scroll up or down one item) .....	202
ScrollPage (scroll up or down one page) .....	203
SetAlerts (alert keystrokes for list and locator controls) .....	204
SetLocatorField (set sort free element to passed field).....	205
SetLocatorFromSort (use sort like locator field).....	205
SetQueueRecord (copy data from file buffer to queue buffer:BrowseClass).....	206
SetSort (apply a sort order to the browse) .....	207
TakeAcceptedLocator (apply an accepted locator value) .....	208
TakeEvent (process the current ACCEPT loop event:BrowseClass) .....	209
TakeKey (process an alerted keystroke:BrowseClass) .....	210
TakeNewSelection (process a new selection:BrowseClass) .....	211
TakeScroll (process a scroll event) .....	212
TakeVCRScroll (process a VCR scroll event) .....	213
UpdateBuffer (copy selected item from queue buffer to file buffer) .....	214
UpdateQuery (set default query interface) .....	215
UpdateResets (copy reset fields to file buffer) .....	216
UpdateThumb (position the scrollbar thumb) .....	217
UpdateThumbFixed (position the scrollbar fixed thumb) .....	218
UpdateViewRecord (get view data for the selected item) .....	219
UpdateWindow (update display variables to match browse) .....	220
<b>BrowseQueue Interface</b> .....	<b>221</b>
BrowseQueue Concepts .....	221
Relationship to Other Application Builder Classes .....	221
BrowseQueue Source Files .....	221
BrowseQueue Methods .....	222

Delete(remove entry in LIST queue)	222
Fetch(retrieve entry from LIST queue)	222
Free(clear contents of LIST queue)	222
GetViewPosition(retrieve VIEW position)	222
Insert(add entry to LIST queue)	223
Records(return number of records)	223
SetViewPosition(set VIEW position)	223
Update(update entry in LIST queue)	224
Who(returns field name)	224
<b>BrowseToolBarClass</b>	<b>225</b>
BrowseToolBarClass Overview	225
BrowseToolBarClass Concepts	225
Relationship to Other Application Builder Classes	225
BrowseToolBarClass ABC Template Implementation	225
BrowseToolBarClass Source Files	225
BrowseToolBarClass Properties	226
Browse (BrowseClass object)	226
Button (toolbar buttons FEQ values)	227
Window (WindowManager object)	227
BrowseToolBarClass Methods	228
Init (initialize the BrowseToolBarClass object)	228
InitBrowse (initialize the BrowseToolBarClass update buttons)	228
InitMisc (initialize the BrowseToolBarClass miscellaneous buttons)	229
InitVCR (initialize the BrowseToolBarClass VCR buttons)	230
ResetButton (synchronize toolbar with a corresponding browse control)	231
ResetFromBrowse(synchronize toolbar controls with browse controls)	232
TakeEvent(process the current event)	232
<b>BufferedPairsClass</b>	<b>233</b>
BufferedPairsClass Overview	233
BufferedPairsClass Properties	235
BufferedPairsClass Properties	235
RealList (recognized field pairs)	235
BufferedPairsClass Methods	236
BufferedPairsClass Methods	236
BufferedPairsClass Functional Organization Expected Use	236
AddPair (add a field pair:BufferedPairsClass)	238
AssignBufferToLeft (copy from "buffer" fields to "left" fields)	240
AssignBufferToRight (copy from "buffer" fields to "right" fields)	241
AssignLeftToBuffer (copy from "left" fields to "buffer" fields)	242
AssignRightToBuffer (copy from "right" fields to "buffer" fields)	243
EqualLeftBuffer (compare "left" fields to "buffer" fields)	244
EqualRightBuffer (compare "right" fields to "buffer" fields)	245
Init (initialize the BufferedPairsClass object)	246
Kill (shut down the BufferedPairsClass object)	247

<b>ConstantClass</b>	<b>249</b>
ConstantClass Overview .....	249
ConstantClass Properties .....	253
TerminatorField (identify the terminating field) .....	253
TerminatorInclude (include matching terminator record) .....	253
TerminatorValue (end of data marker) .....	254
ConstantClass Methods .....	255
ConstantClass Functional Organization--Expected Use .....	255
AddItem (set constant datatype and target variable) .....	256
Init (initialize the ConstantClass object) .....	257
Kill (shut down the ConstantClass object) .....	259
Next (load all constant items to file or queue) .....	260
Next (copy next constant item to targets) .....	262
Reset (reset the object to the beginning of the constant data) .....	263
Set (set the constant data to process) .....	264
<b>Crystal8 Class</b>	<b>265</b>
Crystal8 Class Properties .....	266
Crystal8 Methods .....	267
AllowPrompt (prompt for runtime parameter data) .....	267
CanDrillDown(allow Crystal drill down support ) .....	268
HasCancelButton (display cancel button on report preview) .....	269
HasCloseButton (display close button on report preview) .....	270
HasExportButton (display export button on report preview) .....	271
HasLaunchButton (display launch button on report preview) .....	272
HasNavigationControls (display navigation controls on report preview) .....	273
HasPrintButton (display print button on report preview) .....	274
HasPrintSetupButton (display print setup button on report preview) .....	275
HasProgressControls (display progress controls on report preview) .....	276
HasRefreshButton (display refresh button on report preview) .....	277
HasSearchButton (display search button on report preview) .....	278
HasZoomControl (display zoom control on report preview) .....	279
Init (initialize Crystal8 object) .....	280
Kill (shut down Crystal8 object) .....	281
Preview (preview a Crystal Report) .....	282
_Print (print a Crystal Report) .....	283
Query (retrieve or set the SQL data query) .....	284
SelectionFormula (retrieve or set the Crystal formula ) .....	285
ShowDocumentTips (show tips on document in the preview window) .....	286
ShowReportControls (show print controls) .....	287
ShowToolBarTips (show tips on preview window toolbar) .....	288
<b>cwRTF Class</b>	<b>289</b>
cwRTF Overview .....	289
cwRTF Class Concepts .....	289
cwRTF Relationship to Other Application Builder Classes .....	290

cwRTF ABC Template Implementation .....	290
cwRTF Source Files .....	290
cwRTF Properties .....	291
cwRTF Properties .....	291
hRTFWindow(RTF control handle) .....	291
cwRTF Methods .....	292
AlignParaCenter (center paragraph) .....	292
AlignParaLeft (left justify paragraph) .....	292
AlignParaRight (right justify paragraph) .....	292
ChangeFontStyle (set current font style) .....	293
CanRedo (check for redo data) .....	293
CanUndo (check for undo data) .....	294
Color (set text color) .....	294
Copy (copy selected text to clipboard) .....	294
Cut (cut selected text) .....	295
Find (set current font style) .....	295
FlatButton (use flat button style) .....	296
Font (apply font attributes) .....	297
GetText (copy text to variable) .....	298
Init (initialize the cwRTF object) .....	299
IsDirty (indicates modified data) .....	300
Kill (shut down the csRTF object) .....	301
LeftIndent (indent the current or selected paragraph) .....	301
LimitTextSize (limit amount of text) .....	302
LoadField (load rich text data from field) .....	302
LoadFile (load rich text data from file) .....	303
NewFile (clear rich text data) .....	303
Offset (offset current or selected paragraph) .....	303
PageSetup (set page attributes) .....	304
ParaBulletsOff (turn off paragraph bullets) .....	304
ParaBulletsOn (turn on paragraph bullets) .....	304
Paste (paste text from clipboard) .....	305
_Print (print rich text control contents) .....	305
Redo (reapply action) .....	306
Replace (find and replace search) .....	306
ResizeControls (used internally) .....	306
RightIndent (indent the current or selected paragraph) .....	307
SaveField (save rich text data to field) .....	307
SaveFile (save rich text data to file) .....	308
SelectText (select characters) .....	308
SetDirtyFlag (set modified flag) .....	309
SetFocus (give rich text control focus) .....	309
SetText (place text into rich text control) .....	310
ShowControl (hide/unhide RTF control) .....	311
Undo (undo action) .....	311

<b>DbAuditManager</b>	<b>313</b>
DbAuditManager Overview .....	313
Relationship to Other Application Builder Classes .....	313
DbAuditManager ABC Template Implementation .....	313
DbAuditManager Source Files .....	313
DbAuditManager Properties.....	314
Action (log file action column) .....	314
ColumnInfo (log file column queue) .....	314
LogFiles (log file queue) .....	314
FM (DbLogFileManager object) .....	315
Errors (ErrorClass object) .....	315
DbAuditManager Methods .....	316
AddItem (maintain the columninfo structure) .....	316
AddLogFile (maintain log file structure) .....	317
AppendLog (initiate audit log file update) .....	317
BeforeChange (update audit log file before file change) .....	318
CreateHeader (create log file header records) .....	318
Init (initialize the DbAuditManager object) .....	319
Kill (shut down DbAuditManger object) .....	319
OnChange (update audit log file after a record change) .....	320
OnDelete (update audit log file when a record is deleted) .....	320
OnFieldChange (virtual method for each field change) .....	321
OnInsert (update audit log file when a record is added) .....	322
OpenLogFile (open the audit log file) .....	322
SetFM (determine log file status) .....	323
<b>DbChangeManager</b>	<b>325</b>
DbChangeManager Overview .....	325
Relationship to Other Application Builder Classes .....	325
DbChangeManager ABC Template Implementation .....	325
DbChangeManager Source Files .....	325
DbChangeManager Properties .....	326
NameQueue (pointer into trigger queue) .....	326
TriggerQueue (pointer to BFP for field changes) .....	326
DbChangeManager Methods.....	327
AddItem (maintain the namequeue structure) .....	327
AddThread (maintains the triggerqueue) .....	327
CheckChanges(check record for changes) .....	328
CheckPair(check field pairs for changes) .....	328
Equal(checks for equal before and after values) .....	329
Init (initialize the DbChangeManager object) .....	329
Kill (shut down DbChangeManger object) .....	330
SetThread (read triggerqueue) .....	330
Update (update the audit log file buffer) .....	331

<b>DbLogFileManager</b>	<b>333</b>
DbLogFileManager Overview .....	333
Relationship to Other Application Builder Classes .....	333
DbLogFileManager ABC Template Implementation .....	333
DbLogFileManager Source Files .....	333
DbLogFileManager Properties .....	334
DbLogFileManager Properties .....	334
Opened (file opened flag) .....	334
DbLogFileManager Methods.....	335
DbLogFileManager Methods .....	335
Init (initialize the DbLogFileManager object) .....	335
<b>EditClass</b>	<b>337</b>
EditClass Overview .....	337
EditClass Concepts .....	337
Relationship to Other Application Builder Classes .....	337
ABC Template Implementation .....	338
EditClass Source Files .....	338
EditClass Conceptual Example .....	339
EditClass Properties .....	343
FEQ (the edit-in-place control number) .....	343
ReadOnly ( edit-in-place control is read-only).....	343
EditClass Methods .....	344
Functional Organization--Expected Use .....	344
CreateControl (a virtual to create the edit control) .....	345
Init (initialize the EditClass object) .....	346
Kill (shut down the EditClass object) .....	347
SetAlerts (alert keystrokes for the edit control) .....	347
SetReadOnly (set edit control to read-only) .....	348
TakeAccepted (validate EIP field) .....	348
TakeEvent (process edit-in-place events) .....	349
<b>EditSpinClass</b>	<b>351</b>
EditSpinClass--Overview .....	351
EditSpinClass Concepts .....	351
EditSpinClass -- Relationship to Other Application Builder Classes .....	351
EditSpinClass--ABC Template Implementation .....	351
EditSpinClass--Conceptual Example .....	352
EditSpinClass Properties .....	355
EditSpinClass Properties .....	355
EditSpinClass Methods.....	356
EditSpinClass Methods .....	356
EditSpinClass--Functional Organization--Expected Use .....	356
CreateControl (create the edit-in-place SPIN control) .....	357

---

<b>EditCheckClass</b>	<b>359</b>
EditCheckClass Overview .....	359
EditCheckClass Concepts .....	359
EditCheckClass Relationship to Other Application Builder Classes .....	359
EditCheckClass ABC Template Implementation .....	359
EditCheckClass Source Files .....	359
EditCheckClass Conceptual Example .....	361
EditCheckClass Properties .....	364
EditCheckClass Methods.....	365
EditCheckClass Functional Organization—Expected Use .....	365
CreateControl (create the edit-in-place CHECK control) .....	366
<b>EditColorClass</b>	<b>367</b>
EditColorClassOverview .....	367
EditColorClass Concepts .....	367
EditColorClass Relationship to Other Application Builder Classes .....	367
EditColorClass ABC Template Implementation .....	367
EditColorClass Source Files .....	368
EditColorClass Conceptual Example .....	369
EditColorClass Properties.....	373
EditColorClass Properties .....	373
Title (color dialog title text) .....	373
EditColorClass Methods .....	374
EditColorClass Functional Organization--Expected Use .....	374
CreateControl (create the edit-in-place control) .....	375
TakeEvent (process edit-in-place events:EditColorClass) .....	376
<b>EditDropComboClass</b>	<b>377</b>
EditDropComboClass Overview .....	377
EditDropComboClass Concepts .....	377
Relationship to Other Application Builder Classes .....	377
EditDropComboClass ABC Template Implementation .....	377
EditDropComboClass Source Files .....	378
EditDropComboClass Conceptual Example .....	379
EditDropComboClass Properties .....	382
EditDropComboClass Methods .....	383
EditDropComboClass Functional Organization .....	383
CreateControl (create the edit-in-place COMBO control) .....	384
<b>EditDropListClass</b>	<b>385</b>
EditDropListClass Overview .....	385
EditDropListClass Concepts .....	385
EditDropListClass Relationship to Other Application Builder Classes .....	385
EditDropListClass ABC Template Implementation .....	385
EditDropListClass Source Files .....	385
EditDropListClass Conceptual Example .....	387

EditDropListClass Properties .....	390
EditDropListClass Properties .....	390
EditDropListClass Methods.....	391
EditDropListClass Functional Organization--Expected Use .....	391
CreateControl (create the edit-in-place DROPLIST control) .....	392
SetAlerts (alert keystrokes for the edit control:EditDropListClass) .....	393
SetReadOnly (set edit control to read-only:EditDropClass) .....	394
TakeEvent (process edit-in-place events:EditDropList Class) .....	395
<b>EditEntryClass</b> .....	<b>397</b>
EditEntryClass Overview .....	397
EditEntryClass Concepts .....	397
EditEntryClass Relationship to Other Application Builder Classes .....	397
EditEntryClass ABC Template Implementation .....	398
EditEntryClass Source Files .....	398
EditEntryClass Conceptual Example .....	398
EditEntryClass Properties .....	402
EditEntryClass Properties .....	402
EditEntryClass Methods.....	403
EditEntryClass Methods .....	403
EditEntryClass Functional Organization--Expected Use .....	403
CreateControl (create the edit-in-place ENTRY control) .....	404
<b>EditFileClass</b> .....	<b>405</b>
EditFileClass Overview .....	405
EditFileClass Concepts .....	405
EditFileClass Relationship to Other Application Builder Classes .....	405
EditFileClass ABC Template Implementation .....	405
EditFileClass Source Files .....	406
EditFileClass Conceptual Example .....	407
EditFileClass Properties.....	411
EditFileClass Properties .....	411
FileMask (file dialog behavior) .....	411
FilePattern (file dialog filter) .....	411
Title (file dialog title text) .....	412
EditFileClass Methods .....	413
EditFileClass Functional Organization--Expected Use .....	413
CreateControl (create the edit-in-place control:EditFileClass) .....	414
TakeEvent (process edit-in-place events:EditFileClass) .....	415
<b>EditFontClass</b> .....	<b>417</b>
EditFontClass Overview .....	417
EditFontClass Concepts .....	417
EditFontClass Relationship to Other Application Builder Classes .....	417
EditFontClass ABC Template Implementation .....	417
EditFontClass Source Files .....	418

EditFontClass Conceptual Example .....	419
EditFontClass Properties .....	423
EditFontClass Properties .....	423
Title (font dialog title text) .....	423
EditFontClass Methods.....	424
EditFontClass Methods .....	424
EditFontClass Functional Organization--Expected Use .....	424
CreateControl (create the edit-in-place control:EditFontClass) .....	425
TakeEvent (process edit-in-place events:EditFontClass) .....	426
<b>EditMultiSelectClass</b> .....	<b>427</b>
EditMultiSelectClass Overview .....	427
EditMultiSelectClass Concepts .....	427
EditMultiSelectClass Relationship to Other Application Builder Classes .....	427
EditMultiSelectClass ABC Template Implementation .....	428
EditMultiSelectClass Source Files .....	428
EditMultiSelectClass Conceptual Example .....	429
EditMultiSelectClass Properties.....	434
EditMultiSelectClass Properties .....	434
Title (font dialog title text:EditMultiSelectClass) .....	434
EditMultiSelectClass Methods .....	435
EditMultiSelectClass Methods .....	435
EditMultiSelectClass Functional Organization--Expected Use .....	435
AddValue (prime the MultiSelect dialog) .....	437
CreateControl (create the edit-in-place control:EditMultiSelectClass) .....	438
Reset (reset the EditMultiSelectClass object) .....	438
TakeAction (process MultiSelect dialog action) .....	439
TakeEvent (process edit-in-place events:EditMultiSelectClass) .....	442
<b>EditTextClass</b> .....	<b>443</b>
EditTextClass: Overview .....	443
EditTextClass Properties .....	445
Title (text dialog title text) .....	445
EditTextClass Methods .....	446
CreateControl (create the edit-in-place control:EditTextClass) .....	447
TakeEvent (process edit-in-place events:EditTextClass) .....	448
<b>EIPManagerClass</b> .....	<b>449</b>
EIPManagerClass--Overview .....	449
EIPManagerClass Concepts .....	449
EIPManagerClass--Relationship to Other Application Builder Classes .....	449
EIPManagerClass--ABC Template Implementation .....	450
EIPManagerClass Source Files .....	450
EIPManagerClass--Conceptual Example .....	451
EIPManagerClass Properties.....	454
Again (column usage flag) .....	454

Arrow (edit-in-place action on arrow key) .....	454
Column (listbox column) .....	454
Enter (edit-in-place action on enter key) .....	455
EQ (list of edit-in-place controls) .....	455
Fields (managed fields) .....	456
FocusLoss ( action on loss of focus) .....	456
Insert (placement of new record) .....	457
ListControl (listbox control number) .....	458
LastColumn (previous edit-in-place column) .....	458
Repost (event synchronization) .....	459
RepostField (event synchronization field) .....	459
Req (database request) .....	460
SeekForward (get next field flag) .....	460
Tab (action on a tab key) .....	460
EIPManagerClass Methods .....	461
EIPManagerClass--Functional Organization--Expected Use .....	461
AddControl (register edit-in-place controls) .....	463
ClearColumn (reset column property values:EIPManagerClass) .....	464
GetEdit (identify edit-in-place field) .....	465
Init (initialize the EIPManagerClass object) .....	466
InitControls (initialize edit-in-place controls) .....	466
Kill (shut down the EIPManagerClass object) .....	467
Next (get the next edit-in-place field) .....	468
ResetColumn (reset edit-in-place object to selected field) .....	469
Run (run the EIPManager) .....	470
TakeAcceptAll (validate completed row) .....	470
TakeAction (process edit-in-place action) .....	471
TakeCompleted (process completion of edit:EIPManagerClass) .....	472
TakeEvent (process window specific events) .....	473
TakeFieldEvent (process field specific events) .....	474
TakeFocusLoss (a virtual to process loss of focus) .....	475
TakeNewSelection (reset edit-in-place column:EIPManagerClass) .....	476

## **EntryLocatorClass**

**477**

EntryLocatorClass Overview .....	477
EntryLocatorClass Properties .....	481
EntryLocatorClass Properties .....	481
Shadow (the search value) .....	481
EntryLocatorClass Methods.....	482
GetShadow(return shadow value) .....	482
Init (initialize the EntryLocatorClass object) .....	483
Set (restart the locator:EntryLocatorClass) .....	484
SetShadow(set shadow value) .....	485
TakeAccepted (process an accepted locator value:EntryLocatorClass) .....	485
TakeKey (process an alerted keystroke:EntryLocatorClass) .....	486
Update (update the locator control and free elements) .....	487
UpdateWindow (redraw the locator control) .....	487

<b>ErrorClass</b>	<b>489</b>
ErrorClass Overview .....	489
ErrorClass Source Files .....	489
Multiple Customizable Levels of Error Treatment .....	490
Predefined Windows and Database Errors .....	491
Dynamic Extensibility of Errors.....	491
ErrorClass ABC Template Implementation .....	491
ErrorClass Relationship to Other Application Builder Classes.....	491
ErrorClass Macro Expansion.....	492
ErrorClass Multi-Language Capability.....	493
ErrorClass Conceptual Example .....	494
ErrorClass Properties.....	495
ErrorClass Properties .....	495
DefaultCategory (error category) .....	495
ErrorLog (errorlog interface) .....	495
Errors (recognized error definitions) .....	496
FieldName (field that produced the error) .....	497
FileName (file that produced the error) .....	497
History (error history structure) .....	497
HistoryResetOnView(clear error history log file) .....	498
HistoryThreshold (determine size of error history) .....	498
HistoryViewLevel (trigger error history) .....	498
KeyName (key that produced the error) .....	498
LogErrors (turn on error history logging) .....	499
MessageText (custom error message text) .....	499
Silent (silent error flag) .....	499
ErrorClass Methods .....	500
ErrorClass Functional Organization--Expected Use .....	500
AddErrors (add or override recognized errors) .....	502
AddHistory (update History structure) .....	503
GetCategory (retrieve error category) .....	503
GetError (Retrieve the current error message) .....	503
GetErrorcode (Retrieve the current Errorcode).....	504
GetProcedureName (return procedure name ) .....	504
HistoryMsg (initialize the message window) .....	505
Init (initialize the ErrorClass object) .....	506
Kill (perform any necessary termination code) .....	507
Message (display an error message) .....	508
Msg (initiate error message destination) .....	509
MessageBox (display error message to window) .....	510
RemoveErrors (remove or restore recognized errors) .....	511
ResetHistory(clear History structure) .....	512
SetCategory (set error category) .....	512
SetErrors (save the error state) .....	513
SetFatality (set severity level for a particular error) .....	514
SetField (set the substitution value of the %Field macro) .....	515

SetFile (set the substitution value of the %File macro) .....	516
SetKey (set the substitution value of the %Key macro) .....	517
SetId (make a specific error current) .....	518
SetProcedureName ( stores procedure names) .....	519
SubsString (resolves error message macros) .....	520
TakeBenign (process benign error) .....	521
TakeError (process specified error) .....	522
TakeFatal (process fatal error) .....	523
TakeNotify (process notify error) .....	524
TakeOther (process other error) .....	525
TakeProgram (process program error) .....	526
TakeUser (process user error) .....	527
Throw (process specified error) .....	528
ThrowFile (set value of %File, then process error) .....	529
ThrowMessage (set value of %Message, then process error) .....	530
ViewHistory (initiates the view of the current errors) .....	530
<b>ErrorLogInterface</b> .....	<b>531</b>
ErrorLogInterface Concepts .....	531
Relationship to Other Application Builder Classes .....	531
ErrorLogInterface Source Files .....	531
ErrorLogInterface Methods .....	532
ErrorLogInterface Methods .....	532
Close (initiate close of log file) .....	532
Open (method to initiate open of log file) .....	533
Take (update the log file) .....	533
<b>FieldPairsClass</b> .....	<b>535</b>
FieldPairsClass Overview .....	535
FieldPairsClass Properties.....	538
List (recognized field pairs) .....	538
FieldPairsClass Methods .....	539
FieldPairsClass Functional Organization--Expected Use .....	539
AddItem (add a field pair from one source field) .....	540
AddPair (add a field pair:FieldPairsClass) .....	541
AssignLeftToRight (copy from "left" fields to "right" fields) .....	543
AssignRightToLeft (copy from "right" fields to "left" fields) .....	544
ClearLeft (clear each "left" field) .....	545
ClearRight (clear each "right" field) .....	546
Equal (return 1 if all pairs are equal) .....	547
EqualLeftRight (return 1 if all pairs are equal) .....	548
Init (initialize the FieldPairsClass object) .....	548
Kill (shut down the FieldPairsClass object) .....	549
<b>FileDropComboClass</b> .....	<b>551</b>
Overview:FileDropComboClass .....	551

FileDropComboClass Properties ..... 556

    FileDropComboClass Properties ..... 556

    AskProcedure (update procedure) ..... 556

    ECon (current state of entry completion) ..... 556

    EntryCompletion (automatic fill-ahead flag) ..... 557

    RemoveDuplicatesFlag (remove duplicate data) ..... 557

    UseField (COMBO USE variable) ..... 558

FileDropComboClass Methods ..... 559

    FileDropComboClass Methods ..... 559

    FileDropComboClass Functional Organization--Expected Use ..... 559

    AddRecord (add a record filedrop queue) ..... 561

    Ask (add a record to the lookup file) ..... 561

    GetQueueMatch (locate a list item) ..... 562

    Init (initialize the FileDropComboClass object) ..... 563

    KeyValid (check for valid keystroke) ..... 565

    Kill (shut down the FileDropComboClass object) ..... 565

    ResetFromList (reset VIEW) ..... 565

    ResetQueue (refill the filedrop queue) ..... 566

    TakeAccepted (process accepted event) ..... 567

    TakeEvent (process the current ACCEPT loop event:FileDropComboClass) ... 567

    TakeNewSelection (process NewSelection events:FileDropComboClass) ..... 568

    UniquePosition (check queue for duplicate record by key position) ..... 569

**FileDropClass 571**

FileDropClass Overview ..... 571

FileDropClass Properties ..... 576

    FileDropClass Properties ..... 576

    AllowReset (allow a reset) ..... 576

    DefaultFill (initial display value) ..... 576

    InitSyncPair (initial list position) ..... 576

FileDropClass Methods..... 577

    FileDropClass Methods ..... 577

    FileDropClass Functional Organization--Expected Use ..... 577

    AddField (specify display fields) ..... 579

    AddRecord (update filedrop queue) ..... 580

    AddUpdateField (specify field assignments) ..... 580

    Init (initialize the FileDropClass object) ..... 581

    Kill (shut down the FileDropClass object) ..... 582

    ResetQueue (fill filedrop queue) ..... 583

    SetQueueRecord (copy data from file buffer to queue buffer:FileDropClass) . 584

    TakeAccepted (a virtual to accept data) ..... 584

    TakeEvent (process the current ACCEPT loop event--FileDropClass) ..... 585

    TakeNewSelection (process EVENT:NewSelection events:FileDropClass) .... 586

    ValidateRecord (a virtual to validate records) ..... 587

<b>FileManager</b>	<b>589</b>
FileManager Overview .....	589
FileManager Properties.....	594
AliasedFile (the primary file) .....	594
Buffer (the record buffer) .....	595
Buffers (saved record buffers) .....	595
Create (create file switch) .....	596
Errors (the ErrorManager) .....	596
File (the managed file) .....	596
FileName (variable filename) .....	597
FileNameValue (constant filename) .....	598
LazyOpen (delay file open until access) .....	599
LockRecover (/RECOVER wait time parameter) .....	599
OpenMode (file access/sharing mode) .....	600
SkipHeldRecords (HELD record switch) .....	600
FileManager Methods .....	601
Naming Conventions and Dual Approach to Database Operations .....	601
FileManager Functional Organization--Expected Use .....	602
AddField(track fields in a structure) .....	604
AddKey (set the file's keys) .....	605
BindFields (bind fields when file is opened) .....	606
CancelAutoInc (undo PrimeAutoInc) .....	607
ClearKey (clear specified key components) .....	610
Close (close the file) .....	612
Deleted (return record status) .....	613
DeleteRecord (delete a record) .....	614
Destruct (automatic destructor) .....	615
EqualBuffer (detect record buffer changes) .....	616
Fetch (get a specific record by key value) .....	617
GetComponents (return the number of key components) .....	618
GetEOF (return end of file status) .....	619
GetError (return the current error ID) .....	620
GetField (return a reference to a key component) .....	621
GetFieldName (return a key component field name) .....	623
GetFields(get number of fields) .....	625
GetFieldPicture(get field picture) .....	625
GetFieldType(get field type) .....	625
GetName (return the filename) .....	626
Init (initialize the FileManager object) .....	627
Insert (add a new record) .....	629
KeyToOrder (return ORDER expression for a key) .....	630
Kill (shutdown the FileManager object) .....	632
Next (get next record in sequence) .....	633
Open (open the file) .....	634
Position (return the current record position) .....	635
PostDelete(trigger delete action post-processing) .....	636

---

PostInsert(trigger insert action post-processing).....	637
PostUpdate(trigger update action post-processing).....	638
PreDelete(trigger delete action pre-processing) .....	639
PreInsert(trigger insert action pre-processing).....	640
PreUpdate(trigger update action pre-processing).....	642
Previous (get previous record in sequence) .....	644
PrimeAutoInc (prepare an autoincremented record for adding) .....	645
PrimeFields (a virtual to prime fields) .....	647
PrimeRecord (prepare a record for adding:FileManager) .....	648
RestoreBuffer (restore a previously saved record buffer) .....	650
RestoreFile (restore a previously saved file state) .....	651
SaveBuffer (save a copy of the record buffer) .....	652
SaveFile (save the current file state) .....	653
SetError (save the specified error and underlying error state) .....	654
SetErrors (set the error class used) .....	655
SetKey (set current key) .....	656
SetName (set current filename) .....	657
Throw (pass an error to the error handler for processing) .....	659
ThrowMessage (pass an error and text to the error handler) .....	660
TryFetch (try to get a specific record by key value) .....	662
TryInsert (try to add a new record) .....	663
TryNext (try to get next record in sequence) .....	664
TryOpen (try to open the file) .....	665
TryPrevious (try to get previous record in sequence) .....	666
TryPrimeAutoInc (try to prepare an autoincremented record for adding) .....	667
TryReget (try to get a specific record by position) .....	669
TryUpdate (try to change the current record) .....	669
TryValidateField(validate field contents) .....	670
Update (change the current record) .....	671
UseFile (use LazyOpen file) .....	672
ValidateField (validate a field) .....	674
ValidateFields (validate a range of fields) .....	675
ValidateFieldServer(validate field contents) .....	676
ValidateRecord (validate all fields) .....	677
<b>FilterLocatorClass</b> .....	<b>679</b>
FilterLocatorClass Overview .....	679
FilterLocatorClass Properties.....	683
FilterLocatorClass Properties .....	683
FloatRight ("contains" or "begins with" flag) .....	683
FilterLocatorClass Methods .....	684
FilterLocatorClass Methods .....	684
TakeAccepted (process an accepted locator value:FilterLocatorClass) .....	684
UpdateWindow (apply the search criteria) .....	685

<b>FuzzyClass</b>	<b>687</b>
FuzzyClass Overview .....	687
Relationship to Other Application Builder Classes .....	687
FuzzyClass ABC Template Implementation .....	687
FuzzyClass Source Files .....	687
FuzzyClass Properties .....	688
FuzzyClass Methods.....	688
Construct (initialize FuzzyClass object) .....	688
Init (initialize FuzzyClass object) .....	688
Kill (shutdown FuzzyClass object) .....	688
Match (find query matches) .....	689
SetOption (set fuzzymatch options) .....	690
<b>FormVCRClass</b>	<b>691</b>
FormVCRClass Overview .....	691
FormVCRClass Concepts.....	691
FormVCRClass Relationship to Other Application Builder Classes .....	691
FormVCRClass ABC Template Implementation.....	691
FormVCRClass Source Files .....	692
FormVCRClass Properties.....	692
QuickScan (buffered reads flag).....	692
Toolbar (FormVCR Toolbar object).....	693
ToolbarItem (FormVCR ToolbarTarget object) .....	693
ViewPosition (store the current record position) .....	694
FormVCRClass Methods: .....	695
AddToolbarTarget (set the FormVCR toolbar).....	695
Init (initialize the FormVCR object).....	696
InitSort (initialize locator values).....	696
Kill (shut down the FormVCR object) .....	697
CheckBorders (check for existence of records) .....	697
GetAction (return FormVCR action) .....	698
GetActionAllowed (validate a requested FormVCR action) .....	699
Next (get the next FormVCR item).....	700
Previous (get the previous FormVCR item) .....	701
ResetSort (apply sort order to FormVCR).....	701
SetAlerts (alert keystrokes for FormVCR controls) .....	702
SetRequestControl (assign field equates to FormVCR actions) .....	702
SetVCRControls (assign field equates to FormVCR scrolling) .....	703
SetSort (apply a sort order to the FormVCR group).....	704
TakeAcceptedLocator (apply an accepted FormVCR locator value) .....	705
TakeEvent (process the current ACCEPT loop event).....	705
TakeLocate (a FormVCR virtual to process each sort) .....	706
TakeScroll (process a FormVCR scroll event) .....	706
UpdateWindow (update display variables to match FormVCR action) .....	707

<b>GraphClass</b>	<b>709</b>
GraphClass Overview .....	709
Relationship to Other Application Builder Classes .....	709
GraphClass ABC Template Implementation .....	709
GraphClass Source Files .....	709
GraphClass Properties.....	710
eShowSBonFirstThread (display on base status bar).....	710
eSumYMax (calculated maximum node value).....	710
gShowDiagramName (show diagram name on target).....	711
gShowDiagramNameV (show diagram value on target).....	712
gShowMouse (show mouse coordinates on target).....	713
gShowMouseX (show mouse X coordinate on target).....	714
gShowMouseY (show mouse Y coordinate on target).....	715
gShowNodeName (show node name on target).....	716
gShowNodeNameV (show node name value on target).....	717
gShowNodeValue (show node axis values on target).....	718
gShowNodeValueX (show node x-axis value on target).....	719
gShowNodeValueY (show node y-axis value on target).....	720
GraphClass Methods .....	721
AllText (return full graph text information).....	721
BeginRefresh (prepare drawing of graph class object).....	721
CalcBestPositionNodeText (calculate graph text best fit position).....	722
CalcCurrentGraph (calculates values for current graph type).....	722
CalcCurrentNode (calculates values of current node).....	723
CalcGraph (calculates all graph object values).....	723
CalcPopup (create popup menu for graph object).....	724
CalcPopupAdd2 (create popup menu item text for graph object).....	725
DiagramNameText (create diagram name text).....	726
DiagramText (create diagram name text with prompts).....	726
DiagramNameText (create diagram name text).....	727
DiagramText (create diagram name text with prompts).....	727
Draw (calculate and draw GraphClass object).....	728
DrawGraph (draws calculated values).....	728
DrawReport (draw graph object on report).....	729
DrawWallpaper (draw background wallpaper for graph object).....	729
DrillDown (transfer control to new graph object).....	730
FindNearbyNodes (locate nodes based on mouse position).....	731
GetMouse (get mouse coordinates in all formats).....	732
GetValueFromField (get contents of specified field).....	733
GetValueFromStatusBar (return status bar zone contents).....	733
ImageToWMF (Save object and return WMF file name).....	734
Init (Initialize the graph object).....	734
Interactivity (process mouse location data to tool tip or control.....	735
IsOverNode ( is mouse over node location).....	735
Kill (shut down the GraphClass object).....	736
MouseEvent (creates text and mouse coordinate information).....	736

MouseXText (generate X coordinate text only) .....	736
MouseYText (generate Y coordinate text only) .....	737
NodeNameText (generate current node name identifier) .....	737
NodeText (generate label, name, and node value) .....	738
NodeTipText (generate node information for tool tip) .....	738
NodeValueText (generate current node value text) .....	739
NodeXText (generate X node text value) .....	739
NodeYText (generate Y node text value) .....	740
Popup (GraphClass object popup menu manager) .....	740
PopupAsk (Display popup menu for graph object) .....	741
PostEvent (send an event to the GraphClass object) .....	742
PrintGraph (send graph object to printer) .....	744
Refresh (refresh drawing of GraphClass object) .....	745
Resize (conditional refresh when size changed) .....	745
ReturnFromDrillDown ( transfer control to graph object after drilldown) .....	746
SaveAsGraph (save graph to WMF file selected) .....	747
SaveGraph (auto-save graph to WMF file) .....	747
SetDefault (initialize selected graph properties) .....	748
ShowOnField (show text contents to specified field) .....	748
ShowOnStatusBar (show text to status bar zone) .....	749
TakeEvent (process graph control events) .....	749
TakeEventofParent (process all graph events) .....	750
ToolTip (show all text to tool tips) .....	750
ToShowValues (show all composite text to all graph targets) .....	751

**GridClass****753**

GridClass Overview .....	753
Relationship to Other Application Builder Classes .....	753
GridClass ABC Template Implementation .....	753
GridClass Source Files .....	753
GridClass Properties .....	754
GridClass Properties .....	754
Children (reference to child group controls) .....	754
Chosen (current browse queue element) .....	754
ClickPress (forward control) .....	755
ControlBase (base control number) .....	755
ControlNumber (number of controls) .....	755
GroupColor (background color of group fields) .....	756
GroupControl (GROUP control number) .....	756
GroupTitle (title of group element) .....	756
SelColor (color of selected element) .....	757
Selectable (element selectable flag) .....	757
UpdateControl (file update trigger) .....	757
UpdateControlEvent .....	757
GridClass Methods .....	758
AddLocator (specify a locator) .....	758
FetchRecord (retrieve selected record) .....	758

GetAcross (number of horizontal grids) .....	759
GetClickPress (forward click control) .....	759
GetDown (number of vertical grids) .....	759
GetPosition (retrieve group control position) .....	760
IfGroupField (determine if current control is a GROUP) .....	760
Init (initialize the GridClass object) .....	761
IsSkelActive .....	761
Kill (shutdown the GridClass object) .....	762
SetAlerts (initialize and create child controls) .....	762
SyncGroup (initialize GROUP field properties) .....	762
TakeEvent (process the current ACCEPT loop event) .....	763
UpdateRecord (refresh BrowseGrid) .....	763
UpdateWindow (refresh window display) .....	763
<b>HistHandlerClass</b> .....	<b>765</b>
HistHandlerClass Source Files .....	765
HistHandlerClass Properties.....	765
Err (errorclass object) .....	765
History (error history structure) .....	765
LBColumns (number of listbox columns) .....	766
Win (reference to window) .....	766
HistHandlerClass Methods .....	767
Init (initialize the HistHandlerClass object) .....	767
TakeEvent (process window events) .....	767
VLBProc (retrieve LIST and error history information.) .....	768
<b>IDbChangeAudit Interface</b> .....	<b>769</b>
IDbChangeAudit Concepts .....	769
Relationship to Other Application Builder Classes .....	769
IDbChangeAudit Source Files .....	769
IDbChangeAudit Methods.....	769
BeforeChange (update audit log file before file change) .....	769
ChangeField (virtual method for managing field changes) .....	770
OnChange (update audit log file after a record change) .....	771
<b>IListControl Interface</b> .....	<b>773</b>
IListControl Concepts .....	773
Relationship to Other Application Builder Classes .....	773
IListControl Source Files .....	773
IListControl Methods.....	773
Choice(returns current selection number) .....	773
GetControl(returns control number) .....	774
GetItems(returns number of entries) .....	774
GetVisible(returns visibility of control) .....	774
SetChoice(change selected entry) .....	775
SetControl(change selected entry) .....	775

<b>IncrementalLocatorClass</b>	<b>777</b>
IncrementalLocatorClass Overview .....	777
IncrementalLocatorClass Properties.....	781
IncrementalLocatorClass Properties .....	781
IncrementalLocatorClass Methods .....	782
IncrementalLocatorClass Methods .....	782
SetAlerts (alert keystrokes for the LIST control:IncrementalLocatorClass) .....	782
TakeKey (process an alerted keystroke:IncrementalLocatorClass) .....	783
<b>INIClass</b>	<b>785</b>
INIClass Overview .....	785
INIClass Properties .....	788
INIClass Properties .....	788
FileName .....	788
INIClass Methods.....	789
Fetch (get INI file entries) .....	789
FetchField (return comma delimited INI file value) .....	791
FetchQueue (get INI file queue entries) .....	792
Init (initialize the INIClass object) .....	793
TryFetch (get a value from the INI file) .....	795
TryFetchField (return comma delimited INI file value) .....	796
Update (write INI file entries) .....	797
<b>IReportGenerator Interface</b>	<b>799</b>
IReportGenerator Interface.....	799
IReportGenerator Concepts.....	799
IReportGenerator Methods .....	799
AskProperties (pop up window to set properties).....	799
CloseDocument (end document printing).....	800
ClosePage (end a page print) .....	800
GetProperty (get a property value).....	801
Init (initialize error class before printing) .....	801
OpenDocument (begin document printing) .....	802
OpenPage (begin a page print).....	802
Opened (file opened flag) .....	802
ProcessArc (print an arc).....	803
ProcessBand (begin/end report band processing).....	804
ProcessCheck (print a checkbox).....	804
ProcessChord (print a section of an ellipse) .....	805
ProcessEllipse (print an ellipse) .....	806
ProcessImage (print an image) .....	806
ProcessLine (print a line).....	807
ProcessOption (print an option control).....	807
ProcessRadio (print a radio button).....	808
ProcessRectangle (print a box control) .....	808
ProcessText (print a string or text control) .....	809

SetProperty (set a property value) .....	810
WhoAml (identify the report generator type) .....	810
<b>LocatorClass</b>	<b>811</b>
LocatorClass Overview .....	811
LocatorClass Properties.....	813
Control (the locator control number) .....	813
FreeElement (the locator's first free key element) .....	813
NoCase (case sensitivity flag) .....	814
ViewManager (the locator's ViewManager object) .....	814
LocatorClass Methods .....	815
GetShadow(return shadow value) .....	815
Init (initialize the LocatorClass object) .....	816
Reset (reset the locator for next search) .....	817
Set (restart the locator:LocatorClass) .....	818
SetAlerts (alert keystrokes for the LIST control:LocatorClass) .....	818
SetEnabled (enable or disable the locator control) .....	819
SetShadow (update shadow value) .....	819
TakeAccepted (process an accepted locator value:LocatorClass) .....	820
TakeKey (process an alerted keystroke:LocatorClass) .....	820
UpdateWindow (redraw the locator control with its current value) .....	821
<b>MsgBoxClass</b>	<b>823</b>
MsgBoxClass Overview .....	823
MsgBoxClass Source Files .....	823
MsgBoxClass Properties.....	824
ButtonTypes (standard windows buttons) .....	824
Caption (window title) .....	824
Err (errorclass object) .....	824
Icon (icon for image control).....	824
HistoryHandler (windowcomponent interface) .....	825
MsgRVal (message box return value) .....	825
Style (font style) .....	825
Win (reference to window) .....	825
MsgBoxClass Methods .....	826
FetchFeq (retrieve button feq) .....	826
FetchStdButton (determine button pressed) .....	826
Init (initialize the MsgBoxClass object) .....	827
Kill (perform any necessary termination code) .....	827
SetupAdditionalFeqs (initialize additional control properties) .....	828
TakeAccepted (process accepted event) .....	828
<b>PopupClass</b>	<b>829</b>
PopupClass Overview .....	829
PopupClass Properties .....	833
ClearKeycode (clear KEYCODE character) .....	833

PopupClass Methods.....	834
PopupClass Functional Organization--Expected Use .....	834
AddItem (add menu item) .....	835
AddItemEvent (set menu item action) .....	837
AddItemMimic (tie menu item to a button) .....	838
AddMenu (add a menu) .....	839
AddSubMenu (add submenu) .....	840
Ask (display the popup menu) .....	841
DeleteItem (remove menu item) .....	842
DeleteMenu (remove a popup submenu) .....	843
GetItemChecked (return toggle item status) .....	844
GetItemEnabled (return item status) .....	845
GetItems(returns number of popup entries) .....	846
GetLastNumberSelection (get last menu item number selected) .....	846
GetLastSelection (return selected item) .....	847
Init (initialize the PopupClass object) .....	848
Kill (shut down the PopupClass object) .....	848
Restore (restore a saved menu) .....	849
Save (save a menu for restoration) .....	850
SetIcon (set icon name for popup menu item) .....	851
SetItemCheck (set toggle item status) .....	852
SetItemEnable (set item status) .....	853
SetLevel (set menu item level) .....	854
SetText (set menu item text) .....	855
SetToolbox (set menu item toolbox status).....	856
SetTranslator (set run-time translator:PopupClass) .....	857
Toolbox (start the popup toolbox menu) .....	858
ViewMenu (popup menu debugger) .....	859
<b>PrintPreviewClass</b>	<b>861</b>
PrintPreviewClass Overview .....	861
PrintPreviewClass Properties .....	866
AllowUserZoom (allow any zoom factor) .....	866
ConfirmPages (force 'pages to print' confirmation) .....	866
CurrentPage (the selected report page) .....	867
Maximize (number of pages displayed horizontally) .....	867
PagesAcross (number of pages displayed horizontally) .....	867
PagesDown (number of vertical thumbnails) .....	868
PagesToPrint (the pages to print) .....	868
UserPercentile (custom zoom factor) .....	868
WindowPosSet (use a non-default initial preview window position) .....	869
WindowSizeSet (use a non-default initial preview window size) .....	869
ZoomIndex (index to applied zoom factor) .....	870
PrintPreviewClass Methods.....	871
PrintPreviewClass Functional Organization--Expected Use .....	871
AskPage (prompt for new report page) .....	872
AskPrintPages (prompt for pages to print) .....	873

AskThumbnails (prompt for new thumbnail configuration) ..... 875  
 DeletelImageQueue (remove non-selected pages) ..... 876  
 Display (preview the report) ..... 877  
 Init (initialize the PrintPreviewClass object) ..... 878  
 InPageList (check page number) ..... 879  
 Kill (shut down the PrintPreviewClass object) ..... 880  
 Open (prepare preview window for display) ..... 881  
 SetlNIManager (save and restore window coordinates) ..... 882  
 SetPosition (set initial preview window coordinates) ..... 883  
 SetZoomPercentile (set user or standard zoom factor) ..... 884  
 SetDefaultPages (set the default pages to print) ..... 885  
 SyncImageQueue (sync image queue with PagesToPrint) ..... 885  
 TakeAccepted (process EVENT:Accepted events:PrintPreviewClass) ..... 886  
 TakeEvent (process all events:PrintPreviewClass) ..... 887  
 TakeFieldEvent (a virtual to process field events:PrintPreviewClass) ..... 888  
 TakeWindowEvent (process non-field events:PrintPreviewClass) ..... 889

**ProcessClass 891**

ProcessClass Overview ..... 891  
 ProcessClass Properties..... 895  
     CaseSensitiveValue (case sensitive flag) ..... 895  
     Percentile (portion of process completed) ..... 895  
     PText (progress control number) ..... 896  
     RecordsProcessed (number of elements processed) ..... 896  
     RecordsToProcess (number of elements to process) ..... 896  
 ProcessClass Methods ..... 897  
     ProcessClass Functional Organization--Expected Use ..... 897  
     Init (initialize the ProcessClass object) ..... 899  
     Kill (shut down the ProcessClass object) ..... 901  
     Next (get next element) ..... 902  
     Reset (position to the first element) ..... 903  
     SetProgressLimits (calibrate the progress monitor) ..... 903  
     TakeLocate (a virtual to process each filter) ..... 904  
     TakeRecord (a virtual to process each report record) ..... 904

**QueryClass 905**

QueryClass Overview ..... 905  
 QueryClass Concepts ..... 905  
 QueryClass Relationship to Other Application Builder Classes ..... 905  
 QueryClass ABC Template Implementation ..... 906  
 QueryClass Source Files ..... 906  
 QueryClass Conceptual Example ..... 907  
 QueryClass Properties..... 910  
     QKCurrentQuery ( popup menu choice ) ..... 910  
     QKIcon ( icon for popup submenu ) ..... 910  
     QKMenulcon ( icon for popup menu ) ..... 910

QKSupport ( quickqbe flag) .....	911
Window ( browse window:QueryClass ) .....	911
QueryClass Methods .....	912
QueryClass Functional Organization--Expected Use .....	912
AddItem (add field to query) .....	913
Ask (a virtual to accept query criteria) .....	915
ClearQuery ( remove loaded query ) .....	916
Delete ( remove saved query ) .....	917
GetFilter (return filter expression) .....	918
GetLimit (get searchvalues) .....	920
Init (initialize the QueryClass object) .....	921
Kill (shut down the QueryClass object) .....	922
Reset (reset the QueryClass object) .....	923
Restore ( retrieve saved query ) .....	924
Save ( save a query ) .....	925
SetLimit (set search values) .....	926
SetQuickPopup ( add QuickQBE to browse popup ) .....	928
Take ( process QuickQBE popup menu choice ) .....	929
<b>QueryFormClass</b> .....	<b>931</b>
QueryFormClass Overview .....	931
QueryFormClass Concepts .....	931
QueryFormClass Relationship to Other Application Builder Classes .....	931
QueryFormClass ABC Template Implementation .....	932
QueryFormClass Source Files .....	932
QueryFormClass Conceptual Example .....	933
QueryFormClass Properties .....	936
QueryFormClass Methods .....	937
QueryFormClass Functional Organization--Expected Use .....	937
Ask (solicit query criteria) .....	938
Init (initialize the QueryFormClass object) .....	939
Kill (shut down the QueryFormClass object) .....	940
<b>QueryFormVisual</b> .....	<b>941</b>
QueryFormVisual Overview .....	941
QueryFormVisual Concepts .....	941
QueryFormVisual Relationship to Other Application Builder Classes .....	941
QueryFormVisual ABC Template Implementation .....	941
QueryFormVisual Source Files .....	942
QueryFormVisual Conceptual Example .....	943
QueryFormVisual Properties.....	946
QFC (reference to the QueryFormClass) .....	946
QueryFormVisual Methods .....	947
QueryFormVisual Functional Organization--Expected Use .....	947
GetButtonFeq(returns a field equate label) .....	948
Init (initialize the QueryFormVisual object) .....	949

ResetFromQuery ( reset the QueryFormVisual object ) .....	950
SetText ( set prompt text:QueryFormVisual ) .....	951
TakeAccepted (handle query dialog Accepted events: QueryFormVisual).....	952
TakeCompleted (complete the query dialog: QueryFormVisual) .....	953
TakeFieldEvent (a virtual to process field events:QueryFormVisual) .....	954
UpdateFields ( process query values ) .....	955
<b>QueryListClass</b> .....	<b>957</b>
QueryListClass--Overview .....	957
QueryListClass Concepts .....	957
QueryListClass--Relationship to Other Application Builder Classes .....	957
QueryListClass--ABC Template Implementation .....	957
QueryListClass Source Files .....	958
QueryListClass--Conceptual Example .....	959
QueryListClass Properties .....	962
QueryListClass Methods.....	963
QueryListClass--Functional Organization--Expected Use .....	963
Ask (solicit query criteria:QueryListClass) .....	964
Init (initialize the QueryListClass object) .....	965
Kill (shut down the QueryListClass object) .....	966
<b>QueryListVisual</b> .....	<b>967</b>
QueryListVisual--Overview .....	967
QueryListVisual Concepts .....	967
QueryListVisual--Relationship to Other Application Builder Classes .....	967
QueryListVisual--ABC Template Implementation .....	967
QueryListVisual Source Files .....	967
QueryListVisual--Conceptual Example .....	969
QueryListVisual Properties .....	972
QFC (reference to the QueryListClass) .....	972
OpsEIP (reference to the EditDropListClass) .....	972
FldsEIP (reference to the EditDropListClass) .....	972
ValueEIP(reference to QEditEntryClass) .....	973
QueryListVisual Methods.....	974
QueryListVisual--Functional Organization--Expected Use .....	974
Init (initialize the QueryListVisual object) .....	975
Kill (shutdown the QueryListVisual object) .....	976
ResetFromQuery ( reset the QueryList Visual object ) .....	977
SetAlerts (alert keystrokes for the edit control:QueryListVisual) .....	978
TakeAccepted (handle query dialog EVENT:Accepted events) .....	979
TakeCompleted (complete the query dialog) .....	980
TakeEvent (process edit-in-place events:QueryListVisual) .....	981
TakeFieldEvent (a virtual to process field events:QueryListVisual) .....	982
UpdateControl(updates the edit-in-place entry control) .....	983
UpdateFields ( process query values ) .....	983

<b>QueryVisualClass</b>	<b>985</b>
QueryVisualClass: Overview .....	985
QueryVisualClass Properties .....	986
QC (reference to the QueryClass) .....	986
Resizer (reference to the WindowResizeClass:QueryVisualClass) .....	986
QueryVisualClass Methods.....	987
Init (initialize the QueryVisual object ) .....	987
Kill (shut down the QueryVisual object) .....	988
Reset ( reset the dialog for display:QueryVisualClass ) .....	989
TakeAccepted (handle query dialog EVENT:Accepted events) .....	990
TakeFieldEvent (a virtual to process field events:QueryVisualClass) .....	991
TakeWindowEvent (a virtual to process non-field events:QueryVisualClass) .....	992
 <b>RelationManager</b>	 <b>993</b>
RelationManager Overview .....	993
RelationManager Properties .....	998
RelationManager Properties .....	998
Me (the primary file's FileManager object) .....	998
UseLogout (transaction framing flag) .....	998
RelationManager Methods.....	999
RelationManager Functional Organization--Expected Use .....	999
AddRelation (set a file relationship) .....	1000
AddRelationLink (set linking fields for a relationship) .....	1002
CancelAutoInc (undo autoincrement) .....	1004
Close (close a file and any related files) .....	1005
Delete (delete record subject to referential constraints) .....	1006
GetNbFiles(returns number of children) .....	1007
GetNbRelations(returns number of relations) .....	1007
GetRelation(returns reference to relation manager) .....	1008
GetRelationType(returns relation type) .....	1008
Init (initialize the RelationManager object) .....	1009
Kill (shut down the RelationManager object) .....	1010
ListLinkingFields (map pairs of linked fields) .....	1011
Open (open a file and any related files) .....	1012
Save (copy the current record and any related records) .....	1012
SetAlias (set a file alias) .....	1013
SetQuickScan (enable QuickScan on a file and any related files) .....	1014
Update (update record subject to referential constraints) .....	1015
 <b>ReportManager Class</b>	 <b>1017</b>
ReportManager Overview .....	1017
ReportManager Concepts .....	1017
ReportManager Properties.....	1022
Attribute (ReportAttributeManager object) .....	1022
BreakMan (BreakManagerClass object) .....	1022
DeferOpenReport (defer open) .....	1023

DeferWindow (defer progress window) .....	1023
KeepVisible (keep progress window visible) .....	1024
OutputFileQueue (advanced report generation filenames) .....	1024
Preview (PrintPreviewClass object) .....	1025
PreviewQueue (report metafile pathnames) .....	1025
Process (ProcessClass object) .....	1026
QueryControl (query button) .....	1026
Report (the managed REPORT) .....	1027
ReportTarget (IReportGenerator interface) .....	1027
SkipPreview (print rather than preview) .....	1027
TargetSelector (ReportTargetSelectorClass object) .....	1028
TargetSelectorCreated (report target active) .....	1028
WaitCursor (activate Wait cursor during report processing) .....	1029
WMFParser (WMFDocumentParser object) .....	1029
Zoom (initial report preview magnification) .....	1030
ReportManager Methods .....	1031
ReportManager Functional Organization--Expected Use .....	1031
AddItem (program the ReportManager object) .....	1032
Ask (display window and process its events:ReportManager) .....	1032
AskPreview (preview or print the report) .....	1033
CancelPrintReport (cancel report printing) .....	1034
EndReport (close the report) .....	1035
Kill (shut down the ReportManager object) .....	1037
Next (get next report record) .....	1038
Open (a virtual to execute on EVENT:OpenWindow--ReportManager) .....	1039
OpenReport (prepare report for execution) .....	1040
PrintReport (print the report) .....	1041
ProcessResultFiles (process generated output files) .....	1042
SetReportTarget (set ReportGenerator target) .....	1043
SetStaticControlsAttributes (set report's static controls) .....	1043
SetDynamicControlsAttributes (set report's static controls) .....	1044
TakeAccepted (process Accepted event) .....	1044
TakeCloseEvent (a virtual to process EVENT:CloseWindow) .....	1045
TakeNoRecords (process empty report) .....	1046
TakeRecord(process each record) .....	1046
TakeWindowEvent (a virtual to process non-field events:ReportManager) ..	1047

**RuleManager 1049**

Overview .....	1049
RuleManager Concepts .....	1049
RuleManager ABC Template Implementation .....	1051
Implementation Steps using hand code .....	1056
Rule Class Properties .....	1058
Rule Class Methods .....	1059
SetGlobalRule (post address to GlobalRule) .....	1061
ResetGlobalRule (clear address in GlobalRule) .....	1062
RuleIsBroken (test rule and return result) .....	1063

SetIndicator (set error indicator) .....	1064
RulesCollection Class Properties .....	1065
RulesCollection Class Methods.....	1065
Construct (initialize RulesCollection object).....	1065
Destruct (shut down RulesCollection object) .....	1066
RuleCount (count rules in the collection) .....	1066
BrokenRuleCount (count rules in the collection which are broken) .....	1066
AddRule (add a rule to this collection) .....	1067
AddControl (add managed control ).....	1068
AddControlToRule (add managed control ) .....	1069
CheckRule (check a particular rule).....	1070
CheckAllRules (check all rules in this collection) .....	1070
Item (locate a particular rule) .....	1071
TakeAccepted (handle acceptance of error indicators) .....	1072
SetEnumerateIcons (set icons for broken rules display) .....	1073
EnumerateBrokenRules (display a list of rules with status of each).....	1074
SetControlsStatus (set status of managed controls) .....	1075
NeedChangeControlStatus (check if control status needs to change) .....	1076
RulesManager Properties.....	1077
RulesManager Methods .....	1077
Construct (initialize RulesManager object) .....	1077
Destruct (shut down RulesManager object).....	1077
RulesManagerCount (count rules in the collection) .....	1078
BrokenRulesCount (count rules in the collection which are broken) .....	1078
AddRulesCollection (add a rule to this collection) .....	1079
CheckAllRules (check all rules in all collections) .....	1079
TakeAccepted (handle acceptance of error indicators) .....	1080
SetEnumerateIcons (set icons for broken rules display) .....	1081
EnumerateBrokenRules (display a list of rules with status of each).....	1082
SetControlsStatus (set status of managed controls) .....	1083
<b>SelectFileClass</b> .....	<b>1085</b>
SelectFileClass Concepts .....	1085
SelectFileClass Properties.....	1088
DefaultDirectory (initial path) .....	1088
DefaultFile (initial filename/filemask) .....	1088
Flags (file dialog behavior) .....	1089
WindowTitle (file dialog title text) .....	1089
SelectFileClass Methods .....	1090
AddMask (add file dialog file masks) .....	1090
Ask (display Windows file dialog) .....	1091
Init (initialize the SelectFileClass object) .....	1092
SetMask (set file dialog file masks) .....	1093
<b>StandardBehavior Class</b> .....	<b>1095</b>
StandardBehavior Overview .....	1095

StandardBehavior Class Concepts .....	1095
Relationship to Other Application Builder Classes .....	1095
StandardBehavior Source Files .....	1095
StandardBehavior Properties.....	1096
StandardBehavior Methods .....	1097
StandardBehavior Methods .....	1097
Init(initialize the StandardBehavior object) .....	1097
<b>StandardErrorLogClass</b> .....	<b>1099</b>
StandardErrorLogClass Overview .....	1099
StandardErrorLogClass Source Files .....	1099
ABC Template Implementation .....	1099
StandardErrorLogClass Properties.....	1100
StandardErrorLogClass Methods .....	1101
Close (close standarderrorlog file) .....	1101
Construct (initialize StandardErrorLogClass object) .....	1101
Destruct (remove the StandardErrorLogClass object) .....	1101
Open (open standarderrorlog file) .....	1102
<b>StepClass</b> .....	<b>1103</b>
StepClass Overview .....	1103
StepClass Properties .....	1105
Controls (the StepClass sort sequence) .....	1105
StepClass Methods.....	1106
GetPercentile (return a value's percentile:StepClass) .....	1106
GetValue (return a percentile's value:StepClass) .....	1107
Init (initialize the StepClass object) .....	1108
Kill (shut down the StepClass object) .....	1109
SetLimit (set smooth data distribution:StepClass) .....	1110
SetLimitNeeded (return static/dynamic boundary flag:StepClass) .....	1111
<b>StepCustomClass</b> .....	<b>1113</b>
StepCustomClass Overview .....	1113
StepCustomClass Properties.....	1117
Entries (expected data distribution) .....	1117
StepCustomClass Methods .....	1118
StepCustomClass Methods .....	1118
AddItem (add a step marker) .....	1118
GetPercentile (return a value's percentile:StepCustomClass) .....	1119
GetValue (return a percentile's value:StepCustomClass) .....	1120
Init (initialize the StepCustomClass object) .....	1121
Kill (shut down the StepCustomClass object) .....	1122
<b>StepLongClass</b> .....	<b>1123</b>
StepLongClass Overview .....	1123
StepLongClass Properties .....	1126

Low (lower boundary:StepLongClass) .....	1126
High (upper boundary:StepLongClass) .....	1126
StepLongClass Methods.....	1127
GetPercentile (return a value's percentile:StepLongClass) .....	1127
GetValue (return a percentile's value:StepLongClass) .....	1128
SetLimit (set smooth data distribution:StepLongClass) .....	1129
<b>StepLocatorClass</b>	<b>1131</b>
StepLocatorClass Overview .....	1131
StepLocatorClass Properties .....	1134
StepLocatorClass Methods.....	1135
StepLocatorClass Methods .....	1135
Set (restart the locator:StepLocatorClass) .....	1135
TakeKey (process an alerted keystroke:StepLocatorClass) .....	1136
<b>StepRealClass</b>	<b>1137</b>
StepRealClass Overview .....	1137
StepRealClass Properties.....	1140
StepRealClass Properties .....	1140
Low (lower boundary:StepRealClass) .....	1140
High (upper boundary:StepRealClass) .....	1140
StepRealClass Methods .....	1141
StepRealClass Methods .....	1141
GetPercentile (return a value's percentile:StepRealClass) .....	1141
GetValue (return a percentile's value:StepRealClass) .....	1142
SetLimit (set smooth data distribution:StepRealClass) .....	1143
<b>StepStringClass</b>	<b>1145</b>
StepStringClass Overview .....	1145
StepStringClass Properties.....	1150
LookupMode (expected data distribution) .....	1150
Root (the static portion of the step) .....	1151
SortChars (valid sort characters) .....	1151
TestLen (length of the static step portion) .....	1152
StepStringClass Methods .....	1153
GetPercentile (return a value's percentile) .....	1153
GetValue (return a percentile's value) .....	1154
Init (initialize the StepStringClass object) .....	1155
Kill (shut down the StepStringClass object) .....	1156
SetLimit (set smooth data distribution:StepStringClass) .....	1157
SetLimitNeeded (return static/dynamic boundary flag:StepStringClass) .....	1158
<b>TagHTMLHelp Class</b>	<b>1159</b>
TagHTMLHelpOverview .....	1159
TagHTMLHelp Class Concepts .....	1159
Relationship to Other Application Builder Classes .....	1159

TagHTMLHelp ABC Template Implementation .....	1159
TagHTMLHelp Source Files .....	1159
TagHTMLHelp Methods.....	1160
AlinkLookup (associative link lookup) .....	1160
CloseHelp (close HTML help file) .....	1161
GetHelpFile (get help file name) .....	1161
GetTopic (get current topic name) .....	1161
Init (initialize HTML Help object) .....	1162
KeyWordLookup (lookup keyword) .....	1163
Kill (shutdown the TagHTMLHelp object) .....	1163
SetHelpFile (set the current HTML Help file name) .....	1164
SetTopic (set the current HTML Help file topic) .....	1165
ShowIndex (open the HTML Help index tab) .....	1165
ShowSearch (open the HTML Help search tab) .....	1166
ShowTOC (open the HTML Help contents tab) .....	1166
ShowTopic (display a help topic) .....	1166
<b>TextWindowClass</b> .....	<b>1167</b>
TextWindowClass Overview .....	1167
TextWindowClass Concepts .....	1167
Relationship to Other Application Builder Classes .....	1167
ABC Template Implementation .....	1167
TextWindowClass Source Files .....	1167
TextWindowClass Properties.....	1168
SelE (ending edit position) .....	1168
SelS (starting edit position) .....	1168
Txt (field equate number) .....	1168
TextWindowClass Methods .....	1169
Init (initialize TextWindow object) .....	1169
Kill (shutdown TextWindow object) .....	1169
TakeAccepted (process window controls) .....	1170
<b>ToolbarClass</b> .....	<b>1171</b>
ToolbarClass Overview .....	1171
ToolbarClass Methods.....	1178
ToolbarClass Functional Organization--Expected Use .....	1178
AddTarget (register toolbar driven entity) .....	1179
DisplayButtons (enable appropriate toolbar buttons:ToolbarClass) .....	1180
Init (initialize the ToolbarClass object) .....	1181
Kill (shut down the ToolbarClass object) .....	1181
SetTarget (sets the active target) .....	1182
TakeEvent (process toolbar event:ToolbarClass) .....	1183
<b>ToolbarListBoxClass</b> .....	<b>1185</b>
ToolbarListBoxClass Overview .....	1185
ToolbarListboxClass Properties.....	1190

Browse (BrowseClass object) .....	1190
ToolbarListboxClass Methods.....	1191
DisplayButtons (enable appropriate toolbar buttons:ToolbarListboxClass) ...	1191
TakeEvent (convert toolbar events:ToolbarListboxClass ) .....	1192
TakeToolbar (assume control of the toolbar) .....	1193
TryTakeToolbar (return toolbar control indicator:ToolbarListBoxClass) .....	1194
<b>ToolbarReltreeClass</b> .....	<b>1195</b>
ToolbarReltreeClass Overview .....	1195
ToolbarReltreeClass Properties.....	1200
ToolbarReltreeClass Methods .....	1201
DisplayButtons (enable appropriate toolbar buttons:ToolbarReltreeClass) ...	1201
TakeToolbar (assume control of the toolbar:ToolbarReltreeClass) .....	1202
<b>ToolbarTargetClass</b> .....	<b>1203</b>
ToolbarTarget Overview .....	1203
ToolbarTarget Properties .....	1205
ChangeButton (change control number) .....	1205
Control (window control) .....	1205
DeleteButton (delete control number) .....	1206
HelpButton (help control number) .....	1206
InsertButton (insert control number) .....	1207
LocateButton(query control number) .....	1207
SelectButton (select control number) .....	1208
ToolbarTarget Methods.....	1209
ToolbarTarget Functional Organization--Expected Use .....	1209
DisplayButtons (enable appropriate toolbar buttons:ToolbarTarget) .....	1210
TakeEvent (convert toolbar events:ToolbarTarget) .....	1211
TakeToolbar (assume control of the toolbar:ToolbarTarget) .....	1213
TryTakeToolbar (return toolbar control indicator:ToolbarTarget) .....	1213
<b>ToolbarUpdateClass</b> .....	<b>1215</b>
ToolbarUpdateClass Overview .....	1215
ToolbarUpdateClass Properties.....	1223
Request (requested database operation) .....	1223
History (enable toolbar history button) .....	1224
ToolbarUpdateClass Methods .....	1225
DisplayButtons (enable appropriate toolbar buttons:ToolbarUpdateClass) ...	1225
TakeEvent (convert toolbar events:ToolbarUpdateClass) .....	1226
TakeToolbar (assume control of the toolbar:ToolbarUpdateClass) .....	1227
TryTakeToolbar (return toolbar control indicator:ToolbarUpdateClass) .....	1228
<b>TranslatorClass</b> .....	<b>1229</b>
TranslatorClass Overview .....	1229
TranslatorClass Properties .....	1234
ExtractText (identify text to translate) .....	1234

TranslatorClass Methods .....	1235
AddTranslation (add translation pairs) .....	1235
Init (initialize the TranslatorClass object) .....	1237
Kill (shut down the TranslatorClass object) .....	1237
TranslateControl (translate text for a control) .....	1238
TranslateControls (translate text for range of controls) .....	1239
TranslateString (translate text) .....	1240
TranslateWindow (translate text for a window) .....	1241

**ViewManager 1243**

ViewManager Overview .....	1243
ViewManager Properties.....	1248
Order (sort, range-limit, and filter information) .....	1248
PagesAhead (buffered pages) .....	1249
PagesBehind (buffered pages) .....	1249
PageSize (buffer page size) .....	1250
Primary (the primary file RelationManager ) .....	1250
SavedBuffers (saved record buffers).....	1251
TimeOut (buffered pages freshness) .....	1252
View (the managed VIEW) .....	1252
ViewManager Methods .....	1253
ViewManager Functional Organization--Expected Use .....	1253
AddRange (add a range limit) .....	1255
AddSortOrder (add a sort order) .....	1256
AppendOrder (refine a sort order) .....	1257
ApplyFilter (range limit and filter the result set) .....	1258
ApplyOrder (sort the result set) .....	1259
ApplyRange (conditionally range limit and filter the result set) .....	1260
Close (close the view) .....	1261
GetFirstSortField (return first field of current sort).....	1261
GetFreeElementName (return free key element name) .....	1262
GetFreeElementPosition (return free key element position) .....	1263
Init (initialize the ViewManager object) .....	1264
Kill (shut down the ViewManager object) .....	1265
Next (get the next element) .....	1265
Open (open the view) .....	1266
Previous (get the previous element) .....	1267
PrimeRecord (prepare a record for adding:ViewManager) .....	1268
Reset (reset the view position) .....	1269
RestoreBuffers (restore VIEW file buffers).....	1270
SaveBuffers (save VIEW file buffers) .....	1270
SetFilter (add, change, or remove active filter) .....	1271
SetOrder (replace a sort order) .....	1273
SetSort (set the active sort order) .....	1274
UseView (use LazyOpen files) .....	1275
ValidateRecord (validate an element) .....	1276

<b>WindowComponent Interface</b>	<b>1277</b>
WindowComponent Overview.....	1277
WindowComponent Concepts .....	1277
Relationship to Other Application Builder Classes .....	1277
WindowComponent Source Files.....	1277
WindowComponent Methods.....	1278
WindowComponent Methods .....	1278
Kill(shutdown the parent object) .....	1278
PrimaryBufferRestored(confirm restore of primary buffer) .....	1279
PrimaryBufferRestoreRequired(flag restore of primary buffer) .....	1279
PrimaryBufferSaved(confirm save of primary buffer) .....	1280
PrimaryBufferSaveRequired(flag save of primary buffer) .....	1280
Reset(reset object's data).....	1281
ResetRequired(determine if screen refresh needed) .....	1282
SetAlerts(alert keystrokes for window component) .....	1283
TakeEvent(process the current ACCEPT loop event).....	1283
Update(get VIEW data for the selected item).....	1284
UpdateWindow(update window controls).....	1284
<b>WindowResizeClass</b>	<b>1285</b>
WindowResizeClass Overview .....	1285
WindowResizeClass Properties.....	1289
AutoTransparent (optimize redraw) .....	1289
DeferMoves (optimize resize) .....	1289
WindowResizeClass Methods .....	1290
WindowResizeClass Functional Organization--Expected Use .....	1290
GetParentControl (return parent control) .....	1291
GetPositionStrategy (return position strategy for a control type) .....	1292
GetResizeStrategy (return resize strategy for a control type) .....	1293
Init (initialize the WindowResizeClass object) .....	1294
Kill (shut down the WindowResizeClass object) .....	1297
Reset (resets the WindowResizeClass object) .....	1298
Resize (resize and reposition controls) .....	1299
RestoreWindow (restore window to initial size) .....	1300
SetParentControl (set parent control) .....	1301
SetParentDefaults (set default parent controls) .....	1302
SetStrategy (set control resize strategy) .....	1303
<b>WindowManager</b>	<b>1305</b>
WindowManager Overview .....	1305
WindowManager Properties.....	1313
AutoRefresh (reset window as needed flag) .....	1313
AutoToolbar (set toolbar target on new tab selection) .....	1313
CancelAction (response to cancel request) .....	1314
ChangeAction (response to change request) .....	1315
Dead (shut down flag) .....	1315

DeleteAction (response to delete request) .....	1316
Errors (ErrorClass object) .....	1317
FilesOpened(files opened by procedure) .....	1317
FirstField (first window control) .....	1317
ForcedReset (force reset flag) .....	1318
HistoryKey (restore field key) .....	1318
InsertAction (response to insert request) .....	1319
LastInsertedPosition (hold position of last inserted record) .....	1319
OKControl (window acceptance control--OK button) .....	1320
Opened (window opened flag) .....	1320
OriginalRequest (original database request) .....	1321
Primary (RelationManager object) .....	1321
Request (database request) .....	1322
ResetOnGainFocus (gain focus reset flag) .....	1323
Resize (WindowResize object) .....	1323
Response (response to database request) .....	1324
Saved (copy of primary file record buffer) .....	1324
Translator (TranslatorClass object:WindowManager) .....	1325
VCRRequest (delayed scroll request) .....	1325
WindowManager Methods .....	1326
WindowManager Functional Organization--Expected Use .....	1326
AddHistoryField (add restorable control and field) .....	1328
AddHistoryFile (add restorable history file) .....	1329
AddItem (program the WindowManager object) .....	1330
AddUpdateFile (register batch add files) .....	1331
Ask (display window and process its events:WindowManager) .....	1332
ChangeRecord(execute change record process) .....	1333
DeleteRecord(execute delete record process).....	1334
Init (initialize the WindowManager object) .....	1335
InsertRecord (execute insert record activity).....	1337
Kill (shut down the WindowManager object) .....	1338
Open (a virtual to execute on EVENT:OpenWindow--WindowManager) .....	1339
PostCompleted (initiates final Window processing) .....	1340
PrimeFields (a virtual to prime form fields) .....	1341
PrimeUpdate (update or prepare for update) .....	1342
RemoveItem(remove WindowComponent object) .....	1343
Reset (reset the window for display) .....	1344
RestoreField (restore field to last saved value) .....	1345
Run (run this procedure or a subordinate procedure) .....	1346
SaveHistory (save history fields for later restoration) .....	1348
SaveOnChangeAction(execute change record process and remain active) ...	1349
SaveOnInsertAction(execute insert record activity and remain active).....	1350
SetAlerts (alert window control keystrokes) .....	1351
SetResponse (OK or Cancel the window) .....	1352
TakeAccepted (a virtual to process EVENT:Accepted--WindowManager) ....	1353
TakeCloseEvent (a virtual to Cancel the window) .....	1354
TakeCompleted (a virtual to complete an update form) .....	1356

---

TakeEvent (a virtual to process all events:WindowManager) .....	1358
TakeFieldEvent (a virtual to process field events:WindowManager) .....	1359
TakeNewSelection (a virtual to process EVENT:NewSelection) .....	1360
TakeRejected (a virtual to process EVENT:Rejected) .....	1361
TakeSelected (a virtual to process EVENT:Selected) .....	1362
TakeWindowEvent (a virtual to process non-field events:WindowManager) .	1363
Update (prepare records for writing to disk) .....	1364

**Index:****1365**





---

# Foreword

## Welcome

Welcome to the Application Builder Class Library Reference! This book is designed to be your every day reference to the Classes that lie beneath the templates.

Once you've become familiar with the Clarion development environment, through *Getting Started*, *Learning Clarion* and the *Online User's Guide*, you will refer to those books less and less frequently. However, in your day-to-day work, we think you will continue to need information on the finer points of the various Application Builder Class methods.

That's why we created this Application Builder Class Library Reference—for every Clarion developer who wants a quick, ready reference to those Clarion components you use over and over again.

This book provides in-depth discussions of the ABC Library. It shows you how the ABC Templates use the powerful ABC Library objects—and how you can use, reuse, and modify the classes with the ABC Templates or within your hand-coded project.

These are the tools you'll continue to refer to regardless of your expertise with Clarion. The depths of information on these tools and the consequent versatility you can achieve with them is virtually unlimited.

## Documentation Conventions

### Typeface Conventions

<i>Italics</i>	Indicates what to type at the keyboard and variable information, such as <i>Enter This</i> or <i>filename.TXT</i> . Also identifies the title information of dialog windows, like <i>Procedure Properties</i> .
CAPS	Indicates keystrokes to enter at the keyboard such as ENTER or ESCAPE, and mouse operations such as RIGHT-CLICK.
<b>Boldface</b>	Indicates commands or options from a menu or text in a dialog window.
UPPERCASE	Clarion language keywords such as MAX or USE.
<code>Courier New</code>	Used for diagrams, source code listings, to annotate examples, and for examples of the usage of source statements.

### Keyboard Conventions

F1	Indicates a single keystroke. In this case, press and release the F1 key.
ALT+X	Indicates a combination of keystrokes. In this case, hold down the ALT key and press the X key, then release both keys.

### Other Conventions

**Tip****Note:**

Special Tips, Notes, and Warnings—information that is not immediately evident from the topic explanation.

# ABC Library Overview

## About This Book

This book describes the Application Builder Class (ABC) Library.

It provides an overview of each class or related group of classes. Then it provides specific information on the public properties and methods of each class, plus examples for using them. It also shows you the source files for each class and describes some of the relationships between the classes.

## Application Builder Class (ABC) Library

### **Class Libraries Generally**

The purpose of a class library in an Object Oriented system is to help programmers work more efficiently by providing a safe, efficient way to reuse pieces of program code. In other words, a class library should relieve programmers of having to write certain routines by letting them use already written generic routines to perform common or repetitive program tasks.

In addition, a class library can reduce the amount of programming required to implement changes to an existing class based program. By deriving classes that incrementally add to or subtract from the classes in the library, programmers can accomplish substantial changes without having to rewrite the base classes or the programs that rely on the base classes.

### **Application Builder Classes—The ABCs of Rapid Application Development**

#### **Typical Reusability and Maintenance Benefits**

The Application Builder Classes (ABC Library) provide all the benefits of class libraries in general. Clarion's ABC Templates automatically generate code that uses and reuses the robust, flexible, and solid (pre-tested) objects defined by the ABC Library. Further, the templates are designed to help you easily derive your own classes based on the ABC Library.

Of course, you need not use the templates to use the Application Builder Classes. However, the template generated code certainly provides appropriate examples for using the ABC Library in hand coded programs. Either way, the bottom line for you is more powerful programs with less coding.

### **Database and Windows Program Orientation**

The Application Builder Classes have a fairly specific focus or scope. That is, *its objects are designed to process databases within a Windows environment*. Even more specifically, these objects are designed to support all the standard functionality provided by prior versions of Clarion, plus a lot more.

As such, there are database related objects that open, read, write, view, search, sort, and print data files. There are objects that enforce relational integrity between related data files.

In addition there are general purpose Windows related objects that display error messages, manage popup menus, perform edit-in-place, manage file-loaded drop-down lists, perform language translation on windows, resize windows and controls, process toolbars across execution threads, read and write INI files, and manage selection and processing of DOS/Windows files.

The point is, the class library supports general purpose database Windows programs; it does not support, say, real-time process control for oil refineries.

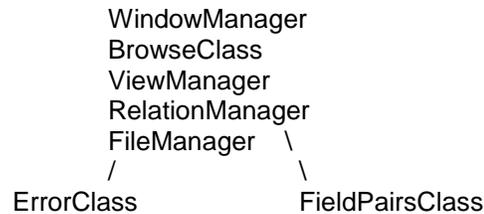
### **Core Classes**

The Application Builder Classes may be logically divided into “core” classes and “peripheral” classes. The core classes are central to the ABC Library—everything else is built from them or hangs off them. If you intend to study the Application Builder Classes, you should begin with the core classes. Further, a thorough understanding of these classes should give you an excellent foundation for understanding the ABC Template generated programs and procedures that use these classes.

Even if you want to stay as far away from the ABC Library as possible, you should keep a couple of things in mind with regard to the core classes:

- The core classes are ErrorClass, FieldPairsClass, FileManager, RelationManager, ViewManager, WindowManager, and BrowseClass.
- Core classes are used repeatedly, so if you must modify them, try to keep them efficient.
- Core classes are almost certainly in any template based program, so additional references to them generally won't affect the size of your executable.

There is a hierarchy within the core classes. The ErrorClass and the FieldPairsClass form the foundation upon which the FileManager, RelationManager, and ViewManager rest. Finally, the BrowseClass, which is derived from the ViewManager, tops off the core classes. The WindowManager is programmed to understand these core classes and manages window procedures that use them.



To understand these core classes, we recommend you tackle the core classes first (ErrorClass and FieldPairsClass), then work your way up to the WindowManager.

### ABC Library Source Files

The Application Builder Classes are installed by default to the Clarion \LIBSRC folder. The specific classes reside in the following respective files. The core classes are shown in bold.

The class declarations reside in the .INC files, and their method definitions reside in the specified .CLW files.

#### ABASCII.INC

AsciiFileClass	MODULE('ABASCII.CLW')
AsciiPrintClass	MODULE('ABASCII.CLW')
AsciiSearchClass	MODULE('ABASCII.CLW')
AsciiViewerClass	MODULE('ABASCII.CLW')

#### ABBROWSE.INC

StepClass	MODULE('ABBROWSE.CLW')
StepLongClass	MODULE('ABBROWSE.CLW')
StepRealClass	MODULE('ABBROWSE.CLW')
StepStringClass	MODULE('ABBROWSE.CLW')
StepCustomClass	MODULE('ABBROWSE.CLW')
LocatorClass	MODULE('ABBROWSE.CLW')
StepLocatorClass	MODULE('ABBROWSE.CLW')
EntryLocatorClass	MODULE('ABBROWSE.CLW')
IncrementalLocatorClass	MODULE('ABBROWSE.CLW')
ContractingLocatorClass	MODULE('ABBROWSE.CLW')
EditClass	MODULE('ABBROWSE.CLW')
<b>BrowseClass</b>	<b>MODULE('ABBROWSE.CLW')</b>

#### ABDROPS.INC

FileDropClass	MODULE('ABDROPS.CLW')
FileDropComboClass	MODULE('ABDROPS.CLW')

ABEIP.INC	
EditClass	MODULE('ABEIP.CLW')
EditCheckClass	MODULE('ABEIP.CLW')
EditColorClass	MODULE('ABEIP.CLW')
EditDropListClass	MODULE('ABEIP.CLW')
EditEntryClass	MODULE('ABEIP.CLW')
EditFileClass	MODULE('ABEIP.CLW')
EditFontClass	MODULE('ABEIP.CLW')
EditMultiSelectClass	MODULE('ABEIP.CLW')
ABERROR.INC	
<b>ErrorClass</b>	<b>MODULE('ABERROR.CLW')</b>
ABFILE.INC	
<b>FileManager</b>	<b>MODULE('ABFILE.CLW')</b>
<b>RelationUsage</b>	<b>MODULE('ABFILE.CLW')</b>
<b>RelationManager</b>	<b>MODULE('ABFILE.CLW')</b>
<b>ViewManager</b>	<b>MODULE('ABFILE.CLW')</b>
ABPOPOP.INC	
PopupClass	MODULE('ABPOPOP.CLW')
ABQUERY.INC	
QueryClass	MODULE('ABQUERY.CLW')
QueryVisualClass	MODULE('ABQUERY.CLW')
QueryFormVisual	MODULE('ABQUERY.CLW')
ABREPORT.INC	
ProcessClass	MODULE('ABREPORT.CLW')
PrintPreviewClass	MODULE('ABREPORT.CLW')
<b>ReportManager</b>	<b>MODULE('ABREPORT.CLW')</b>
ABRESIZE.INC	
WindowResizeClass	MODULE('ABRESIZE.CLW')
ABTOOLBA.INC	
ToolbarTargetClass	MODULE('ABTOOLBA.CLW')
ToolbarListboxClass	MODULE('ABTOOLBA.CLW')
ToolbarReltreeClass	MODULE('ABTOOLBA.CLW')
ToolbarUpdateClass	MODULE('ABTOOLBA.CLW')
ToolbarClass	MODULE('ABTOOLBA.CLW')
ABUTIL.INC	
ConstantClass	MODULE('ABUTIL.CLW')
<b>FieldPairsClass</b>	<b>MODULE('ABUTIL.CLW')</b>
<b>BufferedPairsClass</b>	<b>MODULE('ABUTIL.CLW')</b>
INIClass	MODULE('ABUTIL.CLW')
DOSFileLookupClass	MODULE('ABUTIL.CLW')
TranslatorClass	MODULE('ABUTIL.CLW')
ABWINDOW.INC	
<b>WindowManager</b>	<b>MODULE('ABWINDOW.CLW')</b>

### Including the right files in your data section

Many of the class declarations directly reference other classes. To resolve these references, each class header (.INC file) INCLUDEs only the headers containing the directly referenced classes. This convention maximizes encapsulation, minimizes compile times, and ensures that all necessary components are present for the make process. We recommend you follow this convention too.

The Application Builder Classes source code is structured so that you can INCLUDE either the header or the definition (.CLW file) in your program's data section. If you include the header, it references the required definitions and vice versa.

A good rule of thumb is to INCLUDE as little as possible. The compiler will let you know if you have omitted something.

## ABC Library and the ABC Templates

The ABC Templates rely heavily on the ABC Library. However, the templates are highly configurable and are designed to let you substitute your own class definitions if you wish. See *Part I—Classes Tab Options (Global)* for more information on configuring the global level interaction between the ABC Templates and the ABC Library. See *Part I—Classes Tab Options (Local)* for more information on configuring the local (module level) interaction between the ABC Templates and the ABC Library.

### Classes and Their Template Generated Objects

The ABC Templates instantiate objects from the ABC Library. The default template generated *object* names are usually related to the corresponding *class* names, but they are not exactly the same. Your ABC applications' generated code may contain data declarations and executable statements similar to these:

```
GlobalErrors          ErrorClass
Hide:Access:Customer CLASS(FileManager)
INIMgr               INIClass
ThisWindow           CLASS(ReportManager)
ThisWindow           CLASS(WindowManager)
ThisReport           CLASS(ProcessClass)
ThisProcess          CLASS(ProcessClass)
BRW1                 CLASS(BrowseClass)
EditInPlace::CUS:NAME EditClass
Resizer              WindowResizeClass
Toolbar              ToolbarClass
CODE
GlobalResponse = ThisWindow.Run()
BRW1.AddSortOrder(BRW1::Sort0:StepClass,ST:StKey)
BRW1.AddToolbarTarget(Toolbar)
GlobalErrors.Throw()
Resizer.AutoTransparent=True
Previewer.AllowUserZoom=True
```

These data declarations instantiate objects from the ABC Library, and the executable statements reference the instantiated objects. The various ABC classes and their template instantiations are listed below so you can identify ABC objects in your applications' generated code and find the corresponding ABC Library documentation.

<u>Template Generated Object</u>	<u>Application Builder Class</u>
Access:file	FileManager
BRWn	BrowseClass
BRWn::Sortn:Locator	LocatorClass
BRWn::Sortn:StepClass	StepClass
EditInPlace::field	EditClass
FDBn	FileDropClass
FDCBn	FileDropComboClass
FileLookupN	SelectFileClass
GlobalErrors	ErrorClass
INIMgr	INIClass
QBEn	QueryClass
QBVn	QueryVisualClass
Popup	PopupClass
Previewer	PrintPreviewClass
ProgressMgr	StepClass
Relate:file	RelationManager
RELn::Toolbar	ToolbarReltreeClass
Resizer	WindowResizeClass
ThisProcess	ProcessClass
ThisReport	ProcessClass
ThisWindow	WindowManager, ReportManager
Toolbar	ToolbarClass
ToolbarForm	ToolbarUpdateClass
Translator	TranslatorClass
ViewerN	ASCIIViewerClass

## ABC Coding Conventions

The ABC Library uses several coding conventions. You may see instances of these code constructions in ABC applications' generated code and in the ABC Library code. We recommend that you follow these conventions within your embedded code.

### Method Names

The following names have a specific meaning in the ABC Library. The names and their meanings are described below.

#### *AddItem*

The object adds an item to its datastore. The item may be a field, a key, a sort order, a range limit, another object, etc. The item may be anything the object needs to do its job.

#### *Ask[Information]*

The method interacts with the end user to get the Information.

#### *Fetch*

The method retrieves data from a file.

#### *GetItem*

The method returns the value of the named item.

#### *Init*

The method does whatever is required to initialize the object.

#### *Kill*

The method does whatever is required to shut down the object, including freeing any memory allocated during its lifetime.

#### *Reset[what or how]*

The method resets the object and its controls. This includes reloading data, resetting sort orders, redrawing window controls, etc.

#### *SetItem*

The method sets the value of the named item, or makes the named item active so that other object methods operate on the active item.

#### *TakeItem*

The method "takes" the item from another method or object and continues processing it. The item may be a window event (Accepted, Rejected, OpenWindow, CloseWindow, Resize, etc.), a record, an error condition, etc.

Throw[*Item*]

The method “throws” the item to another object or method for handling. The item is usually an error condition.

### *TryAction*

The method makes one attempt to carry out the action, then returns a value indicating success or failure. A return value of zero (0 or Level:Benign) indicates success; any other value indicates failure.

## **Where to Initialize & Kill Objects**

There are generally two factors to consider when initializing and killing objects:

- Generally, objects should live as short a life as possible
- Objects should always be Killed (to free any memory allocated during its lifetime)

Balancing these two (sometimes conflicting) factors dictates that objects Initialized with EVENT:OpenWindow are usually Killed with EVENT:CloseWindow. Objects Initialized with ThisWindow.Init are usually Killed with ThisWindow.Kill.

## **Return Values**

Many ABC methods return a value indicating success or failure. A return value of zero (0 or Level:Benign) indicates success. Any other return value indicates a problem whose severity may vary. Other return values and their ABC severity EQUATES (Level:User, Level:Cancel, Level:Notify, Level:Fatal, Level:Program) are documented in the *Error Class* chapter and in the individual methods' documentation. This convention produces code like the following:

```
IF ABCObject.Method()  
  !handle failure / error  
ELSE  
  !continue normally  
END
```

```
IF ~ABCObject.Method()  
  !continue normally  
END
```

## **Event Processing Method Return Values**

Some ABC methods process ACCEPT loop events. The names of these methods begin with "Take" and usually indicate the type of events they handle. These event processing methods execute within an ACCEPT loop (as implemented by the WindowManager.Ask method) and return a value indicating how the ACCEPT loop should proceed.

A return value of Level:Benign indicates processing of this event should continue normally. A return value of Level:Notify indicates processing is completed for this event and the ACCEPT loop should CYCLE. A return value of Level:Fatal indicates the event could not be processed and the ACCEPT loop should BREAK.

If you (or the ABC Templates) derive a class with any of these methods, you should use this return value convention to control ACCEPT loop processing.

Following is the `WindowManager.Ask` method code that implements this convention. See *WindowManager Concepts* for more information.

```
ACCEPT
CASE SELF.TakeEvent()
OF Level:Fatal
  BREAK
OF Level:Notify
  CYCLE
END
END
```

### Ending a Procedure

In your embedded code you may encounter a condition that requires the procedure to end immediately (that is, it cannot wait for an `EVENT:CloseWindow`, or an `EVENT:CloseWindow` is not appropriate).

In some cases, a simple `RETURN` will not end your procedure (because a `RETURN` embedded within a derived method ends the method, not the calling procedure), and even if it would, it might not be appropriate (because the procedure may have allocated memory or started other tasks that should be ended in a controlled manner).

There are several ways you can initiate the normal shut down of your procedure, depending on where in the procedure your code is embedded. Following are the conventional ways to shut down your procedure normally.

```
RETURN(Level:Fatal)           !normal shutdown from ABC derived method
ReturnValue = Level:Fatal    !normal shutdown at end of ABC derived method
ThisWindow.Kill              !normal shutdown from Procedure Routine
ThisWindow.Kill;RETURN      !normal shutdown from Procedure Routine
                             ! called from within ACCEPT loop
```

### PRIVATE (undocumented) Items

Some of the properties and methods in the ABC Library have the `PRIVATE` attribute. These `PRIVATE` items are not documented. These items are `PRIVATE` because they are likely to change or disappear completely in future ABC Library releases. Making some items `PRIVATE`, gives TopSpeed the flexibility to change and improve these areas without affecting applications developed with the ABC Library. We strongly recommend that you do not remove the `PRIVATE` attributes on ABC Library items.

## **PROTECTED, VIRTUAL, DERIVED, and PROC Attributes**

Some of the ABC Library properties and methods have special attributes that enhance their functionality, usability, and maintainability. Each property and method topic shows any applicable attributes in the syntax diagram (gray box). The purpose and effect of these attributes are documented here and in the Language Reference, but not in individual property and method topics.

### **PROTECTED Attribute**

The **PROTECTED** attribute specifies that the property or method on which it is placed is visible only to the methods of the same CLASS or of derived CLASSES. This simply suggests that the property or method is important to the correct functioning of the CLASS, and that any changes to these items should be done with care. See *PROTECTED* in the *Language Reference*.

### **VIRTUAL Attribute**

The **VIRTUAL** attribute allows methods in a parent CLASS to call methods in a derived CLASS. This has two primary benefits. First, it allows parent CLASSES to delegate the implementation of certain actions to derived classes; and second, it makes it easy for derived classes to override these same parent class actions. See *VIRTUAL* in the *Language Reference*.

Virtual methods let you insert custom code into an existing class, without copying or duplicating the existing code. Furthermore, the existing class calls the virtual methods (containing the custom code) as part of its normal operation, so you don't have to explicitly call them. When TopSpeed updates the existing class, the updates are automatically integrated into your application simply by recompiling. The existing class continues to call the virtual methods containing the custom code as part of its normal operation. This approach gives you many opportunities to customize your ABC applications while minimizing maintenance issues.

### **DERIVED Attribute**

The **DERIVED** attribute is similar to the *VIRTUAL* attribute, except that it must have a matching prototype in the parent class.

### **PROC Attribute**

The **PROC** attribute may be placed on a method prototyped with a return value, so you can call the method and ignore the return value without compiler warnings. See *PROC* in the *Language Reference*.

# Documentation Conventions

## Reference Item and Syntax Diagram

The documentation formats for Properties and Methods are illustrated in the following syntax diagrams.

### Property (short description of intended use)

**Property**      **Datatype [, PROTECTED ]**

A complete description of the Property and its uses.

**Datatype** shows the datatype of the property such as LONG or &BrowseClass.

Implementation: A discussion of specific implementation issues. The implementation may change with each release / version of Internet Connect.

```
ComplexDataType STRUCTURE      !actual structure declaration
                               END
```

See Also:      Related Methods and Properties

### Method (short description of what the method does)

```
Method( | parameter1 | [, parameter2 ] ) [, PROTECTED ] [, VIRTUAL ] [, PROC ]
        | alternate |
        | parameters |
```

**Method**      A brief statement of what the method does.

*parameter1*      A complete description of parameter1, along with how it relates to parameter2 and the Method.

*parameter2*      A complete description of parameter2, along with how it relates to parameter1 and the Method. Brackets [ ] indicate optional parameters.

A concise description of what the Method does.

Implementation: A description of how the method currently accomplishes its objective. The implementation may change with each release / version of Clarion.

Return Data Type: The data type returned if applicable.

## Example:

```
FieldOne = FieldTwo + FieldThree      !This is a source code example  
FieldThree = Method(FieldOne,FieldTwo) !Comments follow the "!" character
```

See Also:        [Related Methods and Properties](#)

### **Conceptual Example**

A description of the type of example to be illustrated. Examples show the concept of how a specific class is implemented in source code. The demands of brevity and concision often force the removal of structures which are not essential in illustrating the class.

**PROGRAM**

**MAP**

**END**

**! Data structures**

**CODE**

**! Code Statements**

# ASCIIFileClass

## ASCIIFileClass Overview

The ASCIIFileClass identifies, opens (read-only), indexes, and page-loads a file's contents into a QUEUE. The indexing function speeds any additional access of records and supports page-loading, which in turn allows browsing of very large files.

## ASCIIFileClass Relationship to Other Application Builder Classes

There are several related classes whose collective purpose is to provide reusable, read-only, viewing, scrolling, searching, and printing capability for files, including variable length files. Although these classes are primarily designed for ASCII text and they anticipate using the Clarion ASCII Driver to access the files, they also work with binary files and with other database drivers. These classes can be used to build other components and functionality as well.

The classes that provide this read-only functionality and their respective roles are:

ASCIIViewerClass	ASCIIFileClass plus user interface
ASCIIFileClass	Open, read, filter, and index the file
ASCIIPrintClass	Print one or more lines
ASCIISearchClass	Locate and scroll to text

The ASCIIViewerClass is derived from the ASCIIFileClass. See *ASCIIViewerClass* for more information.

## ASCIIFileClass ABC Template Implementation

The ASCIIFileClass serves as the foundation to the Viewer procedure template; however, the ABC Templates do not instantiate the ASCIIFileClass independently of the ASCIIViewerClass.

The ASCIIViewerClass is derived from the ASCIIFileClass, and the Viewer Procedure Template instantiates the derived ASCIIViewerClass.

## ASCIIFileClass Source Files

The ASCIIFileClass source code is installed by default to the Clarion \LIBSRC folder. The ASCIIFileClass source code are contained in:

ABASCII.INC	ASCIIFileClass declarations
ABASCII.CLW	ASCIIFileClass method definitions

## ASCIIFileClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an ASCIIFileClass object and related objects.

This example lets the end user select a file, then display it's pathname, total line count, and the text at a given percentage point within the file.

```

PROGRAM
MAP
END
INCLUDE('ABASCII.INC')           !declare ASCIIFileClass

Percentile  BYTE(50)              !a value between 1 & 100
GlobalErrors ErrorClass          !declare GlobalErrors object
Afile       AsciiFileClass,THREAD !declare Afile object
FileActive  BYTE(False),THREAD   !Afile initialized flag
Filename    STRING(255),THREAD   !FileName variable

AsciiFile FILE,DRIVER('ASCII'),NAME(Filename),PRE(A1),THREAD
RECORD     RECORD,PRE()
Line       STRING(255)
           END
           END

window WINDOW('View a text file'),AT(3,7,203,63),SYSTEM,GRAY,DOUBLE
        PROMPT('Show Line at Percentile'),AT(5,4),USE(?Prompt:Pct)
        SPIN(@s3),AT(84,3,25,,),USE(Percentile),RANGE(1,100)
        BUTTON('New File'),AT(113,2),USE(?NewFileButton)
        BUTTON('File Size'),AT(157,2),USE(?FileSizeButton)
        PROMPT('Line:'),AT(4,26),USE(?Prompt:Line)
        PROMPT(' '),AT(26,26,172,32),USE(?Line)
        END

CODE
GlobalErrors.Init          !initialize GlobalErrors object
OPEN(window)              !Initialize Afile with:
FileActive=Afile.Init( AsciiFile, | ! file label,
                        A1:line,   | !file field to display
                        Filename,   | !variable file NAME attribute
                        GlobalErrors) !GlobalErrors object

IF FileActive
  window{PROP:Text}=Afile.GetFileName()
ELSE
  window{PROP:Text}='no file selected'
END

ACCEPT

```

```
CASE FIELD()
OF ?NewFileButton          !on New File button
  IF EVENT() = EVENT:Accepted
    CLEAR(FileName)
    FileActive=AFile.Reset(FileName) !reset AFile to a new file
    IF FileActive
      window{PROP:Text}=AFile.GetFileName() !show filename in titlebar
    ELSE
      window{PROP:Text}='no file selected'
    END
  END
END
OF ?Percentile             !on Percentile SPIN
  CASE EVENT()
  OF EVENT:Accepted OROF EVENT:NewSelection
    IF FileActive          !calculate lineno and get the line
      ?Line{PROP:Text}=AFile.GetLine(Percentile/100*AFile.GetLastLineNo())
    ELSE
      ?Line{PROP:Text}='no file selected'
    END
  END
END
OF ?FileSizeButton        !on File Size button
  IF EVENT() = EVENT:Accepted
    IF FileActive          !display total line count
      ?FileSizeButton{PROP:Text}=AFile.GetLastLineNo()&' Lines'
    ELSE
      ?FileSizeButton{PROP:Text}='0 Lines'
    END
  END
END
END
IF FileActive THEN AFile.Kill. !shut down AFile object
GlobalErrors.Kill
```

## AsciiFileClass Properties

### ASCIIFile (the ASCII file)

**ASCIIFile**      **&FILE**

The **File** property is a reference to the managed file. The File property simply identifies the managed file for the various ASCIIFileClass methods.

Implementation:      The Init method initializes the File property.

See Also:              Init

### ErrorMgr (ErrorClass object)

**ErrorMgr**          **&ErrorClass, PROTECTED**

The **ErrorMgr** property is a reference to the ErrorClass object for this ASCIIFileClass object. The ASCIIFileClass uses the ErrorMgr to handle various errors and conditions it encounters when processing the file.

Implementation:      The Init method initializes the ErrorMgr property.

See Also:              Init

### OpenMode (file access/sharing mode)

**OpenMode**          **BYTE**

The **OpenMode** property contains a value that determines the level of access granted to both the user opening the file and other users in a multi-user system.

Implementation:      The Init method sets the OpenMode property to a hexadecimal value of 42h (ReadWrite/DenyNone). The ABC Templates override this default with the appropriate value from the application generator.

The Open method uses the OpenMode property when it OPENS the file for processing. See the *Language Reference* for more information on OPEN.

See Also:              Init

## AsciiFileClass Methods

### **ASCIIFileClass Functional Organization--Expected Use**

As an aid to understanding the ASCIIFileClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the ASCIIFileClass methods.

#### **Non-Virtual Methods**

---

The non-virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the ASCIIFileClass object
Kill	shut down the ASCIIFileClass object

##### **Mainstream Use:**

GetLastLineNo	return last line number
GetLine	return line of text
GetPercentile	convert file position to percentage
SetPercentile	convert percentage to file position

##### **Occasional Use:**

GetFilename	return the filename
Reset	reset the ASCIIFileClass object

#### **Virtual Methods**

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

GetDOSFilename	prompt end user to select a file
FormatLine	a virtual to format text
SetLine	position to specific line
ValidateLine	a virtual to implement a filter

**FormatLine (a virtual to format text)**

**FormatLine**( *line* [, *line number*] ), **PROTECTED**, **VIRTUAL**

---

<b>FormatLine</b>	A virtual placeholder method to format text.
<i>line</i>	The label of the STRING variable containing the text to reformat.
<i>line number</i>	An integer constant, variable, EQUATE or expression that contains the offset or position of the line of text being formatted. If omitted, FormatLine operates on the current line.

The **FormatLine** method is a virtual placeholder method to reformat text prior to display at runtime.

Implementation: The FormatLine method is a placeholder for derived classes. It provides an easy way for you to reformat the text prior to display. The GetLine method calls the FormatLine method.

Example:

```

    INCLUDE('ABASCII.INC')
MyViewer  CLASS(AsciiViewerClass),TYPE
FormatLine PROCEDURE(*STRING),VIRTUAL
            END
Viewer    MyViewer,THREAD
AsciiFile FILE,DRIVER('ASCII'),NAME('MyText'),PRE(A1),THREAD
RECORD    RECORD,PRE()
Line      STRING(255)
            END
            END

CODE
!program code

MyViewer.FormatLine PROCEDURE(*STRING line) !called by ASCIIViewerClass
CODE
line = line[1:5]' '&line[5:55]           !reformat the text

```

See Also: [GetLine](#)

**GetDOSFilename (let end user select file)**

**GetDOSFilename**( *filename* ), VIRTUAL

---

**GetDOSFilename**      Prompts the end user to select the file to process.

*filename*            The label of the ASCIIFile property's NAME attribute variable which receives the selected filename.

The **GetDOSFilename** method prompts the end user to select the file to process and returns a value indicating whether the end user selected a file or did not select a file. A return value of one (1) indicates a file was selected and *filename* contains its pathname; a return value of zero (0) indicates no file was selected and *filename* is empty.

Implementation:      The GetDOSFileName method uses a SelectFileClass object to get the filename from the end user.

Return Data Type:    BYTE

Example:

```
MyAsciiFileClass.Reset FUNCTION(*STRING FName)
RVal            BYTE(True)
SavePath      CSTRING(FILE:MaxFilePath+1),AUTO
CODE
CLOSE(SELF.AsciiFile)
SavePath=PATH()
LOOP
  IF ~FName AND ~SELF.GetDOSFilename(FName)
    RVal=False
    BREAK
  END
  OPEN(SELF.AsciiFile,ReadOnly+DenyNone)
  IF ERRORCODE()
    MESSAGE('Can't open ' & FName)
    RVal=False
  ELSE
    BREAK
  END
END
IF RVal
  SELF.FileSize=BYTES(SELF.AsciiFile)
END
SETPATH(SavePath)
RETURN RVal
```

See Also:            ASCIIFile, SelectFileClass

**GetFilename (return the filename)****GetFilename**

The **GetFilename** method returns the name of the ASCII file.

Implementation: The GetFileName method uses the NAME function. See the *Language Reference* for more information.

Return Data Type: STRING

Example:

```

    INCLUDE('ABASCII.INC')
Viewer   AsciiViewerClass,THREAD           !declare ASCIIViewerClass
Filename STRING(255),THREAD               !declare Viewer object
AsciiFile FILE,DRIVER('ASCII'),NAME(Filename),PRE(A1),THREAD !declare filename variable
RECORD   RECORD,PRE()
Line     STRING(255)
        END
        END
CODE
!program code
MESSAGE('Filename:&Viewer.GetFilename()') !get the ASCII filename

```

## GetLastLineNo (return last line number)

### GetLastLineNo, PROC

The **GetLastLineNo** method returns the number of the last line in the file, and indexes the entire file.

Return Data Type: LONG

Example:

```
MyViewer.TakeScroll PROCEDURE(UNSIGNED EventNo)
```

```
LineNo LONG
```

```
CODE
```

```
IF FIELD()=SELF.ListBox
```

```
IF EVENT() = EVENT:ScrollBottom           !on scroll bottom
```

```
LineNo = SELF.GetLastLineNo()             !index to end of file
```

```
SELF.DisplayPage(LineNo-SELF.ListBoxItems+1) !display last page
```

```
SELECT(SELF.ListBox,SELF.ListBoxItems)     !highlight last row
```

```
END
```

```
END
```

**GetLine (return line of text)****GetLine**( *line number* ), PROC

---

<b>GetLine</b>	Returns a line of text.
<i>line number</i>	An integer constant, variable, EQUATE or expression that contains the offset or position of the line of text to return.

The **GetLine** method returns the line of text specified by *line number*.

Implementation: The GetLine method gets a line at position *line number* from the ASCII file, extending the index queue if needed. If the index queue already contains the requested *line number* then the file is read using the existing offset, otherwise the index is extended. If the requested *line number* does not exist in the file, the text line is cleared and ERRORCODE() set.

Return Data Type: STRING

Example:

```
MyViewer.DisplayPage PROCEDURE(LONG LineNo)
LineOffset USHORT,AUTO
CODE
IF LineNo > 0                                !line specified?
  SELF.ListBoxItems=SELF.ListBox{PROP:Items} !note size of list box
  FREE(SELF.DisplayQueue)                    !free the display queue
  SELF.GetLine(LineNo+SELF.ListBoxItems-1)    !index to end of page
  LOOP LineOffset=0 TO SELF.ListBoxItems-1    !for each listbox line
    SELF.DisplayQueue.Line=SELF.GetLine(LineNo+LineOffset) !read ASCII file record
    IF ERRORCODE()                               !on end of file
      BREAK                                       ! stop reading
    END
  ADD(SELF.DisplayQueue)                       !add to display queue
  END
  SELF.TopLine=LineNo                          !note 1st line displayed
  DISPLAY(SELF.ListBox)                        !redraw the list box
END
```

See Also: GetLine

---

**GetPercentile (convert file position to percentage:ASCIIFileClass)**

**GetPercentile**( *line number* )

---

**GetPercentile** Returns the specified position in the file as a percentage.

*line number* An integer constant, variable, EQUATE or expression that contains the offset or position to convert to a percentage.

The **GetPercentile** method returns the specified position in the file as an approximate percentage which can be used to position a vertical scroll bar thumb.

Return Data Type: USHORT

Example:

```
SetThumb ROUTINE
!current line is what % thru the file?
PctPos=MyASCIIFile.GetPercentile(MyASCIIFile.TopLine+CHOICE(?ASCIIBox)-1)
?ASCIIBox{PROP:VScrollPos}=PctPos      !set thumb to corresponding % position
```

**Init (initialize the ASCIIFileClass object)**

**Init**( *file*, *field* [, *filename*], *error handler* )

---

<b>Init</b>	Initializes the ASCIIFileClass object.
<i>file</i>	The label of the file to display.
<i>field</i>	The fully qualified label of the <i>file</i> field to display.
<i>filename</i>	The label of the <i>file</i> 's NAME attribute variable. If omitted, the <i>file</i> has a constant NAME attribute. If null (""), the ASCIIFileClass prompts the end user to select a file.
<i>error handler</i>	The label of the ErrorClass object to handle errors encountered by this ASCIIFileClass object.

The **Init** method initializes the ASCIIFileClass object and returns a value indicating whether it successfully accessed the *file* and is ready to proceed.

Implementation: The Init method returns one (1) if it accessed the *file* and is ready to proceed; it returns zero (0) and calls the Kill method if unable to access the *file* and cannot proceed. If the Init method returns zero (0), the ASCIIFileClass object is not initialized and you should not call its methods.

Return Data Type: BYTE

Example:

```

Filename      STRING(255),THREAD      !declare filename variable
FileActive    BYTE                    !declare success/fail switch
AsciiFile     FILE,DRIVER('ASCII'),NAME(Filename),PRE(A1)
RECORD        RECORD,PRE()
Line          STRING(255)
              END
              END

CODE
FileActive=ASCIIFile.Init(AsciiFile,    |
                          A1:Line,      |
                          Filename,     |
                          GlobalErrors)  !init ASCIIFileClass object with:
IF ~FileActive THEN RETURN.             ! file label
ACCEPT                                  ! file field to display
  IF EVENT() = EVENT:CloseWindow        ! NAME attribute variable
    IF FileActive THEN ASCIIFile.Kill.   ! ErrorClass object
  END !program code                     !If init failed, don't proceed
                                        !If init succeeded, proceed
                                        !If init succeeded, shut down
END
END

```

See Also: Kill

**Kill (shut down the ASCIIFileClass object)****Kill**

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code.

Example:

```

Filename  STRING(255),THREAD           !declare filename variable
FileActive BYTE                        !declare success/fail switch
AsciiFile FILE,DRIVER('ASCII'),NAME(Filename),PRE(A1)
RECORD    RECORD,PRE()
Line      STRING(255)
          END
          END

CODE
FileActive=ASCIIFile.Init(AsciiFile, | !init ASCIIFileClass object with:
          A1:Line,                    | ! file label
          Filename,                   | ! file field to display
          GlobalErrors)               | ! NAME attribute variable
! ErrorClass object
IF ~FileActive THEN RETURN.          !If init failed, don't proceed
ACCEPT                               !If init succeeded, proceed
  IF EVENT() = EVENT:CloseWindow
    IF FileActive THEN ASCIIFile.Kill. !If init succeeded, shut down
  END
!program code
END

```

## Reset (reset the ASCIIFileClass object)

**Reset**( *filename* )

---

**Reset**            Resets the ASCIIFileClass object.

*filename*            The label of the ASCIIFile property's NAME attribute variable.

The **Reset** method resets the ASCIIFileClass object and returns a value indicating whether the end user selected a file or did not select a file. A return value of one (1) indicates a file was selected and *filename* contains its pathname; a return value of zero (0) indicates no file was selected and *filename* is empty.

Implementation:    The Reset method calls the GetDOSFileName method to get the filename from the end user. Reset opens the file and resets any statistics and flags associated with the selected file.

Return Data Type:    BYTE

Example:

```
AsciiViewerClass.Reset FUNCTION(*STRING Filename)
  CODE
  FREE(SELF.DisplayQueue)
  DISPLAY(SELF.ListBox)
  IF ~PARENT.Reset(Filename) THEN RETURN False.
  SELF.TopLine=1
  SELF.DisplayPage
  SELECT(SELF.ListBox,1)
  RETURN True
```

See Also:            ASCIIFile, GetDOSFilename

## SetLine (a virtual to position the file)

**SetLine**( *line number* ), PROTECTED, VIRTUAL

---

**SetLine**        A virtual placeholder method to position the file.

*line number*    The offset or position of the line in the file.

The **SetLine** method is a virtual placeholder method to position the file.

Implementation:    The SetLine method is a placeholder for derived classes. The SetPercentile, the ASCIIViewerClass.AskGotoLine, and the ASCIISearchClass.Ask methods call the SetLine method.

Example:

```
MyViewerClass.SetLine PROCEDURE(LONG LineNo)   !synchronize LIST with line
number
CODE
SELF.DisplayPage(LineNo)                       !scroll list to LineNo
!highlight the LineNo line
SELECT( SELF.ListBox, CHOOSE( SELF.TopLine=LineNo, 1, LineNo-SELF.TopLine+1) )
```

See Also:         SetPercentile, ASCIIViewerClass.AskGoToLine, ASCIISearchClass.Ask

## SetPercentile (set file to relative position)

**SetPercentile**( *percentile* )

---

**SetPercentile** Positions the file to the record nearest to  
file size \* *percentile* / 100.

*percentile* A value between 0 and 100 that indicates a relative position within the file. This value may be set by a vertical scrollbar thumb position.

The **SetPercentile** method positions the file to the record nearest to  
file size \* *percentile* / 100. You may use SetPercentile to position the file based on the end user's  
vertical scrollbar thumb setting.

Implementation: The SetPercentile method positions the file based on a given percentage (usually  
determined by the vertical thumb position). SetPercentile extends the index as  
required and calls the virtual SetLine method to position the file.

SetPercentile calculates the position by dividing *percentile* by 100 then  
multiplying the resulting percentage times the file size.

Example:

```
MyViewerClass.TakeDrag PROCEDURE(UNSIGNED EventNo)
CODE
IF FIELD()=SELF.ListBox
  IF EventNo = EVENT:ScrollDrag
    SELF.SetPercentile(SELF.ListBox{PROP:VScrollPos}) !reposition based on thumb
  END
END
```

See Also: SetLine

## ValidateLine (a virtual to implement a filter)

**ValidateLine( *line* ), PROTECTED, VIRTUAL**

---

**ValidateLine** A virtual placeholder method to implement a filter.

*line* The offset or position of the line of text to evaluate.

The **ValidateLine** method is a virtual placeholder method to implement a filter. **ValidateLine** returns one (1) to include the *line* and zero (0) to exclude the *line*.

Implementation: The **ValidateLine** method is a placeholder method for derived classes. The **ASCIIFileClass** calls the **ValidateLine** method when it initially reads a record.

Return Data Type: **BYTE**

Example:

**MyFileClass.ValidateLine FUNCTION(STRING LineToTest)**

```
CODE
IF LineToTest[1] = '!'           !check for ! in column 1
  RETURN False                  !exclude lines with !
ELSE
  RETURN True                   !include all other lines
END
```



# ASCIIPrintClass

## ASCIIPrintClass Overview

The ASCIIPrintClass provides the user interface--a simple Print Options dialog--to print one or more lines from a text file. The ASCIIPrintClass interface lets the end user specify a range of lines to print, then optionally previews the lines before printing them. The ASCIIPrintClass interface also provides access to the standard Windows Print Setup dialog.

## ASCIIPrintClass Relationship to Other Application Builder Classes

The ASCIIPrintClass relies on the ASCIIFileClass to read and index the file that it prints. It also relies on the PrintPreviewClass to provide the on-line preview. It also uses the TranslatorClass to translate its Print Options dialog text if needed.

The ASCIIViewerClass uses the ASCIIPrintClass to provide the end user with a Print Options dialog to print one or more lines from the viewed file.

There are several related classes whose collective purpose is to provide reusable, read-only, viewing, scrolling, searching, and printing capability for files, including variable length files. Although these classes are primarily designed for ASCII text and they anticipate using the Clarion ASCII Driver to access the files, they also work with binary files and with other database drivers. These classes can be used to build other components and functionality as well.

The classes that provide this read-only functionality are:

ASCIIViewerClass	ASCIIFileClass plus user interface
ASCIIFileClass	Open, read, filter, and index the file
ASCIIPrintClass	Print one or more lines
ASCIISearchClass	Locate and scroll to text

## ASCIIPrintClass ABC Template Implementation

Both the Viewer procedure template and the ASCIIPrintButton control template generate code to instantiate an ASCIIPrintClass object. The Viewer template accomplishes this by adding a parameter to the ASCIIViewerClass.Init method. The ASCIIPrintButton template accomplishes this by declaring an ASCIIPrintClass object and calling the ASCIIViewerClass.AddItem method to register the ASCIIPrintClass object with the ASCIIViewerClass object.

## ASCIIPrintClass Source Files

The ASCIIPrintClass source code is installed by default to the Clarion \LIBSRC folder. The specific ASCIIPrintClass source code and their respective components are contained in:

ABASCII.INC	ASCIIPrintClass declarations
ABASCII.CLW	ASCIIPrintClass method definitions

## ASCIIPrintClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate an ASCIIPrintClass object and related objects.

This example lets the end user select a file, then search and print from it.

```

MEMBER('viewer.clw')

INCLUDE('ABASCII.INC')
INCLUDE('ABWINDOW.INC')

MAP
  MODULE('VIEWE002.CLW')
BrowseFiles PROCEDURE
  END
END

BrowseFiles PROCEDURE

FilesOpened BYTE
ViewerActive BYTE(False)
Filename     STRING(FILE:MaxFilePath),AUTO,STATIC,THREAD
AsciiFile   FILE,DRIVER('ASCII'),NAME(Filename),PRE(A1),THREAD
RECORD      RECORD,PRE()
Line        STRING(255)
            END
            END

ViewWindow  WINDOW('View an ASCII File'),AT(3,7,296,136),SYSTEM,GRAY
            LIST,AT(5,5,285,110),USE(?AsciiBox),IMM,FROM('')
            BUTTON('&Print...'),AT(7,119),USE(?Print)
            BUTTON('&Search...'),AT(44,119),USE(?Search)
            END

ThisWindow  CLASS(WindowManager)
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
TakeAccepted PROCEDURE(),BYTE,PROC,VIRTUAL
            END

Viewer      AsciiViewerClass          !declare Viewer object

```

```

Searcher  AsciiSearchClass      !declare Searcher object
Printer   AsciiPrintClass       !declare Printer object

CODE
GlobalResponse = ThisWindow.Run()

ThisWindow.Init PROCEDURE()
ReturnValue  BYTE,AUTO
CODE
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?AsciiBox
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
OPEN(ViewWindow)
SELF.Opened=True
CLEAR(FileName)
ViewerActive=Viewer.Init(AsciiFile,A1:Line,FileName,?AsciiBox,GlobalErrors)
IF ~ViewerActive THEN RETURN Level:Fatal.
Viewer.AddItem(Searcher)          !register Searcher with Viewer
Viewer.AddItem(Printer)          !register Printer with Viewer
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.TakeAccepted PROCEDURE()
ReturnValue  BYTE,AUTO
CODE
ReturnValue = PARENT.TakeAccepted()
CASE ACCEPTED()
OF ?Print
  ThisWindow.Update
  IF ViewerActive THEN Viewer.Printer.Ask.  !display Print Options dialog
OF ?Search
  ThisWindow.Update
  IF ViewerActive
    IF CHOICE(?AsciiBox)>0                !search from current line
      Viewer.Searcher.Ask(Viewer.TopLine+CHOICE(?AsciiBox)-1)
    ELSE
      Viewer.Searcher.Ask(1)              !search from line 1
  END
END
END
RETURN ReturnValue

```

## AsciiPrintClass Properties

### FileMgr (AsciiFileClass object:AsciiPrintClass)

#### **FileMgr &AsciiFileClass, PROTECTED**

The **FileMgr** property is a reference to the AsciiFileClass object that manages the file to print. The AsciiPrintClass object uses the FileMgr to read the file, manage print range line numbers and to handle error conditions and messages.

Implementation: The Init method initializes the FileMgr property.

See Also: Init

### PrintPreview (print preview switch)

#### **PrintPreview BYTE**

The **PrintPreview** property contains the print preview setting for the AsciiPrintClass object. A value of one (1 or True) initially "checks" the print preview box (default is preview); a value of zero (0 or False) "clears" the print preview box (default is no preview).

Implementation: The Init method sets the PrintPreview property to false. The PrintLines method implements the action specified by the PrintPreview property.

See Also: Init, PrintLines

### Translator (TranslatorClass object:AsciiPrintClass)

#### **Translator &TranslatorClass, PROTECTED**

The **Translator** property is a reference to the TranslatorClass object for the AsciiPrintClass object. The AsciiPrintClass object uses this property to translate text in the object's Print Options dialog to the appropriate language.

Implementation: The AsciiPrintClass does not initialize the Translator property. The AsciiPrintClass only invokes the Translator if the Translator property is not null. You can use the AsciiViewerClass.SetTranslator method or a reference assignment statement to set the Translator property.

See Also: AsciiViewerClass.SetTranslator

## AsciiPrintClass Methods

### Ask (solicit print specifications)

#### **Ask, VIRTUAL**

The **Ask** method displays a Print Options dialog that prompts the end user for print specifications, then prints the selected lines subject to those specifications (printer destination, paper orientation, etc.).

Implementation: The Ask method prompts the end user for print specifications (including the Windows standard Print Setup dialog), print preview, plus a range of lines to print. If the user CLICKS the Print button, the Ask method prints the requested lines to the printer specified by the end user.

Example:

```
ACCEPT
CASE FIELD()
OF ?PrintButton           !on "Print" button
  IF EVENT() = EVENT:Accepted  !call the Printer.Ask method
    IF ViewerActive THEN Viewer.Printer.Ask. !to gather specs and print lines
  END
END
END
END
```

**Init (initialize the ASCIIPrintClass object)**

`Init( ASCIIFileMgr ), VIRTUAL`

---

**Init**            Initializes the ASCIIPrintClass object.

*ASCIIFileMgr*    The label of the ASCIIFileClass object that manages the file to print. The ASCIIPrintClass object uses the *ASCIIFileMgr* to read from the file and handle line numbers and error conditions.

The **Init** method initializes the ASCIIPrintClass object.

Example:

```
MyViewerClass.Init FUNCTION(FILE AsciiFile,*STRING FileLine,*STRING Filename,|
    UNSIGNED ListBox,ErrorClass ErrHandler,BYTE Enables)
CODE
!program code
IF BAND(Enables,EnableSearch)            !if Search flag is on
    SELF.Searcher &= NEW AsciiSearchClass !instantiate Searcher object
    SELF.Searcher.Init(SELF)              !initialize Searcher object
END
IF BAND(Enables,EnablePrint)            !if Print flag is on
    SELF.Printer &= NEW AsciiPrintClass    !instantiate Printer object
    SELF.Printer.Init(SELF)                !initialize Printer object
END
```

## PrintLines (print or preview specified lines)

**PrintLines**( *first*, *last* ), VIRTUAL

---

**PrintLines** Prints or previews the specified lines.

*first* An integer constant, variable, EQUATE, or expression containing the number of the first line of the range of lines to print.

*last* An integer constant, variable, EQUATE, or expression containing the number of the last line of the range of lines to print.

If the PrintPreview property is True, the **PrintLines** method previews the specified lines, then prints the lines or not, depending on the end user's response to the preview.

If the PrintPreview property is False, the **PrintLines** method prints the specified lines to the selected printer.

Example:

```
IF EVENT() = EVENT:Accepted
  IF ACCEPTED() = ?PrintButton
    FirstLine=1
    LastLine=HighestLine
    SELF.PrintLines(FirstLine,LastLine)
    POST(EVENT:CloseWindow)
  END
END
```

See Also:           PrintPreview



# ASCIISearchClass

## ASCIISearchClass Overview

The ASCIISearchClass provides the user interface--a persistent non-MDI Find dialog--to locate specific text within the browsed file. The ASCIISearchClass interface lets the end user specify the direction and case sensitivity of the search, and it allows repeating searches ("find next").

## ASCIISearchClass Relationship to Other Application Builder Classes

The ASCIISearchClass relies on the ASCIIFileClass to read and index the file that it searches. It also uses the TranslatorClass to translate its Find dialog text if needed.

The ASCIIViewerClass uses the ASCIISearchClass to provide the end user with a Find dialog to locate text in the viewed file.

There are several related classes whose collective purpose is to provide reusable, read-only, viewing, scrolling, searching, and printing capability for files, including variable length files. Although these classes are primarily designed for ASCII text and they anticipate using the Clarion ASCII Driver to access the files, they also work with binary files and with other database drivers. These classes can be used to build other components and functionality as well.

The classes that provide this read-only functionality and their respective roles are:

ASCIIViewerClass	ASCIIFileClass plus user interface
ASCIIFileClass	Open, read, filter, and index the file
ASCIISearchClass	Print one or more lines
ASCIISearchClass	Locate and scroll to text

## ASCIISearchClass ABC Template Implementation

Both the Viewer procedure template and the ASCIISearchButton control template generate code to instantiate an ASCIISearchClass object. The Viewer template accomplishes this by adding a parameter to the ASCIIViewerClass.Init method. The ASCIISearchButton template accomplishes this by declaring an ASCIISearchClass object and calling the ASCIIViewerClass.AddItem method to register the ASCIISearchClass object with the ASCIIViewerClass object.

## ASCIISearchClass Source Files

The ASCIISearchClass source code is installed by default to the Clarion \LIBSRC folder. The specific ASCIISearchClass source code and their respective components are contained in:

ABASCII.INC	ASCIISearchClass declarations
ABASCII.CLW	ASCIISearchClass method definitions

## ASCIISearchClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate an ASCIISearchClass object and related objects.

This example lets the end user select a file, then search and print from it.

```

MEMBER('viewer.clw')

INCLUDE('ABASCII.INC')
INCLUDE('ABWINDOW.INC')

MAP
  MODULE('VIEWE002.CLW')
BrowseFiles  PROCEDURE
  END
  END

BrowseFiles  PROCEDURE

FilesOpened  BYTE
ViewerActive BYTE(False)
Filename     STRING(FILE:MaxFilePath),AUTO,STATIC,THREAD
AsciiFile    FILE,DRIVER('ASCII'),NAME(Filename),PRE(A1),THREAD
RECORD       RECORD,PRE()
Line         STRING(255)
            END
            END

ViewWindow   WINDOW('View an ASCII File'),AT(3,7,296,136),SYSTEM,GRAY
            LIST,AT(5,5,285,110),USE(?AsciiBox),IMM,FROM('')
            BUTTON('&Print...'),AT(7,119),USE(?Print)
            BUTTON('&Search...'),AT(44,119),USE(?Search)
            END
ThisWindow   CLASS(WindowManager)
Init         PROCEDURE(),BYTE,PROC,VIRTUAL
TakeAccepted PROCEDURE(),BYTE,PROC,VIRTUAL
            END

Viewer       AsciiViewerClass           !declare Viewer object
Searcher     AsciiSearchClass           !declare Searcher object
Printer      AsciiPrintClass            !declare Printer object

CODE
GlobalResponse = ThisWindow.Run()

ThisWindow.Init PROCEDURE()
ReturnValue     BYTE,AUTO

```

```
CODE
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?AsciiBox
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
OPEN(ViewWindow)
SELF.Opened=True
CLEAR(Filename)
ViewerActive=Viewer.Init(AsciiFile,A1:Line,Filename,?AsciiBox,GlobalErrors)
IF ~ViewerActive THEN RETURN Level:Fatal.
Viewer.AddItem(Searcher)           !register Searcher with Viewer
Viewer.AddItem(Printer)            !register Printer with Viewer
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.TakeAccepted PROCEDURE()
ReturnValue          BYTE,AUTO
CODE
ReturnValue = PARENT.TakeAccepted()
CASE ACCEPTED()
OF ?Print
  ThisWindow.Update
  IF ViewerActive THEN Viewer.Printer.Ask.    !display Print Options dialog
OF ?Search
  ThisWindow.Update
  IF ViewerActive
    IF CHOICE(?AsciiBox)>0                    !search from current line
      Viewer.Searcher.Ask(Viewer.TopLine+CHOICE(?AsciiBox)-1)
    ELSE
      Viewer.Searcher.Ask(1)                  !search from line 1
  END
END
END
RETURN ReturnValue
```

## AsciiSearchClass Properties

### Find (search constraints)

#### **Find FindGroup, PROTECTED**

The **Find** property contains the current search criteria or specification.

Implementation: The search specification includes the text to search for, the direction in which to search, and whether or not the search is case sensitive.

The Ask method sets the values of the Find property based on end user input to the Find dialog. The Setup method sets the values of the Find property for use without the Ask method. The Next method implements the search specified by the Find property.

The FindGroup data type is declared in ABASCII.INC as follows:

```
FindGroup GROUP,TYPE
What      PSTRING(64) !text to look for
MatchCase BYTE      !case sensitive?
Direction STRING(4)  !either 'Up ' or 'Down'
END
```

See Also: Ask, Next, Setup

## **FileMgr (AsciiFileClass object:AsciiSearchClass)**

**FileMgr &AsciiFileClass, PROTECTED**

The **FileMgr** property is a reference to the AsciiFileClass object tyhat manages the file to search. The AsciiSearchClass object uses the FileMgr to read the file, and to handle error conditions and messages.

Implementation: The Init method initializes the FileMgr property.

See Also: Init

## **LineCounter (current line number)**

**LineCounter LONG, PROTECTED**

The **LineCounter** property contains the current line number of the searched file.

Implementation: The ASCIISearchClass object uses the LineCounter property to implement "find next" searches--searches that continue from the current line.

## **Translator (TranslatorClass object:ASCIISearchClass)**

**Translator &TranslatorClass, PROTECTED**

The **Translator** property is a reference to the TranslatorClass object for the ASCIISearchClass object. The ASCIISearchClass object uses this property to translate window text to the appropriate language.

Implementation: The ASCIISearchClass does not initialize the Translator property. The ASCIISearchClass only invokes the Translator if the Translator property is not null. You can use the AsciiViewerClass.SetTranslator method to set the Translator property.

See Also: AsciiViewerClass.SetTranslator

## AsciiSearchClass Methods

### Ask (solicit search specifications)

**Ask**( [*startline*] ), VIRTUAL

---

<b>Ask</b>	Prompts the end user for search specifications then positions to the specified search value.
<i>startline</i>	The offset or position of the line at which to begin the search, typically the current line position. If omitted, <i>startline</i> defaults to one (1).

The **Ask** method prompts the end user for search specifications then positions the file to the next line subject to the search specifications, or issues an appropriate message if the search value is not found.

Implementation: The Ask method prompts the end user for search specifications including a value to search for, the direction of the search, and whether the search is case sensitive. If the user invokes the search (doesn't cancel), the Ask method positions the file to the next line containing the search value, or issues an appropriate message if the search value is not found.

Example:

```
ACCEPT
CASE FIELD()
OF ?PrintButton
  IF EVENT() = EVENT:Accepted
    IF ViewerActive THEN Viewer.Printer.Ask.
  END
OF ?Search
  !on "search" button
  IF EVENT() = EVENT:Accepted
    IF ViewerActive
      StartSearch=CHOOSE(CHOICE(?AsciiBox)>0, | !call Searcher.Ask method
      Viewer.TopLine+CHOICE(?AsciiBox)-1,1) ! passing the currently
      Viewer.Searcher.Ask(StartSearch) ! selected line as the
      ! search's starting point
    END
  END
END
END
```

**Init (initialize the ASCIIsearchClass object)**

`Init( ASCIIFileMgr ), VIRTUAL`

---

**Init**            Initializes the ASCIIsearchClass object.

*ASCIIFileMgr*    The label of the ASCIIFileClass object that manages the file to search. The ASCIIsearchClass object uses the *ASCIIFileMgr* to read from the file.

The **Init** method initializes the ASCIIsearchClass object.

Example:

```
MyViewerClass.Init FUNCTION(FILE AsciiFile,*STRING FileLine,*STRING Filename,|
    UNSIGNED ListBox,ErrorClass ErrHandler,BYTE Enables)
CODE
!program code
IF BAND(Enables,EnableSearch)            !if Search flag is on
    SELF.Searcher &= NEW AsciiSearchClass !instantiate Searcher object
    SELF.Searcher.Init(SELF)            !initialize Searcher object
END
IF BAND(Enables,EnablePrint)            !if Print flag is on
    SELF.Printer &= NEW AsciiPrintClass !instantiate Printer object
    SELF.Printer.Init(SELF)            !initialize Printer object
END
```

## Next (find next line containing search text)

### Next, VIRTUAL

The **Next** method returns the line number of the next line containing the search value specified by the Ask method.

Implementation: The Ask method calls the Next method. The Next method searches for the search value and in the direction set by the Ask method. Alternatively, you can use the Setup method to set the search constraints.

Return Data Type: LONG

Example:

```
MyAsciiSearchClass.Ask PROCEDURE
CODE
!procedure code
CASE EVENT()
OF EVENT:Accepted
CASE FIELD()
OF ?NextButton
SELF.LineCounter=SELF.Next()
IF SELF.LineCounter
SELF.FileMgr.SetLine(SELF.LineCounter)
END
!procedure code
```

See Also: Ask, Setup

**Setup (set search constraints)**

**Setup**( *constraints* [, *startline*] )

---

<b>Setup</b>	Sets the search constraints.
<i>constraints</i>	The label of a structure containing the search constraints. The structure must have the same structure as the FindGroup GROUP declared in ABASCII.INC.
<i>startline</i>	The offset or position of the line at which to begin the search, typically the current line position. If omitted, <i>startline</i> defaults to one (1).

The **Setup** method sets the search constraints. The AsciiSearchClass object applies the constraints when searching the text file.

Implementation: The ABC Library does not call the Setup method. The Setup method is provided so you can do custom searches outside the normal AsciiViewerClass process (without using the Ask method).

The Next method applies the search constraints set by the Setup method. The constraints include the text to search for, the direction in which to search, and whether or not the search is case sensitive.

The FindGroup GROUP is declared in ABASCII.INC as follows:

```
FindGroup  GROUP,TYPE
What       PSTRING(64) !text to look for
MatchCase  BYTE       !case sensitive?
Direction  STRING(4)  !either 'Up ' or 'Down'
END
```

Example:

```
MyAsciiSearchClass.Ask PROCEDURE
Constraints LIKE(FindGroup)
CODE
Constraints.MatchCase = False      !never case sensitive
Constraints.Direction = 'Down'     !always search downward
!prompt end user for search value
SELF.Setup(Constraints,StartLine) !set search constraints
SELF.LineCounter=SELF.Next()      !execute search
IF SELF.LineCounter
  SELF.FileMgr.SetLine(SELF.LineCounter) !set to next line containing search value
ELSE
  MESSAGE('''&CLIP(SELF.Constraints.What)&'' not found.')
END
```

See Also: Ask, Next



---

# ASCIIViewerClass

## ASCIIViewerClass Overview

There are several related classes whose collective purpose is to provide reusable, read-only, viewing, scrolling, searching, and printing capability for files, including variable length files. Although these classes are primarily designed for ASCII text and they anticipate using the Clarion ASCII Driver to access the files, they also work with binary files and with other database drivers. These classes can be used to build other components and functionality as well.

The classes that provide this read-only functionality are the ASCII Viewer classes. The ASCII Viewer classes and their respective roles are:

ASCIIViewerClass	Supervisor class
ASCIIFileClass	Open, read, filter, and index the file
ASCIIPrintClass	Print one or more lines
ASCIISearchClass	Locate and scroll to text

These classes are fully documented in the remainder of this chapter.

---

### ASCIIViewerClass

The ASCIIViewerClass uses the ASCIIFileClass, the ASCIIPrintClass, and the ASCIISearchClass to create a single full featured ASCII file viewer object. This object uses a LIST control to display, scroll, search, and print the contents of the file. Typically, you instantiate only the ASCIIViewerClass in your program which, in turn, instantiates the other classes as needed.

---

### ASCIIFileClass

The ASCIIFileClass identifies, opens (read-only), indexes, and page-loads a file's contents into a QUEUE. The indexing function speeds any reaccess of records and supports page-loading, which in turn allows browsing of very large files.

---

### ASCIIPrintClass

The ASCIIPrintClass lets the end user specify a range of lines to print, then prints them. It also provides access to the standard Windows Print Setup dialog.

---

### ASCIISearchClass

The ASCIISearchClass lets the end user specify a search value, case sensitivity, and a search direction, then scrolls to the next instance of the search value within the file.

## ASCIIViewerClass Relationship to Other Application Builder Classes

The ASCIIViewerClass is derived from the ASCIIFileClass, plus it relies on the ASCIIPrintClass, ASCIISearchClass, ErrorClass, and PopupClass to accomplish some user interface tasks. Therefore, if your program instantiates the ASCIIViewerClass, it must also instantiate these other classes. Much of this is automatic when you INCLUDE the ASCIIViewerClass header (ABASCII.INC) in your program's data section. See the *Conceptual Example*.

## ASCIIViewerClass ABC Template Implementation

The ABC Templates declare a local ASCIIViewer class *and* object for each instance of the ASCIIViewControl template. The ABC Templates automatically include all the classes necessary to support the functionality specified in the ASCIIViewControl template.

The templates *derive* a class from the ASCIIViewerClass for *each* ASCIIViewerClass in the application. The derived class is called *procedure:Viewer#* where *procedure* is the procedure name and # is the instance number of the ASCIIViewControl template. The templates provide the derived class so you can use the ASCIIViewControl template **Classes** tab to easily modify the viewer's behavior on an instance-by-instance basis.

The object is named Viewer# where # is the instance number of the control template. The derived ASCIIViewerClass is local to the procedure, is specific to a single ASCIIViewerClass and relies on the global ErrorClass object.

## ASCIIViewerClass Source Files

The ASCIIViewerClass source code is installed by default to the Clarion \LIBSRC folder. The specific ASCIIViewerClass source code and their respective components are contained in:

ABASCII.INC	ASCIIViewerClass declarations
ABASCII.CLW	ASCIIViewerClass method definitions



## AsciiViewerClass Properties

### Popup (PopupClass object)

#### **Popup &PopupClass**

The **Popup** property is a reference to the PopupClass object for this AsciiViewerClass object. The AsciiViewerClass object uses the Popup property to define and manage its popup menus.

Implementation: The Init method initializes the Popup property.

See Also: Init

### Printer (ASCIIPrintClass object)

#### **Printer &ASCIIPrintClass**

The **Printer** property is a reference to the ASCIIPrintClass object for this AsciiViewerClass object. The AsciiViewerClass object uses the Printer property to solicit print ranges and specifications from the end user, then print from the file subject to the specifications.

Implementation: The AddItem and Init methods initialize the Printer property.

The **Printer** property is added to the AsciiViewer by calling the AddItem method. The AsciiViewer does not take ownership of these objects, it just uses them if supplied. It is up to the owner of the objects to destroy them when they are no longer required.

Since these objects are generated by the templates in local procedure scope, they will be destroyed when the enclosing generated procedure dies. If these objects are created by hand coding, then they should be destroyed by whoever creates them.

The exception to this is when the **EnablePrint** parameter is set on the **Init** call. In this case the AsciiViewer may create it's own "internal" printer and viewer. In this case, they are destroyed in the **Kill** method of the AsciiViewer.

See Also: AddItem, Init

## **Searcher (ASCIISearchClass object)**

**Searcher**      **&ASCIISearchClass**

The **Searcher** property is a reference to the ASCIISearchClass object for this ASCIIViewerClass object. The ASCIIViewerClass object uses the Searcher property to solicit search values from the end user, then locate the values within the browsed file.

Implementation:      The AddItem and Init methods initialize the Searcher property.

The **Searcher** property is added to the AsciiViewer by calling the AddItem method. The AsciiViewer does not take ownership of these objects, it just uses them if supplied. It is up to the owner of the objects to destroy them when they are no longer required.

Since these objects are generated by the templates in local procedure scope, they will be destroyed when the enclosing generated procedure dies. If these objects are created by hand coding, then they should be destroyed by whoever creates them.

The exception to this is when the **EnableSearch** parameter is set on the **Init** call. In this case the AsciiViewer may create it's own "internal" viewer. In this case, they are destroyed in the **Kill** method of the AsciiViewer.

See Also:              AddItem, Init

## **TopLine (first line currently displayed)**

**TopLine**      **UNSIGNED**

The **TopLine** property contains the offset or position of the first line currently displayed by the ASCIIViewerClass object. The ASCIIViewerClass object uses the TopLine property to manage scrolling and scrollbar thumb positioning.

## AsciiViewerClass Methods

### AsciiViewerClass Functional Organization--Expected Use

As an aid to understanding the ASCIIViewerClass, it is useful to organize the its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the ASCIIViewerClass methods.

#### Non-Virtual Methods

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the ASCIIViewerClass object
Kill	shut down the ASCIIViewerClass object

##### **Mainstream Use:**

AskGotoLine	go to user specified line
DisplayPage	display new page
PageDown	scroll down one page
PageUp	scroll up one page
TakeEvent	process ACCEPT loop event

##### **Occasional Use:**

AddItem	add printer or searcher object
GetFilename <sub>i</sub>	return the filename
GetLastLineNo <sub>i</sub>	return last line number
GetLine <sub>i</sub>	return line of text
GetPercentile <sub>i</sub>	convert file position to percentage
Reset	reset the ASCIIViewerClass object
SetPercentile <sub>i</sub>	convert percentage to file position
SetLine <sub>v</sub>	position to specific line
SetLineRelative	move N lines

<sub>i</sub> These methods are inherited from the ASCIIFileClass.

<sub>v</sub> These methods are also virtual.

---

**Virtual Methods**

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

FormatLine	format text
SetLine	position to specific line
ValidateLine	implement a filter

| These methods are inherited from the ASCIIFileClass.

**AddItem (program the AsciiViewer object)**

```
AddItem( | printer | )
          | searcher |
```

---

**AddItem** Adds specific functionality to the AsciiViewer object.

*printer* The label of an AsciiPrintClass object.

*searcher* The label of an AsciiSearchClass object.

The **AddItem** method adds specific functionality to the AsciiViewer object. This method provides an alternative to the Init method for adding or changing the print and search capability of the AsciiViewer object.

Implementation: The AddItem method sets the value of the Printer or Searcher property, initializes the *printer* or *searcher*, then enables the corresponding popup menu item.

Example:

```
MyPrinter CLASS(AsciiPrintClass)           !declare custom printer object
NewMethod PROCEDURE
          END
MySearcher CLASS(AsciiSearchClass)         !declare custom searcher object
NewMethod PROCEDURE
          END
```

CODE

```
Viewer.Init(AsciiFile,A1:line,Filename,?AsciiBox,GlobalErrors)
Viewer.AddItem(MyPrinter)                   !add print functionality
Viewer.AddItem(MySearcher)                 !add search functionality
```

See Also: Init, Printer, Searcher

## AskGotoLine (go to user specified line)

### AskGotoLine

The **AskGotoLine** method prompts the end user for a specific line number to display, then positions the file to the line nearest the one specified.

Implementation: The ASCIIViewerClass invokes the AskGotoLine method from a RIGHT-CLICK popup menu. The AskGotoLine method calls the SetLine method to position to the requested record.

Example:

```
MyViewerClass.TakeEvent PROCEDURE(UNSIGNED EventNo)
CODE
CASE EventNo
OF EVENT:AlertKey
  IF KEYCODE()=MouseRight
    CASE SELF.Popup.Ask()
    OF 'Print'
      SELF.Printer.Ask
    OF 'Goto'
      SELF.AskGotoLine
    END
  END
END
END
```

See Also: SetLine

## **DisplayPage (display new page)**

**DisplayPage**( [*line number*] )

---

**DisplayPage** Displays a new page from the file.

*line number* An integer constant, variable, EQUATE or expression that contains the offset or position of the line of text to include in the display. If omitted, *line number* defaults to the value of the TopLine property.

The **DisplayPage** method displays a new page from the file. The display includes the line at *line number*, or the line specified by the TopLine property, if *line number* is omitted.

Example:

```
MyViewerClass.Reset PROCEDURE(*STRING Filename)
  CODE
  FREE(SELF.DisplayQueue)
  DISPLAY(SELF.ListBox)
  PARENT.Reset(Filename)
  SELF.TopLine=1
  SELF.DisplayPage
  SELECT(SELF.ListBox,1)
```

See Also:        TopLine

**Init (initialize the ASCIIViewerClass object)**

**Init**( *file*, *field*, [*filename*], *list control*, *error handler* [, *features*] )

---

<b>Init</b>	Initializes the ASCIIViewerClass object.
<i>file</i>	The label of the file to display.
<i>field</i>	The fully qualified label of the <i>file</i> field to display.
<i>filename</i>	The label of the <i>file</i> 's NAME attribute variable. If omitted, the file has a constant NAME attribute. If null (""), the Init method prompts the end user to select a file.
<i>list control</i>	An integer constant, variable, EQUATE, or expression containing the control number of the LIST that displays the <i>file</i> contents.
<i>error handler</i>	The label of the ErrorClass object to handle errors encountered by this ASCIIViewerClass object.
<i>features</i>	An integer constant, variable, EQUATE, or expression that tells the ASCIIViewerClass object which features to implement; for example, printing ( <i>EnablePrint</i> ), searching ( <i>EnableSearch</i> ), or both. If omitted, no additional features are implemented.

The **Init** method initializes the ASCIIViewerClass object and returns a value indicating whether it successfully accessed the *file* and is ready to proceed.

Implementation: The Init method returns one (1) if it accessed the *file* and is ready to proceed; it returns zero (0) and calls the Kill method if unable to access the *file* and cannot proceed. If the Init method returns zero (0), the ASCIIViewerClass object is not initialized and you should not call its methods.

You can set the *features* value with the following EQUATEs declared in ASCII.INC. Pass either EQUATE to implement its feature (search or print), or add the EQUATEs together to implement both features.

**EnableSearch**            **BYTE(001b)**

**EnablePrint**            **BYTE(010b)**

Return Data Type: **BYTE**

Example:

```

PROGRAM
MAP
END

INCLUDE('ABASCII.INC')           !declare ASCIIViewer Class

ViewWindow WINDOW('View an ASCII File'),AT(3,7,296,136),SYSTEM,GRAY
    LIST,AT(5,5,285,110),USE(?AsciiBox),IMM
    BUTTON('&Print'),AT(5,120),USE(?Print)
    BUTTON('&Search'),AT(45,120),USE(?Search)
    BUTTON('&Close'),AT(255,120),USE(?Close)
END

GlobalErrors ErrorClass          !declare GlobalErrors object
Viewer   AsciiViewerClass,THREAD !declare Viewer object

ViewerActive BYTE(False),THREAD  !Viewer initialized flag
Filename     STRING(255),THREAD  !FileName variable

AsciiFile FILE,DRIVER('ASCII'),NAME(Filename),PRE(A1),THREAD
RECORD     RECORD,PRE()
Line       STRING(255)
          END
          END

CODE

GlobalErrors.Init                !initialize GlobalErrors object
OPEN(ViewWindow)                !open the window
!Initialize Viewer with:
ViewerActive=Viewer.Init( AsciiFile, | ! file label,
    A1:line,                        | ! file field to display
    Filename,                       | ! variable file NAME attribute
    ?AsciiBox,                      | ! LIST control number
    GlobalErrors,                   | ! ErrorClass object
    EnableSearch+EnablePrint)      | ! features to implement flag
IF ~ViewerActive THEN RETURN.     !if init unsuccessful, don't
                                ! call other Viewer methods
ACCEPT                            !If init succeeded, proceed
    IF EVENT() = EVENT:CloseWindow
    IF ViewerActive THEN Viewer.Kill. !If init succeeded, shut down
    END
    !program code
END

```

See Also: Kill



```
IF EVENT() = EVENT:CloseWindow
  IF ViewerActive THEN Viewer.Kill.      !If init succeeded, shut down
END
!program code
END
```

## PageDown (scroll down one page)

### PageDown, PROTECTED

The **PageDown** method scrolls the display down one "page." A page is the number of lines displayed in the ASCIIViewerClass object's LIST control.

Example:

```
MyViewerClass.TakeEvent PROCEDURE(UNSIGNED EventNo)
CODE
  IF FIELD()=SELF.ListBox
    CASE EventNo
      OF EVENT:Scrollup
        SELF.SetLineRelative(-1)
      OF EVENT:ScrollDown
        SELF.SetLineRelative(1)
      OF EVENT:PageUp
        SELF.PageUp
      OF EVENT:PageDown
        SELF.PageDown
    END
  END
END
```

## PageUp (scroll up one page)

### PageUp, PROTECTED

The **PageUp** method scrolls the display up one "page." A page is the number of lines displayed in the ASCIIViewerClass object's LIST control.

Example:

```
MyViewerClass.TakeEvent PROCEDURE(UNSIGNED EventNo)
CODE
  IF FIELD(=SELF.ListBox
    CASE EventNo
      OF EVENT:Scrollup
        SELF.SetLineRelative(-1)
      OF EVENT:ScrollDown
        SELF.SetLineRelative(1)
      OF EVENT:PageUp
        SELF.PageUp
      OF EVENT:PageDown
        SELF.PageDown
    END
  END
END
```

## Reset (reset the ASCIIViewerClass object)

**Reset**( *filename* )

---

**Reset**            Resets the ASCIIViewerClass object.

*filename*            The label of the ASCIIFile property's NAME attribute variable.

The **Reset** method resets the ASCIIViewerClass object and returns a value indicating whether the end user selected a file or did not select a file. A return value of one (1) indicates a file was selected and *filename* contains its pathname; a return value of zero (0) indicates no file was selected and *filename* is empty.

Implementation:    The Reset method frees the display QUEUE and calls the ASCIIFileClass.Reset method to get a new filename from the end user. Reset refills the display QUEUE and redisplay the list box if a new file was selected.

Return Data Type:    BYTE

Example:

```
AsciiFileClass.Init FUNCTION |
  (FILE AsciiFile,*STRING FileLine,*STRING FName,ErrorClass ErrorHandler)

  CODE
  SELF.AsciiFile&=AsciiFile
  SELF.Line&=FileLine
  SELF.ErrorMgr&=ErrorHandler
  SELF.IndexQueue&=NEW(IndexQueue)
  IF ~SELF.Reset(FName)
    SELF.Kill
    RETURN False
  END
  RETURN True
```

See Also:            ASCIIFile, ASCIIFileClass.Reset

**SetLine (position to specific line)**

**SetLine**( *line number* ), PROTECTED, VIRTUAL

---

<b>SetLine</b>	Positions the ASCIIViewerClass object to a specific line.
<i>line number</i>	An integer constant, variable, EQUATE or expression that contains the offset or position of the line of text to position to.

The **SetLine** method positions the ASCIIViewerClass object to a specific line within the browsed file.

Implementation: The AskGotoLine method, the ASCIIFileClass.SetPercentile method, and the ASCIIsearchClass.Ask method all use the SetLine method to position to the required text line.

Example:

```
MyViewerClass.AskGotoLine PROCEDURE
LineNo LONG,STATIC
OKGo   BOOL(False)
GotoDialog WINDOW('Goto'),AT(,,96,38),GRAY,DOUBLE
        SPIN(@n_5),AT(36,4,56,13),USE(LineNo),RANGE(1,99999)
        PROMPT('&Line No:'),AT(4,9,32,10),USE(?Prompt1)
        BUTTON('&Go'),AT(8,22,40,14),USE(?GoButton)
        BUTTON('&Cancel'),AT(52,22,40,14),USE(?CancelButton)
        END
CODE
OPEN(GotoDialog)
ACCEPT
CASE EVENT()
OF EVENT:Accepted
CASE ACCEPTED()
OF ?GoButton
OKGo=True
POST(EVENT:CloseWindow)
OF ?CancelButton
POST(EVENT:CloseWindow)
END
END
END
CLOSE(GotoDialog)
IF OKGo THEN SELF.SetLine(LineNo).
```

See Also: AskGoToLine, ASCIIFileClass.SetPercentile, ASCIIsearchClass.Ask

## SetLineRelative (move *n* lines)

**SetLineRelative**( *lines* ), PROTECTED

---

**SetLineRelative**      Positions the ASCIIViewerClass object to a relative line.

*lines*              An integer constant, variable, EQUATE or expression containing the number of lines to move from the current position. A positive value moves downward; a negative value moves upward.

The **SetLineRelative** method repositions the ASCIIViewerClass object *lines* lines from the current position.

Example:

```
MyViewerClass.TakeScrollOne PROCEDURE(UNSIGNED EventNo)
  CODE
  IF FIELD()=SELF.ListBox
    CASE EventNo
    OF EVENT:Scrollup
      SELF.SetLineRelative(-1)
    OF EVENT:ScrollDown
      SELF.SetLineRelative(1)
    END
  END
END
```

**SetTranslator (set run-time translator:ASCIIViewerClass)****SetTranslator**( *translator* )**SetTranslator** Sets the TranslatorClass object for the AsciiViewerClass object.*translator* The label of the TranslatorClass object for this AsciiViewerClass object.

The **SetTranslator** method sets the TranslatorClass object for the AsciiViewerClass object. By specifying a TranslatorClass object for the AsciiViewerClass object, you automatically translate any window or popup menu text displayed by the viewer.

Implementation: The SetTranslator method sets the TranslatorClass object for the PopupClass, AsciiPrintClass, and AsciiSearchClass objects.

Example:

```

Viewer      AsciiViewerClass      !declare Viewer object
Translator  TranslatorClass       !declare Translator object
CODE
  Translator.Init                  !initialize Translator object
ViewerActive=Viewer.Init( AsciiFile,
                          |       ! file label,
                          |       ! file field to display
                          |       ! variable file NAME attribute
                          |       ! LIST control number
                          |       ! ErrorClass object
                          |       ! features to implement flag
                          |       ! if init unsuccessful, don't
                          |       ! call other Viewer methods
                          |       ! enable text translation
IF ~ViewerActive THEN RETURN.

Viewer.SetTranslator(Translator)
!program code

```

## TakeEvent (process ACCEPT loop event:ASCIIViewerClass)

**TakeEvent**( *event* ), PROC

---

**TakeEvent** Processes an ACCEPT loop event.

*event* An integer constant, variable, EQUATE or expression containing the event number.

The **TakeEvent** method processes an ACCEPT loop event on behalf of the ASCIIViewerClass object and returns a value indicating whether a CYCLE to the top of the ACCEPT loop is required to properly refresh the display.

Implementation: The TakeEvent method handles resizing, RIGHT-CLICKS, LEFT-CLICKS, and scrolling events.

A return value of zero (0) indicates no CYCLE is needed; any other return value requires a CYCLE.

Return Data Type: BYTE

Example:

```
ACCEPT
CASE FIELD()
OF ?AsciiBox
IF ViewerActive
IF Viewer.TakeEvent(EVENT())
CYCLE
END
END
END
END
```



---

# BreakManagerClass

## BreakManagerClass - Overview

The BreakManagerClass handles embedded break events for a target report. Each break can perform totaling based on a data element's contents changing. Breaks can be nested, allowing the contents of one break result to determine another break result. Conditional headers and footers can be printed by any break. Each break is totally customizable through available embed points defined in virtual methods.

## BreakManagerClass - Concepts

Each embedded break generated by BreakManager is controlled by a template interface in the Report Properties dialog and through a set of embed points generated. Embedded breaks are designed to give the developer more control "behind the scenes". The BreakManager is not related to the traditional Break logic supported by the structure of the report.

Each break inherits a set of methods and properties defined by an internal LevelManager. An internal break queue stores a unique break and level id which triggers the appropriate break logic.

## BreakManagerClass - Relationship to Other Application Builder Classes

### ReportManagerClass

---

The BreakManager class is derived from the ReportManager class.

### LevelManagerClass

---

Each ReportManagerClass utilizing embedded breaks declares a LevelManagerClass to manage the nesting and execution of the embedded break logic.

## BreakManagerClass - ABC Template Implementation

The Report template declares a BreakManager when a break is added through the Break tab located in the Report Properties dialog.

## BreakManagerClass - Source Files

The BreakManagerClass source code is installed by default to the Clarion \LIBSRC folder. The specific BreakManagerClass source code and their respective components are contained in:

ABBreak.INC	EditClass declarations
ABBreak.CLW	EditClass method definitions

## BreakManagerClass - Conceptual Example

The following example shows a sequence of statements to declare, and instantiate a BreakManager object.

```
MEMBER('people.clw')                                ! This is a MEMBER module

INCLUDE('ABBREAK.INC'),ONCE
INCLUDE('ABBROWSE.INC'),ONCE
INCLUDE('ABREPORT.INC'),ONCE

MAP
  INCLUDE('PEOPL005.INC'),ONCE                      !procedure decl
END

PrintPEO:KeyLastName PROCEDURE    ! Generated from procedure template - Report

Progress:Thermometer BYTE          !
FilesOpened           BYTE          !
TestCount             LONG          !
Process:View          VIEW(people)
  PROJECT(PEO:FirstName)
  PROJECT(PEO:Gender)
  PROJECT(PEO:Id)
  PROJECT(PEO:LastName)
END
LocMyFocusControlExt SHORT(0)      !Used by the ENTER Instead of Tab template
LocEnableEnterByTab  BYTE(1)       !Used by the ENTER Instead of Tab template
ProgressWindow       WINDOW('Progress...'),AT(, ,142,59),CENTER,TIMER(1),GRAY,DOUBLE
  PROGRESS,USE(Progress:Thermometer),AT(15,15,111,12),RANGE(0,100)
  STRING(''),AT(0,3,141,10),USE(?Progress:UserString),CENTER
  STRING(''),AT(0,30,141,10),USE(?Progress:PctText),CENTER
  BUTTON('Cancel'),AT(45,42,50,15),USE(?Progress:Cancel)
END
```

```

report    REPORT,AT(1000,1540,6000,7458),PRE(RPT),FONT('Arial',10,,),THOUS
          HEADER,AT(1000,1000,6000,542)
          STRING('People by Last Name'),AT(0,20,6000,220),CENTER,FONT(,,FONT:bold)
          BOX,AT(0,260,6000,280),COLOR(COLOR:Black),FILL(COLOR:Silver)
          LINE,AT(1500,260,0,280),COLOR(COLOR:Black)
          LINE,AT(3000,260,0,280),COLOR(COLOR:Black)
          LINE,AT(4500,260,0,280),COLOR(COLOR:Black)
          STRING('Id'),AT(50,310,1400,170),TRN
          STRING('First Name'),AT(1550,310,1400,170),TRN
          STRING('Last Name'),AT(3050,310,1400,170),TRN
          STRING('Gender'),AT(4550,310,1400,170),TRN
          END
detail    DETAIL,AT(,6000,281),USE(?unnamed)
          LINE,AT(1500,0,0,280),COLOR(COLOR:Black)
          LINE,AT(3000,0,0,280),COLOR(COLOR:Black)
          LINE,AT(4500,0,0,280),COLOR(COLOR:Black)
          STRING(@S10),AT(50,50,1400,170),USE(PEO:Id),RIGHT(1)
          STRING(@S30),AT(1550,50,1400,170),USE(PEO:FirstName)
          STRING(@S30),AT(3050,50,1400,170),USE(PEO:LastName)
          STRING(@S1),AT(4552,52,240,167),USE(PEO:Gender)
          STRING(@n_4),AT(5208,42),USE(TestCount),RIGHT(1)
          LINE,AT(50,280,5900,0),COLOR(COLOR:Black)
          END
          FOOTER,AT(1000,9000,6000,219)
          STRING(@pPage <<<#p),AT(5250,30,700,135),PAGE NO,USE(?PageCount)
          END
          END

ThisWindow    CLASS(ReportManager)
Init          PROCEDURE(),BYTE,PROC,DERIVED! Method added to host embed code
Kill         PROCEDURE(),BYTE,PROC,DERIVED! Method added to host embed code
OpenReport   PROCEDURE(),BYTE,PROC,DERIVED! Method added to host embed code
SetAlerts    PROCEDURE(),DERIVED          ! Method added to host embed code
TakeWindowEvent PROCEDURE(),BYTE,PROC,DERIVED! Method added to host embed code
          END

ThisReport    CLASS(ProcessClass)          ! Process Manager
TakeRecord   PROCEDURE(),BYTE,PROC,DERIVED! Method added to host embed code
          END

ProgressMgr   StepStringClass              ! Progress Manager
Previewer    PrintPreviewClass            ! Print Previewer
BreakMgr     BreakManagerClass            ! Break Manager

CODE
GlobalResponse = ThisWindow.Run()          !Opens the window and starts an Accept Loop

!-----
DefineListboxStyle ROUTINE
!|
!| This routine create all the styles to be shared in this window
!| It's called after the window open
!|

```

```
!-----  
ThisWindow.Init PROCEDURE  
  
ReturnValue          BYTE,AUTO  
  
CODE  
GlobalErrors.SetProcedureName('PrintPEO:KeyLastName')  
SELF.Request = GlobalRequest          ! Store the incoming request  
ReturnValue = PARENT.Init()  
IF ReturnValue THEN RETURN ReturnValue.  
SELF.FirstField = ?Progress:Thermometer  
SELF.VCRRequest &= VCRRequest  
SELF.Errors &= GlobalErrors  
CLEAR(GlobalRequest)  
CLEAR(GlobalResponse)  
SELF.AddItem(Translator)  
Relate:people.Open  
! File people used by this procedure, so make sure it's RelationManager is open  
SELF.FilesOpened = True  
BreakMgr.Init()                      !Initialize the BreakManager object  
BreakMgr.AddBreak()                  !A break exists  
BreakMgr.AddLevel()                  !People Count is the name of the brteak  
BreakMgr.AddResetField(PEO:Gender) !resets when gender changes  
BreakMgr.AddTotal(TestCount,1)      !Performs a count when break occurs  
SELF.AddItem(BreakMgr)  
  
OPEN(ProgressWindow)                ! Open window
```

## BreakManagerClass - Properties

The BreakManagerClass contains no public properties.

## BreakManagerClass - Methods

The BreakManagerClass contains the following methods:

### **AddBreak (add a Break)**

#### **AddBreak( )**

---

**AddBreak** Identifies to the corresponding report that an embedded break object is active.

The **AddBreak** method identifies that an embedded break object is active for a corresponding report. An internal BreakID number is assigned for each break object created. This BreakID is incremented when a subsequent **AddBreak** method is executed.

Implementation: The AddBreak method is called for each break object that is active for a report. Each object is instantiated from the LevelManagerClass. An AddLevel method follows each AddBreak method.

Example:

```
BreakMgr.AddBreak()           !First Break
BreakMgr.AddLevel()          !Auto assign level 1
BreakMgr.AddResetField(PEO:Gender)
BreakMgr.AddTotal(TestCount,1)
BreakMgr.AddBreak()         !Second Break
BreakMgr.AddLevel() !Break2
```

See Also: Add Level

## **AddLevel (add a Level to the BreakId Break)**

**AddLevel**( | *breakid* | )

---

**AddLevel** Assign a level number to the active break.

*breakid* An integer that identifies which internal break to assign this level to.

The **AddLevel** method identifies the order of break execution that is assigned to a corresponding report. This is used when nested breaks are assigned, and controls which break is executed first.

Implementation: The **AddLevel** method is called after each break is added, directly following the **AddBreak** method.

Example:

```
BreakMgr.AddBreak()           !First Break
BreakMgr.AddLevel()          !Auto assign level 1
BreakMgr.AddResetField(PEO:Gender)
BreakMgr.AddTotal(TestCount,1)
BreakMgr.AddBreak()           !Second Break
BreakMgr.AddLevel() !Break2
```

See Also: Add Break

## AddResetField (add a reset field to the last Level added)

**AddResetField**( *fieldlabel* )

---

**AddResetField**            Identify a field to reset on the active break.

*fieldlabel*                The label of a field that the break will reset

The **AddResetField** method identifies the field whose contents will be reset when a break is executed.

Implementation:    The **AddResetField** method is called after each break and level is added, directly following the AddLevel method. In the template interface, it is only active when the **Reset on Break** option is set to TRUE.

Example:

```
BreakMgr.AddBreak()                    !First Break
BreakMgr.AddLevel()                   !Auto assign level 1
BreakMgr.AddResetField(PEO:Gender)   !Reset on break is active
BreakMgr.AddTotal(TestCount,1)
BreakMgr.AddBreak()                   !Second Break
BreakMgr.AddLevel() !Break2
```

See Also: Add Break

**AddTotal (add a total field to the last Level added)**

**AddTotal** (*target, source, type, reset*)  
 (*target, source, type, reset, condition*)  
 (*target, reset*)  
 (*target, reset, condition*)

---

<b>AddTotal</b>	Identify the break total type and optional source/target fields and condition.
<i>target</i>	The label of a field that the total result will be stored into.
<i>source</i>	The label of a field that the total result will be calculated from.
<i>type</i>	A byte that identifies the total type defined for the break. A count type = 1, average type = 2, sum type = 3.
<i>reset</i>	A byte that specifies whether or not the target variable is reset on each break detected. A value of 1 specifies a reset, zero will not reset the target value.
<i>condition</i>	A valid expression that will force a reset when evaluated to TRUE.

The **AddTotal** method is an overloaded method that calculates three types of conditional or unconditional totaling.

Implementation: The **AddTotal** method is called after each break and level is added, directly following the AddResetField method. In the template interface, it is only active when totaling is added to the break.

Example:

```
BreakMgr.Init()
BreakMgr.AddBreak()
BreakMgr.AddLevel() !Count Break
BreakMgr.AddResetField(PEO:Gender)
BreakMgr.AddTotal(CountValue,1)
BreakMgr.AddBreak()
BreakMgr.AddLevel() !SumBreak
BreakMgr.AddResetField(PEO:Gender)
BreakMgr.AddTotal(SumValue,SourceToTotal,eBreakTotalSum,1)
BreakMgr.AddBreak()
BreakMgr.AddLevel() !ConditionalAverage
BreakMgr.AddResetField(PEO:Gender)
BreakMgr.AddTotal(AverageValue,SourceToAverage,eBreakTotalAve,0,'PEO:Gender = 'M')
```

## **Construct (automatic initialization of the BreakManager object)**

### **Construct**

The **Construct** method performs the necessary code to initialize the BreakManager object.

Implementation: The Construct method is automatically called when the object is instantiated. The internal break queue is created, and the Break Id is reset to zero.

## **Destruct (automatic destroy of the Breakmanager object)**

### **Destruct**

The **Destruct** method performs the necessary code to destroy the BreakManager object.

Implementation: The Destruct method is automatically called when the object is instantiated. Each break object instantiated in the Init method is cleared from memory.

## **Init (initialize the BreakManager object)**

### **Init( )**

---

**Init** Initialize the BreakManager object.

The **Init** method initializes the BreakManager by clearing any prior entries in the internal Break Queue, which holds any breaks defined. Normally, the Destruct method also performs a similar function, but the **Init** method ensures that the Break Queue is clean before calling the report.

Implementation: The **Init** method is called prior to the first AddBreak method.

Example:

```
CODE
GlobalErrors.SetProcedureName('PrintPEO:KeyLastName')
SELF.Request = GlobalRequest    ! Store the incoming request
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?Progress:Thermometer
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
CLEAR(GlobalRequest)           ! Clear GlobalRequest after storing locally
CLEAR(GlobalResponse)
SELF.AddItem(Translator)
Relate:people.Open
SELF.FilesOpened = True
BreakMgr.Init()
BreakMgr.AddBreak()
```

## TakeEnd (Break closed)

TakeEnd ( )

---

**TakeEnd** A virtual method called prior to closing an active break.

The **TakeEnd** method is a virtual method that performs any necessary code action prior to the close of the active break.

Implementation: The **TakeEnd** method is used by the ABC templates to print a custom footer detail for the active break. There are embed points available if you need to prime any variables prior to its printing.

Example:

```
BreakMgr.TakeEnd PROCEDURE(SHORT BreakId,SHORT LevelId)
```

```
CODE
  CASE BreakId
    END
  OF 2
    CASE LevelId
      OF 1 !SumBreak
        PRINT(RPT:sumdetail)
      END
    OF 3
      CASE LevelId
        OF 1 !ConditionalAverage
          PRINT(RPT:AverageDetail)
        END
      END
    END
  PARENT.TakeEnd(BreakId,LevelId)
```

## TakeStart (Break opened)

### TakeStart ( )

---

**TakeStart** A virtual method called prior to the start of an active break.

The **TakeStart** method is a virtual method that performs any necessary code action prior to the start of the active break.

Implementation: The **TakeStart** method is used by the ABC templates to print a custom header detail for the active break. There are embed points available if you need to prime any variables prior to its printing.

Example:

```
BreakMgr.TakeStart PROCEDURE(SHORT BreakId,SHORT LevelId)
```

```
CODE
  CASE BreakId
  OF 1
    CASE LevelId
    OF 1 !Count Break
      PRINT(RPT:CountDetail)
    END
  OF 3
    CASE LevelId
    OF 1 !ConditionalAverage
      PRINT(RPT:AverageDetail)
    END
  END
PARENT.TakeStart(BreakId,LevelId)
```

## **UpdateTotal (Calculate Break totaling)**

**UpdateTotal( )**

---

**UpdateTotal** A virtual method called prior to the start and ending of any break totaling.

The **UpdateTotal** method is a virtual method that performs any necessary code action prior to the start and finish of any totaling defined in an active break.

Implementation:

The **UpdateTotal** method is used by the ABC templates to allow any modifications or extra cleanup of any totaling performed for an active break. The virtual method embeds provide a "Before Totaling" and "After Totaling" embed point for every total defined in each break.

---

# BrowseEIPManagerClass

## BrowseEIPManagerClass--Overview

The BrowseEIPManagerClass is an EIPManager that displays an Edit-in-place dialog, and handles events for that dialog. Each BrowseClass utilizing Edit-in-place declares a BrowseEIPManagerClass to manage the events and processes that are used by each Edit-in-place field in the browse.

See Also:

[BrowseEIPManagerClass Concepts](#)

[BrowseEIPManagerClass--Relationship to Other Application Builder Classes](#)

[BrowseEIPManagerClass--ABC Template Implementation](#)

[BrowseEIPManagerClass Source Files](#)

[BrowseEIPManagerClass--Conceptual Example](#)

## BrowseEIPManagerClass Concepts

Each Edit-in-place control is a window created on top of the browse from which it is called. The EIPManager dynamically creates an Edit-in-place object and control upon request (Insert, Change, or Delete) by the end user. When the end user accepts the edited record the EIPManager destroys the edit-in-place object and control.

See Also:

[BrowseEIPManagerClass--Relationship to Other Application Builder Classes](#)

[BrowseEIPManagerClass--ABC Template Implementation](#)

[BrowseEIPManagerClass Source Files](#)

[BrowseEIPManagerClass--Conceptual Example](#)

---

## BrowseEIPManagerClass--Relationship to Other Application Builder Classes

### EIPManagerClass

---

The BrowseEIPManager class is derived from the EIPManager class.

### BrowseClass

---

Each BrowseClass utilizing edit-in-place declares a BrowseEIPManagerClass to manage the events and processes that are used by each edit-in-place field in the browse.

The BrowseClass.AskRecord method is the calling method for edit-in-place functionality when edit-in-place is enabled.

### EditClass

---

The BrowseEIPManager provides the basic or "under the hood" interface between the Edit classes and the Browse class. All fields in the browse utilizing customized edit-in-place controls are kept track of by the BrowseEIPManager via the BrowseEditQueue.

## BrowseEIPManagerClass--ABC Template Implementation

The Browse template declares a BrowseEIPManager when the BrowseUpdateButtons control template enables default edit-in-place support for the BrowseBox.

See *Control Templates--BrowseBox*, and *BrowseUpdateButtons* for more information.

## BrowseEIPManagerClass Source Files

The BrowseEIPManagerClass source code is installed by default to the Clarion \LIBSRC folder. The specific BrowseEIPManagerClass source code and their respective components are contained in:

ABBrowse.INC	EditClass declarations
ABBrowse.CLW	EditClass method definitions
ABBrowse.TRN	EditClass translation strings

## BrowseEIPManagerClass--Conceptual Example

The following example shows a sequence of statements to declare, and instantiate a BrowseEIPManager object. The example page-loads a LIST of actions and associated priorities, then edits the list items via edit-in-place. Note that the BrowseClass object declares a BrowseEIPManager which is a reference to the EIPManager as declared in ABBrowse.INC.

```

PROGRAM

_ABCEllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)

INCLUDE('ABBROWSE.INC'),ONCE
INCLUDE('ABEIP.INC'),ONCE
INCLUDE('ABWINDOW.INC'),ONCE
MAP
END

Actions          FILE,DRIVER('TOPSPEED'),PRE(ACT),CREATE,BINDABLE,THREAD
KeyAction        KEY(ACT:Action),NOCASE,OPT
Record           RECORD,PRE()
Action           STRING(20)
Priority          DECIMAL(2)
Completed        DECIMAL(1)
                END
                END

Access:Actions   &FileManager
Relate:Actions   &RelationManager
GlobalErrors     ErrorClass
GlobalRequest    BYTE(0),THREAD
ActionsView      VIEW(Actions)
                END

Queue:Browse     QUEUE
ACT:Action       LIKE(ACT:Action)
ACT:Priority      LIKE(ACT:Priority)
ViewPosition     STRING(1024)
                END

BrowseWindow     WINDOW('Browse Records'),AT(0,0,247,140),SYSTEM,GRAY
                LIST,AT(5,5,235,100),USE(?List),IMM,HVSCROLL,MSG('Browsing Records'),|
                FORMAT('80L~Action~@S20@8R~Priority~L@N2@'),FROM(Queue:Browse)
                BUTTON('&Insert'),AT(5,110,40,12),USE(?Insert),KEY(InsertKey)
                BUTTON('&Change'),AT(50,110,40,12),USE(?Change),KEY(CtrlEnter),DEFAULT
                BUTTON('&Delete'),AT(95,110,40,12),USE(?Delete),KEY(DeleteKey)
                END

```

```

ThisWindow CLASS(WindowManager)
Init          PROCEDURE(),BYTE,PROC,DERIVED
Kill         PROCEDURE(),BYTE,PROC,DERIVED
            END

BRW1 CLASS(BrowseClass)
Q          &Queue:Browse
Init PROCEDURE|
(SIGNED ListBox,*STRING Posit,VIEW V,QUEUE Q,RelationManager RM,WindowManager WM)
            END

BRW1::EIPManager      BrowseEIPManager

CODE
GlobalErrors.Init
Relate:Actions.Init
GlobalResponse = ThisWindow.Run()
Relate:Actions.Kill
GlobalErrors.Kill

ThisWindow.Init PROCEDURE

ReturnValue          BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue =PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?List
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(Toolbar)
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
Relate:Actions.Open
FilesOpened = True
BRW1.Init|
(?List,Queue:Browse.ViewPosition,BRW1::View:Browse,Queue:Browse,Relate:Actions,SELF)
OPEN(BrowseWindow)
SELF.Opened=True
BRW1.Q &= Queue:Browse
BRW1.AddSortOrder(,ACT:KeyAction)
BRW1.AddLocator(BRW1::Sort0:Locator)
BRW1::Sort0:Locator.Init(,ACT:Action,1,BRW1)
BRW1.AddField(ACT:Action,BRW1.Q.ACT:Action)
BRW1.AddField(ACT:Priority,BRW1.Q.ACT:Priority)
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change

```

```
BRW1.DeleteControl=?Delete
BRW1.AddToolBarTarget(ToolBar)
SELF.SetAlerts()
RETURN ReturnValue
```

ThisWindow.Kill PROCEDURE

```
ReturnValue          BYTE,AUTO
CODE
ReturnValue =PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
IF FilesOpened
    Relate:Actions.Close
END
RETURN ReturnValue
```

BRW1.Init PROCEDURE(SIGNED ListBox,\*STRING Posit,VIEW V,QUEUE Q,RelationManager |  
RM,WindowManager WM)

```
CODE
PARENT.Init(ListBox,Posit,V,Q,RM,WM)
SELF.EIP &= BRW1::EIPManager
```

## BrowseEIPManagerClass Properties

### **BrowseEIPManagerClass Properties**

The BrowseEIPManagerClass contains the following property and inherits all the properties of the EIPManagerClass.

### **BC (browse class)**

**BC**     **&BrowseClass, PROTECTED**

The **BC** property is a reference to the calling BrowseClass object.

## BrowseEIPManagerClass Methods

### **BrowseEIPManagerClass--Functional Organization--Expected Use**

As an aid to understanding the EIPManagerClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EIPManagerClass methods.

#### **Non-Virtual Methods**

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	▷	initialize the BrowseEditClass object
Kill	▷	shut down the BrowseEditClass object

##### **Mainstream Use:**

TakeNewSelection▷	handle Event:NewSelections
-------------------	----------------------------

##### **Occasional Use:**

ClearColumn▷	reset column property values
TakeCompleted▷	process completion of edit

▷ These methods are also Derived

#### **Derived Methods**

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are derived, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init	▷	initialize the BrowseEditClass object
Kill	▷	shut down the BrowseEditClass object
TakeNewSelection▷	handle Event:NewSelections	
ClearColumn▷	reset column property values	
TakeCompleted▷	process completion of edit	

## ClearColumn (reset column property values)

### ClearColumn, DERIVED

The **ClearColumn** method checks for a value in the LastColumn property and conditionally sets the column values to 0.

The TakeCompleted method calls the ClearColumn method.

Example:

```
BrowseEIPManager.TakeCompleted PROCEDURE(BYTE Force)
SaveAns UNSIGNED,AUTO
Id      USHORT,AUTO
CODE
SELF.Again = 0
SELF.ClearColumn
SaveAns = CHOOSE(Force = 0,Button:Yes,Force)
IF SELF.Fields.Equal()
    SaveAns = Button:No
ELSE
    IF ~Force
        SaveAns = SELF.Errors.Message(Msg:SaveRecord,|
            Button:Yes+Button:No+Button:Cancel,Button:Yes)
    END
END
! code to handle user input from SaveRecord message
```

See Also: Column

## **Init (initialize the BrowseEIPManagerClass object)**

**Init, DERIVED, PROC**

The **Init** method initializes the BrowseEIPManagerClass object.

Implementation: The Init method primes variables and calls the InitControls method which then initializes the appropriate edit-in-place controls. It is indirectly called by the BrowseClass.AskRecord method via a call to an inherited Run method.

Return Data Type: BYTE

Example:

**WindowManager.Run PROCEDURE**

**CODE**

**IF ~SELF.Init()**

**SELF.Ask**

**END**

**SELF.Kill**

**RETURN CHOOSE(SELF.Response=0,RequestCancelled,SELF.Response)**

See Also: BrowseClass.ResetFromAsk

**Kill (shut down the BrowseEIPManagerClass object)****Kill, DERIVED, PROC**

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code. The Kill method must leave the object in a state in which it can be initialized.

Implementation: The Kill method calls the BrowseClass.ResetFromAsk method.

Return Data Type: BYTE

Example:

**WindowManager.Run PROCEDURE**

**CODE**

**IF ~SELF.Init()**

**SELF.Ask**

**END**

**SELF.Kill**

**RETURN CHOOSE(SELF.Response=0,RequestCancelled,SELF.Response)**

See Also: BrowseClass.ResetFromAsk

## TakeCompleted (process completion of edit)

**TakeCompleted( *force* ), DERIVED**

**TakeCompleted** Determines the edit-in-place dialog's action after either a loss of focus or an end user action.

*force* An integer constant, variable, EQUATE, or expression that indicates the record being edited should be saved without prompting the end user.

The **TakeCompleted** method either saves the record being edited and end the edit-in-place process, or prompts the end user to save the record and end the edit-in-place process, cancel the changes and end the edit-in-place process, or remain editing.

Implementation: The EIPManager.TakeFocusLoss and EIPManager.TakeAction methods call the TakeCompleted method.

**Note:** TakeCompleted does not override the WindowManager.TakeCompleted method.

Example:

```
EIPManager.TakeFocusLoss PROCEDURE
CODE
CASE CHOOSE(SELF.FocusLoss&=NULL,EIPAction:Default,BAND(SELF.FocusLoss,EIPAction:Save))
  OF EIPAction:Always OROF EIPAction:Default
    SELF.TakeCompleted(Button:Yes)
  OF EIPAction:Never
    SELF.TakeCompleted(Button:No)
  ELSE
    SELF.TakeCompleted(0)
END
```

See Also: EIPManager.TakeFocusLoss, EIPManager.TakeAction

## TakeNewSelection (reset edit-in-place column)

TakeNewSelection, DERIVED, PROC

The **TakeNewSelection** method resets the edit-in-place column selected by the end user.

Implementation: TakeNewSelection calls ResetColumn, and returns a Level:Benign.

Return Data Type: BYTE

Example:

```
WindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
CODE
  IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
  END
  CASE EVENT()
  OF EVENT:Accepted
    RVal = SELF.TakeAccepted()
  OF EVENT:Rejected
    RVal = SELF.TakeRejected()
  OF EVENT:Selected
    RVal = SELF.TakeSelected()
  OF EVENT:NewSelection
    RVal = SELF.TakeNewSelection()
  OF EVENT:AlertKey
    IF SELF.HistoryKey AND KEYCODE() = SELF.HistoryKey
      SELF.RestoreField(FOCUS())
    END
  END
  IF RVal THEN RETURN RVal.
```

See Also: ResetColumn



# BrowseClass

## BrowseClass Overview

The BrowseClass is a ViewManager with a user interface for navigating through the result set of the underlying VIEW.

## BrowseClass Concepts

The BrowseClass uses several related classes to provide standard browse functionality--that is, file-loaded or page-loaded lists with automatic scrolling, searching, ranging, filtering, resets, conditional colors, conditional icons, etc. These classes can be used to build other components and functionality as well.

Added to this standard functionality, is Edit-In-Place--that is, you can update the VIEW's primary file by typing directly into the browse list. No separate update procedure is required, and the updates are appropriately autoincremented, referentially constrained, and field validated.

Following are the classes that provide this browse functionality. The classes and their respective roles are:

BrowseClass	Browse list "supervisor" class
StepClass	Scrollbar/Progress Bar base class
LongStepClass	Numeric Runtime distribution
RealStepClass	Numeric Runtime distribution
StringStepClass	Alpha/Lastname distribution
CustomStepClass	Custom distribution
LocatorClass	Locator base class
StepLocatorClass	Step Locator
EntryLocatorClass	Entry Locator
IncrementalLocatorClass	Incremental Locator
FilterLocatorClass	Filter Locator
EditClass	Edit-In-Place

The BrowseClass is fully documented in the remainder of this chapter. Each related class is documented in its own chapter.

## BrowseClass Relationship to Other Application Builder Classes

The BrowseClass is closely integrated with several other ABC Library objects--in particular the WindowManager and ToolbarClass objects. These objects register their presence with each other, set each other's properties, and call each other's methods as needed to accomplish their respective tasks.

The BrowseClass is derived from the ViewManager, plus it relies on many of the other Application Builder Classes (RelationManager, FieldPairsClass, ToolbarClass, PopupClass, etc.) to accomplish its tasks. Therefore, if your program instantiates the BrowseClass, it must also instantiate these other classes. Much of this is automatic when you INCLUDE the BrowseClass header (ABBROWSE.INC) in your program's data section. See the *Conceptual Example*.

## BrowseClass ABC Template Implementation

The ABC Templates automatically include all the classes and generate all the code necessary to support the functionality specified in your application's Browse Procedure and BrowseBox Control templates.

The templates *derive* a class from the BrowseClass for *each* BrowseBox in the application. By default, the derived class is called BRW# where # is the BrowseBox control template instance number. This derived class object supports all the functionality specified in the BrowseBox template.

The derived BrowseClass is local to the procedure, is specific to a single BrowseBox and relies on the global file-specific RelationManager and FileManager objects for the browsed files. The templates provide the derived class so you can customize the BrowseBox behavior on a per-instance basis. See *Control Templates--BrowseBox* for more information.

## BrowseClass Source Files

The BrowseClass source code is installed by default to the Clarion \LIBSRC folder. The specific BrowseClass source code and their respective components are contained in:

ABBROWSE.INC	BrowseClass declarations
ABBROWSE.CLW	BrowseClass method definitions
ABBROWSE.TRN	BrowseClass translation strings

## BrowseClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a BrowseClass object and related objects. The example initializes and page-loads a LIST, then handles a number of associated events, including searching, scrolling, and updating. When they are initialized properly, the BrowseClass and WindowManager objects do most of the work (default event handling) internally.

```

PROGRAM
  INCLUDE('ABWINDOW.INC')           !declare WindowManager class
  INCLUDE('ABBROWSE.INC')          !declare BrowseClass
MAP
END

State      FILE,DRIVER('TOPSPEED'),PRE(ST),THREAD
StateCodeKey  KEY(ST:STATECODE),NOCASE,OPT
Record      RECORD,PRE()
STATECODE   STRING(2)
STATENAME   STRING(20)
            END
            END

StView     VIEW(State)              !declare VIEW for BrowseSt
            END

StateQ     QUEUE                    !declare Q for LIST
ST:STATECODE  LIKE(ST:STATECODE)
ST:STATENAME  LIKE(ST:STATENAME)
ViewPosition STRING(512)
            END

GlobalErrors ErrorClass             !declare GlobalErrors object
Access:State CLASS(FileManager)     !declare Access:State object
Init        PROCEDURE
            END
Relate:State CLASS(RelationManager)  !declare Relate:State object
Init        PROCEDURE
            END
VCRRequest  LONG(0),THREAD

ThisWindow CLASS(WindowManager)     !declare ThisWindow object
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL
            END
BrowseSt    CLASS(BrowseClass)       !declare BrowseSt object
Q           &StateQ
            END

```

```

StLocator  StepLocatorClass          !declare StLocator object
StStep     StepStringClass          !declare StStep object

StWindow WINDOW('Browse States'),AT(, ,123,152),IMM,SYSTEM,GRAY
LIST,AT(8,5,108,124),USE(?StList),IMM,HVSCROLL,FROM(StateQ),|
FORMAT('27L(2)|M-CODE~@s2@80L(2)|M-STATENAME~@s20@')
BUTTON('&Insert'),AT(8,133),USE(?Insert)
BUTTON('&Change'),AT(43,133),USE(?Change),DEFAULT
BUTTON('&Delete'),AT(83,133),USE(?Delete)
END

CODE
ThisWindow.Run()                    !run the window procedure

ThisWindow.Init PROCEDURE()         !initialize things
ReturnValue  BYTE,AUTO
CODE
ReturnValue = PARENT.Init()         !call base class init
IF ReturnValue THEN RETURN ReturnValue.
GlobalErrors.Init                   !initialize GlobalErrors object
Relate:State.Init                   !initialize Relate:State object
SELF.FirstField = ?StList           !set FirstField for ThisWindow
SELF.VCRRequest &= VCRRequest      !VCRRequest not used
SELF.Errors &= GlobalErrors         !set error handler for ThisWindow
Relate:State.Open                   !open State and related files
!Init BrowseSt object by naming its LIST,VIEW,Q,RelationManager & WindowManager
BrowseSt.Init(?StList,StateQ.ViewPosition,StView,StateQ,Relate:State,SELF)
OPEN(StWindow)
SELF.Opened=True
BrowseSt.Q &= StateQ                !reference the browse QUEUE
!initialize the StStep object
StStep.Init(+ScrollSort:AllowAlpha,ScrollBy:Runtime)
!set the browse sort order
BrowseSt.AddSortOrder(StStep,ST:StateCodeKey)
BrowseSt.AddLocator(StLocator)      !plug in the browse locator
StLocator.Init(,ST:STATECODE,1,BrowseSt) !initialize the locator
BrowseSt.AddField(ST:STATECODE,BrowseSt.Q.ST:STATECODE) !set a column to browse
BrowseSt.AddField(ST:STATENAME,BrowseSt.Q.ST:STATENAME) !set a column to browse
BrowseSt.InsertControl=?Insert      !set the control to add records
BrowseSt.ChangeControl=?Change      !set the control to change records
BrowseSt.DeleteControl=?Delete      !set the control to delete records
SELF.SetAlerts()                   !alert any keys for ThisWindow
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()         !shut down things
ReturnValue  BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()         !call base class shut down

```

```
IF ReturnValue THEN RETURN ReturnValue.
Relate:State.Close           !close State and related files
Relate:State.Kill           !shut down Relate:State object
GlobalErrors.Kill          !shut down GlobalErrors object
RETURN ReturnValue

Access:State.Init PROCEDURE
CODE
PARENT.Init(State,GlobalErrors)
SELF.FileNameValue = 'State'
SELF.Buffer &= ST:Record
SELF.AddKey(ST:StateCodeKey, 'ST:StateCodeKey', 0)

Relate:State.Init PROCEDURE
CODE
Access:State.Init
PARENT.Init(Access:State, 1)
```

## BrowseClass Properties

### **BrowseClass Properties**

The BrowseClass inherits all the properties of the ViewManager from which it is derived. See *ViewManager Properties* for more information.

In addition to the inherited properties, the BrowseClass contains the following properties:

### **ActiveInvisible (obscured browse list action)**

#### **ActiveInvisible BYTE**

The **ActiveInvisible** property indicates whether to fill or refill the browse queue when the browse LIST is "invisible" because it is on a non-selected TAB or is otherwise hidden. A value of one (1) refills the queue when the LIST is invisible; a value of zero (0) suppresses the refill.

Setting ActiveInvisible to zero (0) improves performance for procedures with "invisible" browse lists; however, buffer contents for the invisible browse list are not current and should not be relied upon.

Implementation: The ResetQueue method implements the behavior specified by the ActiveInvisible property.

See Also: ResetQueue

## **AllowUnfilled (display filled list)**

### **AllowUnfilled    BYTE**

The **AllowUnfilled** property indicates whether to always display a "full" list, or to allow a partially filled list when the result set "ends" in mid-list. A value of one (1) displays a partially filled list and improves performance by suppressing any additional reads needed to fill the list; a value of zero (0) always displays a filled list.

Setting AllowUnfilled to one (1) improves performance for browse lists, especially for those using SQL data.

Implementation:    The ResetQueue method implements the behavior specified by the AllowUnfilled property.

See Also:            ResetQueue

**ArrowAction (edit-in-place action on arrow key)****ArrowAction**    **BYTE**

The **ArrowAction** property indicates the action to take when the end user presses the up or down arrow key during an edit-in-place process. There are three types of actions that ArrowAction controls:

- what to do with any changes (default, save, abandon, or prompt),
- what mode to use next (continue editing or revert to non-edit mode),
- what column to edit next (current column or first editable column).

The specified actions are implemented by the Ask method. Set the actions by assigning, adding, or subtracting the following EQUATED values to ArrowAction. The following EQUATES are in ABBROWSE.INC:

```

ITEMIZE,PRE(EIPAction)
Default      EQUATE(0)      !save according to the Ask method
Always       EQUATE(1)      !always save the changes
Never        EQUATE(2)      !never save the changes
Prompted     EQUATE(4)      !ask whether to save the changes
Remain       EQUATE(8)      !continue editing
RetainColumn EQUATE(16)     !maintain column position in new row
END

```

Example:

```

BRW1.ArrowAction = EIPAction:Prompted      !ask to save changes
BRW1.ArrowAction = EIPAction:Prompted+EIPAction:Remain !ask to save, keep editing
                                                    !1st editable column
BRW1.ArrowAction = EIPAction:Remain+EIPAction:RetainColumn!default save, keep editing
                                                    !same column

```

See Also:        Ask

## **AskProcedure (update procedure)**

### **AskProcedure USHORT**

The **AskProcedure** property identifies the procedure to update a browse item. A value of zero (0) uses the BrowseClass object's own AskRecord method to do updates. Any other value uses a separate procedure registered with the WindowManager object.

Implementation: Typically, the WindowManager object (Init method) sets the value of the AskProcedure property when a separate update procedure is needed. The Ask method passes the AskProcedure value to the WindowManager.Run method to indicate which procedure to execute.

See Also: Ask, AskRecord, WindowManager.Run

## **ChangeControl (browse change/update control number)**

### **ChangeControl SIGNED**

The **ChangeControl** property contains the number of the browse's change/update control. This is typically the value of the Change BUTTON's field equate. The BrowseClass methods use this value to enable and disable the control when appropriate, to post events to the control, to map change behavior to corresponding popup menu choices, etc.

Implementation: The BrowseClass.Init method does not initialize the ChangeControl property. You should initialize the ChangeControl property after the BrowseClass.Init method is called. See the *Conceptual Example*.

See Also: UpdateToolBarButtons

**CurrentChoice (current LIST control entry number)****CurrentChoice** LONG, PROTECTED

The **CurrentChoice** property represents the entry number of the highlighted record in a LIST control.

Implementation: The **CurrentChoice** property is updated as the scroll bar for the LIST control is moved within the Listbox.

**DeleteAction (edit-in-place action on delete key)****DeleteAction** BYTE

The **DeleteAction** property indicates the action to take when the end user presses the DELETE key during an edit-in-place process. **DeleteAction** controls what mode to use next (continue editing or revert to non-edit mode).

The specified actions are implemented by the Ask method. Set the actions by assigning, adding, or subtracting the following EQUATED values to ArrowAction. The following EQUATES are in ABBROWSE.INC:

```

ITEMIZE,PRE(EIPAction)
Default   EQUATE(0)       !save according to the Ask method
Always    EQUATE(1)       !always save the changes
Never     EQUATE(2)       !never save the changes
Prompted  EQUATE(4)       !ask whether to save the changes
Remain    EQUATE(8)       !continue editing

END

```

**DeleteControl (browse delete control number)****DeleteControl** SIGNED

The **DeleteControl** property contains the number of the browse's delete control. This is typically the value of the Delete BUTTON's field equate. The BrowseClass methods use this value to enable and disable the control when appropriate, to post events to the control, to map delete behavior to corresponding popup menu choices, etc.

Implementation: The BrowseClass.Init method does not initialize the DeleteControl property. You should initialize the DeleteControl property after the BrowseClass.Init method is called. See the *Conceptual Example*.

See Also: UpdateToolBarButtons

## EditList (list of edit-in-place controls)

### **EditList &BrowseEditQueue, PROTECTED**

The **EditList** property is a reference to a structure containing a list of edit-in-place classes for use with specific browse list columns.

The AddEditControl method adds new edit-in-place classes and their associated browse list columns to the EditList property.

Implementation: You do not need to initialize this property to implement the default edit-in-place controls. The EditList property supports custom edit-in-place controls.

The EditList property is a reference to a QUEUE declared in ABBROWSE.INC as follows:

```
BrowseEditQueue QUEUE,TYPE
Field           UNSIGNED
FreeUp         BYTE
Control        &EditClass
                END
```

See Also: AddEditControl

## EIP (edit-in-place manager)

### **EIP &BrowseEIPManager**

The **EIP** property is a reference to the BrowseEIPManager class used by this BrowseClass object.

See Also: Init

## EnterAction (edit-in-place action on enter key)

**EnterAction**    **BYTE**

The **EnterAction** property indicates the action to take when the end user presses the ENTER key during an edit-in-place process. There are two types of actions that EnterAction controls:

- what to do with any changes (default, save, abandon, or prompt),
- what mode to use next (continue editing or revert to non-edit mode).

The specified actions are implemented by the Ask method. Set the actions by assigning, adding, or subtracting the following EQUATEd values to ArrowAction. The following EQUATEs are in ABBROWSE.INC:

```

ITEMIZE,PRE(EIPAction)
Default  EQUATE(0)      !save according to the Ask method
Always   EQUATE(1)      !always save the changes
Never    EQUATE(2)      !never save the changes
Prompted EQUATE(4)      !ask whether to save the changes
Remain   EQUATE(8)      !continue editing
END

```

Example:

```

BRW1.EnterAction = EIPAction:Prompted      !ask to save changes
BRW1.EnterAction = EIPAction:Prompted+EIPAction:Remain !ask to save, keep
editing

```

See Also:        Ask

## FocusLossAction (edit-in-place action on lose focus)

**FocusLossAction**      **BYTE**

The **FocusLossAction** property indicates the action to take with regard to pending changes when the edit control loses focus during an edit-in-place process.

The specified action is implemented by the Ask method. Set the action by assigning, adding, or subtracting one of the following EQUATEd values to FocusLossAction. The following EQUATEs are in ABBROWSE.INC:

```
ITEMIZE,PRE(EIPAction)
Default   EQUATE(0)      !save according to the Ask method
Always    EQUATE(1)      !always save the changes
Never     EQUATE(2)      !never save the changes
Prompted  EQUATE(4)      !ask whether to save the changes
END
```

Example:

```
BRW1.FocusLossAction = EIPAction:Prompted      !ask to save changes
```

See Also:      Ask

### **HasThumb (vertical scroll bar flag)**

**HasThumb**      **BYTE**

The **HasThumb** property indicates whether BrowseClass object's LIST control has a vertical scroll bar. A value of one (1) indicates a scroll bar; a value of zero (0) indicates no scroll bar.

Implementation:      The SetAlerts method sets the value of the HasThumb property. The UpdateThumb method uses the HasThumb property to implement correct thumb and scroll bar behavior.

See Also:              ListControl, SetAlerts, UpdateThumb

### **HideSelect (hide select button)**

**HideSelect**      **BYTE**

The **HideSelect** property indicates whether to HIDE the Select button (as indicated by the SelectControl property) when the browse is called for update purposes (as indicated by the Selecting property). A value of one (1) hides the select button; a value of zero (0) always displays the select button.

Implementation:      The ResetQueue method implements the behavior specified by the HideSelect property.

See Also:              ResetQueue, SelectControl, Selecting

### **ILC(reference to IListControl interface)**

**ILC**                      **&IListControl**

The **ILC** property is a reference to the IListControl interface which is passed to the Init method. The IListControl interface is used to implement standard behaviors for a listbox. See the IListControl interface section for more information.

Implementation:      The Init method creates an instance of ILC for the list control. The Kill method disposes of that interface instance.

See Also:              BrowseClass.Init, BrowseClass.Kill

## InsertControl (browse insert control number)

**InsertControl**    **SIGNED**

The **InsertControl** property contains the number of the browse's insert control. This is typically the value of the Insert BUTTON's field equate. The BrowseClass methods use this value to enable and disable the control when appropriate, to post events to the control, to map Insert behavior to corresponding popup menu choices, etc.

Implementation:    The BrowseClass.Init method does not initialize the InsertControl property. You should initialize the InsertControl property after the BrowseClass.Init method is called. See the *Conceptual Example*.

See Also:    UpdateToolBarButtons

## ListQueue (browse data queue reference)

**ListQueue**    **&BrowseQueue, PROTECTED**

The **ListQueue** property is a reference to a structure that is the source of the data elements displayed in the browse LIST.

The BrowseClass.Init method initializes the ListQueue property. See the *Conceptual Example*.

See Also:          Init

## Loaded (browse queue loaded flag)

**Loaded**    **BYTE, PROTECTED**

The **Loaded** property contains a value that indicates whether or not the BrowseClass object has tried to load the browse list queue. The BrowseClass uses this property to ensure the browse queue gets loaded and to avoid redundant reloads.

## **Popup (browse popup menu reference)**

### **Popup &PopupClass**

The **Popup** property is a reference to the PopupClass class used by this BrowseClass object.

Implementation: Because it directly references the PopupClass, the BrowseClass header INCLUDEs the PopupClass header. That is, the BrowseClass's implementation of the PopupClass is automatic. You need take no action.

The BrowseClass.Init method instantiates the PopupClass object referenced by the Popup property. See the *Conceptual Example*.

See Also: Init

## **PrevChoice (prior LIST control entry number)**

### **PrevChoice LONG, PROTECTED**

The **PrevChoice** property represents the entry number of the previously selected record in a LIST control.

Implementation: The PrevChoice property is updated as the scroll bar for the LIST control is moved within the Listbox.

## **PrintControl (browse print control number)**

### **PrintControl SIGNED**

The **PrintControl** property contains the number of the browse's print control. This is typically the value of the Print BUTTON's field equate. The BrowseClass methods use this value to enable and disable the control when appropriate, to post events to the control, to map Print behavior to corresponding popup menu choices, etc.

Implementation: The BrowseClass.Init method does not initialize the PrintControl property. You should initialize the PrintControl property after the BrowseClass.Init method is called. See the *Conceptual Example*.

See Also: UpdateToolBarButtons

## **PrintProcedure (print procedure)**

### **PrintProcedure USHORT**

The **PrintProcedure** property identifies the procedure to execute when the Browse Print action is called. The procedure name is registered with the WindowManager object.

Implementation: Typically, the WindowManager object (Init method) sets the value of the PrintProcedure property when a Browse Print is called. The Ask method passes the AskProcedure value to the WindowManager.Run method to indicate which procedure to execute.

See Also: Ask  
AskRecord  
WindowManager.Run

## **Processors (reference to ProcessorQueue)**

### **ListQueue &QUEUE, PROTECTED**

The **Processors** property is a reference to the ProcessorQueue queue which contains references to the RecordProcessor interface and manages several processes.

See Also: BrowseClass.Init, BrowseClass.Kill

## **Query (reference to QueryClass)**

### **Query &QueryClass**

The **Query** property is a reference to the QueryClass which manages Query-by-Example processes for the BrowseClass.

See Also: BrowseClass.Init  
BrowseClass.Kill

## **QueryControl (browse query control number)**

**QueryControl**    **SIGNED**

The **QueryControl** property contains the number of the browse's query (QBE) control. This is typically the value of the QBE BUTTON's field equate. The BrowseClass methods use this value to enable and disable the control when appropriate, to post events to the control, to map QBE behavior to corresponding popup menu choices, etc.

Implementation:    The BrowseClass.Init method does not initialize the QueryControl property. You should initialize the QueryControl property after the BrowseClass.Init method is called. See the *Conceptual Example*.

See Also: UpdateToolBarButtons

## **QueryShared (share query with multiple sorts)**

**QueryShared**    **BYTE**

The **QueryShared** property contains a value that tells the BrowseClass whether or not to keep queries active when changing different sort orders. An example of this is when switching from one tab control to another in the active browse.

A value of zero (0) disables sharing (the query is only active for the tab in which it is set); a value of 1 enables sharing.

Implementation:    The QueryShared property is set by the Query Button template interface when the **Auto-share between tabs** option is checked.

## QuickScan (buffered reads flag)

**QuickScan**     **BYTE**

The **QuickScan** property contains a value that tells the BrowseClass whether or not to quickscan when page-loading the browse list queue. Quick scanning only affects file systems that use multi-record buffers. See *Database Drivers* for more information.

A value of zero (0) disables quick scanning; a non-zero value enables quick scanning. Quick scanning is the normal way to read records for browsing. However, rereading the buffer may provide slightly improved data integrity in some multi-user circumstances at the cost of substantially slower reads.

Implementation:     TheBrowseClass.Fetch method implements the faster reads only during the page-loading process, and only if the QuickScan property is not zero. The BrowseClass.Fetch method SENDs the 'QUICKSCAN=ON' driver string to the applicable files' database drivers with the RelationManager.SetQuickScan method.

**Note:**     The RelationManager.SetQuickScan method does *not* set the BrowseClass.QuickScan property. However if you set the BrowseClass.QuickScan property to 1, the BrowseClass uses the RelationManager.SetQuickScan method to SEND the QUICKSCAN driver string to the appropriate files.

See Also:             Fetch, RelationManager.SetQuickScan.

### **RetainRow (highlight bar refresh behavior)**

**RetainRow**     **BYTE**

The **RetainRow** property indicates whether the BrowseClass object tries to maintain the highlight bar in the same list row following a change in sort order, an update, or other browse refresh action. A value of one (1) maintains the current highlight bar row; a value of zero (0) lets the highlight bar move to the first row.

Setting RetainRow to one (1) can cause a performance penalty in applications using TopSpeed's pre-Accelerator ODBC driver.

Implementation:     The Init method sets the RetainRow property to one (1). The ResetQueue method implements the behavior specified by the RetainRow property.

See Also:             Init, ResetQueue

### **SelectControl (browse select control number)**

**SelectControl**     **SIGNED**

The **SelectControl** property contains the number of the browse's select control. This is typically the value of the Select BUTTON's field equate. The BrowseClass methods use this value to enable and disable the control when appropriate, to post events to the control, to map Select behavior to corresponding popup menu choices, etc.

Implementation:     The BrowseClass.Init method does not initialize the SelectControl property. You should initialize the SelectControl property after the BrowseClass.Init method is called. See the *Conceptual Example*.

See Also:     UpdateToolbarButtons

**Selecting (select mode only flag)****Selecting**      **BYTE**

The **Selecting** property indicates whether the BrowseClass object selects a browse item or updates browse items. A value of zero (0) sets update mode; a value of one (1) sets select only mode.

The HideSelect property is only effective when the Selecting property indicates update mode.

Implementation:      In select mode, a DOUBLE-CLICK or ENTER selects the item; otherwise, a DOUBLE-CLICK or ENTER updates the item.

See Also:              HideSelect

**SelectWholeRecord (select entire record flag)****SelectWholeRecord**      **BYTE**

The **SelectWholeRecord** property indicates whether an UpdateViewRecord should be called in the TakeEvent method. A value of one (1) will get the whole record from the view; a value of zero (0), the default, gets the record from the buffer.

See Also:              UpdateViewRecord, TakeEvent

## Sort (browse sort information)

### Sort    &BrowseSortOrder

The **Sort** property is a reference to a structure containing all the sort information for this BrowseClass object. The BrowseClass methods use this property to implement multiple sort orders, range limits, filters, and locators for a single browse list.

Implementation:    The BrowseClass.Sort property mimics or shadows the inherited ViewManager.Order property. The Sort property is a reference to a QUEUE declared in ABBROWSE.INC as follows:

```

BrowseSortOrder  QUEUE(SortOrder),TYPE  !browse sort information
Locator          &LocatorControl        !locator for this sort order
Resets          &FieldPairsClass        !reset fields for this sort order
Thumb           &ThumbClass             !ThumbClass for this sort order
                END

```

Notice this BrowseSortOrder queue contains all the fields in the SortOrder queue declared in ABFILE.INC as follows:

```

SortOrder        QUEUE,TYPE             !VIEW sort information
Filter           &FilterQueue           !ANDed filter expressions
FreeElement      ANY                     !The Free key element
LimitType        BYTE                    !Range limit type flag
MainKey          &KEY                    !The KEY
Order            &STRING                 !ORDER expression (equal to KEY)
RangeList        &FieldPairsClass       !fields in the range limit
                END

```

And the SortOrder queue contains a reference to the FilterQueue declared in ABFILE.INC as follows:

```

FilterQueue      QUEUE,TYPE             !VIEW filter information
ID               STRING(30)             !filter ID
Filter           &STRING                 !filter expression
                END

```

So, the BrowseSortOrder queue is, among other things, a queue of queues.

The AddSortOrder method defines sort orders for the browse. The SetSort method applies or activates a sort order for the browse. Only one sort order is active at a time.

See Also:            AddSortOrder, SetSort

## StartAtCurrent (initial browse position)

### StartAtCurrent BYTE

The **StartAtCurrent** property indicates whether the BrowseClass object initially positions to the first item in the sort order or positions to the item specified by the contents of the Browse's view buffer. A value of zero (0 or False) positions to the first item; a value of one (1 or True) positions to the item specified by the contents of the view buffer.

Implementation: The SetSort method implements the StartAtCurrent initial position. The SetSort method positions the browse list based on the contents of the fields in the active sort order, including the free element field.

Example:

```
BRW1.StartAtCurrent = True
ST:StateCode = 'K'           !set key component value
BrowseSt.Init(?StList,StateQ.ViewPosition,StView,StateQ,Relate:State,SELF)
```

See Also: SetSort

## TabAction (edit-in-place action on tab key)

**TabAction**      **BYTE**

The **TabAction** property indicates the action to take when the end user presses the TAB key during an edit-in-place process. There are two types of actions that TabAction controls:

- what to do with pending changes (default, save, abandon, or prompt),
- what mode to use next (continue editing or revert to non-edit mode).

The specified actions are implemented by the Ask method. Set the actions by assigning, adding, or subtracting the following EQUATEd values to TabAction. The following EQUATEs are in ABBROWSE.INC:

```

ITEMIZE,PRE(EIPAction)
Default  EQUATE(0)      !save according to the Ask method
Always   EQUATE(1)      !always save the changes
Never    EQUATE(2)      !never save the changes
Prompted EQUATE(4)      !ask whether to save the changes
Remain   EQUATE(8)      !continue editing
END

```

Example:

```

BRW1.TabAction = EIPAction:Prompted      !ask to save changes
BRW1.TabAction = EIPAction:Prompted+EIPAction:Remain !ask to save, keep
editing

```

See Also:      Ask

## Toolbar (browse Toolbar object)

### Toolbar &ToolbarClass

The **Toolbar** property is a reference to the ToolbarClass for this BrowseClass object. The ToolbarClass object collects toolbar events and passes them on to the active ToolbarTarget object for processing.

The AddToolbarTarget method registers a ToolbarTarget, such as a ToolbarListBoxClass object, as a potential target of a ToolbarClass object.

The ToolbarClass.SetTarget method sets the active target for a ToolbarClass object.

Implementation: The ToolbarClass object for a browse is the object that detects toolbar events, such as scroll down or page down, and passes them on to the *active* ToolbarListBoxClass (ToolbarTarget) object. In the standard template implementation, there is a single global toolbar, and a ToolbarClass object per procedure that may drive several different browses and forms, each of which is a ToolbarTarget. Only one ToolbarTarget is active at a time.

See Also: ToolbarItem, AddToolbarTarget, ToolbarClass.SetTarget

## **ToolbarItem (browse ToolbarTarget object)**

### **ToolbarItem    &ToolbarListBoxClass**

The **ToolbarItem** property is a reference to the `ToolbarListBoxClass` for this `BrowseClass` object. The `ToolbarListBoxClass` (`ToolbarTarget`) object receives toolbar events (from a `ToolbarClass` object) and processes them.

The `AddToolbarTarget` method registers a `ToolbarTarget`, such as a `ToolbarListBoxClass` object, as a potential target of a `ToolbarClass` object.

The `ToolbarClass.SetTarget` method sets the active target for a `ToolbarClass` object.

Implementation:    The `ToolbarClass` object for a browse is the object that detects toolbar events, such as scroll down or page down, and passes them on to the *active* `ToolbarListBoxClass` (`ToolbarTarget`) object. In the standard template implementation, there is a single global toolbar, and a `ToolbarClass` object per procedure that may drive several different browses and forms, each of which is a `ToolbarTarget`. Only one `ToolbarTarget` is active at a time.

See Also:            `Toolbar`, `AddToolbarTarget`, `ToolbarClass.SetTarget`

## **ToolControl (browse toolbox control number)**

**ToolControl**      **SIGNED**

The **ToolControl** property contains the number of the browse's toolbox control. This is typically the value of the Toolbox BUTTON's field equate. The BrowseClass methods use this value to enable and disable the control when appropriate, to post events to the control, to map Toolbox behavior to corresponding popup menu choices, etc.

Implementation:      The BrowseClass.Init method does not initialize the ToolControl property. You should initialize the ToolControl property after the BrowseClass.Init method is called. See the *Conceptual Example*.

See Also: UpdateToolBarButtons

## **ViewControl (view button)**

**ViewControl**      **SIGNED**

The **ViewControl** property contains the number (field equate) of the browse's view button control.

## **Window (WindowManager object)**

**Window**              **&WindowManager**

The **Window** property is a reference to the WindowManager object for this BrowseClass object. The WindowManager object forwards events to the active BrowseClass object for processing.

The WindowManager.AddItem method registers the BrowseClass object with the WindowManager object, so the WindowManager object can forward events.

The Init method sets the value of the Window property.

Implementation:      The WindowManager object calls the BrowseClass.TakeEvent method so the BrowseClass object can handle the events as needed.

See Also:              Init, WindowManager.AddItem

## BrowseClass Methods

### BrowseClass Methods

The BrowseClass inherits all the methods of the ViewManager from which it is derived. See *ViewManager Methods* for more information.

### BrowseClass Functional Organization--Expected Use

As an aid to understanding the BrowseClass, it is useful to organize its various methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the BrowseClass methods.

#### Non-Virtual Methods

---

The non-virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the BrowseClass object
AddEditControl	specify custom edit-in-place for a browse field
AddField	identify corresponding FILE and QUEUE fields
AddLocator	associate a locator with its sort order
AddResetField	specify a field that refreshes the browse list
AddSortOrder	add a sort order to the browse list
AddToolbarTarget	associate the browse list with a toolbar object
SetAlertsv	alert keys for list, locator, and edit controls
Kill v	shut down the BrowseClass object

##### **Mainstream Use:**

Nextv	get the next view record in sequence
Previousv	get the previous view record in sequence
Ask	update the selected item
TakeEventv	process the current ACCEPT loop event
TakeNewSelectionv	process a new browse list item selection

v These methods are also Virtual.

##### **Occasional Use:**

ApplyRange	refresh browse list to specified range limit
AskRecord	edit-in-place the selected item
PostNewSelection	post an EVENT:NewSelection to the browse list
Records	return the number of records in the browse list
ResetResets	snapshot the current value of the Reset fields
ResetThumbLimits	reset thumb limits to match the result set
TakeAcceptedLocator	apply an entered locator value

UpdateResets	copy reset fields to file buffer
UpdateThumb	position the scrollbar thumb
UpdateThumbFixed	position the scrollbar fixed thumb
UpdateWindow v	apply pending scroll, locator, range, etc.

## Virtual Methods

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

ApplyRange	conditionally range limit and filter the records
Fetch	loads a page of items into the browse list
Kill	shut down the BrowseClass object
Next	get the next record from the browse view
Previous	get the previous record from the browse view
Reset	reset the view position
ResetFromAsk	reset browse object after update
ResetFromBuffer	refill queue based on current record buffer
ResetFromFile	refill queue based on FILE POSITION
ResetFromView	reset browse object to its result set
ResetQueue	fill or refill the browse queue
ScrollEnd	scroll to the first or last item
ScrollOne	scroll up or down one item
ScrollPage	scroll up or down one page of items
SetAlerts	alert keys for list, locator, and edit controls
SetQueueRecord	copy data from file buffer to queue buffer
SetSort	apply sort order to browse
ResetSort	apply sort order to browse
TakeKey	process an alerted keystroke
TakeEvent	process the current ACCEPT loop event
TakeNewSelection	process a new browse list item selection
TakeScroll	process a scroll event
TakeVCRScroll	process a VCR scroll event
UpdateBuffer	copy data from queue buffer to file buffer
UpdateViewRecord	copy selected item to corresponding file buffers
UpdateWindow	apply pending scroll, locator, range, etc.

### Tip

**Use ResetSort followed by UpdateWindow to refresh and redisplay your ABC BrowseBoxes. Or, use the WindowManager.Reset method.**

**AddEditControl (specify custom edit-in-place class)**

**AddEditControl**( [*editclass*], *column* [, *autofree*] )

---

<b>AddEditControl</b>	Specifies a custom edit-in-place class for a browse field.
<i>editclass</i>	The label of the EditClass. If omitted, the specified <i>column</i> is not editable.
<i>column</i>	An integer constant, variable, EQUATE, or expression that indicates the browse list column to edit with the specified <i>editclass</i> object. A value of one (1) indicates the first column; a two (2) indicates the second column, etc.
<i>autofree</i>	A numeric constant, variable, EQUATE, or expression that indicates whether the BrowseClass.Kill method DISPOSEs of the <i>editclass</i> object. A zero (0) value leaves the object intact. A non-zero value DISPOSEs the object. If omitted, <i>autofree</i> defaults to zero (0).

The **AddEditControl** method specifies the *editclass* that defines the edit-in-place control for the browse *column*. Use *autofree* with caution; you should only DISPOSE of memory allocated with a NEW statement. See the *Language Reference* for more information on NEW and DISPOSE.

Implementation: You do not need to call this method to use the default *editclass*. If you do not call the AddEditControl method for a browse list column, the BrowseClass automatically instantiates the EditClass declared in ABBROWSE.INC for that column.

The *autofree* parameter defaults to zero (0). The BrowseClass.Kill method DISPOSEs the editclass objects only if *autofree* contains a non-zero value.

The BrowseClass.Ask method instantiates the *editclass* objects as needed, then creates and deletes the edit-in-place control upon the end user's insert or change request.

Example:

```

INCLUDE('ABBROWSE.INC')           !declare browse & related classes
INCLUDE('MYCOMBO.INC')           !declare custom Edit-in-place control class
!other browse class declarations
CODE
MyBrowse.AddEditControl(,1)       !column 1 not editable
MyBrowse.AddEditControl(ComboClass,2) !edit column 2 with combo control

```

See Also: Ask

**AddField (specify a FILE/QUEUE field pair)**

**AddField**( *filefield*, *queuefield* )

---

<b>AddField</b>	Identifies the corresponding FILE and QUEUE fields for a browse list column.
<i>filefield</i>	The fully qualified label of the FILE field or memory variable. The <i>filefield</i> is the original source of the browse LIST's data.
<i>queuefield</i>	The fully qualified label of the corresponding QUEUE field. The <i>queuefield</i> is loaded from the <i>filefield</i> , and is the immediate source of the browse LIST's data.

The **AddField** method identifies the corresponding FILE and QUEUE fields for a browse list column. You must call AddField for each column displayed in the browse list.

You may also use the AddField method to pair memory variables with QUEUE fields by specifying a variable label as the *filefield* parameter.

Implementation: For browses with edit-in-place, you must add fields (call the AddField method) in the same sequence that you declare the browse QUEUE fields.

Example:

```

INCLUDE('ABBROWSE.INC')                !declare browse & related classes
States FILE,DRIVER('TOPSPEED'),PRE(StFile) !declare States file
ByCode KEY(StFile:Code),NOCASE,OPT
Record RECORD,PRE()
Code STRING(2)
Name STRING(20)
END
END
StQType QUEUE,TYPE                    !declare the St QUEUE type
Code LIKE(StFile:Code)
Name LIKE(StFile:Name)
Position STRING(512)
END
BrowseStClass CLASS(BrowseClass),TYPE !declare the BrowseSt CLASS
Q &StQType
END
StQ StQType                            !declare the (real) StQ QUEUE
BrowseSt BrowseStClass                 !declare the BrowseSt object
CODE
BrowseSt.AddField(StFile:Code,BrowseSt.Q.Code) !pair up fields in
BrowseSt.AddField(StFile:Name,BrowseSt.Q.Name) !FILE & QUEUE

```

## AddItem(program the BrowseClass object)

### AddItem(*RecordProcessor*)

---

**AddItem** Adds specific functionality to the BrowseClass object.

*RecordProcessor* The label of a RecordProcessor interface.

The **AddItem** method registers an ABC Library interface with the BrowseClass object and adds the interface's specific functionality to the BrowseClass.

See Also: BrowseClass.Kill

## AddLocator (specify a locator)

### AddLocator( *locator* )

---

**AddLocator** Specifies a locator object for a specific sort order.

*locator* The label of the locator object.

The **AddLocator** method specifies a locator object for the sort order defined by the preceding call to the AddSortOrder or SetSort method. Typically, you call the AddLocator method immediately after the AddSortOrder method.

Implementation: The specified *locator* is sort order specific--it is enabled only when the associated sort order is active. The SetSort method applies or activates a sort order for the browse. Only one sort order is active at a time.

Example:

```

BrowseSt.AddSortOrder(BrowseSt:Step,StFile:ByCode) !add sort order and
BrowseSt.AddLocator(BrowseSt:Locator) !associated locator
BrowseSt:Locator.Init(?Loc,StFile:StCode,1,BrowseSt)!init locator object

```

See Also: AddSortOrder, LocatorClass, SetSort

## AddResetField (set a field to monitor for changes)

**AddResetField**( *resetfield* )

---

**AddResetField** Specifies a field that resets the browse list when the contents of the field changes.

*resetfield*        The label of the field to monitor for changes.

For the active sort order (defined by the preceding call to the AddSortOrder or SetSort method), the **AddResetField** method specifies a field that the browse object monitors for changes, then, when the contents of the field changes, refreshes the browse list. Typically, you call the AddResetField method immediately after the AddSortOrder method.

You may call AddResetField multiple times to establish multiple reset fields for a sort order.

Implementation:    The specified *resetfield* is sort order specific--it is enabled only when the associated sort order is active. The SetSort method sets the active sort order for the browse. SetSort also calls ApplyRange to monitor the reset fields for changes and SetSort resets the browse when a change occurs.

The WindowManager.Reset method also initiates an evaluation of the reset fields and a subsequent browse reset if needed for any browse objects registered with the WindowManager.

Example:

```
BrowseSt.AddSortOrder(BrowseSt:Step,StFile:ByCode) !add sort order  
BrowseSt.AddLocator(BrowseSt:Locator) !and associated locator  
BrowseSt.AddResetField(Local:StFilter) !and associated reset field
```

See Also:            AddSortOrder, SetSort, WindowManager.Reset

**AddSortOrder (specify a browse sort order)**

**AddSortOrder**( [*thumbstep*] [, *key*]), **PROC**

---

**AddSortOrder** Specifies an additional sort order for the browse list.

*thumbstep*      The label of the StepClass object that controls vertical scroll bar and thumb behavior. If omitted, the vertical scroll bar exhibits Fixed Thumb behavior. See *Control Templates--BrowseBox* for more information on thumb behavior.

*key*              The label of the KEY to sort by. If omitted, the browse list is not sorted--the items appear in physical order, or in the order specified by the inherited AppendOrder method.

The **AddSortOrder** method specifies an additional sort order for the browse list and returns the sort order's sequence number for use with the SetSort method. You must call the AddSortOrder method for each different sort order applied to the browse list.

The AddLocator method adds an associated locator for the sort order defined by the preceding call to AddSortOrder.

The AddResetField method adds an associated reset field for the sort order defined by the preceding call to AddSortOrder. You may add multiple reset fields for each sort order with multiple calls to AddResetField.

The inherited AddRange method adds an associated range limit for the sort order defined by the preceding call to AddSortOrder.

Implementation:    The AddSortOrder method adds an entry at a time to the Sort property.

Return Data Type:    **BYTE**

Example:

```
BrowseSt.AddSortOrder(BrowseSt:Step,StFile:ByCode) !add sort order
BrowseSt.AddLocator(BrowseSt:Locator) !and associated locator
BrowseSt.AddResetField(Local:StFilter) !and associated reset field
```

See Also:            AddLocator, AddResetField, Sort, StepClass, SetSort, ViewManager.AddRange, ViewManager.AppendOrder

**AddToolbarTarget (set the browse toolbar)**

**AddToolbarTarget**( *toolbar* )

---

**AddToolbarTarget** Registers the browse list as a potential target of the specified *toolbar*.

*toolbar* The label of the ToolbarClass object that directs toolbar events to this BrowseClass object.

The **AddToolbarTarget** method registers the BrowseClass object as a potential target of the specified *toolbar*.

The ToolbarClass.SetTarget method sets the active target for a ToolbarClass object.

Implementation: The Toolbar object for a browse is the object that detects toolbar events, such as scroll down or page down, and passes them on to the *active* ToolbarTarget object. In the standard template implementation, there is a single global toolbar, and a Toolbar object per procedure that may drive several different browses and forms, each of which is a ToolbarTarget. Only one ToolbarTarget is active at a time.

Example:

```

BrowseSt.AddToolbarTarget(Browse:Toolbar) !tie BrowseSt object to Toolbar object
BrowseZIP.AddToolbarTarget(Browse:Toolbar) !tie BrowseZIP object to Toolbar object
!program code
Browse:Toolbar.SetTarget(?StList) !state list is current toolbar target
!program code
Browse:Toolbar.SetTarget(?ZIPList) !ZIP list is current toolbar target

```

See Also: Toolbar, ToolbarItem, ToolbarClass.SetTarget

## **ApplyRange (refresh browse based on resets and range limits)**

### **ApplyRange, VIRTUAL, PROC**

The **ApplyRange** method checks the current status of reset fields and range limits and refreshes the browse list if necessary. Then it returns a value indicating whether a screen redraw is required.

The inherited **AddRange** method adds an associated range limit for each sort order. The **AddResetField** method establishes reset fields for each browse sort order.

Implementation: The **ApplyRange** method returns one (1) if a screen redraw is required or zero (0) if no redraw is required.

Return Data Type: **BYTE**

Example:

```
IF BrowseSt.ApplyRange()      !refresh browse queue if things changed  
  DISPLAY(?StList)          !redraw LIST if queue refreshed  
END
```

See Also: **AddResetField, ViewManager.AddRange**

**Ask (update selected browse item)**

**Ask**( *request* ), VIRTUAL, PROC

---

**Ask** Updates the selected browse record.

*request* A numeric constant, variable, EQUATE, or expression that indicates the requested update action. Valid actions are Insert, Change, and Delete.

The **Ask** method updates the selected browse record and returns a value indicating whether the requested update was completed or cancelled.

Implementation: Depending on the value of the AskProcedure property, the Ask method either calls the WindowManager.Run method to execute a specific update procedure, or it calls the AskRecord method to do an edit-in-place update.

The TakeEvent method calls the Ask method. The Ask method assumes the UpdateViewRecord method has been called to ensure correct record buffer contents.

Return value EQUATEs are declared in \LIBSRC\TPLEQU.CLW:

```
RequestCompleted    EQUATE (1)    !Update Completed
RequestCancelled    EQUATE (2)    !Update Aborted
```

EQUATEs for *request* are declared in \LIBSRC\TPLEQU.CLW:

```
InsertRecord        EQUATE (1)    !Add a record to table
ChangeRecord        EQUATE (2)    !Change the current record
DeleteRecord        EQUATE (3)    !Delete the current record
```

Return Data Type: BYTE

Example:

```
BrowseClass.TakeEvent PROCEDURE
!procedure data
  CODE
  !procedure code
  CASE ACCEPTED()
  OF SELF.DeleteControl
    SELF.Window.Update()
    SELF.Ask(DeleteRecord)          !delete a browse item
  OF SELF.ChangeControl
    SELF.Window.Update()
    SELF.Ask(ChangeRecord)         !change a browse item
  OF SELF.InsertControl
    SELF.Window.Update()
    SELF.Ask(InsertRecord)         !insert a browse item
```

```
OF SELF.SelectControl
  SELF.Window.Response = RequestCompleted
  POST(EVENT:CloseWindow)
ELSE
  SELF.TakeAcceptedLocator
END
```

See Also:           AskProcedure, AskRecord, TakeEvent

**AskRecord (edit-in-place selected browse item)****AskRecord( *request* ), VIRTUAL, PROC**


---

**AskRecord** Does edit-in-place update of the selected browse record.

*request* A numeric constant, variable, EQUATE, or expression that indicates the requested edit-in-place action. Valid edit-in-place actions are Insert, Change, and Delete.

The **AskRecord** method does edit-in-place updates for the selected browse row and column, then returns a value indicating whether the requested edit was completed or cancelled. The **AddEditControl** method specifies a custom **EditClass** for a specific browse column.

Implementation: The **AskRecord** method assumes the **UpdateViewRecord** method has been called to ensure correct record buffer contents. **AskRecord** should be followed by the **ResetFromAsk** method. The **Ask** method calls the **AskRecord** method.

Return value EQUATEs are declared in \LIBSRC\TPLEQU.CLW:

```
RequestCompleted EQUATE (1) !Update Completed
RequestCancelled EQUATE (2) !Update Aborted
```

EQUATEs for *request* are declared in \LIBSRC\TPLEQU.CLW:

```
InsertRecord EQUATE (1) !Add a record to table
ChangeRecord EQUATE (2) !Change the current record
DeleteRecord EQUATE (3) !Delete the current record
```

Return Data Type: BYTE

Example:

```
BrowseClass.Ask PROCEDURE(BYTE Req)
Response BYTE
CODE
LOOP
    SELF.Window.VCRRequest = VCR:None
    IF Req=InsertRecord THEN
        SELF.PrimeRecord
    END
    IF SELF.AskProcedure
        Response = SELF.Window.Run(SELF.AskProcedure,Req) !do edit-in-place update
        SELF.ResetFromAsk(Req,Response)
    ELSE
        Response = SELF.AskRecord(Req)
    END
UNTIL SELF.Window.VCRRequest = VCR:None
RETURN Response
```

See Also: **AddEditControl**, **Ask**, **ResetFromAsk**

**Fetch (get a page of browse items)****Fetch( *direction* ), VIRTUAL, PROTECTED**


---

**Fetch** Loads a page of items into the browse list queue.

*direction* A numeric constant, variable, EQUATE, or expression that indicates whether to get the next set of items or the previous set of items.

The **Fetch** method loads the next or previous page of items into the browse list queue.

Implementation: Fetch is called by the ResetQueue, ScrollOne, ScrollPage, and ScrollEnd methods. A page of items is as many items as fits in the LIST control.

BrowseClass.Fetch *direction* value EQUATEs are declared in ABBROWSE.INC as follows:

```
FillBackward EQUATE(1)
FillForward EQUATE(2)
```

Example:

```
ScrollOne PROCEDURE(SIGNED Event)
CODE
  IF Event = Event:ScrollUp AND CurrentChoice > 1
    CurrentChoice -= 1
  ELSIF Event = Event:ScrollDown AND CurrentChoice < RECORDS(ListQueue)
    CurrentChoice += 1
  ELSE
    ItemsToFill = 1
    MyBrowse.Fetch( CHOOSE( Event = EVENT:ScrollUp, FillForward, FillBackward ))
  END
```

See Also: ResetQueue, ScrollOne, ScrollPage, ScrollEnd

**Init(initialize the BrowseClass object)**

<b>Init</b> ( <i>listcontrol, viewposition, view, listqueue, relationmanager, windowmanager</i> )	
<i> listcontrol, view, browsequeue, relationmanager, windowmanager</i>	
<b>Init</b>	Initializes the BrowseClass object.
<i>listcontrol</i>	A numeric constant, variable, EQUATE, or expression containing the control number of the browse's LIST control.
<i>ilistcontrol</i>	A reference to an IListControl interface.
<i>viewposition</i>	The label of a string field within the listqueue containing the POSITION of the view.
<i>view</i>	The label of the browse's underlying VIEW.
<i>listqueue</i>	The label of the listcontrol's data source QUEUE.
<i>browsequeue</i>	A reference to a BrowseQueue interface.
<i>relationmanager</i>	The label of the browse's primary file RelationManager object. See Relation Manager for more information.
<i>windowmanager</i>	The label of the browse's WindowManager object. See Window Manager for more information.

The **Init** method initializes the BrowseClass object.

**Init**(*listcontrol, viewposition, view, listqueue, relationmanager, windowmanager*)  
 Initializes the BrowseClass object using the Standard Behaviour interface. For more information see the StandardBehaviour section.

**Init**(*ilistcontrol, view, browsequeue, relationmanager, windowmanager*)  
 Initializes the BrowseClass object without the Standard Behaviour interface. The Init method calls the PARENT.Init (ViewManager.Init) method to initialize the browse's ViewManager object. See View Manager for more information. The Init method instantiates a PopupClass object for the browse. The Init method calls the WindowManager.AddItem method to register its presence with the WindowManager.

Example:

```
CODE                                     !Setup the BrowseClass object:
BrowseState.Init(?StateList, |         ! identify its LIST control,
    StateQ.Position, |         ! its VIEW position string,
    StateView, |         ! its source/target VIEW,
    StateQ, |         ! the LIST's source QUEUE,
    Relate:State |         ! the primary file RelationManager
    ThisWindow)         ! the WindowManager
```

See Also:            ViewManager.Init, PopupClass, WindowManager.AddItem, StandardBehavior  
                      Interface

### **InitSort (initialize locator values)**

**InitSort** (*neworder*), VIRTUAL

The **InitSort** method initializes locator values when a new sort order is applied to a browse list.

See Also:            SetSort

## Kill (shut down the BrowseClass object)

### Kill, VIRTUAL

The **Kill** method shuts down the BrowseClass object.

Implementation: Among other things, the BrowseClass.Kill method calls the PARENT.Kill (ViewManager.Kill) method to shut down the browse's ViewManager object. See *View Manager* for more information.

Example:

```
CODE
BrowseState.Init(?StateList, |           !Setup the BrowseClass object:
    StateQ.Position, |           ! identify its LIST control,
    StateView, |           ! its VIEW position string,
    StateQ, |           ! its source/target VIEW,
    Relate:State |           ! the LIST's source QUEUE,
    ThisWindow) |           ! the primary file RelationManager
                                ! the WindowManager

!program code

BrowseState.Kill                !shut down the BrowseClass object
```

See Also: ViewManager.Kill

**Next (get the next browse item)****Next, VIRTUAL**

The **Next** method gets the next record from the browse view and returns a value indicating its success or failure.

Next returns Level:Benign if successful, Level:Notify if it reached the end of the file, and Level:Fatal if it encountered a fatal error.

Implementation: Corresponding return value EQUATEs are declared in ABERROR.INC. See *Error Class* for more information on these severity level EQUATEs.

Level:Benign	EQUATE(0)
Level:User	EQUATE(1)
Level:Program	EQUATE(2)
Level:Fatal	EQUATE(3)
Level:Cancel	EQUATE(4)
Level:Notify	EQUATE(5)

The Next method is called by the Fetch and ResetThumbLimits(PRIVATE) methods. Among other things, Next calls the PARENT.Next (ViewManager.Next) method. See *ViewManager* for more information.

Return Data Type: BYTE

Example:

```

CASE MyBrowse.Next()      !get next record
OF Level:Benign          !if successful, continue
OF Level:Fatal           !if fatal error
  RETURN                 ! end this procedure
OF Level:Notify          !if end of file reached
  MESSAGE('Reached end of file.') ! acknowledge EOF
END

```

See Also: Fetch

## NotifyUpdateError (throw error on update)

**NotifyUpdateError( ), BYTE, VIRTUAL**

The **NotifyUpdateError** method returns an error code to the active ErrorManager when an attempted update to refresh a record has failed.

Implementation: The NotifyUpdateError is called from the BrowseClass UpdateViewRecord method, which is used to retrieve the VIEW record that corresponds to a chosen listbox record.

Return Data Type: BYTE

Example:

```
IF SELF.ListQueue.Records()  
  SELF.CurrentChoice = SELF.ILC.Choice()  
  SELF.ListQueue.Fetch(SELF.CurrentChoice)  
  WATCH(SELF.View)  
  REGET(SELF.View,SELF.ListQueue.GetViewPosition())  
  
  RC = ERRORCODE()  
  
  IF RC = NoDriverSupport  
    Pos = POSITION (SELF.View)  
    RESET(SELF.View,SELF.ListQueue.GetViewPosition())  
    WATCH(SELF.View)  
    NEXT(SELF.View)  
    RC = ERRORCODE()  
    RESET(SELF.View,Pos)  
  END  
  IF RC <> 0  
    SELF.NeedRefresh = SELF.NotifyUpdateError()  
  END  
END
```

See Also: UpdateViewRecord

## **PostNewSelection (post an EVENT:NewSelection to the browse list)**

### **PostNewSelection**

The **PostNewSelection** method posts an EVENT:NewSelection to the browse list to support scrolling, inserts, deletes, and other changes of position within the browse list.

Implementation: Event EQUATEs are declared in EQUATES.CLW.

Example:

#### **UpdateMyBrowse ROUTINE**

**!update code**

**MyBrowse.ResetFromFile**

**!after insert or change, reload Q from file**

**MyBrowse.PostNewSelection**

**!after update, post a new selection event**

**!so window gets properly refreshed**

## Previous (get the previous browse item)

### Previous, VIRTUAL

The **Previous** method gets the previous record from the browse view and returns a value indicating its success or failure.

Implementation: Returns Level:Benign if successful, Level:Notify if it reached the end of the file, and Level:Fatal if it encountered a fatal error. Corresponding severity level EQUATEs are declared in ABERROR.INC. See *Error Class* for more information on error severity levels.

Level:Benign	EQUATE(0)
Level:User	EQUATE(1)
Level:Program	EQUATE(2)
Level:Fatal	EQUATE(3)
Level:Cancel	EQUATE(4)
Level:Notify	EQUATE(5)

The Previous method is called by the Fetch and ResetThumbLimits methods. Among other things, Previous calls the PARENT.Previous (ViewManager.Previous) method. See *ViewManager* for more information.

Return Data Type: BYTE

Example:

```
CASE MyBrowse.Previous()      !get previous record
OF Level:Benign              !if successful, continue
OF Level:Fatal               !if fatal error
  RETURN                     ! end this procedure
OF Level:Notify              !if end of file reached
  MESSAGE('Reached end of file.'){ acknowledge EOF
END
```

See Also: Fetch

## Records (return the number of browse queue items)

### Records, PROC

The **Records** method returns the number of records in the browse list queue *and* disables appropriate controls if the record count is zero.

Return Data Type: LONG

Example:

```
DeleteMyBrowse ROUTINE
  !delete code
  MyBrowse.Records()           !disable delete button (and menu) if no items
```

## ResetFields(reinitialize FieldPairsClass)

### ResetFields

The **ResetFields** method reinitializes the FieldPairs recognized by the FieldPairsClass.

Implementation: The ResetFields method reinitializes the FieldPairs by first disposing the FieldsPairsClass and then initializing the FieldPairsClass.

See Also: FieldPairsClass.Init , FieldPairsClass.Kill

**ResetFromAsk (reset browse after update)****ResetFromAsk( *request*, *response* ), VIRTUAL, PROTECTED****ResetFromAsk** Resets the BrowseClass object following an update.*request* An integer constant, variable, EQUATE, or expression that indicates the type of update requested. Valid updates are insert, change, and delete.*response* An integer constant, variable, EQUATE, or expression that indicates whether the requested update was completed or cancelled.The **ResetFromAsk** method resets the BrowseClass object following an Ask or AskRecord update to a browse item.

Implementation: The Ask and AskRecord methods call ResetFromAsk as needed to reset the BrowseClass object.

ResetFromAsk FLUSHes the BrowseClass object's VIEW if needed, calls the appropriate "reset" method (ResetQueue, ResetFromFile, or ResetFromView) to refill the QUEUE, then carries out any pending scroll request made concurrently with the update. See *WindowManager.VCRRequest*.EQUATEs for the *request* parameter are declared in \LIBSRC\TPLEQU.CLW as follows:

```

InsertRecord EQUATE (1) !Add a record to table
ChangeRecord EQUATE (2) !Change the current record
DeleteRecord EQUATE (3) !Delete the current record

```

EQUATEs for the *response* parameter are declared in \LIBSRC\TPLEQU.CLW as follows:

```

RequestCompleted EQUATE (1) !Update Completed
RequestCancelled EQUATE (2) !Update Aborted

```

Example:

```
BrowseClass.Ask PROCEDURE(BYTE Req)
Response BYTE
CODE
LOOP
  SELF.Window.VCRRequest = VCR:None
  IF Req=InsertRecord THEN
    SELF.PrimeRecord
  END
  IF SELF.AskProcedure
    Response = SELF.Window.Run(SELF.AskProcedure,Req)
    SELF.ResetFromAsk(Req,Response)      !reset the browse after update
  ELSE
    Response = SELF.AskRecord(Req)
  END
UNTIL SELF.Window.VCRRequest = VCR:None
RETURN Response
```

See Also: Ask, AskRecord, ResetQueue, ResetFromFile, ResetFromView,  
WindowManager.VCRRequest

**ResetFromBuffer (fill queue starting from record buffer)****ResetFromBuffer, VIRTUAL**

The **ResetFromBuffer** method fills or refills the browse queue starting from the record in the primary file buffer (and secondary file buffers if applicable). If the record is found, **ResetFromBuffer** fills the browse queue starting from that record. If the record is not found, **ResetFromBuffer** fills the browse queue starting from the nearest matching record.

If the active sort order (key) allows duplicates and duplicate matches exist, **ResetFromBuffer** fills the browse queue starting from the *first* matching record.

**Tip:** Use **ResetFromBuffer** when the primary and secondary file positions and values are valid, but the result set may no longer match the buffer values. For example, after a locator or scrollbar thumb move.

Implementation: **ResetFromBuffer** succeeds even if there is no exactly matching record and is typically used to locate the appropriate record after a thumb movement.

**ResetFromBuffer** calls the **ViewManager.Reset** method for positioning, then calls the **ResetQueue** method to fill the browse queue.

Example:

```

IF EVENT() = EVENT:ScrollDrag           !if thumb moved
  IF ?MyList{PROP:VScrollPos} <= 1      !handle scroll to top
    POST(Event:ScrollTop, ?MyList)
  ELSIF ?MyList{PROP:VScrollPos} = 100 !handle scroll to bottom
    POST(Event:ScrollBottom, ?MyList)
  ELSE                                   !handle intermediate scroll
    MyBrowse.Sort.FreeElement =
MyBrowse.Sort.Step.GetValue(?MyList{PROP:VScrollPos})
    MyBrowse.ResetFromBuffer             !and reload the queue from that point
  END
END

```

See Also: **ViewManager.Reset**, **ResetQueue**

## ResetFromFile (fill queue starting from file POSITION)

### ResetFromFile, VIRTUAL

The **ResetFromFile** method fills or refills the browse queue starting from the current POSITION of the primary file. If no POSITION has been established, ResetFromFile fills the browse queue starting from the beginning of the file.

**Tip:** Use ResetFromFile when the primary file position is valid but secondary records and their contents may not be. For example, when returning from an update.

Implementation: ResetFromFile succeeds even if the record buffer is cleared and is typically used to get the current record after an update.

Example:

```
MyBrowseClass.ResetFromAsk PROCEDURE(*BYTE Request,*BYTE Response)
  CODE
  IF Response = RequestCompleted
    FLUSH(SELF.View)
    IF Request = DeleteRecord
      DELETE(SELF.ListQueue)
      SELF.ResetQueue(Reset:Queue) !refill queue after delete
    ELSE
      SELF.ResetFromFile           !refill queue after insert or change
    END
  ELSE
    SELF.ResetQueue(Reset:Queue)
  END
```

## ResetFromView (reset browse from current result set)

ResetFromView, VIRTUAL

The **ResetFromView** method resets the BrowseClass object to conform to the current result set.

**Tip:** Use **ResetFromView** when you want to reset for any changes that may have happened to the entire record set, such as new records added or deleted by other workstations.

Implementation: The SetSort method calls the ResetFromView method.

The ResetFromView method readjusts the scrollbar thumb if necessary. The ABC Templates override the BrowseClass.ResetFromView method to recalculate totals if needed.

Example:

```
BRW1.ResetFromView PROCEDURE
ForceRefresh:Cnt LONG
CODE
SETCURSOR(Cursor:Wait)
SELF.Reset
LOOP
CASE SELF.Next()
OF Level:Notify
BREAK
OF Level:Fatal
RETURN
END
SELF.SetQueueRecord
ForceRefresh:Cnt += 1
END
ForceRefresh = ForceRefresh:Cnt
SETCURSOR()
```

**ResetQueue (fill or refill queue)****ResetQueue**( *resetmode* ), VIRTUAL**ResetQueue** Fills or refills the browse queue.

*resetmode* A numeric constant, variable, EQUATE, or expression that determines how ResetQueue determines the highlighted record after the reset. A value of Reset:Queue highlights the currently selected item. A value of Reset:Done highlights a record based on the view's current position and other factors, such as the RetainRow property.

The **ResetQueue** method fills or refills the browse queue and appropriately enables or disables Change, Delete, and Select controls. The refill process depends on the value of the *resetmode* parameter and several other BrowseClass properties, including ActiveInvisible, AllowUnfilled, RetainRow, etc.

A *resetmode* value of Reset:Queue usually produces a more efficient queue refill than Reset:Done.

Implementation: ResetQueue calls the Fetch method to fill the queue.

The *resetmode* EQUATEs are declared in ABBROWSE.INC as follows:

```
ITEMIZE,PRE(Reset)
Queue EQUATE
Done EQUATE
END
```

Example:

```
DeleteMyBrowse ROUTINE
!delete code
MyBrowse.ResetQueue(Reset:Queue) !after delete, refresh Q
MyBrowse.PostNewSelection         !after delete, post a new selection event
                                   !so window gets properly refreshed
```

See Also: ActiveInvisible, AllowUnfilled, RetainRow, ChangeControl, DeleteControl, SelectControl, Fetch

## ResetResets (copy the Reset fields)

### ResetResets, PROTECTED

The **ResetResets** method copies the current values of the Reset fields so any subsequent changes in their contents can be detected.

The AddResetField method adds an associated reset field for the sort order defined by the preceding call to AddSortOrder. You may add multiple reset fields for each sort order with multiple calls to AddResetField.

Example:

```
MyBrowse.CheckReset PROCEDURE
  IF NOT SELF.Sort.Resets.Equal() !if reset fields changed,
    SELF.ResetQueue(Reset:Queue) !refresh Q
    SELF.ResetResets             !take a new copy of the reset field values
  END
```

See Also:       AddResetField

**ResetSort (apply sort order to browse)****ResetSort( *force* ), VIRTUAL, PROC**


---

<b>ResetSort</b>	Reapplies the active sort order to the browse list.
<i>force</i>	A numeric constant, variable, EQUATE, or expression that indicates whether to reset the browse conditionally or unconditionally. A value of one (1 or True) unconditionally resets the browse; a value of zero (0 or False) only resets the brose as circumstances require (sort order changed, reset fields changed, first loading, etc.).

The **ResetSort** method reapplies the active sort order to the browse list and returns one (1) if the sort order changed; it returns zero (0) if the order did not change. Any range limits, locators, or reset fields associated with the sort order are enabled.

**Tip: Use ResetSort followed by UpdateWindow to refresh and redisplay your ABC BrowseBoxes. Or, use the WindowManager.Reset method.**

Implementation: The ResetSort method calls the SetSort method to applt the current sort order. The ABC Templates override the ResetSort method to apply the sort order based on the selected tab.

Return Data Type: BYTE

Example:

```
BRW1.ResetSort FUNCTION(BYTE Force)    !apply appropriate sort order

CODE
IF CHOICE(?CurrentTab) = 1            !If 1st tab selected
  RETURN SELF.SetSort(1,Force)        !apply first sort order
ELSE
  RETURN SELF.SetSort(2,Force)        !apply second sort order
END
```

See Also: AddRange, AddResetField, AddSortOrder, Set Sort ,UpdateWindow

**ScrollEnd (scroll to first or last item)****ScrollEnd( *scrollevent* ), VIRTUAL, PROTECTED**


---

<b>ScrollEnd</b>	Scrolls to the first or last browse list item.
<i>scrollevent</i>	A numeric constant, variable, EQUATE, or expression that indicates the requested scroll action. Valid scroll actions for this method are scrolls to the top or bottom of the list.

The **ScrollEnd** method scrolls to the first or last browse list item.

Implementation: The BrowseClass.TakeScroll method calls the ScrollEnd method.

A hexadecimal *scrollevent* value of EVENT:ScrollTop scrolls to the first list item. A value of EVENT:ScrollBottom scrolls to the last list item. Corresponding scroll event EQUATEs are declared in EQUATES.CLW:

```
EVENT:ScrollTop      EQUATE (07H)
EVENT:ScrollBottom  EQUATE (08H)
```

Example:

```
BrowseClass.TakeScroll PROCEDURE( SIGNED Event )
CODE
IF RECORDS(SELF.ListQueue)
CASE Event
OF Event:ScrollUp OROF Event:ScrollDown
SELF.ScrollOne( Event )
OF Event:PageUp OROF Event:PageDown
SELF.ScrollPage( Event )
OF Event:ScrollTop OROF Event:ScrollBottom
SELF.ScrollEnd( Event )
END
END
```

See Also: TakeScroll

**ScrollOne (scroll up or down one item)****ScrollOne**( *scrollevent* ), VIRTUAL, PROTECTED

---

<b>ScrollOne</b>	Scrolls up or down one browse list item.
<i>scrollevent</i>	A numeric constant, variable, EQUATE, or expression that indicates the requested scroll action. Valid scroll actions for this method are scrolls up or down a single list item.

The **ScrollOne** method scrolls up or down one browse list item.

Implementation: The BrowseClass.TakeScroll method calls the ScrollOne method.

A hexadecimal *scrollevent* value of EVENT:ScrollUp scrolls up one list item. A value of EVENT:ScrollDown scrolls down one list item. Corresponding scroll event EQUATEs are declared in EQUATES.CLW:

```
EVENT:ScrollUp      EQUATE (03H)
EVENT:ScrollDown   EQUATE (04H)
```

Example:

```
BrowseClass.TakeScroll PROCEDURE( SIGNED Event )
CODE
IF RECORDS(SELF.ListQueue)
CASE Event
OF Event:ScrollUp OROF Event:ScrollDown
SELF.ScrollOne( Event )
OF Event:PageUp OROF Event:PageDown
SELF.ScrollPage( Event )
OF Event:ScrollTop OROF Event:ScrollBottom
SELF.ScrollEnd( Event )
END
END
```

See Also: TakeScroll

**ScrollPage (scroll up or down one page)****ScrollPage**( *scrollevent* ), VIRTUAL, PROTECTED

---

**ScrollPage**      Scrolls up or down one page of browse list items.

*scrollevent*      A numeric constant, variable, EQUATE, or expression that indicates the requested scroll action. Valid scroll actions for this method are scrolls up one page or down one page of browse list items.

The **ScrollPage** method scrolls up or down one page of browse list items.

Implementation:      The BrowseClass.TakeScroll method calls the ScrollPage method.

A hexadecimal *scrollevent* value of EVENT:PageUp scrolls up one page of browse list items. A value of EVENT:PageDown scrolls down one page of browse list items. Corresponding scroll event EQUATEs are declared in EQUATES.CLW:

```
EVENT:PageUp            EQUATE (05H)
EVENT:PageDown        EQUATE (06H)
```

Example:

```
BrowseClass.TakeScroll PROCEDURE( SIGNED Event )
CODE
IF RECORDS(SELF.ListQueue)
CASE Event
OF Event:ScrollUp OROF Event:ScrollDown
SELF.ScrollOne( Event )
OF Event:PageUp OROF Event:PageDown
SELF.ScrollPage( Event )
OF Event:ScrollTop OROF Event:ScrollBottom
SELF.ScrollEnd( Event )
END
END
```

See Also:              TakeScroll

**SetAlerts (alert keystrokes for list and locator controls)****SetAlerts, VIRTUAL**

The **SetAlerts** method alerts standard keystrokes for the browse's list control and for any associated locator controls.

The BrowseClass.TakeKey method processes the alerted keystrokes.

Implementation: The BrowseClass.SetAlerts method alerts the mouse DOUBLE-CLICK, the INSERT, DELETE and CTRL+ENTER keys for the browse's list control and calls the LocaorClass.SetAlerts method for each associated locator control. Corresponding keycode EQUATEs are declared in KEYCODES.CLW.

The BrowseClass.SetAlerts method also sets up a popup menu for the browse list that mimics the behavior of any control buttons (insert, change, delete, select).

Example:

```
PrepareStateBrowse ROUTINE                !Setup the BrowseClass object:
BrowseState.Init(?StateList, |           ! identify its LIST control,
    StateQ.Position, |                 ! its VIEW position string,
    StateView, |                       ! its source/target VIEW,
    StateQ, |                           ! the LIST's source QUEUE,
    Relate:State)                       ! and primary file RelationManager
BrowseState.SetAlerts                   !alert LIST and locator keystrokes
```

See Also: TakeKey

### **SetLocatorField (set sort free element to passed field)**

**SetLocatorField** (*free*), VIRTUAL

---

**SetLocatorField** Sets the sort free element to the passed field.

*free* An ANY data type, passed by address, that contains the free element that will be used as the locator field.

The **SetLocatorField** method sets the specified locator sort to the browse list. The free element represents a potential sort that has modified the default sort in the browse list. That element now can become the active locator.

Implementation: The BrowseClass.SetLocatorFromSort call the SetLocatorField method.

### **SetLocatorFromSort (use sort like locator field)**

**SetLocatorFromSort** (*free*), VIRTUAL

---

**SetLocatorFromSort** Applies a specified locator to the browse list.

The **SetLocatorFromSort** method uses the first field of the sort as the locator field if there is a sort order active.

Implementation: None

## **SetQueueRecord (copy data from file buffer to queue buffer:BrowseClass)**

### **SetQueueRecord, VIRTUAL**

The **SetQueueRecord** method copies corresponding data from the *filefield* fields to the *queuefield* fields specified by the AddField method. Typically these are the file buffer fields and the browse list's queue buffer fields so that the queue buffer matches the file buffers.

Implementation: The BrowseClass.Fetch and BrowseClass.Ask methods call the SetQueueRecord method.

Example:

**MyBrowseClass.SetQueueRecord PROCEDURE**

**CODE**

```
SELF.Fields.AssignLeftToRight           !copy data from file to q buffer  
SELF.ViewPosition = POSITION( SELF.View )!set the view position  
!your custom code here
```

See Also: Ask, AddField, Fetch

**SetSort (apply a sort order to the browse)****SetSort**( *order*, *force reset* ), VIRTUAL, PROC

---

<b>SetSort</b>	Applies a specified sort order to the browse list.
<i>order</i>	An integer constant, variable, EQUATE, or expression that specifies the sort order to apply.
<i>force reset</i>	A numeric constant, variable, EQUATE, or expression that tells the method whether to reset the browse conditionally or unconditionally. A value of zero (0 or False) resets the browse only if circumstances require (sort order changed, reset fields changed, first time loading); a value of one (1 or True) unconditionally resets the browse.

The **SetSort** method applies the specified sort *order* to the browse list and returns one (1) if the sort order changed; it returns zero (0) if the sort order did not change. Any range limits, locators, and reset fields associated with the sort order are enabled and applied.

The *order* value is typically a value returned by the AddSortOrder method which identifies the particular sort order. Since AddSortOrder returns sequence numbers, a value of one (1) applies the sort order specified by the first call to AddSortOrder; two (2) applies the sort order specified by the next call to AddSortOrder; etc. A value of zero (0) applies the default sort order.

Implementation: The ResetSort method calls the SetSort method.

Return Data Type: BYTE

Example:

```

IF FIELD() = ?FirstTab           !if first tab selected
  IF MyBrowse.SetSort(1,0)       !apply the first sort order
    MyBrowse.ResetThumbLimits   !if sort changed, reset thumb limits
  END
  MyBrowse.UpdateBuffer         !update file buffer from selected item
END

```

See Also: AddRange, AddResetField, AddSortOrder, ResetSort

## TakeAcceptedLocator (apply an accepted locator value)

### TakeAcceptedLocator, VIRTUAL

The **TakeAcceptedLocator** method applies an accepted locator value to the browse list--the BrowseClass object scrolls the list to the requested item.

Locators with entry controls are the only locators whose values are accepted. Other types of locators are invoked in other ways, for example, with alerted keys. Locator values are accepted when the end user TABS off or otherwise switches focus away from the locator's entry control.

The AddLocator method establishes locators for the browse.

Implementation: The TakeAcceptedLocator method calls the appropriate LocatorClass.TakeAccepted method.

Example:

```
IF FIELD() = ?MyLocator           !focus on locator field
  IF EVENT() = EVENT:Accepted     !if accepted
    MyBrowse.TakeAcceptedLocator  !BrowseClass object handles it
  END
END
```

See Also: AddLocator

## TakeEvent (process the current ACCEPT loop event:BrowseClass)

### TakeEvent, VIRTUAL

The **TakeEvent** method processes the current ACCEPT loop event for the BrowseClass object. The TakeEvent method handles all events associated with the browse list except a new selection event. The TakeNewSelection method handles new selection events for the browse.

Implementation: The WindowManager.TakeEvent method calls the TakeEvent method. The TakeEvent method calls the TakeScroll or TakeKey method as appropriate.

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
!procedure code
LOOP I = 1 TO RECORDS(SELF.Browses)
    GET(SELF.Browses,I)
    SELF.Browses.Browse.TakeEvent
END
LOOP i=1 TO RECORDS(SELF.FileDrops)
    GET(SELF.FileDrops,i)
    ASSERT(~ERRORCODE())
    SELF.FileDrops.FileDrop.TakeEvent
END
RETURN RVal
```

See Also: TakeKey, TakeNewSelection, TakeScroll, WindowManager.TakeEvent

## TakeKey (process an alerted keystroke:BrowseClass)

### TakeKey, VIRTUAL, PROC

The **TakeKey** method processes an alerted keystroke for the BrowseClass object, including DOUBLE-CLICK, INSERT, CTRLENTER, or DELETE, and returns a value indicating whether any action was taken.

Implementation: TakeKey returns one (1) if any action is taken, otherwise it returns zero (0).

The TakeEvent method calls the TakeKey method as appropriate. The BrowseClass.TakeKey method calls the Locator.TakeKey method as appropriate.

Return Data Type: BYTE

Example:

```
IF FIELD() = ?MyBrowseList      !focus on browse list
  IF EVENT() EVENT:AlertKey     !if alerted keystroke
    MyBrowse.TakeKey           !BrowseClass object handles it
  END
END
```

See Also: TakeEvent

## TakeNewSelection (process a new selection:BrowseClass)

### TakeNewSelection, VIRTUAL, PROC

The **TakeNewSelection** method processes a new browse list item selection and returns a value indicating whether a window redraw is needed.

Implementation: TakeNewSelection returns one (1) if a window redraw is needed, otherwise it returns zero (0).

The TakeEvent method calls the TakeNewSelection method when appropriate.  
The BrowseClass.TakeNewSelection method calls the appropriate Locator.TakeNewSelection method.

Return Data Type: BYTE

Example:

```
IF FIELD() = ?MyBrowse           !focus on browse list
  IF EVENT() = EVENT:NewSelection !if new selection
    MyBrowse.TakeNewSelection()   !BrowseClass object handles it
  ELSE                             !if other event
    MyBrowse.TakeEvent           !BrowseClass object handles it
  END
END
END
```

**TakeScroll (process a scroll event)****TakeScroll**( [*scrollevent*] ), VIRTUAL**TakeScroll** Processes a scroll event for the browse list.

*scrollevent* An integer constant, variable, EQUATE, or expression that specifies the scroll event. Valid scroll events are up one item, down one item, up one page, down one page, up to the first item, and down to the last item. If omitted, no scrolling occurs.

The **TakeScroll** method processes a scroll event for the browse list.

Implementation: A *scrollevent* value of EVENT:ScrollUp scrolls up one item; EVENT:ScrollDown scrolls down one item; EVENT:PageUp scrolls up one page; EVENT:PageDown scrolls down one page; EVENT:ScrollTop scrolls to the first list item; EVENT:ScrollBottom scrolls to the last list item. Corresponding *scrollevent* EQUATEs are declared in EQUATES.CLW.

```
EVENT:ScrollUp      EQUATE (03H)
EVENT:ScrollDown    EQUATE (04H)
EVENT:PageUp        EQUATE (05H)
EVENT:PageDown      EQUATE (06H)
EVENT:ScrollTop     EQUATE (07H)
EVENT:ScrollBottom  EQUATE (08H)
```

The TakeScroll method calls the ScrollEnd, ScrollOne, or ScrollPage method as needed.

Example:

```
IF FIELD() = ?MyBrowse          !focus on browse list
CASE EVENT()                    !scroll event
OF EVENT:ScrollUp
OROF EVENT:ScrollDown
OROF EVENT:PageUp
OROF EVENT:PageDown
OROF EVENT:ScrollTop
OROF EVENT:ScrollBottom
  MyBrowse.TakeScroll          !BrowseClass object handles it
END
END
```

See Also: ScrollEnd, ScrollOne, ScrollPage

**TakeVCRScroll (process a VCR scroll event)**

**TakeVCRScroll**( [*vcrevent*] ), VIRTUAL

---

**TakeVCRScroll** Processes a VCR scroll event for the browse list.

*vcrevent* An integer constant, variable, EQUATE, or expression that specifies the scroll event. Valid scroll events are up one item, down one item, up one page, down one page, up to the first item, and down to the last item. If omitted, no scrolling occurs.

The **TakeVCRScroll** method processes a VCR scroll event for the browse

Implementation: A *vcrevent* value of VCR:Forward scrolls down one item; VCR:Backward scrolls up one item; VCR:PageForward scrolls down one page; VCR:PageBackward scrolls up one page; VCR:Last scrolls to the last list item; VCR:First scrolls to the first list item. Corresponding *vcrevent* EQUATEs are declared in \LIBSRC\ABTOOLBA.INC.

```

ITEMIZE, PRE(VCR)
Forward      EQUATE(Toolbar:Down)
Backward     EQUATE(Toolbar:Up)
PageForward  EQUATE(Toolbar:PageDown)
PageBackward EQUATE(Toolbar:PageUp)
First        EQUATE(Toolbar:Top)
Last         EQUATE(Toolbar:Bottom)
Insert       EQUATE(Toolbar:Insert)
None         EQUATE(0)
END

```

The TakeVCRScroll method calls the TakeScroll method, translating the *vcrevent* to the appropriate *scrollevent*.

Example:

```

LOOP                !process repeated scroll events
  IF VCRRequest = VCR:None !if no more events
    BREAK           !break out of loop
  ELSE              !if scroll event
    MyBrowse.TakeVCRScroll( VCRRequest ) !BrowseClass object handles it
  END
END

```

See Also: TakeScroll

## UpdateBuffer (copy selected item from queue buffer to file buffer)

### UpdateBuffer, VIRTUAL

The **UpdateBuffer** method copies corresponding data from the *queuefield* fields to the *filefield* fields specified by the AddField method for the currently selected browse item. Typically these are the browse list's queue buffer fields and the file buffer fields so that the file buffers match the currently selected browse list item.

Implementation: Many of the BrowseClass methods call the UpdateBuffer method.

Example:

```
IF FIELD() = ?FirstTab           !if first tab selected
  IF MyBrowse.SetSort(1,0)       !apply the first sort order
    MyBrowse.ResetThumbLimits    !if sort changed, reset thumb limits
  END
  MyBrowse.UpdateBuffer          !update file buffer from selected item
  MyBrowse.UpdateResets          !update file buffer from reset fields
END
```

See Also: AddField

**UpdateQuery (set default query interface)**

**UpdateQuery**( *querymanager*, [*casesensitive*])

---

**UpdateQuery** Defines a default query interface for the BrowseClass object.

*querymanager* The label of the BrowseClass object's QueryClass object. See *QueryClass* for more information.

*casesensitive* A numeric constant, variable, EQUATE, or expression that indicates the case sensitivity of the query expression. If this parameter is omitted the query is case insensitive.

The **UpdateQuery** method defines a default query interface (dialog) for the BrowseClass object.

**Tip:** You may use the **UpdateQuery** method in combination with the **QueryClass.AddItem** method to define a query interface that contains the displayed fields plus other queryable items.

Implementation: The UpdateQuery method sets the value of the Query property, then calls the QueryClass.AddItem method for each displayed field, so that each displayed field accepts filter criteria in the query dialog.

Example:

```
QueryForm QueryFormClass
QueryVis QueryFormVisual
BRW1 CLASS(BrowseClass)
Q &CusQ
END
```

```
CusWindow.Init PROCEDURE()
CODE
!open files, views, window, etc.
IF DefaultQuery
BRW1.UpdateQuery(QueryForm)
ELSE
BRW1.Query &= QueryForm
QueryForm.AddItem('UPPER(CUS:NAME)', '', '')
QueryForm.AddItem('UPPER(CUS:CITY)', '', '')
QueryForm.AddItem('CUS:ZIP_CODE', '', '')
END
RETURN Level:Benign
```

See Also: Query, QueryClass.AddItem

## UpdateResets (copy reset fields to file buffer)

### UpdateResets, PROTECTED

The **UpdateResets** method copies reset field values to corresponding file buffer fields.

The **AddResetField** method defines the reset fields for the **BrowseClass** object.

Implementation: The **BrowseClass.Next** and **BrowseClass.Previous** methods call the **UpdateResets** method.

Example:

```
MyBrowseClass.Next PROCEDURE          !method of class derived from BrowseClass
CODE
  IF Level:Fatal = PARENT.Next()      !do parent method
    POST(EVENT:CloseWindow)          !if fails, shut down
  ELSE                                  !otherwise
    SELF.UpdateResets                 !update file buffer from reset fields
  END
```

See Also: [AddResetField](#), [Next](#), [Previous](#)

## UpdateThumb (position the scrollbar thumb)

### UpdateThumb

The **UpdateThumb** method positions the scrollbar thumb and enables or disables the vertical scroll bar depending on the number of items in the browse list, the currently selected item, and the active step distribution method. See *Control Templates--BrowseBox* for more information on thumb behavior.

Implementation: The AddSortOrder method sets the stepdistribution methods for the BrowseClass object.

Example:

```
IF FIELD() = ?MyBrowse           !focus on browse list
  IF EVENT() = EVENT:NewSelection !if new selection
    IF MyBrowse.TakeNewSelection() !BrowseClass object handles it
      MyBrowse.UdateThumb         !Reposition the thumb
    END
  END
END
END
END
```

## UpdateThumbFixed (position the scrollbar fixed thumb)

### UpdateThumbFixed, PROTECTED

The **UpdateThumbFixed** method positions the scrollbar fixed thumb and enables or disables the vertical scroll bar depending on the number of items in the browse list, the currently selected item, and the active step distribution method. See *Control Templates--BrowseBox* for more information on fixed thumb behavior.

Implementation: The AddSortOrder method sets the step distribution methods for the BrowseClass object.

Example:

```
MyBrowseClass.UpdateThumb PROCEDURE
```

```
CODE
```

```
IF SELF.Sort.Thumb &= NULL
```

```
!if no step object
```

```
    SELF.UpdateThumbFixed
```

```
!reposition thumb as fixed
```

```
ELSE
```

```
!reposition thumb per step object
```

```
END
```

## UpdateViewRecord (get view data for the selected item)

### UpdateViewRecord, VIRTUAL

The **UpdateViewRecord** method regets the browse's VIEW record for the selected browse list item so the VIEW record can be written to disk. The UpdateViewRecord method arms automatic optimistic concurrency checking so the eventual write (PUT) to disk returns an error if another user changed the data since it was retrieved by UpdateViewRecord.

Implementation: The UpdateViewRecord method uses WATCH and REGET to implement optimistic concurrency checking; see the *Language Reference* for more information.

Example:

```
IF FIELD() = ?ChangeButton           !on change button
IF EVENT() = EVENT:Accepted           !if button clicked
  MyBrowse.UpdateViewRecord           !refresh buffers and arm WATCH
  DO MyBrowse:ButtonChange           !call the update routine
END
END
```

## UpdateWindow (update display variables to match browse)

### UpdateWindow, VIRTUAL

The **UpdateWindow** method updates display variables to match the current state of the browse list.

**Tip:** Use **ResetSort** followed by **UpdateWindow** to refresh and redisplay your **ABC BrowseBoxes**. Or, use the **WindowManager.Reset** method.

Implementation: The **BrowseClass.UpdateWindow** method calls the appropriate **LocatorClass.UpdateWindow** method, which ensures the locator field contains the current search value.

Example:

```
IF FIELD() = ?MyBrowse           !focus on browse list
  IF EVENT) = EVENT:NewSelection !if new selection
    IF MyBrowse.TakeNewSelection() !BrowseClass object handles it
      MyBrowse.SetSort(0,1)        !reapply sort order
      MyBrowse.UpdateBuffer        !refresh file buffer from selected item
      MyBrowse.UpdateWindow        !update display variables (locator)
      DISPLAY()                    !and redraw the window
    END
  END
END
```

---

## BrowseQueue Interface

### BrowseQueue Concepts

The BrowseQueue interface is a defined set of behaviors that relate to the VIEW and QUEUE that the LIST control uses.

### Relationship to Other Application Builder Classes

The StandardBehavior class implements the BrowseQueue interface. For more information, see the StandardBehavior class.

### BrowseQueue Source Files

The BrowseQueue source code is installed by default to the Clarion \LIBSRC folder. The specific BrowseQueue source code and their respective components are contained in:

ABBROWSE.INC	BrowseQueue interface declaration
ABBROWSE.CLW	BrowseQueue method definitions

## BrowseQueue Methods

### Delete(remove entry in LIST queue)

#### Delete

The **Delete** method removes an entry in the queue that the LIST control is using.

### Fetch(retrieve entry from LIST queue)

#### Fetch(*position*)

<b>Fetch</b>	Retrieves an entry from the queue for the LIST control.
<i>position</i>	An integer constant, variable, EQUATE, or expression that indicates the relative position in the queue for the LIST.

The **Fetch** method retrieves an entry from the queue that the LIST control is using at the relative position specified.

### Free(clear contents of LIST queue)

#### Free

The **Free** method clears all entries from the queue that the LIST control is using.

### GetViewPosition(retrieve VIEW position)

#### GetViewPosition

The **GetViewPosition** method retrieves the VIEW's POSITION.

Return Data Type:      STRING

## Insert(add entry to LIST queue)

**Insert**(*[position]*)

<b>Insert</b>	Adds an entry to the queue for the LIST control.
<i>position</i>	An integer constant, variable, EQUATE, or expression that indicates the relative position in the queue for the LIST.

The **Insert** method adds an entry to the queue that the LIST control is using. If no position parameter is used, the entry is added to the end of the queue. If the position parameter is used, the entry is added at the relative position specified. If an entry exists at that position, it is moved down to make room for the new entry.

## Records(return number of records)

**Records**

The **Records** method returns the number of records available in the queue for the LIST control.

Return Data Type:      UNSIGNED

## SetViewPosition(set VIEW position)

**SetViewPosition**(*position*)

<b>SetViewPosition</b>	Sets the POSITION of the VIEW.
<i>position</i>	A string constant, variable, EQUATE, or expression containing the POSITION to set in the VIEW.

The **SetViewPosition** sets the POSITION of the VIEW based on the position parameter.

**Update(update entry in LIST queue)****Update**

The **Update** method updates an entry in the queue that the LIST control is using.

**Who(returns field name)****Who**(*column*)

<b>Who</b>	Returns the queue field name for the specified column.
<i>column</i>	An integer constant, variable, EQUATE, or expression that contains a column number from the queue.

The **Who** method returns the queue field name for the column specified by the column parameter.

Return Data Type:      **STRING**

---

# BrowseToolBarClass

## BrowseToolBarClass Overview

The BrowseToolBarClass handles events for specialized buttons for scrolling in the associated BrowseBox. This class works with the BrowseClass and the WindowManager objects to accomplish these tasks.

## BrowseToolBarClass Concepts

The BrowseToolBarClass object interacts with the BrowseClass and WindowManager to allow the toolbar buttons to scroll the browse highlight bar within the BrowseBox. When a toolbar button is pressed and EVENT:Accepted is posted to the associated Browse control.

## Relationship to Other Application Builder Classes

The BrowseToolBarClass works with the BrowseClass and WindowManager to accomplish its tasks.

## BrowseToolBarClass ABC Template Implementation

The BrowseToolBarControl control template generates code to declare a BrowseToolBarClass object in the Browse procedure that the control template is placed. The templates also generate code to register the BrowseToolBarClass object with the BrowseClass and WindowManager objects, as well as initializing all toolbar button controls.

## BrowseToolBarClass Source Files

The BrowseToolBarClass source code is installed by default to the Clarion \LIBSRC folder. The specific BrowseToolBarClass source code and their respective components are contained in:

ABTOOLBA.INC  
ABTOOLBA.CLW

BrowseToolBarClass declarations  
BrowseToolBarClass method definitions

## BrowseToolbarClass Properties

### Browse (BrowseClass object)

**Browse**      **&BrowseClass, PROTECTED**

The **Browse** property is a reference to the BrowseClass object. The BrowseToolbarClass object uses this property to access the BrowseClass object's properties and methods.

Implementation:      The BrowseToolbarClass.Init method sets the value of the Browse property.

See Also:      BrowseToolbarClass.Init

## Button (toolbar buttons FEQ values)

**Button**                    **SIGNED, DIM(ToolBar:Last+1-ToolBar:First), PROTECTED**

The **Button** property is a dimensioned variable that holds the control numbers (FEQ) of the buttons that are represented on the toolbar. A value of zero (0) disables the individual toolbar button

Implementation:            The BrowseToolBarClass object uses this property to enable or disable a toolbar control.

## Window (WindowManager object)

**Window**                    **&WindowManager, PROTECTED**

The **Window** property is a reference to the WindowManager object for this BrowseToolBarClass object. The WindowManager object forwards events to the BrowseToolBarClass object for processing.

Implementation:            The BrowseToolBarClass.Init method sets the value of the Window property.

See Also:                  BrowseToolBarClass.Init

## BrowseToolBarClass Methods

### Init (initialize the BrowseToolBarClass object)

**Init**(*WindowManager, BrowseClass*)

<b>Init</b>	Initializes the BrowseToolBarClass object.
<i>WindowManager</i>	The label of the toolbar's WindowManager object. See Window Manager for more information.
<i>BrowseClass</i>	The label of the toolbar's BrowseClass object. See BrowseClass for more information.

The **Init** method initializes the BrowseToolBarClass object by declaring a reference to both the WindowManager and BrowseClass objects. All toolbar buttons are initialized to zero (0).

### InitBrowse (initialize the BrowseToolBarClass update buttons)

**InitBrowse**(*insert, change, delete, select*)

<b>InitBrowse</b>	Initializes the BrowseToolBarClass update buttons.
<i>insert</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Insert control.
<i>change</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Change control.
<i>delete</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Delete control.
<i>select</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Select control.

The **InitBrowse** method initializes the Button property with the control numbers (FEQ) for the update controls on the toolbar.

Implementation: This method is called automatically by the ABC BrowseToolBarControl template.

**InitMisc (initialize the BrowseToolBarClass miscellaneous buttons)**

**InitMisc**(*history, help*)

<b>InitMisc</b>	Initializes the BrowseToolBarClass miscellaneous buttons.
<i>history</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the History control.
<i>help</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Help control.

The **InitMisc** method initializes the Button property with the control numbers (FEQ) for the History and Help controls on the toolbar.

Implementation: This method is called automatically by the ABC BrowseToolBarControl template.

## **InitVCR (initialize the BrowseToolBarClass VCR buttons)**

**InitVCR**(*top, bottom, pageup, pagedown, up, down, locate*)

<b>InitVCR</b>	Initializes the BrowseToolBarClass VCR buttons.
<i>top</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Top control.
<i>bottom</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Bottom control.
<i>pageup</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the PageUp control.
<i>pagedown</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the PageDown control.
<i>up</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Up control.
<i>down</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Down control.
<i>locate</i>	An integer constant, variable, EQUATE, or expression that identifies the FEQ for the Locate control.

The **InitVCR** method initializes the Button property with the control numbers (FEQ) for the VCR controls on the toolbar.

Implementation: This method is called automatically by the ABC BrowseToolBarControl template.

**ResetButton (synchronize toolbar with a corresponding browse control)****ResetButton**(*toolbar*, *browsebutton*), PROTECTED

<b>ResetButton</b>	Sync Toolbar control properties with its corresponding Browse control.
<i>toolbar</i>	An integer constant, variable, EQUATE, or expression that identifies a particular toolbar control.
<i>browsebutton</i>	An integer constant, variable, EQUATE, or expression that identifies a specific browse control.

The **ResetButton** method enables/disables, hides/unhides a toolbar control based on the current properties of its corresponding browse control.

There are predefined equates that represent each button available on the toolbar. These equates can be found at the top of ABTOOLBA.INC.

Implementation: This method is called by the BrowseToolBarClass.ResetFromBrowse method.

## **ResetFromBrowse(synchronize toolbar controls with browse controls)**

### **ResetFromBrowse, VIRTUAL**

The **ResetFromBrowse** method synchronizes the toolbar controls with their corresponding browse control. These controls include the Select, Insert, Change, Delete, History, Help and Locate buttons.

Implementation: This method is called from the BrowseToolBarClass.TakeEvent method.

## **TakeEvent(process the current event)**

### **TakeEvent, VIRTUAL**

The **TakeEvent** method processes all accepted events for the toolbar controls. When an accepted event occurs, an EVENT:Accepted is posted to the corresponding Browse control. When a NewSelection event occurs on the BrowseBox, the ResetFromBrowse method is called to redisplay the toolbar controls with the correct properties (hide, unhide, enable, disable).

A Level:Benign is returned from this method.

Return Data Type: BYTE

# BufferedPairsClass

## BufferedPairsClass Overview

The BufferedPairsClass is a FieldPairs class with a third buffer area (a "save" area). The BufferedPairsClass can compare the save area with the primary buffers, and can restore data from the save area to the primary buffers (to implement a standard "cancel" operation).

## BufferedPairsClass Concepts

The BufferedPairsClass lets you move data between field pairs, and lets you compare the field pairs to detect whether any changes occurred since the last operation.

This class provides methods that let you identify or "set up" the targeted field pairs.

**Note:** The paired fields need not be contiguous in memory, nor do they need to be part of a structure. You can build a virtual structure simply by adding a series of otherwise unrelated fields to a BufferedPairsClass object. The BufferedPairsClass methods then operate on this virtual structure.

Once the field pairs are identified, you call a single method to move all the fields in one direction (left to right), and others single methods to move all the fields in the other directions (right to left, left to buffer, etc.). You simply have to remember which entity (set of fields) you described as "left" and which entity you described as "right." Other methods compares the sets of fields and return a value to indicate whether or not they are equivalent.

## BufferedPairsClass Relationship to Other Application Builder Classes

The BufferedPairsClass is derived from the FieldPairsClass. The BrowseClass, ViewManager, and RelationManager use the FieldPairsClass and BufferedPairsClass to accomplish various tasks.

## BufferedPairsClass ABC Template Implementation

Various ABC Library objects instantiate BufferedPairsClass objects as needed; therefore, the template generated code does not directly reference the BufferedPairsClass.

## BufferedPairsClass Source Files

The BufferedPairsClass source code is installed in the Clarion \LIBSRC folder. The BufferedPairsClass source code and their respective components are contained in:

ABUTIL.INC    BufferedPairsClass declarations  
ABUTIL.CLW    BufferedPairsClass method definitions



## BufferedPairsClass Properties

### BufferedPairsClass Properties

The `BufferedPairsClass` inherits the properties of the `FieldPairsClass` from which it is derived. See *FieldPairsClass Properties* for more information.

In addition to (or instead of) the inherited properties, the `BufferedPairsClass` contains the `RealList` property.

### RealList (recognized field pairs)

**RealList**            **&FieldPairsQueue**

The **RealList** property is a reference to the structure that holds all the field pairs recognized by the `BufferedPairsClass` object.

Use the `AddPair` method to add field pairs to the `RealList` property. For each field pair, the `RealList` property includes the designated "Left" field, the designated "Right" field, plus a "Buffer" field you can use as an intermediate storage area (a save area).

The "Left," "Right," and "Buffer" designations are reflected in other `BufferedPairsClass` method names (for example, field assignment methods--`AssignLeftToRight` and `AssignRightToBuffer`) so you can easily and accurately control the movement of data between the three sets of fields.

Implementation:    During initialization, the `BufferedPairsClass` initialization method "points" the inherited `List` property to the `RealList` property so there is, in fact, only one list of fields which may be referred to as `RealList`.

`RealList` is a reference to a `QUEUE` declared in `ABUTIL.INC` as follows:

```
BufferedPairsQueue QUEUE,TYPE
Left                ANY
Right               ANY
Buffer              ANY
                   END
```

The `Init` method creates the `List` and `RealList` properties; the `Kill` method disposes of them. `AddPair` adds field pairs to the `RealList` property.

See Also:            `AddPair`, `Init`, `Kill`

## BufferedPairsClass Methods

### BufferedPairsClass Methods

The BufferedPairsClass inherits all the methods of the FieldPairsClass from which it is derived. See *FieldPairsClass Methods* for more information.

In addition to (or instead of) the inherited methods, the BufferedPairsClass contains other methods listed below.

### BufferedPairsClass Functional Organization Expected Use

As an aid to understanding the BufferedPairsClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the BufferedPairsClass methods.

#### Non-Virtual Methods

---

The non-virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the BufferedPairsClass object
AddPairv	add a field pair to the List property
Kill	shut down the BufferedPairsClass object

v These methods are also Virtual.

##### **Occasional Use:**

AssignLeftToRight	assign each "left" field to its "right" counterpart
AssignLeftToBuffer	assign each "left" field to its "buffer" counterpart
AssignRightToLeft	assign each "right" field to its "left" counterpart
AssignRightToBuffer	assign each "right" field to its "buffer" counterpart
AssignBufferToLeft	assign each "buffer" field to its "left" counterpart
AssignBufferToRight	assign each "buffer" field to its "right" counterpart
EqualLeftRight	return 1 if each left equal right, otherwise return 0
EqualLeftBuffer	return 1 if each left equal buffer, otherwise return 0
EqualRightBuffer	return 1 if right equal buffer, otherwise return 0
ClearLeft	CLEAR each "left" field
ClearRight	CLEAR each "right" field

**Inappropriate Use:**

These methods are inherited from the FieldPairsClass and typically are not used in the context of this (BufferedPairsClass) derived class.

AddItem	add a field pair from one source field
Equal	return 1 if each left equal right, otherwise return 0

**Virtual Methods**

---

Typically you will not call these methods directly. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

**AddPair (add a field pair:BufferedPairsClass)****AddPair**( *left*, *right* ), **VIRTUAL**


---

<b>AddPair</b>	Adds a field pair to the ReaList property.
<i>left</i>	The label of the "left" field of the pair. The field may be any data type, but may not be an array.
<i>right</i>	The label of the "right" field of the pair. The field may be any data type, but may not be an array.

The **AddPair** method adds a field pair to the ReaList property. A third "buffer" field is supplied for you. You may use this third "buffer" as an intermediate storage area (a save area).

The fields need not be contiguous in memory, nor do they need to be part of a structure. Therefore you can build a virtual structure simply by adding a series of otherwise unrelated fields to a BufferedPairs object. The other BufferedPairs methods then operate on this virtual structure.

Implementation: AddPair assumes the ReaList property has already been created by Init or by some other method.

By calling AddPair for a series of fields (for example, the corresponding fields in a RECORD and a QUEUE), you effectively build three virtual structures containing the fields and a (one-to-one-to-one) relationship between the structures.

Example:

```

INCLUDE('ABUTIL.INC')           !declare BufferedPairs Class
Fields &BufferedPairsClass      !declare BufferedPairs reference

Customer FILE, DRIVER('TOPSPEED'), PRE(CUST), CREATE, BINDABLE
ByNumber KEY(CUST:CustNo), NOCASE, OPT, PRIMARY
Record    RECORD, PRE()
CustNo    LONG
Name      STRING(30)
Phone     STRING(20)
          END
          END

CustQ     QUEUE
CustNo    LONG
Name      STRING(30)
Phone     STRING(20)
          END

CODE

```

---

```
Fields &= NEW BufferedPairsClass      !instantiate BufferedPairs object
Fields.Init                            !initialize BufferedPairs object
Fields.AddPair(CUST:CustNo, CustQ.CustNo) !establish CustNo pair
Fields.AddPair(CUST:Name, CustQ.Name)   !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone) !establish Phone pair
```

See Also:        Init, Reallist

## AssignBufferToLeft (copy from "buffer" fields to "left" fields)

### AssignBufferToLeft

The **AssignBufferToLeft** method copies the contents of each "buffer" field to its corresponding "left" field in the RealList property.

Implementation: The "left" field is the *first* (left) parameter of the AddPair method. The "right" field is the *second* (right) parameter of the AddPair method. The BufferedPairsClass automatically supplies the "buffer" field.

Example:

```
Fields.AddPair(CUST:Name, CustQ.Name) !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone) !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP) !establish ZIP pair
!some code
IF ~Fields.EqualRightBuffer !compare QUEUE fields to save buffer
CASE MESSAGE('Abandon Changes?',,,BUTTON:Yes+BUTTON:No)
OF BUTTON:No
Fields.AssignRightToLeft !copy changes to CUST (write) buffer
OF BUTTON:Yes
Fields.AssignBufferToLeft !restore original to CustQ (display) buffer
END
END
```

See Also: AddPair, RealList

## AssignBufferToRight (copy from "buffer" fields to "right" fields)

### AssignBufferToRight

The **AssignBufferToRight** method copies the contents of each "buffer" field to its corresponding "right" field in the RealList property.

Implementation: The "left" field is the *first* (left) parameter of the AddPair method. The "right" field is the *second* (right) parameter of the AddPair method. The BufferedPairsClass automatically supplies the "buffer" field.

Example:

```
Fields.AddPair(CUST:Name, CustQ.Name) !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone) !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP) !establish ZIP pair
!some code
IF ~Fields.EqualRightBuffer !compare QUEUE fields to save buffer
CASE MESSAGE('Abandon Changes?',,,,BUTTON:Yes+BUTTON:No)
OF BUTTON:No
Fields.AssignRightToBuffer
OF BUTTON:Yes
Fields.AssignBufferToRight
END
END
```

See Also: AddPair, RealList

## AssignLeftToBuffer (copy from "left" fields to "buffer" fields)

### AssignLeftToBuffer

The **AssignLeftToBuffer** method copies the contents of each "left" field to its corresponding "buffer" field in the RealList property.

Implementation: The "left" field is the *first* (left) parameter of the AddPair method. The "right" field is the *second* (right) parameter of the AddPair method. The BufferedPairsClass automatically supplies the "buffer" field.

Example:

```
Fields.AddPair(CUST:Name, CustQ.Name) !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone) !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP) !establish ZIP pair
!some code
IF ~Fields.EqualRightBuffer !compare QUEUE fields to save buffer
CASE MESSAGE('Abandon Changes?',,,BUTTON:Yes+BUTTON:No)
OF BUTTON:No
Fields.AssignRightToLeft
OF BUTTON:Yes
Fields.AssignLeftToBuffer
END
END
```

See Also: AddPair, RealList

## AssignRightToBuffer (copy from "right" fields to "buffer" fields)

### AssignRightToBuffer

The **AssignRightToBuffer** method copies the contents of each "right" field to its corresponding "buffer" field in the RealList property.

Implementation: The "left" field is the *first* (left) parameter of the AddPair method. The "right" field is the *second* (right) parameter of the AddPair method. The BufferedPairsClass automatically supplies the "buffer" field.

Example:

```
Fields.AddPair(CUST:Name, CustQ.Name) !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone) !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP) !establish ZIP pair
!some code
IF ~Fields.EqualRightBuffer !compare QUEUE fields to save buffer
CASE MESSAGE('Abandon Changes?',,,BUTTON:Yes+BUTTON:No)
OF BUTTON:No
Fields.AssignRightToBuffer
OF BUTTON:Yes
Fields.AssignBufferToRight
END
END
```

See Also: AddPair, RealList

## EqualLeftBuffer (compare "left" fields to "buffer" fields)

### EqualLeftBuffer

The **EqualLeftBuffer** method returns one (1) if each "left" field equals its corresponding "buffer" field; otherwise it returns zero (0).

Implementation: The "left" field is the *first* (left) parameter of the AddPair method. The "right" field is the *second* (right) parameter of the AddPair method. The BufferedPairsClass automatically supplies the "buffer" field.

Example:

```
Fields.AddPair(CUST:Name, CustQ.Name) !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone) !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP) !establish ZIP pair
!some code
IF ~Fields.EqualLeftBuffer !compare CUST fields to save buffer
CASE MESSAGE('Abandon Changes?',,,BUTTON:Yes+BUTTON:No)
OF BUTTON:No
Fields.AssignRightToLeft !copy changes to CUST (write) buffer
OF BUTTON:Yes
Fields.AssignBufferToLeft !restore original to CustQ (display) buffer
END
END
```

See Also: AddPair, RealList

## EqualRightBuffer (compare "right" fields to "buffer" fields)

### EqualRightBuffer

The **EqualRightBuffer** method returns one (1) if each "right" field equals its corresponding "buffer" field; otherwise it returns zero (0).

Implementation: The "left" field is the *first* (left) parameter of the AddPair method. The "right" field is the *second* (right) parameter of the AddPair method. The BufferedPairsClass automatically supplies the "buffer" field.

Example:

```
Fields.AddPair(CUST:Name, CustQ.Name) !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone) !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP) !establish ZIP pair
!some code
IF ~Fields.EqualRightBuffer !compare CUST fields to save buffer
CASE MESSAGE('Abandon Changes?',,,BUTTON:Yes+BUTTON:No)
OF BUTTON:No
Fields.AssignRightToLeft !copy changes to CUST (write) buffer
OF BUTTON:Yes
Fields.AssignBufferToLeft !restore original to CustQ (display) buffer
END
END
```

See Also: AddPair, RealList

## Init (initialize the BufferedPairsClass object)

### Init

The **Init** method initializes the BufferedPairsClass object.

Implementation: The Init method creates the List and RealList properties. This method "points" the inherited List property to the RealList property so there is, in fact, only one list of fields which may be referred to as RealList.

Example:

```
INCLUDE('ABUTIL.INC')           !declare BufferedPairs Class
Fields &BufferedPairsClass     !declare BufferedPairs reference

CODE
Fields &= NEW BufferedPairsClass !instantiate BufferedPairs object
Fields.Init                    !initialize BufferedPairs object
.
.
.
Fields.Kill                    !terminate BufferedPairs object
DISPOSE(Fields)                !release memory allocated for BufferedPairs object
```

See Also: Kill, List, RealList

## Kill (shut down the BufferedPairsClass object)

### Kill

The **Kill** method disposes any memory allocated during the object's lifetime and performs any other necessary termination code.

Implementation: The Kill method disposes the List and RealList properties created by the Init method.

Example:

```
INCLUDE('ABUTIL.INC')           !declare BufferedPairs Class
Fields &BufferedPairsClass      !declare BufferedPairs reference

CODE
Fields &= NEW BufferedPairsClass !instantiate BufferedPairs object
Fields.Init                     !initialize BufferedPairs object
.
.
.
Fields.Kill                     !terminate BufferedPairs object
DISPOSE(Fields)                 !release memory allocated for BufferedPairs object
```

See Also: Init, List, RealList



# ConstantClass

## ConstantClass Overview

The ConstantClass provides an easy, flexible, and efficient way to "loop through" constant data. That is, the ConstantClass parses structures like the following so you can access each (unlabeled) data item discretely:

```
Errors  GROUP,STATIC
Items   USHORT(40)                !item count
        USHORT(Msg:RebuildKey)   !begin item 1
        BYTE(Level:Notify)
        PSTRING('Invalid Key')
        USHORT(Msg:RebuildFailed) !begin item 2
        BYTE(Level:Fatal)
        PSTRING('Key was not built')
        !38 more USHORT,BYTE,PSTRING combinations
END
```

## ConstantClass Concepts

The ConstantClass parses and loads constant data such as error messages or translation text from the GROUP structure that declares the data into other data structures or memory variables (one item at a time). It can also write all the constant data into a QUEUE or a FILE.

The ConstantClass intelligently handles irregular data--you can declare the constant text data with a series of strings of varying lengths so that no space is wasted. The ConstantClass also handles a variety of numeric datatypes including BYTE, SHORT, USHORT, and LONG.

The ConstantClass provides several ways to stop processing the constant data, including a simple item count, a text match, and a read-to-the-end option.

A single ConstantClass object can process multiple GROUP structures with the same (or incremental) layouts.

## Declaring the Data

---

To use the ConstantClass, you must declare the constant data within a GROUP structure. The GROUP structure may declare a single sequence using any combination of the permitted datatypes, or a series of such sequences (the GROUP repeats the combination of datatypes as many times as needed). The ConstantClass permits CSTRING, PSTRING, BYTE, SHORT, USHORT, and LONG datatypes. The GROUP structure may contain an initial BYTE or USHORT that specifies how many times a sequence of datatypes is repeated. For example:

```
Errors  GROUP,STATIC
Items   BYTE(2)                !optional item count
        USHORT(Msg:RebuildKey) !begin first item
        BYTE(Level:Notify)
        PSTRING('Invalid Key') !end first item
        USHORT(Msg:RebuildFailed) !begin second item
        BYTE(Level:Fatal)
        PSTRING('Key not built') !end second item
        END
```

Here is another example of a structure the ConstantClass can handle:

```
Translation  GROUP,STATIC                !no item count
             PSTRING('&Across')          !default text
             PSTRING('')                !translation text
             PSTRING('Align all window Icons') !default text
             PSTRING('')                !translation text
             PSTRING('Arrange Icons')     !default text
             PSTRING('')                !translation text
             END
```

If the GROUP is declared within a procedure it must have the STATIC attribute. See the *Language Reference* for more information.

## Describing the Data

---

The ConstantClass uses two methods to describe or understand the structure of the constant data it processes: the Init method and the AddItem method. The Init method (*termination* parameter) indicates whether or not the GROUP structure declares an item count as well as the datatype of the item count (see Init). The AddItem method identifies each repeating component of the GROUP structure as well as the target variable that receives the contents of the repeating component (see AddItem).

## ConstantClass Relationship to Other Application Builder Classes

The TranslatorClass, ErrorClass, ToolbarClass, and PrintPreview classes all use the ConstantClass. These classes automatically instantiate the ConstantClass as needed.

## ConstantClass ABC Template Implementation

All ABC Library references to the ConstantClass are encapsulated with ABC Library methods--the ABC Templates do not directly reference the ConstantClass.

## ConstantClass Source Files

The ConstantClass source code is installed by default to the Clarion \LIBSRC. The specific ConstantClass source code and their respective components are contained in:

ABUTIL.INC	ConstantClass declarations
ABUTIL.CLW	ConstantClass method definitions

## ConstantClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a ConstantClass object. The example loads translation pairs from a constant GROUP into two CSTRINGs, which are then passed as parameters to another TranslatorClass method. Note that the target CSTRINGs could just as easily be fields in a QUEUE or FILE buffer.

```

    INCLUDE('ABUTIL.INC')                !declare ConstantClass, TranslatorClass
Spanish  GROUP                          !declare constant data
Items    BYTE(50)                        !item count
        PSTRING('One')                  !begin first item
        PSTRING('Uno')
        PSTRING('Two')                  !begin second item
        PSTRING('Dos')
        !48 more PSTRING pairs
    END

LangQ    QUEUE
Text     CSTRING(50)
Repl     CSTRING(50)
Done     BYTE
    END

Const    ConstantClass                  !declare & instantiate Const object
Text     CSTRING(255),AUTO              !a variable to receive a constant value
Repl     CSTRING(255),AUTO              !a variable to receive a constant value

```



## ConstantClass Properties

### TerminatorField (identify the terminating field)

**TerminatorField**      **USHORT**

The **TerminatorField** property contains a value that can be set to a number that represents the number (1 based) of the field that will tested to see if it contains the termination value. The default value is 1.

See Also:

TerminatorInclude

TerminatorValue

ConstantClass.Next

Conceptual Example

### TerminatorInclude (include matching terminator record)

7

**TerminatorInclude**      **BOOL**

The **TerminatorInclude** property, when set to true (1), will include the record that matches the terminator value in the "returned records". When set to FALSE (0), the record containing the termination value will not be included in the returned records. By default, this property is set to FALSE (0)

See Also:

TerminatorField

TerminatorValue

Conceptual Example

ConstantClass.Next

## **TerminatorValue (end of data marker)**

**TerminatorValue**      **ANY**

The **TerminatorValue** property contains a value that the ConstantClass object looks for within the constant data. When the ConstantClass object finds the TerminatorValue, it stops processing the constant data (inclusive).

The TerminatorValue property is only one of several techniques you can use to mark the end of the constant data. See the Init method for more information on this and other techniques.

Implementation:      The Init method CLEARS the TerminatorValue property; therefore, you should set the TerminatorValue property *after* the Init method executes.

The Next() method returns Level:Notify when the characters of the constant data matches the value of the TerminatorValue property. The Next(FILE) and Next(QUEUE) methods stop processing when the ConstantClass object finds the TerminatorValue.

The TerminatorField property can be set to a number that represents the number (1 based) of the field that will tested to see if it contains the termination value.

When the TerminatorInclude property is set to true, this will include the record that matches the terminator value in the 'returned records'. When false, the record containing termination value will not be included in the returned records. By default this is set to false.

See Also:              Init, Next

## ConstantClass Methods

### ConstantClass Functional Organization--Expected Use

As an aid to understanding the ConstantClass, it is useful to organize the its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the ConstantClass methods.

#### Non-Virtual Methods

---

The non-virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the ConstantClass object
AddItem	set constant datatype and target variable
Set	set the constant data to process
Kill	shut down the ConstantClass object

##### **Mainstream Use:**

Next	copy one or all constant items to targets
------	---

##### **Occasional Use:**

Reset	reset the object to beginning of the constant data
-------	--

#### Virtual Methods

---

The ConstantClass has no virtual methods.

**AddItem (set constant datatype and target variable)****AddItem**( *datatype*, *target* )**AddItem** Sets the (repeating) constant datatype and its corresponding target variable.*datatype* An integer constant, variable, EQUATE or expression that identifies the datatype of a repeating constant within the constant GROUP structure. Valid *datatype* values are ConstType:Cstring, ConstType:Pstring, ConstType:Byte, ConstType:Short, ConstType:Ushort, and ConstType:Long.*target* The label of the variable that receives the constant value.

The **AddItem** method sets a (repeating) constant datatype and its corresponding target variable. Use multiple calls to the AddItem method to "describe" the constant data structure as well as the target variables that receive the constant data.

Implementation: You should call AddItem for each repeating datatype declared in the constant GROUP structure. The Next method processes the constant data items described by the AddItem calls. EQUATEs for the *datatype* parameter are declared in ABUTIL.INC:

```

ITEMIZE(1),PRE(ConstType)
First EQUATE
Cstring EQUATE(ConstType:First)
Pstring EQUATE
Byte EQUATE          !1 byte unsigned integer
Short EQUATE         !2 byte signed integer
Ushort EQUATE        !2 byte unsigned interger
Long EQUATE          !4 byte signed integer
Last EQUATE(ConstType:Long)
END

```

Example:

```

Errors GROUP,STATIC
    USHORT(Msg:RebuildKey)      !begin first item
    PSTRING('Invalid Key')     !end first item
    USHORT(Msg:RebuildFailed)  !begin second item
    PSTRING('Key not built')   !end second item
END
ErrorQ QUEUE
ID LONG
Text CSTRING(255)
END
CODE
!The following describes the Errors GROUP and its corrsponding target variables
Const.AddItem(ConstType:Ushort, ErrorQ.ID) !USHORT constant maps to error ID
Const.AddItem(ConstType:PString, ErrorQ.Text)!PSTRING constant maps to error text

```

See Also: Next

**Init (initialize the ConstantClass object)****Init** (*termination*)

---

<b>Init</b>	Initializes the ConstantClass object.
<i>termination</i>	An integer constant, variable, EQUATE or expression that controls when the Next(FILE) and Next(QUEUE) methods stop processing the constant data. If omitted, <i>termination</i> defaults to Term:Ushort. Valid <i>termination</i> values are Term:Ushort, Term:Byte, Term:EndGroup, and Term:FieldValue

The **Init** method initializes the ConstantClass object.

The termination parameter provides two important pieces of information to the ConstantClass object: it tells the ConstantClass object whether there is a non-repeating item count declared at the beginning of the constant data (describes the structure of the constant data), and it tells the ConstantClass object how to recognize the end of the constant data. Valid *termination* values are:

Term:Ushort	The GROUP declares a USHORT containing the item count--stops reading when item count reached.
Term:Byte	The GROUP declares a BYTE containing the item count--stops reading when item count reached.
Term:EndGroup	The GROUP does not declare an item count—stops reading at end of GROUP structure.
Term:FieldValue	The GROUP does not declare an item count—stops reading when it finds the TerminatorValue within the constant data.

Implementation: The Init method CLEARs the TerminatorValue property. The Init method allocates memory and should always be paired with the Kill method, which frees the memory.

EQUATEs for the *termination* parameter are declared in ABUTIL.INC:

```

ITEMIZE(1),PRE(Term)
EndGroup EQUATE !Stops reading at end of GROUP
UShort EQUATE !Reads number of items specified by USHORT at start of group
Byte EQUATE !Reads number of items specified by BYTE at start of group
FieldValue EQUATE !Stops when specified value is found in first AddItem field,
!only first 32 chars are compared
END

```

Example:

```

Const.Init(Term:BYTE)           !Initialize the Const object,
                                !the first BYTE contains item count

Const.AddItem(ConstType:PString, LangQ.Text)

```

!Describe constant structure and variables to accept the values  
Const.AddItem(ConstType:PString, LangQ.Repl)

Const.Set(Spanish)                   !pass the constant data to Const object  
Const.Next(LangQ)                   !copy all constant items to the LangQ  
Const.Kill                           !shut down Const object

See Also:           Kill, Next, TerminatorValue

## Kill (shut down the ConstantClass object)

### Kill

The **Kill** method frees any memory allocated during the life of the object and does any other required termination code.

Example:

```
Const.Init(Term:BYTE)    !Initialize the Const object,
                        ! the first BYTE contains item count
Const.AddItem(ConstType:PString, LangQ.Text) !Describe constant structure and
Const.AddItem(ConstType:PString, LangQ.Repl) !variables to accept the values
Const.Set(Spanish)      !pass the constant data to Const object
Const.Next(LangQ)       !copy all constant items to the LangQ
Const.Kill              !shut down Const object
```



---

```
Const.AddItem(ConstType:PString, LangQ.Text) !Describe constant structure and
Const.AddItem(ConstType:PString, LangQ.Repl) ! variables to accept the values
Const.Set(Spanish)      !pass the constant data to Const object
Const.Next(LangQ)      !copy all constant items to the LangQ
Const.Kill              !shut down Const object
```

See Also:           AddItem, Init, Next

**Next (copy next constant item to targets)****Next, PROC**

The **Next** method copies the next constant item to its respective targets (as defined by the AddItem method) and returns a value indicating whether the item was copied. A return value of Level:Benign indicates the item was copied successfully; a return value of Level:Notify indicates the item was not copied because the end of the constant data, as defined by the Init method, was reached.

Prior calls to the AddItem method determine the makeup of the item as well as the target variables that receive the item.

Implementation: The Next method parses a single item in the constant data, performing any required datatype conversions, and increments appropriate internal counters.

Return Data Type: BYTE

Example:

```

Spanish GROUP          !declare constant data
Items  BYTE(50)        !item count
        PSTRING('One') !begin first item
        PSTRING('Uno')
        PSTRING('Two') !begin second item
        PSTRING('Dos')
        !48 more PSTRING pairs
        END

Const  ConstantClass  !declare & instantiate Const object
Text  CSTRING(255),AUTO !a variable to receive a constant value
Repl  CSTRING(255),AUTO !a variable to receive a constant value
CODE
!process items one-at-a-time
Const.Init(Term:BYTE)  !initialize the Const object,
                       ! the first BYTE contains item count
Const.AddItem(ConstType:PString, Text)  !Describe constant structure and
Const.AddItem(ConstType:PString, Repl)  ! variables to accept the values
Const.Set(Spanish)     !pass the constant data to Const object
LOOP WHILE Const.Next()=Level:Benign    !copy constant data one item at a time
!do something with Text and Repl        ! to respective AddItem target variables
END
Const.Kill              !shut down Const object

```

See Also: AddItem, Init, Next

---

**Reset (reset the object to the beginning of the constant data)****Reset**

The **Reset** method resets internal counters to start processing constant data from the beginning.

Implementation: The Set, Next(FILE) and Next(Queue) methods call the Reset method. Typically you will not call this method.

Example:

```
ConstantClass.Set PROCEDURE(*STRING Src)
CODE
DISPOSE(SELF.Str)
SELF.Str &= NEW STRING(LEN(Src))
SELF.Str = Src
SELF.SourceSize=LEN(SELF.Str)
SELF.Reset
```

**Set (set the constant data to process)****Set**( *datasource* )**Set** Sets the GROUP structure to process.*datasource* The label of the GROUP structure the ConstantClass object processes.The **Set** method sets the GROUP structure to process.Implementation: The Set method takes a copy of *datasource* and calls the Reset method to reset internal counters to process *datasource* copy from the beginning.

Example:

```

Spanish  GROUP          !declare constant data
Items    BYTE(50)        !item count
          PSTRING('One') !begin first item
          PSTRING('Uno')
          PSTRING('Two') !begin second item
          PSTRING('Dos')
          !48 more PSTRING pairs
          END

```

```

LangQ    QUEUE
Text     CSTRING(50)
Repl     CSTRING(50)
          END

```

```

Const  ConstantClass  !declare & instantiate Const object

```

```

CODE

```

```

!process all items at a time

```

```

Const.Init(Term:BYTE)      !re initialize the Const object,
                          ! the first BYTE contains item count

```

```

Const.AddItem(ConstType:PString, LangQ.Text) !Describe constant structure and

```

```

Const.AddItem(ConstType:PString, LangQ.Repl) ! variables to accept the values

```

```

Const.Set(Spanish)        !pass the constant data to Const object

```

```

Const.Next(LangQ)         !copy all constant items to the LangQ

```

```

Const.Kill                !shut down Const object

```

See Also: **Reset**

## Crystal8 Class

Seagate Software's Crystal Reports is one of the leading report writers delivering Windows reports. For more information on this product see Seagate Software at [www.seagatesoftware.com](http://www.seagatesoftware.com).

Clarion's Crystal Report interface is comprised of templates, libraries, and DLLs that communicate with Seagate's Crystal Reports, version 8. The DLL is accessed by a Class Interface and is hooked to your application using simple standard Clarion code. This interface allows a seamless integration of previously defined Crystal reports within a Clarion application. The Crystal report engine accesses data and creates the report. The report can be previewed in a Clarion window.

Clarion's Crystal Reports implementation is compatible with both the ABC and Legacy templates. It can only be used in 32-bit applications.

### Crystal8 Class Concepts

Clarion's Crystal Reports implementation is a DLL that communicates with Seagate Software's Crystal Reports report writer. The Crystal 8 Class accesses the DLL. There are several templates available which make the interface to the report writer easily accessible from your Clarion program. Previewing and/or printing reports are simple.

### Relationship to Other Application Builder Classes

The Crystal8 class works independently of all other ABC classes.

### ABC Template Implementation

The PreviewCrystalReport and PrintCrystalReport template extensions instantiate an object based on the object name specified by either of these extensions. The object is instantiated in the procedure where the extension exists.

### Crystal8 Source Files

The Crystal8 class declarations are installed by default to the Clarion \LIBSRC folder. The Crystal8 component is distributed as a LIB/DLL, therefore the source code for the methods is not available. However, the methods are defined in this chapter and may be implemented in applications provided the required LIB/DLL is available at runtime.

C60cr8.INC	Crystal Class Definition
C60cr8l.INC	Crystal Class Definition Local Compile
C60cr8.dll	Crystal DLL
C60cr8.lib	Crystal LIB
C60cr8L.lib	Crystal Local LIB

## Crystal8 Class Properties

There are no properties associated with the Crystal8 Class Library.

## Crystal8 Methods

### AllowPrompt (prompt for runtime parameter data)

**AllowPrompt**(| *allowpromptflag* |)

**AllowPrompt** Allow report runtime prompting for Crystal Parameter fields.

*Allowpromptflag* An integer constant, variable, EQUATE, or expression that specifies whether the report will prompt for runtime parameter fields. A value of one (1) is used to allow prompting; a value of zero (0) is used to disallow field prompting.

The AllowPrompt method can conditionally allow the Crystal report that is being previewed or printed to prompt for runtime parameter fields. These parameter fields must be defined in the Crystal report. This method returns a BYTE representing the value of *allowpromptflag*.

**Return Data Type:** BYTE

**Example:**

```
oCrystal8.AllowPrompt(1)
```

## CanDrillDown(allow Crystal drill down support )

**CanDrillDown**(| *candrilldown* |)

**CanDrillDown** Allows use of Crystal Report's drill down feature.

*candrilldown* An integer constant, variable, EQUATE, or expression that specifies whether the report make use of Crystal's drill down feature. A value of one (1) allows drill down to be used; a value of zero (0) removes the ability to drill down.

The CanDrillDown method allows a Crystal Report to use the defined drill down support. For more information on Crystal's drill down feature refer to the Crystal Report documentation. This method returns a BYTE representing the value of *candrilldown*.

**Return Data Type:** BYTE

Example:

```
oCrystal18.CanDrillDown(1)
```

## HasCancelButton (display cancel button on report preview)

### **HasCancelButton** (| *hascancelbutton* |)

**HasCancelButton** Allow a cancel button on the report preview window.

*hascancelbutton* An integer constant, variable, EQUATE, or expression that specifies whether a cancel button will appear on the report's preview window. A value of one (1) displays the cancel button; a value of zero (0) does not display the cancel button.

The HasCancelButton method is used to optionally display a cancel button on the report preview window. This method returns a BYTE representing the value of *hascancelbutton*.

**Return Data Type:** BYTE

### **Example:**

```
oCrystal8.HasCancelButton(1)
```

## HasCloseButton (display close button on report preview)

**HasCloseButton** (| *hasclosebutton* |)

<b>HasCloseButton</b>	Allow a close button on the report preview window.
<i>hasclosebutton</i>	An integer constant, variable, EQUATE, or expression that specifies whether a close button will appear on the reports preview window. A value of one (1) displays the close button; a value of zero (0) does not display the close button.

The HasCloseButton method is used to optionally display a close button on the report preview window. This method returns a BYTE representing the value of *hasclosebutton*.

**Return Data Type:** BYTE

**Example:**

```
oCrystal18.HasCloseButton(1)
```

## HasExportButton (display export button on report preview)

**HasExportButton** (| *hasexportbutton* |)

**HasExportButton** Allow an export button on the report preview window.

*hasexportbutton* An integer constant, variable, EQUATE, or expression that specifies whether an export button will appear on the reports preview window. A value of one (1) displays the export button; a value of zero (0) does not display the export button.

The HasExportButton method is used to optionally display an export button on the report preview window. This method returns a BYTE representing the value of *hasexportbutton*.

**Return Data Type:** BYTE

### Example:

```
oCrystal8.HasExportButton(1)
```

## HasLaunchButton (display launch button on report preview)

### **HasLaunchButton** (| *haslaunchbutton* |)

**HasLaunchButton** Allow a launch button on the report preview window.

*haslaunchbutton* An integer constant, variable, EQUATE, or expression that specifies whether a launch button will appear on the reports preview window. A value of one (1) displays the launch button; a value of zero (0) does not display the launch button.

The HasLaunchButton method is used to optionally display a launch button on the report preview window. This method returns a BYTE representing the value of haslaunchbutton. The launch button is used to launch the Seagate Analysis tool.

**Return Data Type:** BYTE

### **Example:**

```
oCrystal18.HasLaunchButton(1)
```

## HasNavigationControls (display navigation controls on report preview)

**HasNavigationControls** (| *hasnavigationcontrols* |)

HasNavigationControls	Allows navigation controls on the report preview window.
<i>hasnavigationcontrols</i>	An integer constant, variable, EQUATE, or expression that specifies whether the navigation controls will appear on the report's preview window. A value of one (1) displays the navigation controls; a value of zero (0) does not display the navigation controls.

The HasNavigationControls method is used to optionally display navigation controls on the report preview window. This method returns a BYTE representing the value of *hasnavigationcontrols*. Navigation controls are used to navigate through a report, immediately to the beginning, end or anywhere in between.

**Return Data Type:** BYTE

**Example:**

```
oCrystal8.HasNavigationControls(1)
```

## HasPrintButton (display print button on report preview)

**HasPrintButton** (| *hasprintbutton* |)

**HasPrintButton** Allows a print button on the report preview window.

*Hasprintbutton* An integer constant, variable, EQUATE, or expression that specifies whether a print button will appear on the report's preview window. A value of one (1) displays the print button; a value of zero (0) does not display the print button.

The HasPrintButton method is used to optionally display a print button on the report preview window. This method returns a BYTE representing the value of *hasprintbutton*.

**Return Data Type:** BYTE

**Example:**

```
oCrystal8.HasPrintButton(1)
```

## HasPrintSetupButton (display print setup button on report preview)

**HasPrintSetupButton** (| *hasprintsetupbutton* |)

**HasPrintSetupButton** Allows a print setup button on the report preview window.

*hasprintsetupbutton* An integer constant, variable, EQUATE, or expression that specifies whether a print setup button will appear on the reports preview window. A value of one (1) displays the print setup button; a value of zero (0) does not display the print setup button.

The HasPrintSetupButton method is used to optionally display a print setup button on the report preview window. This method returns a BYTE representing the value of *hasprintsetupbutton*.

**Return Data Type:** BYTE

**Example:**

```
oCrystal8.HasPrintSetupButton(1)
```

## HasProgressControls (display progress controls on report preview)

**HasProgressControls** (| *hasprogresscontrols* |)

**HasProgressControls** Allows navigation controls on the report preview window.

*Hasprogresscontrols* An integer constant, variable, EQUATE, or expression that specifies whether the progress controls will appear on the reports preview window. A value of one (1) displays the progress controls; a value of zero (0) does not display the progress controls.

The HasProgressControls method is used to optionally display progress controls on the report preview window. This method returns a BYTE representing the value of *hasprogresscontrol*. The Progress controls display the progress of the report when it is running. It displays records read, records selected, etc.).

**Return Data Type:** BYTE

### Example:

```
oCrystal8.HasProgressControls(1)
```

## HasRefreshButton (display refresh button on report preview)

**HasRefreshButton** (| *hasrefreshbutton* |)

**HasRefreshButton** Allows a refresh button on the report preview window.

*hasrefreshbutton* An integer constant, variable, EQUATE, or expression that specifies whether a refresh button will appear on the report's preview window. A value of one (1) displays the refresh button; a value of zero (0) does not display the refresh button.

The HasRefreshButton method is used to optionally display a refresh button on the report preview window. This method returns a BYTE representing the value of *hasrefreshbutton*.

**Return Data Type:** BYTE

**Example:**

```
oCrystal8.HasRefreshButton(1)
```

## HasSearchButton (display search button on report preview)

### **HasSearchButton** (| *hassearchbutton* |)

**HasSearchButton** Allows a search button on the report preview window.

*hassearchbutton* An integer constant, variable, EQUATE, or expression that specifies whether a search button will appear on the reports preview window. A value of one (1) displays the search button; a value of zero (0) does not display the search button.

The HasSearchButton method is used to optionally display a search button on the report preview window. This method returns a BYTE representing the value of *hassearchbutton*.

**Return Data Type:** BYTE

#### **Example:**

```
oCrystal18.HasSearchButton(1)
```

## HasZoomControl (display zoom control on report preview)

### **HasZoomControl** (| *haszoomcontrol* |)

**HasZoomControl**      Allows a zoom control on the report preview window.

*haszoomcontrol*      An integer constant, variable, EQUATE, or expression that specifies whether a zoom button will appear on the report's preview window. A value of one (1) displays the zoom control; a value of zero (0) does not display the zoom control.

The HasZoomControl method is used to optionally display a zoom control on the report preview window. This method returns a BYTE representing the value of *haszoomcontrol*.

**Return Data Type:**      BYTE

#### **Example:**

```
oCrystal8.HasZoomControl(1)
```

## Init (initialize Crystal8 object)

### **Init** (*reportname*)

**Init** Initialize the Crystal8 object.

*reportname* A string constant, variable, EQUATE, or expression containing the report name. This report name is used when previewing a report. It is also the window caption (text).

The Init method initializes the Crystal8 object. This method also sets the title of the preview window, if the report is previewed. A BYTE is returned from this method and represents whether the report engine is successfully initialized.

**Return Data Type:** BYTE

### **Example:**

```
oCrystal8.Init( ReportPathName )
```

## Kill (shut down Crystal8 object)

### **Kill**

The Kill method shuts down the Crystal8 object, releasing any memory allocated during the lifetime of the object.

### **Example:**

```
oCrystal8.Kill()
```

## Preview (preview a Crystal Report)

	<b>Preview</b> ( <i>l windowtitle, initstate, frame, icon, systemmenu, maximizebox, 3dflag</i> )
Preview	Preview a Crystal Report.
<i>windowtitle</i>	A string constant, variable, EQUATE, or expression containing the text to display in the report preview window's title bar. This parameter is optional.
<i>initstate</i>	A string constant, variable, EQUATE, or expression containing an <b>N</b> , <b>M</b> , or <b>I</b> . Use <b>N</b> (Normal) to display the preview window at the default size. Use <b>M</b> (Maximized) to display the preview window in a maximized state. Use <b>I</b> (Iconized) to display the preview window in an iconized state. This parameter is optional.
<i>frame</i>	A string constant, variable, EQUATE, or expression containing an <b>S</b> , <b>D</b> , <b>R</b> , or <b>N</b> . Use <b>S</b> to give the preview window a single pixel frame. Use <b>D</b> to give the preview window a thick frame. Use <b>R</b> to give the preview window a thick but resizable frame. Use <b>N</b> to give the preview window no frame under Windows 3.1 or a single pixel frame under Window 95/98. This parameter is optional.
<i>icon</i>	A string constant, variable, EQUATE, or expression containing an icon filename. By specifying an icon file, a minimize button is automatically placed on the preview window. This parameter is optional.
<i>systemmenu</i>	An integer constant, variable, EQUATE, or expression that specifies whether the preview window will contain a system menu. A value of TRUE will give the preview window a system menu. A value of FALSE (the default value) will not include the system menu on the preview window. This parameter is optional.
<i>maximizebox</i>	An integer constant, variable, EQUATE, or expression that specifies whether the preview window will contain a maximize button. A value of TRUE (the default value) will place the maximize button on the preview window. A value of FALSE will not include the maximize box on the preview window. This parameter is optional.
<i>3dflag</i>	An integer constant, variable, EQUATE, or expression that specifies whether the preview window will have a 3D look. The 3D look provides the window with a gray background and chiseled control look. A value of TRUE (the default value) will provide the preview window with the 3D look. A value of FALSE will not provide the preview window with the 3d look. This parameter is optional.

The Preview method is used to preview a Crystal report within a Clarion window. This method supports several preview window options.

### Example:

```
oCrystal8.Preview( 'My Report', 'I', 'R', , 0, 1, 1 )
```

## `_Print` (print a Crystal Report)

	<code>_Print(  <i>copies</i>, <i>printersetup</i>  )</code>
<code>_Print</code>	Print a Crystal Report.
<code>copies</code>	An integer constant, variable, EQUATE, or expression that specifies the number of copies of the report to print. The default for this parameter is 1. This parameter is optional.
<code>printersetup</code>	An integer constant, variable, EQUATE, or expression that specifies whether the Printer Setup dialog is displayed before sending the report to the printer. Specifying TRUE or 1 for this parameter will cause the Printer Setup dialog to be displayed; a value of FALSE or 0 (the default value) will allow the report to go directly to the printer. This parameter is optional.

The `_Print` method prints a Crystal report directly to the printer without any option to preview the report. The printer setup dialog is optional before the report is sent to the printer.

### Example:

```
oCrystal8._Print( 1, 1 )
```

## Query (retrieve or set the SQL data query)

**Query**( | *querystring* )

**Query**            Set or retrieve the SQL data query.

*querystring*    A string constant, variable, EQUATE, or expression containing the SQL query to be sent to the SQL data source. This parameter is optional.

The Query method is used to either get or set the SQL query. If the *querystring* is omitted from the method call, the current query is retrieved.

**Return Data Type:**        STRING

### Example:

```
formula = oCrystal8.Query()!retrieve query into formula variable
formula = '{{Customer.Country} in ["Australia"]' ! SQL query
oCrystal8.Query( formula )        ! Set the query
```

## SelectionFormula (retrieve or set the Crystal formula )

**SelectionFormula**(| *formulastring* |)

**SelectionFormula**      Set or retrieve the Crystal formula.

*Formulastring*      A string constant, variable, EQUATE, or expression containing the Crystal formula. This parameter is optional.

The SelectionFormula method is used to either get or set the report's formula used to limit retrieved records. If the *formulastring* is omitted from the method call, the current formula is retrieved.

**Return Data Type:**      STRING

### Example:

```
formula = oCrystal8.SelectionFormula()!retrieve selection formula into formula v
formula = '{{Customer.Country} in ["Australia"]' ! SQL query
oCrystal8.SelectionFormula( formula )    ! Set the query
```

## ShowDocumentTips (show tips on document in the preview window)

**ShowDocumentTips**(*showdocumenttips*)

ShowDocumentTips    Display tooltips in the document being previewed.

*showdocumenttips*    An integer constant, variable, EQUATE, or expression that specifies whether to enable tooltips on the document in the report preview window. A value of one (1) indicates that tooltips will be shown. A value of zero (0) indicates that tooltips will not be displayed on the document in the preview window. This is the default if the parameter is omitted.

The ShowDocumentTips method is used to enable tooltips on the document being previewed in the report preview window. This method returns a BYTE representing the value of showdocumenttips.

**Return Data Type:**        BYTE

### Example:

```
oCrystal18.ShowDocumentTips(1)
```

## ShowReportControls (show print controls)

### ShowReportControls (| *showreportcontrols* |)

ShowReportControls    Display tooltips in the document being previewed.

*showreportcontrols*    An integer constant, variable, EQUATE, or expression that specifies whether the print controls are displayed. A value of one (1) will cause the print controls to be displayed. A value of zero (0) indicates that will hide the print controls. This parameter is optional and defaults to TRUE if omitted.

The ShowReportControls method is used to display the print controls. The print controls include the First, Previous, Next, and Last Page buttons as well as the buttons for Cancel, Close, Export, and Print to Printer. This method returns a BYTE representing the value of *showreportcontrols*.

**Return Data Type:**        BYTE

#### Example:

```
oCrystal8.ShowReportControls(1)
```

## ShowToolBarTips (show tips on preview window toolbar)

**ShowToolBarTips**(| *showtooltips* |)

ShowToolBarTips      Display tooltips in the preview window's toolbar.

*showtooltips*      An integer constant, variable, EQUATE, or expression that specifies whether to enable tooltips on the toolbar in the report preview window. A value of one (1) indicates that tooltips will be shown. This is the default if the parameter is omitted. A value of zero (0) indicates that tooltips will not be displayed in the report preview window.

The ShowToolBarTips method is used to enable tooltips on the toolbar of the report preview window. This method returns a BYTE representing the value of *showtooltips*.

**Return Data Type:**      BYTE

**Example:**

```
oCrystal8.ShowToolBarTips(1)
```

---

## cwRTF Class

### cwRTF Overview

Clarion's implementation of Rich Text Format (RTF) is derived from Microsoft's Rich Edit Control, version 3.0. Rich Text provides a storage format for text that keeps the text compatible among different operating systems, and software applications. A rich text control allows a user to enter, edit, format, print, and save text.

### cwRTF Class Concepts

The features of Clarion's RTF control include the support for colorized text, fonts, font sizes, bold, italics, underlining, paragraph justification, bullets, undo, redo, cut, copy and paste clipboard functionality, search, print, new open and save to file, ruler, and tab settings.

Clarion's Rich Text support is compatible with both the ABC and Legacy templates. It may only be used in a 32-bit application. Multiple RTF controls may be used on a single window.

Clarion's Rich Text Classes and Templates are wrappers around Microsoft's Rich Edit Control DLL's. The presence of RichEdxx.DLL is required in the Windows/System directory. There are three versions of this DLL. The following list of operating systems shows which DLL each system installs and supports. Other programs may install a newer version of the Rich Edit DLL.

#### Windows NT/Windows 2000

Microsoft® Windows NT® version 4.0 includes Rich Edit 1.0 and 2.0. Microsoft Windows® 2000 includes Rich Edit 3.0 with a Rich Edit 1.0 emulator.

#### Windows 98

Windows 98 includes Rich Edit 1.0 and 2.0.

#### Windows 95

Windows 95 includes only Rich Edit 1.0. However, Riched20.dll is compatible with Windows 95 and may be installed if an application that uses Rich Edit 2.0 has been installed.

## cwRTF Relationship to Other Application Builder Classes

The cwRTF class works independantly of all other ABC classes.

## cwRTF ABC Template Implementation

Once the cwRTF procedure template extension is added to a procedure, the templates instantiate a cwRTF object into the generated code for the procedure. This is also where the cwRTF object is initialized.

## cwRTF Source Files

The cwRTF class declarations are installed by default to the Clarion \LIBSRC folder. The cwRTF component is distributed as a LIB/DLL, therefore the source code for the methods is not available. However, the methods are defined in this chapter and may be implemented in applications provided the required LIB/DLL is available at runtime.

C60RTF.INC	Rich Text Class Definition
C60RTFL.INC	Rich Text Class Definition Local Compile
C60RTF.dll	Rich Text DLL
C60RTF.lib	Rich Text LIB
C60RTFL.lib	Rich Text Local LIB

## cwRTF Properties

### cwRTF Properties

The cwRTF class contains many properties that although are not PRIVATE, should not be used. These methods are used internally.

### hRTFWindow(RTF control handle)

#### **hRTFWindow LONG**

The **hRTFWindow** property is a handle to the RTF control.

## cwRTF Methods

### AlignParaCenter (center paragraph)

#### **AlignParaCenter**

The **AlignParaCenter** method is used to center the current or selected paragraph of text within the rich text control. The current paragraph is the one that the cursor is within.

Example:

```
oRTF_RTFTextBox.AlignParaCenter() !Center paragraph
```

### AlignParaLeft (left justify paragraph)

#### **AlignParaLeft**

The **AlignParaLeft** method is used to left justify the current or selected paragraph of text within the rich text control. The current paragraph is the one that the cursor is within.

Example:

```
oRTF_RTFTextBox.AlignParaLeft() !Left justify paragraph
```

### AlignParaRight (right justify paragraph)

#### **AlignParaRight**

The **AlignParaRight** method is used to right justify the current or selected paragraph of text within the rich text control. The current paragraph is the one that the cursor is within.

Example:

```
oRTF_RTFTextBox.AlignParaRight() !Right justify paragraph
```

## ChangeFontStyle (set current font style)

**ChangeFontStyle**(*fontstyle*)

**ChangeFontStyle** Sets the current font style.

*fontstyle* An integer constant, variable, EQUATE, or expression that contains a value that represents a fontstyle. Valid fontstyle equates can be found in EQUATES.CLW.

The **ChangeFontStyle** method is used to set the current font style. The style of a font represents the strike weight and style of the font.

Example:

```
oRTF_RTFTextBox.ChangeFontStyle(FONT:Italic ) !Font Style
oRTF_RTFTextBox.ChangeFontStyle(loc:fontstyle ) !Variable Font Style
```

## CanRedo (check for redo data)

**CanRedo**

The **CanRedo** method is used to see if there is data in the redo buffer. A return value of one (1 or True) indicates there is redo data in the buffer, a redo operation is available; a return value of zero (0 or False) is returned if there is no redo data in the buffer, a redo operation is not available.

Return Data Type: BYTE

Example:

```
loc:redo =oRTF_RTFTextBox.CanRedo() !Check for data in the Redo buffer
```

## **CanUndo (check for undo data)**

### **CanUndo**

The **CanUndo** method is used to see if there is data in the undo buffer. A return value of one (1 or True) indicates there is undo data in the buffer, an undo operation is available; a return value of zero (0 or False) is returned if there is no undo data in the buffer, an undo operation is not available.

Return Data Type:        BYTE

Example:

```
loc:undo =oRTF_RTFTextBox.CanUndo() !Check for data in the Undo buffer
```

## **Color (set text color)**

### **Color**

The **Color** method calls the Windows standard color choice dialog box in order to allow a user to choose a color. If text is selected prior to calling the color dialog, the selected text will be colored the selected color.

Example:

```
oRTF_RTFTextBox.Color()!Color text
```

## **Copy (copy selected text to clipboard)**

### **Copy**

The **Copy** method is used to copy selected text from the rich text control into the clipboard.

Example:

```
oRTF_RTFTextBox.Copy() !Copy text to clipboard
```

**Cut (cut selected text )****Cut**

The **Cut** method is used to cut the selected text from the rich text control into the clipboard.

Example:

```
oRTF_RTFTextBox.Cut() !Cut text to clipboard
```

**Find (set current font style)****Find([findtext])**

<b>Find</b>	Search for text within the rich text field.
<i>findtext</i>	A string constant, variable, EQUATE, or expression containing the text for search for. If this is omitted the Find dialog is presented.

The **Find** method is used to search the Rich Text field for the text specified in *findtext*. If no *findtext* is specified, the Find dialog is presented. The user can type in the text to search for as well as specify other search options such as Whole Words Only, Case Sensitive Search, and Search Direction (up or down).

This method returns a value indicating the character position where the text was found.

Return Data Type:           LONG

Example:

```
loc:position =oRTF_RTFTextBox.Find(loc:find ) ! Search for contents of
loc:find
loc:position =oRTF_RTFTextBox.Find('mytext ' ) ! Search for 'mytext'
loc:position =oRTF_RTFTextBox.Find()           ! Show find dialog
```

## **FlatButtons (use flat button style)**

**FlatButtons**(*buttonstatus*)

<b>FlatButtons</b>	Set the FormatBar and Toolbar buttons to appear in a flat or raised mode.
<i>buttonstatus</i>	A numeric constant, variable, EQUATE, or expression that indicates whether to use flat or raised buttons. A TRUE value is passed to the method to use flat buttons. A FALSE value is passed to the FlatButtons method to raised buttons.

The **FlatButtons** method is used to set the FormatBar and Toolbar buttons to appear in a flat or raised mode. A flat button remains flat until the mouse cursor passes over it.

Example:

```
oRTF_RTFTextBox.FlatButtons(1 ) !Flat buttons  
oRTF_RTFTextBox.FlatButtons(0 ) !Raised buttons
```

## Font (apply font attributes)

**Font**(*[fontname]*, *[fontsize]*, *[fontcolor]*, *[fontstyle]*)

<b>Font</b>	Apply font attributes to rich text.
<i>fontname</i>	A string constant or variable containing a font name.
<i>fontsize</i>	An integer constant containing the size (in points) of the font.
<i>fontcolor</i>	An integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, for an EQUATE for a standard Windows color value.
<i>fontstyle</i>	An integer constant, constant expression, or EQUATE specifying the strike weight and style of the font.

The **Font** method is used to set font attributes at the current insertion point or apply it to the currently selected text. If all of the parameters for this method are omitted the Windows standard font dialog box is called in order to allow a user to set standard font attributes.

Example:

```
!Using variables  
oRTF_RTFTextBox.Font(FontName,FontSize,FontColor,FontStyle )  
!Specific Font Attributes  
oRTF_RTFTextBox.Font('Arial ',10,COLOR:Red,FONT:Underline )  
!Font Dialog  
oRTF_RTFTextBox.Font()
```

## GetText (copy text to variable)

**GetText**(*text*, [*startpos*], [*endpos*])

<b>GetText</b>	Copy indicated text to a variable.
<i>text</i>	A string variable that will receive the specified text. The highlighted text will be assigned to the variable if no starting and ending positions are specified. If positions are specified the text specified by these positions will be assigned to the string variable.
<i>startpos</i>	An integer constant or variable that specifies the starting character position to copy text from.
<i>endpos</i>	An integer constant or variable that specifies the ending character position to copy text from.

The **GetText** method is used to copy the specified text to a string or variable. The method can return a result that specifies the length (number of characters) of the copied text.

Return Data Type:      LONG

Example:

```
oRTF_RTFTextBox.GetText(loc:find, ,)                   !read text into loc:find
loc:length =oRTF_RTFTextBox.GetText(loc:find,,) !return string length

!read text into loc:find from specified position
oRTF_RTFTextBox.GetText(loc:find,10,20)
```

**Init (initialize the cwRTF object)**

**Init**(*window*, *textbox*, [*rulerflag*], [*toolbarflag*], [*formatbarflag*])

<b>Init</b>	Initialize the cwRTF object.
<i>window</i>	The label of the WINDOW that contains the rich text control.
<i>textbox</i>	A numeric constant, variable, EQUATE, or expression containing the control number (FEQ) of the rich text control.
<i>rulerflag</i>	A numeric constant, variable, EQUATE, or expression that indicates whether to initially display the RulerBar with the rich text control.
<i>toolbarflag</i>	A numeric constant, variable, EQUATE, or expression that indicates whether to initially display the ToolBar with the rich text control.
<i>formatbarflag</i>	A numeric constant, variable, EQUATE, or expression that indicates whether to initially display the FormatBar with the rich text control.

The **Init** method initializes the cwRTF object. This method also sets the initial states of the RulerBar, ToolBar, and FormatBar. By default all three bars are displayed.

Example:

```
oRTF_RTFTextBox.Init( Window, ?RTFTextBox, 1, 1, 1 )
```

**IsDirty (indicates modified data)****IsDirty**(*saveflag*)

<b>IsDirty</b>	Determines if the data in the rich text control has been modified.
<i>saveflag</i>	A numeric constant, variable, EQUATE, or expression that indicates whether to prompt the user to save changes to the rich text field if the field has changed data. A one (1 or True) value signifies the user will be prompted. This is the default value. A zero (0 or False) value signifies the user will not be prompted to save the data.

The **IsDirty** method is used to determine if the text in the RTF control has changed. If the text has changed, the user can be prompted to save all changes. A one (1 or True) is returned if the data in the rich text control contains changes; a zero (0 or False) is returned if the data in the rich text control does not contain data changes.

Return Data Type:     **BYTE**

Example:

```
loc:dirty =oRTF_RTFTextBox.IsDirty(1) !Prompt to save changes
loc:dirty =oRTF_RTFTextBox.IsDirty(0) !No prompt to save changes
```

## Kill (shut down the csRTF object)

**Kill**(*isdirty*)

<b>Kill</b>	Shut down the cwRTF object.
<i>isdirty</i>	A numeric constant, variable, EQUATE, or expression that indicates whether the rich text control's modified data should be saved to disk.

The **Kill** method shuts down the cwRTF object, releasing any memory allocated during the life of the object. Passing a one (1 or True) value to this method will force the modified data to be saved upon termination of the method; a value of zero (0 or False) will not save the modified data to the table.

Example:

```
oRTF_RTFTextBox.Kill( TRUE )
```

## LeftIndent (indent the current or selected paragraph)

**LeftIndent**(*indentsize*)

<b>LeftIndent</b>	Indent the current or selected paragraph.
<i>indentsize</i>	A REAL numeric constant, variable, or expression that indicates the number of inches to indent. A negative value will reverse the indent.

The **LeftIndent** method is used to indent the current or selected paragraph. The indentation size is specified in inches and may also be a negative number to reverse the indentation.

Example:

```
oRTF_RTFTextBox.LeftIndent(.5) !Indent 1/2 inch from left
```

## LimitTextSize (limit amount of text)

**LimitTextSize**(*maxtextsize*)

<b>LimitTextSize</b>	Limits the amount of text that can be placed in the rich text control regardless of the method used to save the data field or file.
<i>maxtextsize</i>	A numeric constant, variable, EQUATE, or expression that indicates the maximum number of characters allowed in the rich text field. This includes the rich text formatting characters.

The **LimitTextSize** method is used limit the number of text that can be placed in the rich text control regardless of the method used to save the data field or file. This method should be called before the rich text control is loaded.

Example:

```
oRTF_RTFTextBox.LimitTextSize(2000) !Limit text to 2000 characters
```

## LoadField (load rich text data from field)

**LoadField**(*fieldname*)

<b>LoadField</b>	Load the rich text control with the data contained in the field specified.
<i>fieldname</i>	The fully qualified label of a field to load the data from.

The **LoadField** method is used to load a string or memo rich text field into the rich text control.

Example:

```
oRTF_RTFTextBox.LoadField(let:letter ) !Load RTF field
```

## LoadFile (load rich text data from file)

**LoadFile**(*filename*)

<b>LoadFile</b>	Load the rich text control with the data contained in the file specified.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the .RTF filename. If no <i>filename</i> is specified, the Open file dialog is opened.

The **LoadFile** method is used to load a .RTF file into the rich text control.

Example:

```
oRTF_RTFTextBox.LoadField( let:letter ) !Load RTF field
```

## NewFile (clear rich text data)

**NewFile**

The **NewFile** method is used to clear the contents of the rich text control.

Example:

```
oRTF_RTFTextBox.NewFile()
```

## Offset (offset current or selected paragraph)

**Offset**(*offsetsize*)

<b>Offset</b>	Offset the second and subsequent lines of a paragraph.
<i>offsetsize</i>	A REAL numeric constant, variable, or expression that indicates the number of inches to offset. A negative value will reverse the offset.

The **Offset** method is used to offset the second and subsequent lines of the current or selected paragraph. The *offsetsize* is specified in inches and may also be a negative number to reverse the offset.

Example:

```
oRTF_RTFTextBox.Offset(1 ) !Offset by 1 inch
```

**PageSetup (set page attributes)****PageSetup**

The **PageSetup** method calls the Windows standard page setup dialog box. This dialog allows the setting of the paper size and source, orientation, margins and printer selection and defaults.

Example:

```
oRTF_RTFTextBox.PageSetup( )
```

**ParaBulletsOff (turn off paragraph bullets)****ParaBulletsOff**

The **ParaBulletsOff** method is used turn off bullets for the current or selected paragraph. The current paragraph is the one that the cursor is within.

Example:

```
oRTF_RTFTextBox.ParaBulletsOff() !Turn bullets off
```

**ParaBulletsOn (turn on paragraph bullets)****ParaBulletsOn(*bulletstyle*)**

**ParaBulletsOn** Turn on paragraph bullets.

*bulletstyle* A numeric constant, variable, EQUATE, or expression that identifies the bullet style. Bullet styles are defined in C60RTF.INC.

The **ParaBulletsOn** method is used turn on bullets for the current or selected paragraph. The current paragraph is the one that the cursor is within.

Supported bullet styles are:

Bullets:On (•)

Bullets:Arabic (1, 2, 3,...)

Bullets:LowerLetters (a, b, c,...)

Bullets:UpperLetters (A, B, C,...)

Bullets:LowerRoman (i, ii, iii,...)

Bullets:UpperRoman (I, II, III,...)

Example:

```
oRTF_RTFTextBox.ParaBulletsOn(BULLET:On) !Turn bullets on
```

**Paste (paste text from clipboard)**

**Paste**

The **Paste** method is used to paste clipboard text into the rich text control .

Example:

```
oRTF_RTFTextBox.Paste() !Paste text to clipboard
```

**Print (print rich text control contents)**

***\_Print(showsetup, [jobtitle])***

<b><i>_Print</i></b>	Print rich text control data.
<i>showsetup</i>	A numeric constant, variable, EQUATE, or expression that determines whether to show the Windows Print Setup dialog.
<i>jobtitle</i>	A string constant or variable containing a title of the job being printed. The job title is seen from the print spooler. If no job title is specified, the default job title is 'Untitled'. If the Print icon is used from the toolbar, the job title is set to 'RTF Data'.

The **\_Print** method is used to simply print the rich text data.

If the PageSetup dialog is used, the method will return a one (1 or True) value if the OK button on the dialog is pressed. A zero (0 or False) is returned if the Cancel button on the Page Seup dialog is pressed. If the Page Setup dialog is not used, a one (1 or True) is returned.

Return Data Type:      **BYTE**

Example:

```
oRTF_RTFTextBox._Print(1 )                    !Show Print Setup dialog before printing
oRTF_RTFTextBox._Print(0 )                   !Do not show Print Setup dialog
oRTF_RTFTextBox._Print(0, 'JobTitle' )       !Do not show Print Setup dialog
status = oRTF_RTFTextBox._Print(1, 'JobTitle' )!Do not show Print Setup dialog
```

## Redo (reapply action)

### Redo

The **Redo** method is used to reapply the action that was previously undone on the rich text control. Redo can be used on the previous 100 actions.

Example:

```
oRTF_RTFTextBox.Redo() !Redo previous undo action to rich text control
```

## Replace (find and replace search)

**Replace**([*findtext*], [*replacetext*])

<b>Replace</b>	Search for text within the rich text field.
<i>findtext</i>	A string constant, variable, EQUATE, or expression containing the text for search for. If this is omitted the Find dialog is presented.
<i>replacetext</i>	A string constant, variable, EQUATE, or expression containing the text for search for. If this is omitted the Find dialog is presented.

The **Replace** method is used to do a Find and Replace search option within the rich text control. If the *findtext* and *replacetext* parameters are omitted the Find and Replace dialog will appear.

This method returns a value indicating the number of applied replaces.

Return Data Type:           LONG

Example:

```
!Using Variables
loc:replace =oRTF_RTFTextBox.Replace(loc:find,loc:replace )
!Replace dialog
loc:replace =oRTF_RTFTextBox.Replace()
```

## ResizeControls (used internally)

The **ResizeControls** method is used internally and should not be called otherwise.

## RightIndent (indent the current or selected paragraph)

**RightIndent**(*indentsize*)

<b>RightIndent</b>	Indent the current or selected paragraph.
<i>indentsize</i>	A REAL numeric constant, variable, or expression that indicates the number of inches to indent. A negative value will reverse the indent.

The **RightIndent** method is used to indent the current or selected paragraph from the right side of the rich text control. The indentation size is specified in inches and may also be a negative number to reverse the indentation.

Example:

```
oRTF_RTFTextBox.RightIndent(.5) !Indent 1/2 inch from right
```

## SaveField (save rich text data to field)

**SaveField**(*fieldname*)

<b>SaveField</b>	Save the rich text control data to the field specified.
<i>fieldname</i>	The fully qualified label of a field to save to.

The **SaveField** method is used to save rich text data into a table field.

Example:

```
oRTF_RTFTextBox.SaveField(let:letter )
```

**SaveFile (save rich text data to file)****SaveFile**(*filename*)

<b>SaveFile</b>	Save the rich text data to the file specified.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the .RTF filename to save the data to.

The **SaveFile** method is used to save rich text data to a .RTF file.

Example:

```
oRTF_RTFTextBox.SaveFile(loc:Filename )
oRTF_RTFTextBox.SaveFile('0101SS.RTF ' )
```

**SelectText (select characters)****SelectText**(*startpos, endpos*)

<b>GetText</b>	Select a specific set of characters by character position.
<i>startpos</i>	An integer constant or variable that specifies the starting character position to select text from.
<i>endpos</i>	An integer constant or variable that specifies the ending character position to select text from.

The **SelectText** method is used select a specific set of characters by character position. The text is highlighted when it is selected.

Example:

```
!Select text from position 1 to 20
oRTF_RTFTextBox.SelectText(1,20 )
!Select text from variable positions
oRTF_RTFTextBox.SelectText(loc:startpos,loc:endpos )
```

## SetDirtyFlag (set modified flag)

**SetDirtyFlag**(*status*)

<b>SetDirtyFlag</b>	Sets the modified flag for the rich text control
<i>status</i>	A numeric constant, variable, EQUATE, or expression that indicates whether to set or clear the dirty flag. A one (1 or True) sets the flag; a zero (0 or False) clears the flag.

The **SetDirtyFlag** method is used to set or clear the modified flag for the rich text control. This can be used to clear the flag so the user is not prompted to save data.

Example:

```
oRTF_RTFTextBox.SetDirtyFlag( 0) ! Clear dirty flag
```

## SetFocus (give rich text control focus)

**SetFocus**

The **SetFocus** method is used to give the rich text control focus.

Example:

```
oRTF_RTFTextBox.SetFocus
```

## SetText (place text into rich text control)

**SetText**(*text*, [*allowundo*], [*startpos*], [*endpos*])

<b>SetText</b>	Place indicated text into rich text control.
<i>text</i>	A string constant or variable containing the text to place in the rich text control.
<i>allowundo</i>	A numeric constant, variable, EQUATE, or expression that determines whether the placement of text action is saved into the undo buffer. This allows the undo method to be called for this action. A one (1 or True) allows an undo for this action; a zero (0 or False) does not allow an undo for this action. The default value is False.
<i>startpos</i>	An integer constant or variable that specifies the starting character position to place the text to.
<i>endpos</i>	An integer constant or variable that specifies the ending character position to to place the text to.

The **SetText** method is used to place the text to a specified position within the a rich text control. The text to place comes from the specified string or variable. If the start and end positions are not available the current cursor position is used.

Example:

```
oRTF_RTFTextBox.SetText(loc:find,0,) !place text at current position  
oRTF_RTFTextBox.SetText(loc:find,10,20) !place text at specified position
```

## ShowControl (hide/unhide RTF control)

**ShowControl**(*showstate*)

<b>ShowControl</b>	Hide/Unhide RTF control
<i>showstate</i>	A numeric constant, variable, EQUATE, or expression that indicates whether the RTF control is displayed or hidden. A one (1 or True) unhides the control; a zero (0 or False) hides the control.

The **ShowControl** method is used to hide or unhide the RTF control. When hidden, the ruler bar is also hidden.

Example:

```
oRTF_RTFTextBox.ShowControl( 0 ) ! Hide control  
oRTF_RTFTextBox.ShowControl( 1 ) ! Unhide control
```

## Undo (undo action)

**Undo**

The **Undo** method is used to undo (reverse) the action previously taken on the rich text control.

Example:

```
oRTF_RTFTextBox.Undo()!Undo previous change to rich text control
```



---

# DbAuditManager

## DbAuditManager Overview

The DbAuditManager class declares an audit manager that consistently handles all database auditing operations. This class provides methods that create and update an audit log file.

## Relationship to Other Application Builder Classes

The DbAuditManager class uses the DbFileManager to handle the opening of the log file. This also takes care of any errors that may occur when opening the file. The DbAuditManager class uses the DbChangeManager class to update the audit log file.

This class also implements the DbdChangeAudit interface. This interface aids in the update of the log file.

## DbAuditManager ABC Template Implementation

The DbAuditManager class is used in an application when the Global Database Auditing extension is added to the application. The ABC Templates instantiate the DbAuditManager class in the global module of the application where the class is also initialized.

The DbAuditManager class implements the IDbdChangeAudit interface.

## DbAuditManager Source Files

The DbAuditManager source code is installed by default to the Clarion \LIBSRC folder. The specific DbAuditManager source code and their respective components are contained in:

ABFILE.INC	DbAuditManager declarations
ABFILE.CLW	DbAuditManager method definitions

## DbAuditManager Properties

The DbAuditManager contains the following properties:

### Action (log file action column)

**Action**      **STRING(20)**

The **Action** property contains a string that is used to update the Action column in the audit log file. The action column show the type of update that was performed on the data file.

Implementation:      The Action property is set by the OnInsert, OnDelete, OnChange, and BeforeChange methods. It is used to update the log file by the AddLogFile method.

See Also:              OnInsert, OnDelete, OnChange, and BeforeChange, AddLogFile

### ColumnInfo (log file column queue)

**ColumnInfo**      **&DbColumnQueue, PROTECTED**

The **ColumnInfo** property is a reference to a structure that is the source of the data that is added to the audit log file.

Implementation:      The ColumnInfo queue is initialized by the Init method and updated by the AddItem method.

See Also:              AddItem, AppendLog, CreateHeader, Init, Kill

### LogFiles (log file queue)

**LogFiles**      **&LogFileQueue, PROTECTED**

The **LogFiles** property is a reference to a structure that keeps track of all audit log files.

Implementation:      The LogFiles queue is initialized by the Init method and updated by the AddLogFile method.

See Also:              AddLogFile, Init, Kill, SetFm, OpenLogFile

## **FM (DbLogFileManager object)**

**LFM**      **&DbLogFileManager, PROTECTED**

The **LFM** property is a reference to the DbLogFileManager object that creates and opens the various audit log files.

Implementation:      The LFM property is initialized in the Init method.

See Also:              AppendLog, CreateHeader, Init, Kill, OpenLogFile

## **Errors (ErrorClass object)**

**Errors**      **&ErrorClass, PROTECTED**

The Errors property is a reference to the ErrorClass object that handles unexpected conditions for the DbAuditManager. In an ABC Template generated program, the ErrorClass object is called GlobalErrors by default.

Implementation:      The Init method initializes the Errors property.

See Also:              Init, OpenLogFile

## DbAuditManager Methods

The DbAuditManager class contains the following methods:

### AddItem (maintain the columninfo structure)

**AddItem**(*filename, fieldname, field, fieldheader, fieldpicture*)

<b>AddItem</b>	Adds fields to the columninfo structure so they can be monitored in the audit log file.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.
<i>fieldname</i>	A string constant, variable, EQUATE, or expression containing the label of the field that is to be audited.
<i>field</i>	The fully qualified label for the FILE field.
<i>fieldheader</i>	A string constant, variable, EQUATE, or expression containing the field header (title) field that is to be audited. This is the column header that will appear in the audit log file.
<i>fieldpicture</i>	A string constant, variable, EQUATE, or expression containing the picture of the field that is to be audited. This defines how the column in the audit log file will be formatted.

The **AddItem** method maintains the ColumnInfo queue by updating it with field information needed to create and update the log file.

Implementation: This method is called one time for each field that will appear in the log file. There is one extra column that always appears in the log file. This is the action column that defines the update action that occurred to cause the log file update.

See Also: AddLogFile

## AddLogFile (maintain log file structure)

**AddLogFile**(*filename*, *logfile*)

<b>AddLogFile</b>	Maintain the log file queue.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.
<i>logfile</i>	A string constant, variable, EQUATE, or expression containing the audit log file name.

The **AddLogFile** method maintains the LogFiles structure in order to keep track of each log file and its associated data file.

Implementation: In template generated code, the AddLogFile method is called after the DbAuditManager object is initialized. This method should be called once for each file being audited.

## AppendLog (initiate audit log file update)

**AppendLog**(*filename*)

<b>AppendLog</b>	Initiate audit log file update.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.

The **AppendLog** method initiates the audit log file update by retrieving the field information and setting the date and time of the update. This information is added to the log file by the FileManager.

See Also: LFM

## **BeforeChange (update audit log file before file change)**

**BeforeChange**(*filename*, *BFP*), VIRTUAL

### **BeforeChange**

*filename*            A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.

*BFP*                    The label of a BufferedPairsClass object. See *BufferedPairsClass* for more information.

The **BeforeChange** method saves the values of the record before a change is made. These values are used for comparison after the record is saved back to disk.

Implementation:      The BeforeChange method sets the Action property and saves the current field values to the log file. This method is called from the DbChangeManager.CheckChanges method.

See Also:              BufferedPairsClass, DbChangeManager.CheckChanges

## **CreateHeader (create log file header records)**

**CreateHeader**(*filename*, *LFM*), VIRTUAL

**CreateHeader**        Create the header records in the audit log file.

*filename*            A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.

*LFM*                    The label of the log file's DbLogFileManager object.

The **CreateHeader** method updates the audit log file just after file creation. It adds header records to the log file which serve as column heading information. The header includes the Date, Time, Action, and Field header.

Implementation:      The CreateHeader method is called by the OpenLogFile method.

See Also:              OpenLogFile

## Init (initialize the DbAuditManager object)

**Init**(*error handler*), VIRTUAL

<b>Init</b>	Initializes the DbAuditManager object
<i>error handler</i>	The label of an ErrorClass object. See <i>Error Class</i> for more information.

The **Init** method initializes the DbAuditManager object. The *error handler* object, ColumnInfo, LogFiles, and LFM properties are initialized at this time.

Implementation: In template generated code, the Init method is called in the main application module. It is called with the GlobalErrors error handler.

## Kill (shut down DbAuditManger object)

**Kill**, VIRTUAL

The **Kill** method frees any memory allocated during the life of the object and does any other required termination code.

Implementation: The Kill method frees memory allocated to the ColumnInfo, LogFiles, and LFM properties.

## **OnChange (update audit log file after a record change)**

**OnChange**(*filename*, *file*), VIRTUAL

<b>OnChange</b>	Initiates an update to the audit log file after a Change to the file.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.
<i>file</i>	The label of the FILE being audited.

The **OnChange** method initiates the update to the audit log file after a Change action.

Implementation: The OnChange method sets the Action property and calls the AppendLog method. This method is called from the DbChangeManager.CheckChanges method.

See Also: DbChangeManager.CheckChanges

## **OnDelete (update audit log file when a record is deleted)**

**OnDelete**(*filename*, *file*), VIRTUAL

<b>OnDelete</b>	Initiates an update the audit log file on a Delete action.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.
<i>file</i>	The label of the FILE being audited.

The **OnDelete** method initiates the update to the audit log file on a Delete action.

Implementation: The OnDelete method sets the Action property and calls the AppendLog method. This method is called from the RelationManager's Delete method.

See Also: RelationManager.Delete

## OnFieldChange (virtual method for each field change)

**OnFieldChange**(*left, right, fieldname, filename*), **VIRTUAL**

<b>OnFieldChange</b>	Manage field changes.
<i>left</i>	The label of the "left" field of the pair that contains the original value of the field being updated. The field may be any data type, but may not be an array.
<i>right</i>	The label of the "right" field of the pair that contains the new value of the field being updated. The field may be any data type, but may not be an array.
<i>fieldname</i>	A string constant, variable, EQUATE, or expression containing the label of the field that is to be audited.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is audited.

The **OnFieldChange** method is called for each field in the record that has changed. The before and after values are passed to this method.

Implementation: **OnFieldChange** is a VIRTUAL method so that other base class methods can directly call the OnFieldChange virtual method in a derived class. This lets you easily implement your own custom version of this method.

See Also: DbChangeManager.CheckPair

## **OnInsert (update audit log file when a record is added)**

**OnInsert**(*filename, file*), VIRTUAL

<b>OnInsert</b>	Initiates an update the audit log file on an Insert action.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.
<i>file</i>	The label of the FILE being audited.

The **OnInsert** method initiates the update to the audit log file on an Insert action.

Implementation: The OnInsert method sets the Action property and calls the AppendLog method. This method is called from the FileManager's Insert method.

See Also: FileManager.Insert

## **OpenLogFile (open the audit log file)**

**OpenLogFile**(*filename*), PROTECTED

<b>OpenLogFile</b>	Opens the audit log file.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is audited.

The **OpenLogFile** method opens the audit log file that is to be updated. If the log file does not already exist, the file is created and the headers are added to it.

Implementation: This method is called by the AppendLog method to ensure the log file exists before trying to update it.

See Also: AppendLog, DbLogFileManager class

## SetFM (determine log file status)

**SetFM**(*filename*), PROTECTED

<b>SetFM</b>	Determines log file status.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is audited.

The **SetFM** method is used to determine if the log file has been initialized by the AddLogFile method.

A one (1 or True) value is returned if the log file has been correctly initialized into the LogFiles structure; a zero (0 or False) is returned otherwise.

Implementation:       The SetFM method is called by the AddItem and AppendLog methods.

Return Data Type:        BYTE

See Also:                AddItem, AppendLog



---

# DbChangeManager

## DbChangeManager Overview

The DbChangeManager class declares an audit manager that consistently handles all database change operations.

## Relationship to Other Application Builder Classes

The DbChangeManager class works in conjunctions with the DbAuditManager class and the IDbChangeManager interface.

## DbChangeManager ABC Template Implementation

The DbChangeManager class is used in an application when the Global Database Auditing extension is added to the application. The ABC Templates instantiate the DbChangeManager class in the global module of the application where the class is also initialized.

## DbChangeManager Source Files

The DbChangeManager source code is installed by default to the Clarion \LIBSRC folder. The specific DbChangeManager source code and their respective components are contained in:

ABFILE.INC  
ABFILE.CLW

DbChangeManager declarations  
DbChangeManager method definitions

## DbChangeManager Properties

The DbChangeManager contains the following properties:

### NameQueue (pointer into trigger queue)

**NameQueue**     **&DbNameQueue, PROTECTED**

The **NameQueue** property keeps track of the file and field names when a change is made to a file that is being audited. The NameQueue queue entry also points into the trigger queue for the change data.

Implementation:     The NameQueue structure is updated in the DbChangeManager.AddItem method. This queue contains a pointer into the TriggerQueue.

See Also:             DbChangeManager.Init, DbChangeManager.Kill,  
                          DbChangeManager.AddItem, DbChangeManager.CheckPair

### TriggerQueue (pointer to BFP for field changes)

**TriggerQueue**     **&DbTriggerQueue, PROTECTED**

The **TriggerQueue** property contains the field change information by managing the BufferFieldPairs structure.

## DbChangeManager Methods

The DbChangeManager class contains the following methods:

### AddItem (maintain the namequeue structure)

**AddItem**(*left, name, filename*)

<b>AddItem</b>	Maintains the namequeue structure.
<i>left</i>	The label of the "left" field of the pair that contains the original value of the field being updated. The field may be any data type, but may not be an array.
<i>name</i>	A string constant, variable, EQUATE, or expression containing the label of the field that is to be audited.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is audited.

The **AddItem** method maintains the NameQueue structure by updating it with field information needed to track before and after file changes.

Implementation: This method is called one time for each field that will appear in the log file.

### AddThread (maintains the triggerqueue)

**AddThread**(*filename*)

<b>AddThread</b>	Maintains the TriggerQueue.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is audited.

The **AddThread** method maintains the TriggerQueue by updating it with the current thread number and an instance of the BufferedPairsClass.

## CheckChanges(check record for changes)

**CheckChanges**(*filename, file*)

<b>CheckChanges</b>	Checks the record for changes.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is audited.
<i>file</i>	The label of the FILE being audited.

The **CheckChanges** method checks to see if there were any changes made to the columns in the audited file. This method calls the CheckPairs method to compare the before and after field values of the audited file.

See Also: DbChangeManagerBeforeChange, IDbChangeManagerOnChange, DbChangeManager.CheckPair

## CheckPair(check field pairs for changes)

**CheckPair**(*FP*)

<b>CheckPairs</b>	Checks fields for changes.
<i>FP</i>	The label of a FieldPairsClass object. See <i>FieldPairsClass</i> for more information.

The **CheckPair** method compares the before and after values of a field to see if there were changes made. This is needed to know whether to update the audit log file.

## Equal(checks for equal before and after values)

**Equal**(*filename*)

<b>Equal</b>	Compares before and after values of fields.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is audited.

The **Equal** method returns one (1) if all pairs are equal and returns zero (0) if any pairs are not equal.

Implementation: The Equal method simply calls FieldPairsClass.Equat method which calls the FieldPairsClass.EqualLeftRight method which in turn does all the comparison work.

Return Data Type: BYTE

See Also: FieldPairsClass.Equal

## Init (initialize the DbChangeManager object)

**Init**(*IDBC*), VIRTUAL

<b>Init</b>	Initializes the DbChangeManager object
<i>IDBC</i>	The label of an IDbChangeManager interface. See <i>IDbChangeManager</i> for more information.

The **Init** method initializes the DbChangeManager object. The *TriggerQueue* and *NameQueue* are also initialized at this time.

Implementation: In template generated code, the Init method is called in the main application module.

## Kill (shut down DbChangeManger object)

### Kill, VIRTUAL

The **Kill** method frees any memory allocated during the life of the object and does any other required termination code.

Implementation:       The Kill method frees memory allocated to the NameQueue and TriggerQueue structures.

## SetThread (read triggerqueue)

### SetThread(*filename*)

<b>SetThread</b>	Reads the TriggerQueue for the specified file and current thread.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is audited.

The **SetThread** method reads the TriggerQueue for the specified file and current thread. If a matching entry is found in the queue a one (1 or True) is returned; if an entry is not found in the TriggerQueue a zero (0 or False) is returned.

Return Data Type:     BYTE

## Update (update the audit log file buffer)

**Update**(*filename*)

**Update**

*filename*

A string constant, variable, EQUATE, or expression containing the label of the file that is audited.

The **Update** method calls the BufferFieldPairs class to update the record buffer with the changes made in order for the audit log file to be updated with the before and after values.

See Also:           BFP.AssignLeftToRight



---

# DbLogFileManager

## DbLogFileManager Overview

The DbLogFileManager class declares a file manager that consistently handles the routine database operations on the audit log file. The DbLogFileManager is derived from the FileManager class.

## Relationship to Other Application Builder Classes

The DbLogFileManager class is derived from the FileManager class. This class inherits all FileManager class properties and methods. The DbLogFileManager also relies on the ErrorClass for most of its error handling.

## DbLogFileManager ABC Template Implementation

The DbLogFileManager class is instantiated by the DbAuditManager LFM property in the DbAuditManager.Init method.

## DbLogFileManager Source Files

The DbLogFileManager source code is installed by default to the Clarion \LIBSRC folder. The specific DbLogFileManager source code and their respective components are contained in:

ABFILE.INC	DbLogFileManager declarations
ABFILE.CLW	DbLogFileManager method definitions

## DbLogFileManager Properties

### DbLogFileManager Properties

The DbLogFileManager inherits all the properties of the FileManager class from which it is derived. See FileManager Properties for more information.

In addition to the inherited properties, the DbLogFileManager contains the following properties:

### Opened (file opened flag)

**Opened**      **BYTE**

The **Opened** property indicates whether the DbLogFileManager's FILE (the log file) has been opened. A value of one (1 or True) indicates the FILE is open; a value of zero (0 or False) indicates the FILE is not opened.

## DbLogFileManager Methods

### DbLogFileManager Methods

The DbLogFileManager inherits all the methods of the FileManager class from which it is derived. See FileManager Methods for more information.

In addition to the inherited methods, the DbLogFileManager contains the following methods:

### Init (initialize the DbLogFileManager object)

**Init**(*errorclass*, *logfile*name), VIRTUAL

<b>Init</b>	Initializes the DbAuditManager object
<i>errorclass</i>	The label of an ErrorClass object. See <i>Error Class</i> for more information.
<i>logfile</i> name	A string constant, variable, EQUATE, or expression that contains the audit log file name.

The **Init** method initializes the DbLogFileManager object.



# EditClass

## EditClass Overview

The EditClass lets you implement a dynamic edit-in-place control for each column in a LIST. The EditClass is an abstract class--it is not useful by itself, but serves as the foundation and framework for its derived classes. See *EditCheckClass*, *EditColorClass*, *EditFileClass*, *EditDropListClass*, *EditFontClass*, and *EditMultiSelectClass*.

## EditClass Concepts

The EditClass creates an input control (CHECK, ENTRY, SPIN, COMBO, etc.), accepts input from the end user, then returns the input to a specified variable, typically the variable associated with a specific LIST cell--a field in the LIST control's data source QUEUE. The EditClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditClass provides virtual methods (TakeEvent) to allow you to take control of significant edit-in-place events.

The BrowseClass (AskRecord method) uses the EditClass to accomplish edit-in-place data entry by assigning the EditClass input control to a specific LIST cell--see *BrowseClass.AskRecord*.

## Relationship to Other Application Builder Classes

### Derived Classes

The EditClass serves as the foundation and framework for its derived classes. See *EditCheckClass*, *EditColorClass*, *EditEntryClass*, *EditFileClass*, *EditFileDropClass*, *EditFontClass*, and *EditMultiSelectClass*. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseClass

The EditClass is loosely integrated into the BrowseClass. The BrowseClass depends on the EditClass operating according to this chapter's specification; however, the EditClass may be called by non-BrowseClass procedures and objects.

The BrowseClass.AskProcedure property indicates whether the BrowseClass object uses the EditClass for updates.

The BrowseClass.AskRecord method is the engine for the edit-in-place functionality. This method uses the EditClass to dynamically create the Edit-in-place control upon request (Insert, Change, or Delete) by the end user. When the end user moves off the edited record (enter key, click on another item) the AskRecord method saves or deletes the record and uses the EditClass to destroy the Edit-in-place control.

## ABC Template Implementation

The BrowseUpdateButtons template generates references to EditClass objects as needed. One check box on the BrowseUpdateButtons control template enables default edit-in-place support for a given BrowseBox--any associated Form (update) procedure then becomes redundant.

If you accept the BrowseUpdateButtons default edit-in-place behavior, the generated code does not reference the EditClass, because the default edit-in-place behavior is implemented in the BrowseClass (see BrowseClass.AskRecord), and no additional generated code is needed.

If you use custom (**Configure EditInPlace**) edit-in-place behavior, the BrowseUpdateButtons template generates the code to instantiate the requested object (derived from the EditClass) and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditClass object's methods as needed. See *Control Templates--BrowseUpdateButtons* for more information.

## EditClass Source Files

The EditClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditClass source code and their respective components are contained in:

ABEIP.INC	EditClass declarations
ABEIP.CLW	EditClass method definitions
ABEIP.TRN	EditClass translation strings

## EditClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate several EditClass objects and a related BrowseClass object. The example page-loads a LIST of fieldnames and associated control attributes (such as color, icon, etc.), then edits the list items with a variety of edit-in-place objects. Note that the BrowseClass object calls the "registered" EditClass objects' methods as needed.

```

PROGRAM
_ABCDllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)
  INCLUDE('ABWINDOW.INC')      !declare WindowManager
  INCLUDE('ABBROWSE.INC')      !declare BrowseClass
  INCLUDE('ABEIP.INC')        !declare Edit-in-place classes
MAP
END

Property FILE,DRIVER('TOPSPEED'),PRE(PR),CREATE,BINDABLE,THREAD
NameKey KEY(PR:FieldName),NOCASE,OPT
Record RECORD,PRE()
FieldName STRING(30)
Color STRING(20)
Hidden STRING(1)
IconFile STRING(30)
ControlType STRING(12)
  END
  END
PropView VIEW(Property)
  END

PropQ QUEUE
PR:FieldName LIKE(PR:FieldName)
PR:Color LIKE(PR:Color)
PR:ControlType LIKE(PR:ControlType)
PR:Hidden LIKE(PR:Hidden)
PR:IconFile LIKE(PR:IconFile)
ViewPosition STRING(1024)
  END

PropWindow WINDOW('Browse Field Properties'),AT(, ,318,137),IMM,SYSTEM,GRAY
LIST,AT(8,4,303,113),USE(?PropList),IMM,HVSCROLL,FROM(PropQ),|
FORMAT('50L(2)|_M~Field Name~@s30@[70L(2)|_M~Color~@s20@' &|
'60L(2)|_M~Control Type~@s12@' &|
'20L(2)|_M~Hide~L(0)@s1@/130L(2)|_M~Icon File~@s30@]|M')
BUTTON('&Insert'),AT(169,121),USE(?Insert)
BUTTON('&Change'),AT(218,121),USE(?Change),DEFAULT
BUTTON('&Delete'),AT(267,121),USE(?Delete)
  END

```

```

Edit:PR:FieldName EditEntryClass      !declare Edit:PR:FieldName-EIP ENTRY control

Edit:PR:Color CLASS(EditColorClass)  !declare Edit:PR:Color-EIP color dialog
Init          PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
              END

Edit:PR:Hide CLASS(EditCheckClass) !declare Edit:PR:Color-EIP CHECK control
Init          PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
              END

Edit:PR:IconFile CLASS(EditFileClass)!declare Edit:PR:IconFile-EIP file dialog
Init          PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
              END

Edit:PR:ControlType CLASS(EditDropListClass) !declare Edit:PR:ContolType-EIP droplist
Init          PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
              END

ThisWindow CLASS(WindowManager)
Init          PROCEDURE(),BYTE,PROC,VIRTUAL
Kill          PROCEDURE(),BYTE,PROC,VIRTUAL
              END

BRW1         CLASS(BrowseClass) !declare BRW1, the BrowseClass object
Q            &PropQ             ! that drives the EditClass objects--
              END               ! i.e. calls Init, TakeEvent, Kill

GlobalErrors ErrorClass
Access:Property CLASS(FileManager)
Init            PROCEDURE
              END

Relate:Property CLASS(RelationManager)
Init            PROCEDURE
Kill            PROCEDURE,VIRTUAL
              END

GlobalRequest  BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD
CODE
GlobalErrors.Init
Relate:Property.Init
GlobalResponse = ThisWindow.Run()
Relate:Property.Kill
GlobalErrors.Kill

```

```
ThisWindow.Init PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
SELF.FirstField = ?PropList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:Property.Open
BRW1.Init(?PropList,PropQ.ViewPosition,PropView,PropQ,Relate:Property,SELF)
OPEN(PropWindow)
SELF.Opened=True
?PropList{PROP:LineHeight}=12    !enlarge rows to accomodate EditClass icons
BRW1.Q &= PropQ
BRW1.AddSortOrder(,PR:NameKey)
BRW1.AddField(PR:FieldName,BRW1.Q.PR:FieldName)
BRW1.AddField(PR:Color,BRW1.Q.PR:Color)
BRW1.AddField(PR:ControlType,BRW1.Q.PR:ControlType)
BRW1.AddField(PR:Hidden,BRW1.Q.PR:Hidden)
BRW1.AddField(PR:IconFile,BRW1.Q.PR:IconFile)
BRW1.AddEditControl(Edit:PR:FieldName,1)    !Register Edit:PR:FieldName with BRW1
BRW1.AddEditControl(Edit:PR:Color,2)        !Register Edit:PR:Color with BRW1
BRW1.AddEditControl(Edit:PR:ControlType,3)!Register Edit:PR:ControlType with BRW1
BRW1.AddEditControl(Edit:PR:Hide,4)         !Register Edit:PR:Hide with BRW1
BRW1.AddEditControl(Edit:PR:IconFile,5)    !Register Edit:PR:IconFile with BRW1
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change
BRW1.DeleteControl=?Delete
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Property.Close
RETURN ReturnValue

Edit:PR:Color.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Title='Select field color'    !set EIP color dialog title

Edit:PR:Hide.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
```

```

SELF.Feq{PROP:Text}='Hide '      !set EIP check box text
SELF.Feq{PROP:Value,1}='Y'      !set EIP check box true value
SELF.Feq{PROP:Value,2}='N'      !set EIP check box false value

Edit:PR:IconFile.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Title='Select icon file'      !set EIP file dialog title
SELF.FilePattern='Icon files *.ico|*.ico' !set EIP file dialog file masks
SELF.FileMask=FILE:KeepDir+FILE:LongName !set EIP file dialog behavior flag

Edit:PR:ControlType.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Feq{PROP:From}='ENTRY|SPIN|TEXT|STRING' !set ControlType droplist
choices

Access:Property.Init PROCEDURE      !initialize FileManager
CODE
PARENT.Init(Property,GlobalErrors)
SELF.FileNameValue = 'Property'
SELF.Buffer &= PR:Record
SELF.Create = 1
SELF.AddKey(PR:NameKey, 'PR:NameKey', 0)

Relate:Property.Init PROCEDURE      !initialize RelationManager
CODE
Access:Property.Init
PARENT.Init(Access:Property,1)

Relate:Property.Kill PROCEDURE      !shut down RelationManager
CODE
Access:Property.Kill
PARENT.Kill

```

## EditClass Properties

The EditClass contains the following properties.

### **FEQ (the edit-in-place control number)**

**FEQ**    **UNSIGNED**

The **FEQ** property contains the control number of the edit-in-place control.

The CreateControl method sets the value of the FEQ property when it creates the control.

See Also:            CreateControl

### **ReadOnly ( edit-in-place control is read-only)**

**ReadOnly**    **BYTE**

The **ReadOnly** property is a flag indicating that the edit-in-place control is not editable.

See Also:            SetReadOnly

## EditClass Methods

### Functional Organization--Expected Use

As an aid to understanding the EditClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditClass methods.

#### Non-Virtual Methods

---

The non-virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### Housekeeping (one-time) Use:

Init <sup>v</sup>	initialize the EditClass object
Kill <sup>v</sup>	shut down the EditClass object

##### Mainstream Use:

TakeEvent <sup>v</sup>	handle events for the edit control
------------------------	------------------------------------

##### Occasional Use:

CreateContol <sup>v</sup>	a virtual to create the edit control
SetAlerts <sup>v</sup>	alert appropriate keystrokes for the edit control

<sup>v</sup> These methods are also virtual.

#### Virtual Methods

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init	initialize the EditClass object
CreateContol	a virtual to create the edit control
SetAlerts	alert appropriate keystrokes for the edit control
TakeAccpted	handle accepted events for the edit control
TakeEvent <sup>v</sup>	handle events for the edit control
Kill	shut down the EditClass object

## CreateControl (a virtual to create the edit control)

### CreateControl, VIRTUAL, PROTECTED

The **CreateControl** method is a virtual placeholder to create the appropriate window control for derived classes.

Implementation:       The Init method calls the CreateControl method. The CreateControl method must set the value of the FEQ property.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
  SELF.ListBoxFeq = ListBox
  SELF.CreateControl()
  ASSERT(SELF.Feq)
  SELF.UseVar &= UseVar
  SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
  SELF.Feq{PROP:Use} = UseVar
  SELF.SetAlerts
```

See Also:               FEQ, EditCheckClass.CreateControl, EditColorClass.CreateControl,  
EditEntryClass.CreateControl, EditFileClass.CreateControl,  
EditDropListClass.CreateControl, EditMultiSelectClass.CreateControl

**Init (initialize the EditClass object)****Init( *column*, *listbox*, *editedfield* ), VIRTUAL**

<b>Init</b>	Initializes the EditClass object.
<i>column</i>	An integer constant, variable, EQUATE, or expression that contains the edited column number of the <i>listbox</i> .
<i>listbox</i>	An integer constant, variable, EQUATE, or expression that contains the control number of the edited LIST control--typically a BrowseClass object's LIST.
<i>editedfield</i>	The fully qualifiedlabel of the edited field--typically a field in the BrowseClass object's QUEUE.

The **Init** method initializes the EditClass object.

Implementation: The BrowseClass.AskRecord method calls the Init method. The Init method creates the edit-in-place control, loads it with the selected list item's data, and alerts the appropriate edit-in-place navigation keys.

Example:

```
MyEditClass.Init(1,?MyList,StateQ:StateCode)    !initialize EditClass object
!program code
MyEditClass.Kill                               !shut down EditClass object
```

See Also: BrowseClass.AskRecord

**Kill (shut down the EditClass object)****Kill, VIRTUAL**

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code. The Kill method must leave the object in an Initable state.

Implementation: The BrowseClass.AskRecord method calls the Kill method. The Kill method destroys the edit-in-place control created by the Init method.

Example:

```
MyEditClass.Init(1,?MyList,StateQ:StateCode)    !initialize EditClass object
!program code
MyEditClass.Kill                               !shut down EditClass object
```

See Also: BrowseClass.AskRecord

**SetAlerts (alert keystrokes for the edit control)****SetAlerts, VIRTUAL**

The **SetAlerts** method method alerts appropriate keystrokes for the edit-in-place control.

Implementation: The Init method calls the CreateControl method to create the input control and set the FEQ property. The Init method then calls the SetAlerts method to alert standard edit-in-place keystrokes for the edit control. Alerted keys are:

TabKey	!next field
ShiftTab	!previous field
EnterKey	!complete and save
EscKey	!complete and cancel
DownKey	!complete and save, then edit next row
UpKey	!complete and save, then edit prior row

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts
```

See Also: Init

## **SetReadOnly (set edit control to read-only)**

### **SetReadOnly( *state* ), VIRTUAL**

**SetReadOnly**      The **SetReadOnly** method places the edit-in-place control in a read-only state.

*state*              An integer constant, variable, EQUATE, or expression that indicates whether to disable a droplist control's dropdown button. A value of one (1 or True) disables the button. A value of zero (0 or False) has no effect on the control.

Implementation:      The SetReadOnly method uses PROP:ReadOnly to place the edit-in-place control in a read-only state. After the parent call in the Init method of the EditInPlace object is the recommended place to call SetReadOnly.

Example:

```
EditInPlace::CUS:Number.SetReadOnly()
```

See Also:              ReadOnly, EditDropListClass.SetReadOnly

## **TakeAccepted (validate EIP field)**

### **TakeAccepted (Action), VIRTUAL**

The **TakeAccepted** method processes the accepted EIP field value and returns a value indicating whether to continue editing or to complete the field. If the EIPManager Class attribute SELF.REQ is TRUE, the field will be required, and the row can not be accepted if the field is blank. If the TakeAccepted method returns the EditAction:Cancel equate, the EIP will remain on the current field.

Return Data Type:    BYTE

See Also: EIPManager.TakeAcceptAll

**TakeEvent (process edit-in-place events)****TakeEvent**( *event* ), VIRTUAL

<b>TakeEvent</b>	Processes an event for the EditClass object.
<i>event</i>	An integer constant, variable, EQUATE, or expression that contains the event number (see EVENT in the <i>Language Reference</i> ).

The **TakeEvent** method processes an event for the EditClass object and returns a value indicating the user requested action. Valid actions are none, complete or OK, cancel, next record, previous record, next field, and previous field.

Implementation: The BrowseClass.AskRecord method calls the TakeEvent method. The TakeEvent method process an EVENT:AlertKey for the edit-in-place control and returns a value indicating the user requested action. The BrowseClass.AskRecord method carries out the user requested action.

Corresponding EQUATEs for the possible edit-in-place actions are declared in ABBROWSE.INC as follows:

```

EditAction      ITEMIZE(0),PRE
None            EQUATE                ! no action
Forward        EQUATE                ! next field
Backward       EQUATE                ! previous field
Complete       EQUATE                ! OK
Cancel         EQUATE                ! cancel
Next           EQUATE                ! next record
Previous       EQUATE                ! previous record
END

```

Return Data Type: BYTE

Example:

```
EditClassAction ROUTINE
  CASE SELF.EditList.Control.TakeEvent(EVENT())
  OF EditAction:Forward
    !handle tab forward (new field, same record)
  OF EditAction:Backward
    !handle tab backward (new field, same record)
  OF EditAction:Next
    !handle down arrow (new record, offer to save prior record)
  OF EditAction:Previous
    !handle up arrow (new record, offer to save prior record)
  OF EditAction:Complete
    !handle OK or enter key (save record)
  OF EditAction:Cancel
    !handle Cancel or esc key (restore record)
  END
```

See Also:           BrowseClass.AskRecord

---

# EditSpinClass

## EditSpinClass--Overview

The EditSpinClass is an EditClass that supports a SPIN control. The EditSpinClass lets you implement a dynamic edit-in-place SPIN control for a column in a LIST.

## EditSpinClass Concepts

The EditSpinClass creates a SPIN control, accepts input from the end user, then returns the input to the variable specified by the Init method, typically the variable associated with a specific LIST cell--a field in the LIST control's data source QUEUE. The EditSpinClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditSpinClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## EditSpinClass -- Relationship to Other Application Builder Classes

### EditClass

---

The EditSpinClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseEIPManagerClass

---

The EditClass is managed by the BrowseEIPManagerClass. The BrowseEIPManagerClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

## EditSpinClass--ABC Template Implementation

You can use the BrowseUpdateButtons control template (**Configure EditInPlace**) to generate the code to instantiate an EditSpinClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditSpinClass object's methods as needed. See *Control Templates--BrowseUpdateButtons* for more information. EditSpinClass Source Files

The EditSpinClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditSpinClass source code and their respective components are contained in:

ABEIP.INC	EditSpinClass declarations
ABEIP.CLW	EditSpinClass method definitions

## EditSpinClass--Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditSpinClass object and a related BrowseClass object. The example page-loads a LIST of actions and associated attributes (priority and completed), then edits the "Priority" items with an EditSpinClass object. Note that the BrowseClass object calls the "registered" EditSpinClass object's methods as needed.

**Note:** The EditSpinClass requires values for PROP:RangeLow, PROP:RangeHigh, and PROP:Step to function correctly. The EditSpinClass.Init method is the proper place to set these properties. See *SPIN* in the *Language Reference* for more information.

```

PROGRAM
_ABCEllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)
  INCLUDE('ABWINDOW.INC'),ONCE
  INCLUDE('ABEIP.CLW'),ONCE
  INCLUDE('ABBROWSE.CLW'),ONCE
MAP
END

Actions      FILE,DRIVER('TOPSPEED'),PRE(ACT),CREATE,BINDABLE,THREAD
KeyAction    KEY(ACT:Action),NOCASE,OPT
Record       RECORD,PRE()
Action       STRING(20)
Priority      DECIMAL(2)
Completed    DECIMAL(1)
            END

ViewActions  VIEW(Actions)
            END

ActQ         QUEUE
ACT:Action   LIKE(ACT:Action)
ACT:Priority  LIKE(ACT:Priority)
ACT:Completed LIKE(ACT:Completed)
ViewPosition STRING(1024)
            END

ActionWindow WINDOW('Actions File'),AT(,,164,144),IMM,SYSTEM,GRAY|
LIST,AT(8,6,148,115),USE(?List),IMM,HVSCROLL,FORMAT('80L(2)|~Action~'|&|
 '@S20@31C|~Priority~@N2@40L(2)|~Done~L(0)@N1@'),FROM(ActQ)
BUTTON('&Insert'),AT(10,126,45,14),USE(?Insert:2)
BUTTON('&Change'),AT(59,126,45,14),USE(?Change:2),DEFAULT
BUTTON('&Delete'),AT(108,126,45,14),USE(?Delete:2)
            END

```

```

ThisWindow      CLASS(WindowManager)
Init            PROCEDURE(),BYTE,PROC,DERIVED
Kill           PROCEDURE(),BYTE,PROC,DERIVED
              END
BRW1           CLASS(BrowseClass)
Q              &ActQ
              END

```

```

Edit:ACT:Priority CLASS(EditSpinClass)  ! Edit-in-place class for field ACT:Priority
Init          PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),DERIVED
              END

```

```

CODE
GlobalResponse = ThisWindow.Run()

```

```

ThisWindow.Init PROCEDURE
ReturnValue      BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue =PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?List
SELF.Errors &= GlobalErrors
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
Relate:Actions.Open
FilesOpened = True
BRW1.Init(?List,ActQ.ViewActions,BRW1::ViewActions,ActQ,Relate:Actions,SELF)
OPEN(ActionWindow)
SELF.Opened=True
BRW1.Q &= ActQ
BRW1.AddSortOrder(ACT:KeyAction)
BRW1.AddField(ACT:Action,BRW1.Q.ACT:Action)
BRW1.AddField(ACT:Priority,BRW1.Q.ACT:Priority)
BRW1.AddField(ACT:Completed,BRW1.Q.ACT:Completed)
BRW1.AddEditControl(EditInPlace::ACT:Priority,2)  !Add cutom edit-inplace class
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert:2
BRW1.ChangeControl=?Change:2
BRW1.DeleteControl=?Delete:2
SELF.SetAlerts()
RETURN ReturnValue

```

```

ThisWindow.Kill PROCEDURE
ReturnValue      BYTE,AUTO
CODE
ReturnValue =PARENT.Kill()

```

```
IF ReturnValue THEN RETURN ReturnValue.  
IF FilesOpened  
    Relate:Actions.Close  
END  
RETURN ReturnValue
```

```
Edit:ACT:Priority.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*?  
UseVar)  
CODE  
PARENT.Init(FieldNumber,ListBox,UseVar)  
SELF.FEQ{PROP:RANGE,1} = 1           !Set the Low Range for the Spinbox  
SELF.FEQ{PROP:RANGE,2} = 10        !Set the High Range for the Spinbox  
SELF.FEQ{PROP:Step} = 1            !Set the incremental steps of the Spinbox
```

## EditSpinClass Properties

### **EditSpinClass Properties**

The EditSpinClass inherits all the properties of the EditClass from which it is derived. See *EditClass Properties* and *EditClass Concepts* for more information.

## EditSpinClass Methods

### EditSpinClass Methods

The EditSpinClass inherits all the methods of the EditClass from which it is derived. See *EditClass Methods* and *EditClass Concepts*.

### EditSpinClass--Functional Organization--Expected Use

As an aid to understanding the EditSpinClass it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditSpinClass methods.

#### Non-Virtual Methods

---

The non-virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init <sub>v</sub>	initialize the EditSpinClass object
Kill <sub>v</sub>	shut down the EditSpinClass object

##### **Mainstream Use:**

TakeEvent <sub>v</sub>	handle events for the SPIN control
------------------------	------------------------------------

##### **Occasional Use:**

CreateContol <sub>v</sub>	create the SPIN control
SetAlerts <sub>v</sub>	alert keystrokes for the SPIN control

<sub>v</sub> These methods are also virtual.

<sub>i</sub> These methods are inherited from the EditClass

#### Virtual Methods

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init <sub>i</sub>	initialize the EditSpinClass object
CreateContol <sub>i</sub>	create the SPIN control
SetAlerts <sub>i</sub>	alert keystrokes for the SPIN control
TakeEvent <sub>i</sub>	handle events for the SPIN control
Kill <sub>i</sub>	shut down the EditSpinClass object

## CreateControl (create the edit-in-place SPIN control)

**CreateControl**, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place SPIN control and sets the FEQ property.

Implementation: The Init method calls the CreateControl method. The CreateControl method sets the value of the FEQ property. Use the Init method or the CreateControl method to set any required properties of the SPIN control.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
  SELF.ListBoxFeq = ListBox
  SELF.CreateControl()
  ASSERT(SELF.Feq)
  SELF.UseVar &= UseVar
  SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
  SELF.Feq{PROP:Use} = UseVar
  SELF.SetAlerts
```

See Also: FEQ, EditClass.CreateControl



# EditCheckClass

## EditCheckClass Overview

The EditCheckClass is an EditClass that supports a CHECK control. The EditCheckClass lets you implement a dynamic edit-in-place CHECK control for a column in a LIST.

## EditCheckClass Concepts

The EditCheckClass creates a CHECK control, accepts input from the end user, then returns the input to the variable specified by the Init method, typically the variable associated with a specific LIST cell—a field in the LIST control's data source QUEUE. The EditCheckClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditCheckClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## EditCheckClass Relationship to Other Application Builder Classes

### EditClass

The EditCheckClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseClass

The EditClass is loosely integrated into the BrowseClass. The BrowseClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

## EditCheckClass ABC Template Implementation

You can use the BrowseUpdateButtons control template (**Configure EditInPlace**) to generate the code to instantiate an EditCheckClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditCheckClass object's methods as needed. See *Control Templates—BrowseUpdateButtons* for more information.

## EditCheckClass Source Files

The EditCheckClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditCheckClass source code and their respective components are contained in:

ABEIP.INC  
ABEIP.CLW

EditCheckClass declarations  
EditCheckClass method definitions

## EditCheckClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditCheckClass object and a related BrowseClass object. The example page-loads a LIST of fieldnames and associated control attributes (such as color, icon, etc.), then edits the "Hide" items with an EditCheckClass object. Note that the BrowseClass object calls the "registered" EditCheckClass object's methods as needed.

```

PROGRAM
_ABCLlMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)
  INCLUDE('ABWINDOW.INC')           !declare WindowManager
  INCLUDE('ABBROWSE.INC')           !declare BrowseClass
  INCLUDE('ABEIP.INC')              !declare Edit-in-place classes

MAP
END

Property FILE,DRIVER('TOPSPEED'),PRE(PR),CREATE,BINDABLE,THREAD
NameKey KEY(PR:FieldName),NOCASE,OPT
Record RECORD,PRE()
FieldName STRING(30)
Color STRING(20)
Hidden STRING(1)
IconFile STRING(30)
ControlType STRING(12)
  END
  END

PropView VIEW(Property)
  END

PropQ QUEUE
PR:FieldName LIKE(PR:FieldName)
PR:Color LIKE(PR:Color)
PR:ControlType LIKE(PR:ControlType)
PR:Hidden LIKE(PR:Hidden) !edit this LIST field with a CHECK control
PR:IconFile LIKE(PR:IconFile)
ViewPosition STRING(1024)
  END

PropWindow WINDOW('Browse Field Properties'),AT(, ,318,137),IMM,SYSTEM,GRAY
  LIST,AT(8,4,303,113),USE(?PropList),IMM,HVSCROLL,FROM(PropQ),|
  FORMAT( '50L(2)|_M~Field Name~@s30@[70L(2)|_M~Color~@s20@' &|
    '60L(2)|_M~Control Type~@s12@' &|
    '20L(2)|_M~Hide~L(0)s1@/130L(2)|_M~Icon File~@s30@]|M')
  BUTTON('&Insert'),AT(169,121),USE(?Insert)
  BUTTON('&Change'),AT(218,121),USE(?Change),DEFAULT

```

```

        BUTTON( '&Delete' ), AT(267,121), USE(?Delete)
    END

Edit:PR:Hide CLASS(EditCheckClass) !declare Edit:PR:Color-EIP CHECK control
Init        PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
            END

ThisWindow  CLASS(WindowManager)
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL
            END

BRW1        CLASS(BrowseClass)      !declare BRW1, the BrowseClass object
Q           &PropQ                  ! that drives the EditClass objects--
            END                    ! i.e. calls Init, TakeEvent, Kill

GlobalErrors ErrorClass
Access:Property CLASS(FileManager)
Init          PROCEDURE
            END

Relate:Property CLASS(RelationManager)
Init          PROCEDURE
Kill          PROCEDURE,VIRTUAL
            END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD
CODE
GlobalErrors.Init
Relate:Property.Init
GlobalResponse = ThisWindow.Run()
Relate:Property.Kill
GlobalErrors.Kill

ThisWindow.Init PROCEDURE()
ReturnValue     BYTE,AUTO

CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
SELF.FirstField = ?PropList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:Property.Open
BRW1.Init(?PropList,PropQ.ViewPosition,PropView,PropQ,Relate:Property,SELF)

```

```

OPEN(PropWindow)
SELF.Opened=True
BRW1.Q &= PropQ
BRW1.AddSortOrder(,PR:NameKey)
BRW1.AddField(PR:FieldName,BRW1.Q.PR:FieldName)
BRW1.AddField(PR:Color,BRW1.Q.PR:Color)
BRW1.AddField(PR:ControlType,BRW1.Q.PR:ControlType)
BRW1.AddField(PR:Hidden,BRW1.Q.PR:Hidden)
BRW1.AddField(PR:IconFile,BRW1.Q.PR:IconFile)
BRW1.AddEditControl(Edit:PR:Hide,4)      !Use Edit:PR:Hide to edit BRW1 column 4
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change
BRW1.DeleteControl=?Delete
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue     BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Property.Close
RETURN ReturnValue

Edit:PR:Hide.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Feq{PROP:Text}='Hide '           !set EIP check box text
SELF.Feq{PROP:Value,1}='Y'           !set EIP check box true value
SELF.Feq{PROP:Value,2}='N'           !set EIP check box false value

Access:Property.Init PROCEDURE           !initialize FileManager
CODE
PARENT.Init(Property,GlobalErrors)
SELF.FileNameValue = 'Property'
SELF.Buffer &= PR:Record
SELF.Create = 1
SELF.AddKey(PR:NameKey,'PR:NameKey',0)

Relate:Property.Init PROCEDURE           !initialize RelationManager
CODE
Access:Property.Init
PARENT.Init(Access:Property,1)

Relate:Property.Kill PROCEDURE           !shut down RelationManager
CODE
Access:Property.Kill
PARENT.Kill

```

## EditCheckClass Properties

The EditCheckClass inherits all the properties of the EditClass from which it is derived. See *EditClass Properties* and *EditClass Concepts* for more information.

## EditCheckClass Methods

The EditCheckClass inherits all the methods of the EditClass from which it is derived. See *EditClass Methods* and *EditClass Concepts*.

In addition to (or instead of) the inherited methods, the EditCheckClass contains the following methods:

### **EditCheckClass Functional Organization—Expected Use**

As an aid to understanding the EditCheckClass it is useful to organize its methods into two large categories according to their expected use—the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditCheckClass methods.

#### **Non-Virtual Methods**

The non-virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init <sub>v</sub>	initialize the EditCheckClass object
Kill <sub>v</sub>	shut down the EditCheckClass object

##### **Mainstream Use:**

TakeEvent <sub>v</sub>	handle events for the CHECK control
------------------------	-------------------------------------

##### **Occasional Use:**

CreateControl <sub>v</sub>	create the CHECK control
SetAlerts <sub>v</sub>	alert keystrokes for the CHECK control

<sub>v</sub> These methods are also virtual.

<sub>i</sub> These methods are inherited from the EditClass

#### **Virtual Methods**

Typically you will not call these methods directly—the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init <sub>i</sub>	initialize the EditCheckClass object
CreateControl <sub>i</sub>	create the CHECK control
SetAlerts <sub>i</sub>	alert keystrokes for the CHECK control
TakeEvent <sub>i</sub>	handle events for the CHECK control
Kill <sub>i</sub>	shut down the EditCheckClass object

## CreateControl (create the edit-in-place CHECK control)

**CreateControl**, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place CHECK control and sets the FEQ property.

Implementation: The Init method calls the CreateControl method. The CreateControl method sets the value of the FEQ property. Use the Init method or the CreateControl method to set any required properties of the CHECK control.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts
```

See Also: FEQ, EditClass.CreateControl

# EditColorClass

## EditColorClassOverview

The EditColorClass is an EditClass that supports the Windows Color dialog by way of a dynamic edit-in-place COMBO control.

## EditColorClass Concepts

The EditColorClass creates a COMBO control with an ellipsis button that invokes the Windows Color dialog. The EditColorClass accepts a color selection from the end user, then returns the selection to the variable specified by the Init method, typically the variable associated with a specific LIST cell--a field in the LIST control's data source QUEUE.

The EditColorClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditColorClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## EditColorClass Relationship to Other Application Builder Classes

### EditClass

The EditColorClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseClass

The EditClass is loosely integrated into the BrowseClass. The BrowseClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

## EditColorClass ABC Template Implementation

You can use the BrowseUpdateButtons control template (**Configure EditInPlace**) to generate the code to instantiate an EditColorClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditColorClass object's methods as needed. See *Control Templates--BrowseUpdateButtons* for more information.

## EditColorClass Source Files

The EditColorClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditColorClass source code and their respective components are contained in:

ABEIP.INC	EditColorClass declarations
ABEIP.CLW	EditColorClass method definitions
ABEIP.TRN	EditColorClass translation strings

## EditColorClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditColorClass object and a related BrowseClass object. The example page-loads a LIST of fieldnames and associated control attributes (such as color, icon, etc.), then edits the "Color" items with an EditColorClass object. Note that the BrowseClass object calls the "registered" EditColorClass object's methods as needed.

```

PROGRAM

_ABCEllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)

INCLUDE('ABWINDOW.INC')      !declare WindowManager
INCLUDE('ABBROWSE.INC')      !declare BrowseClass
INCLUDE('ABEIP.INC')         !declare Edit-in-place classes

MAP
END

Property  FILE,DRIVER('TOPSPEED'),PRE(PR),CREATE,BINDABLE,THREAD
NameKey    KEY(PR:FieldName),NOCASE,OPT
Record     RECORD,PRE()
FieldName  STRING(30)
Color      STRING(20)
Hidden     STRING(1)
IconFile   STRING(30)
ControlType STRING(12)
          END
          END

PropView   VIEW(Property)
          END

PropQ      QUEUE
PR:FieldName LIKE(PR:FieldName)
PR:Color     LIKE(PR:Color)      !edit this LIST field with the color dialog
PR:ControlType LIKE(PR:ControlType)
PR:Hidden    LIKE(PR:Hidden)
PR:IconFile  LIKE(PR:IconFile)
ViewPosition STRING(1024)
          END

```

```

PropWindow WINDOW('Browse Field Properties'),AT(, ,318,137),IMM,SYSTEM,GRAY
  LIST,AT(8,4,303,113),USE(?PropList),IMM,HVSCROLL,FROM(PropQ),|
  FORMAT( '50L(2)|_M~Field Name~@s30@[70L(2)|_M~Color~@s20@' &|
    '60L(2)|_M~Control Type~@s12@' &|
    '20L(2)|_M~Hide~L(0)@s1@/130L(2)|_M~Icon File~@s30@]|M')
  BUTTON('&Insert'),AT(169,121),USE(?Insert)
  BUTTON('&Change'),AT(218,121),USE(?Change),DEFAULT
  BUTTON('&Delete'),AT(267,121),USE(?Delete)
END

Edit:PR:Color CLASS(EditColorClass) !declare Edit:PR:Color-EIP color dialog
Init          PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
              END

ThisWindow    CLASS(WindowManager)
Init          PROCEDURE(),BYTE,PROC,VIRTUAL
Kill         PROCEDURE(),BYTE,PROC,VIRTUAL
              END

BRW1          CLASS(BrowseClass)    !declare BRW1, the BrowseClass object
Q             &PropQ                ! that drives the EditClass objects--
              END                    ! i.e. calls Init, TakeEvent, Kill

GlobalErrors  ErrorClass
Access:Property CLASS(FileManager)
Init          PROCEDURE
              END

Relate:Property CLASS(RelationManager)
Init          PROCEDURE
Kill         PROCEDURE,VIRTUAL
              END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD
CODE
GlobalErrors.Init
Relate:Property.Init
GlobalResponse = ThisWindow.Run()
Relate:Property.Kill
GlobalErrors.Kill

```

```

ThisWindow.Init PROCEDURE()
ReturnValue      BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
SELF.FirstField = ?PropList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:Property.Open
BRW1.Init(?PropList,PropQ.ViewPosition,PropView,PropQ,Relate:Property,SELF)
OPEN(PropWindow)
SELF.Opened=True
BRW1.Q &= PropQ
BRW1.AddSortOrder(,PR:NameKey)
BRW1.AddField(PR:FieldName,BRW1.Q.PR:FieldName)
BRW1.AddField(PR:Color,BRW1.Q.PR:Color)
BRW1.AddField(PR:ControlType,BRW1.Q.PR:ControlType)
BRW1.AddField(PR:Hidden,BRW1.Q.PR:Hidden)
BRW1.AddField(PR:IconFile,BRW1.Q.PR:IconFile)
BRW1.AddEditControl(Edit:PR:Color,2)      !Use Edit:PR:Color to edit BRW1 column 2
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change
BRW1.DeleteControl=?Delete
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue      BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Property.Close
RETURN ReturnValue

Edit:PR:Color.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,?* UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Title='Select field color'      !set EIP color dialog title

Access:Property.Init PROCEDURE      !initialize FileManager
CODE
PARENT.Init(Property,GlobalErrors)
SELF.FileNameValue = 'Property'
SELF.Buffer &= PR:Record
SELF.Create = 1
SELF.AddKey(PR:NameKey, 'PR:NameKey', 0)

```

```
Relate:Property.Init PROCEDURE      !initialize RelationManager
CODE
Access:Property.Init
PARENT.Init(Access:Property,1)

Relate:Property.Kill PROCEDURE      !shut down RelationManager
CODE
Access:Property.Kill
PARENT.Kill
```

## EditColorClass Properties

### EditColorClass Properties

The EditColorClass inherits all the properties of the EditClass from which it is derived. See *EditClass Properties* for more information.

In addition to the inherited properties, the EditColorClass contains the following properties:

### Title (color dialog title text)

**Title**    CSTRING(256)

The **Title** property contains a string that sets the title bar text in the Windows color dialog.

Implementation:    The EditColorClass (TakeEvent method) uses the Title property as the *title* parameter to the COLORDIALOG procedure. See *COLORDIALOG* in the *Language Reference* for more information.

See Also:            TakeEvent

## EditColorClass Methods

### EditColorClass Functional Organization--Expected Use

As an aid to understanding the EditColorClass it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditColorClass methods.

#### Non-Virtual Methods

The non-virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init <sub>v</sub>	initialize the EditColorClass object
Kill <sub>v</sub>	shut down the EditColorClass object

##### **Mainstream Use:**

TakeEvent <sub>v</sub>	handle events for the edit control
------------------------	------------------------------------

##### **Occasional Use:**

CreateContol <sub>v</sub>	create the edit (COMBO) control
SetAlerts <sub>v</sub>	alert keystrokes for the edit control

<sub>v</sub> These methods are also virtual.

<sub>i</sub> These methods are inherited from the EditClass

#### Virtual Methods

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init <sub>i</sub>	initialize the EditColorClass object
CreateContol <sub>i</sub>	create the edit (COMBO) control
SetAlerts <sub>i</sub>	alert keystrokes for the edit control
TakeEvent <sub>i</sub>	handle events for the edit control
Kill <sub>i</sub>	shut down the EditColorClass object

## CreateControl (create the edit-in-place control)

**CreateControl**, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place COMBO control and sets the FEQ property.

Implementation: The Init method calls the CreateControl method. The CreateControl method creates a COMBO control with an ellipsis button and sets the value of the FEQ property.

Use the Init method or the CreateControl method to set any required properties of the COMBO control.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts
```

See Also: FEQ, EditClass.CreateControl

**TakeEvent (process edit-in-place events:EditColorClass)****TakeEvent**( *event* ), VIRTUAL

<b>TakeEvent</b>	Processes an event for the EditColorClass object.
<i>event</i>	An integer constant, variable, EQUATE, or expression that contains the event number (see EVENT in the <i>Language Reference</i> ).

The **TakeEvent** method processes an event for the EditColorClass object and returns a value indicating the user requested action. Valid actions are none, complete or OK, cancel, next record, previous record, next field, and previous field.

Implementation: The BrowseClass.AskRecord method calls the TakeEvent method. The TakeEvent method processes an EVENT:AlertKey for the edit-in-place control. On EVENT:DroppingDown, TakeEvent invokes the Windows color dialog and stores the color selection in the edited field specified by the Init method. Finally, TakeEvent returns a value indicating the user requested action. The BrowseClass.AskRecord method carries out the user requested action.

Corresponding EQUATEs for the possible edit-in-place actions are declared in ABBROWSE.INC as follows:

```

EditAction      ITEMIZE(0),PRE
None            EQUATE          ! no action
Forward        EQUATE          ! next field
Backward       EQUATE          ! previous field
Complete       EQUATE          ! OK
Cancel         EQUATE          ! cancel
Next           EQUATE          ! next record
Previous       EQUATE          ! previous record
Ignore        EQUATE          ! no action
END

```

Return Data Type:     BYTE

Example:

```

EditClassAction ROUTINE
CASE SELF.EditList.Control.TakeEvent(EVENT())
OF EditAction:Forward   !handle tab forward (new field, same record)
OF EditAction:Backward !handle tab backward (new field, same record)
OF EditAction:Next     !handle down arrow (new record, offer to save prior record)
OF EditAction:Previous !handle up arrow (new record, offer to save prior record)
OF EditAction:Complete !handle OK or enter key (save record)
OF EditAction:Cancel   !handle Cancel or esc key (restore record)
END

```

See Also:           Init, BrowseClass.AskRecord

---

# EditDropComboClass

## EditDropComboClass Overview

The EditDropComboClass is an EditClass that supports a dynamic edit-in-place COMBO control for a column in a LIST.

## EditDropComboClass Concepts

The EditDropComboClass creates a COMBO control, accepts input from the end user, then returns the input to the variable specified by the Init method, typically the variable associated with a specific LIST cell—a field in the LIST control's data source QUEUE. The EditDropComboClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, canceling the edit, or completing the edit (moving to a new record or row). The EditDropComboClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## Relationship to Other Application Builder Classes

### EditClass

---

The EditDropComboClass is derived from the EditDropListClass which in turn is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseEIPManagerClass

---

The EditClass is managed by the BrowseEIPManagerClass. The BrowseEIPManagerClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

## EditDropComboClass ABC Template Implementation

You can use the BrowseUpdateButtons control template (Configure EditInPlace) to generate the code to instantiate an EditDropComboClass object called EditInPlace:*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditDropComboClass object's methods as needed. See Control Templates—BrowseUpdateButtons for more information.

## EditDropComboClass Source Files

The EditDropComboClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditDropComboClass source code and their respective components are contained in:

ABEIP.INC  
ABEIP.CLW

EditDropComboClass declarations  
EditDropComboClass method definitions

## EditDropComboClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditDropComboClass object and a related BrowseClass object. The example page-loads a LIST of fieldnames and associated control attributes (such as color, icon, etc.), then edits the "ControlType" items with an EditDropComboClass object. Note that the BrowseClass object calls the "registered" EditDropComboClass object's methods as needed.

```

PROGRAM
  _ABCDllMode_ EQUATE(0)
  _ABCLinkMode_ EQUATE(1)

  INCLUDE('ABWINDOW.INC')           !declare WindowManager
  INCLUDE('ABBROWSE.INC')           !declare BrowseClass
  INCLUDE('ABEIP.INC')              !declare Edit-in-place classes

MAP
END

Property FILE,DRIVER('TOPSPEED'),PRE(PR),CREATE,BINDABLE,THREAD
NameKey KEY(PR:FieldName),NOCASE,OPT
Record RECORD,PRE()
FieldName STRING(30)
Color STRING(20)
Hidden STRING(1)
IconFile STRING(30)
ControlType STRING(12)
END
END

PropView VIEW(Property)
END

PropQ QUEUE
PR:FieldName LIKE(PR:FieldName)
PR:Color LIKE(PR:Color)
PR:ControlType LIKE(PR:ControlType) !edit this field with a COMBO control
PR:Hidden LIKE(PR:Hidden)
PR:IconFile LIKE(PR:IconFile)
ViewPosition STRING(1024)
END

PropWindow WINDOW('Browse Field Properties'),AT(, ,318,137),IMM,SYSTEM,GRAY
LIST,AT(8,4,303,113),USE(?PropList),IMM,HVSCROLL,FROM(PropQ),|
FORMAT( '50L(2)|_M~Field Name~@s30@[70L(2)|_M~Color~@s20@' &|
'60L(2)|_M~Control Type~@s12@' &|
'20L(2)|_M~Hide~L(0)s1@/130L(2)|_M~Icon File~@s30@]|M')
BUTTON('&Insert'),AT(169,121),USE(?Insert)

```

```

        BUTTON( '&Change' ), AT( 218, 121 ), USE( ?Change ), DEFAULT
        BUTTON( '&Delete' ), AT( 267, 121 ), USE( ?Delete )
    END
!declare Edit:PR:ControlType-EIP COMBO
Edit:PR:ControlType CLASS(EditDropComboClass)
Init    PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
    END

ThisWindow    CLASS(WindowManager)
Init          PROCEDURE(),BYTE,PROC,VIRTUAL
Kill          PROCEDURE(),BYTE,PROC,VIRTUAL
    END

BRW1 CLASS(BrowseClass) !declare BRW1, the BrowseClass object
Q      &PropQ           ! that drives the EditClass objects-
    END                ! i.e. calls Init, TakeEvent, Kill

GlobalErrors    ErrorClass
Access:Property CLASS(FileManager)
Init            PROCEDURE
    END

Relate:Property CLASS(RelationManager)
Init            PROCEDURE
Kill            PROCEDURE,VIRTUAL
    END

GlobalRequest    BYTE(0),THREAD
GlobalResponse    BYTE(0),THREAD
VCRRequest        LONG(0),THREAD

CODE
GlobalErrors.Init
Relate:Property.Init
GlobalResponse = ThisWindow.Run()
Relate:Property.Kill
GlobalErrors.Kill

ThisWindow.Init    PROCEDURE()
ReturnValue        BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
SELF.FirstField = ?PropList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:Property.Open

```

```

BRW1.Init(?PropList,PropQ.ViewPosition,PropView,PropQ,Relate:Property,SELF)
OPEN(PropWindow)
SELF.Opened=True
BRW1.Q &= PropQ
BRW1.AddSortOrder(,PR:NameKey)
BRW1.AddField(PR:FieldName,BRW1.Q.PR:FieldName)
BRW1.AddField(PR:Color,BRW1.Q.PR:Color)
BRW1.AddField(PR:ControlType,BRW1.Q.PR:ControlType)
BRW1.AddField(PR:Hidden,BRW1.Q.PR:Hidden)
BRW1.AddField(PR:IconFile,BRW1.Q.PR:IconFile)
!Use Edit:PR:ControlType to edit BRW1 col 3
BRW1.AddEditControl(Edit:PR:ControlType,3)
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change
BRW1.DeleteControl=?Delete
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Property.Close
RETURN ReturnValue

Edit:PR:ControlType.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED
ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Feq{PROP:From}='ENTRY|SPIN|TEXT|STRING'!set ControlType droplist
choices

Access:Property.Init PROCEDURE !initialize FileManager
CODE
PARENT.Init(Property,GlobalErrors)
SELF.FileNameValue = 'Property'
SELF.Buffer &= PR:Record
SELF.Create = 1
SELF.AddKey(PR:NameKey,'PR:NameKey',0)
Relate:Property.Init PROCEDURE !initialize RelationManager
CODE
Access:Property.Init
PARENT.Init(Access:Property,1)
Relate:Property.Kill PROCEDURE !shut down RelationManager
CODE
Access:Property.Kill
PARENT.Kill

```

## EditDropComboClass Properties

The EditDropComboClass inherits all of the properties of the EditDropListClass and EditClass from which is derived. See EditClass Properties and EditClass Concepts for more information.



## CreateControl (create the edit-in-place COMBO control)

CreateControl, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place COMBO control and sets the FEQ property.

Implementation: The Init method calls the CreateControl method. The CreateControl method sets the value of the FEQ property. Use the Init method or the CreateControl method to set any required properties of the COMBO control.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts
```

See Also: FEQ, EditClass.CreateControl

---

# EditDropListClass

## EditDropListClass Overview

The EditDropListClass is an EditClass that supports a DROPLIST control. The EditDropListClass lets you implement a dynamic edit-in-place DROPLIST control for a column in a LIST.

## EditDropListClass Concepts

The EditDropListClass creates a DROPLIST control, accepts input from the end user, then returns the input to the variable specified by the Init method, typically the variable associated with a specific LIST cell--a field in the LIST control's data source QUEUE. The EditDropListClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditDropListClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## EditDropListClass Relationship to Other Application Builder Classes

### EditClass

---

The EditDropListClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseClass

---

The EditClass is loosely integrated into the BrowseClass. The BrowseClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

## EditDropListClass ABC Template Implementation

You can use the BrowseUpdateButtons control template (**Configure EditInPlace**) to generate the code to instantiate an EditDropListClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditDropListClass object's methods as needed. See *Control Templates--BrowseUpdateButtons* for more information.

## EditDropListClass Source Files

The EditDropListClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditDropListClass source code and their respective components are contained in:

ABEIP.INC  
ABEIP.CLW

EditDropListClass declarations  
EditDropListClass method definitions

## EditDropListClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditDropListClass object and a related BrowseClass object. The example page-loads a LIST of fieldnames and associated control attributes (such as color, icon, etc.), then edits the "ControlType" items with an EditDropListClass object. Note that the BrowseClass object calls the "registered" EditDropListClass object's methods as needed.

```

PROGRAM
_ABCEllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)
  INCLUDE('ABWINDOW.INC')           !declare WindowManager
  INCLUDE('ABBROWSE.INC')           !declare BrowseClass
  INCLUDE('ABEIP.INC')              !declare Edit-in-place classes

MAP
END

Property FILE,DRIVER('TOPSPEED'),PRE(PR),CREATE,BINDABLE,THREAD
NameKey   KEY(PR:FieldName),NOCASE,OPT
Record    RECORD,PRE()
FieldName STRING(30)
Color     STRING(20)
Hidden    STRING(1)
IconFile  STRING(30)
ControlType STRING(12)
          END
          END
PropView  VIEW(Property)
          END

PropQ     QUEUE
PR:FieldName LIKE(PR:FieldName)
PR:Color     LIKE(PR:Color)
PR:ControlType LIKE(PR:ControlType) !edit this field with a DROPLIST control
PR:Hidden    LIKE(PR:Hidden)
PR:IconFile  LIKE(PR:IconFile)
ViewPosition STRING(1024)
          END

PropWindow WINDOW('Browse Field Properties'),AT(, ,318,137),IMM,SYSTEM,GRAY
LIST,AT(8,4,303,113),USE(?PropList),IMM,HVSCROLL,FROM(PropQ),|
FORMAT( '50L(2)|_M~Field Name~@s30@[70L(2)|_M~Color~@s20@' &|
'60L(2)|_M~Control Type~@s12@' &|
'20L(2)|_M~Hide~L(0)@s1@/130L(2)|_M~Icon File~@s30@]|M')
BUTTON('&Insert'),AT(169,121),USE(?Insert)
BUTTON('&Change'),AT(218,121),USE(?Change),DEFAULT
BUTTON('&Delete'),AT(267,121),USE(?Delete)
          END

```

```

!declare Edit:PR:ControlType-EIP DROPLIST
Edit:PR:ControlType CLASS(EditDropListClass)
Init      PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
          END

ThisWindow CLASS(WindowManager)
Init      PROCEDURE(),BYTE,PROC,VIRTUAL
Kill      PROCEDURE(),BYTE,PROC,VIRTUAL
          END

BRW1      CLASS(BrowseClass) !declare BRW1, the BrowseClass object
Q         &PropQ             ! that drives the EditClass objects--
          END                ! i.e. calls Init, TakeEvent, Kill

GlobalErrors ErrorClass
Access:Property CLASS(FileManager)
Init          PROCEDURE
          END

Relate:Property CLASS(RelationManager)
Init         PROCEDURE
Kill         PROCEDURE,VIRTUAL
          END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD

CODE
GlobalErrors.Init
Relate:Property.Init
GlobalResponse = ThisWindow.Run()
Relate:Property.Kill
GlobalErrors.Kill

ThisWindow.Init PROCEDURE()
ReturnValue     BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
SELF.FirstField = ?PropList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:Property.Open
BRW1.Init(?PropList,PropQ.ViewPosition,PropView,PropQ,Relate:Property,SELF)
OPEN(PropWindow)
SELF.Opened=True

```

```

BRW1.Q &= PropQ
BRW1.AddSortOrder(,PR:NameKey)
BRW1.AddField(PR:FieldName,BRW1.Q.PR:FieldName)
BRW1.AddField(PR:Color,BRW1.Q.PR:Color)
BRW1.AddField(PR:ControlType,BRW1.Q.PR:ControlType)
BRW1.AddField(PR:Hidden,BRW1.Q.PR:Hidden)
BRW1.AddField(PR:IconFile,BRW1.Q.PR:IconFile)
!Use Edit:PR:ControlType to edit BRW1 col 3
BRW1.AddEditControl(Edit:PR:ControlType,3)
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change
BRW1.DeleteControl=?Delete
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Property.Close
RETURN ReturnValue

Edit:PR:ControlType.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Feq{PROP:From}='ENTRY|SPIN|TEXT|STRING'!set ControlType droplist choices

Access:Property.Init PROCEDURE !initialize FileManager
CODE
PARENT.Init(Property,GlobalErrors)
SELF.FileNameValue = 'Property'
SELF.Buffer &= PR:Record
SELF.Create = 1
SELF.AddKey(PR:NameKey,'PR:NameKey',0)

Relate:Property.Init PROCEDURE !initialize RelationManager
CODE
Access:Property.Init
PARENT.Init(Access:Property,1)

Relate:Property.Kill PROCEDURE !shut down RelationManager
CODE
Access:Property.Kill
PARENT.Kill

```

## EditDropListClass Properties

### **EditDropListClass Properties**

The EditDropListClass inherits all the properties of the EditClass from which it is derived. See *EditClass Properties* and *EditClass Concepts* for more information.

## EditDropListClass Methods

### EditDropListClass Functional Organization--Expected Use

As an aid to understanding the EditDropListClass it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditDropListClass methods.

#### Non-Virtual Methods

The non-virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init <sub>v</sub>	initialize the EditDropListClass object
Kill <sub>v</sub>	shut down the EditDropListClass object

##### **Mainstream Use:**

TakeEvent <sub>v</sub>	handle events for the LIST control
------------------------	------------------------------------

##### **Occasional Use:**

CreateControl <sub>v</sub>	create the LIST control
SetAlerts <sub>v</sub>	alert keystrokes for the LIST control

<sub>v</sub> These methods are also virtual.

<sub>i</sub> These methods are inherited from the EditClass

#### Virtual Methods

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init <sub>i</sub>	initialize the EditDropListClass object
CreateControl <sub>i</sub>	create the LIST control
SetAlerts <sub>i</sub>	alert keystrokes for the LIST control
TakeEvent <sub>i</sub>	handle events for the LIST control
Kill <sub>i</sub>	shut down the EditDropListClass object

## **CreateControl (create the edit-in-place DROPLIST control)**

**CreateControl**, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place DROPLIST control and sets the FEQ property.

Implementation:       The Init method calls the CreateControl method. The CreateControl method sets the value of the FEQ property. Use the Init method or the CreateControl method to set any required properties of the DROPLIST control.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts
```

See Also:               FEQ, EditClass.CreateControl

## SetAlerts (alert keystrokes for the edit control:EditDropListClass)

### SetAlerts, VIRTUAL

The **SetAlerts** method alerts appropriate keystrokes for the edit-in-place DROPLIST control.

Implementation: The Init method calls the CreateControl method to create the input control and set the FEQ property. The Init method then calls the SetAlerts method to alert appropriate edit-in-place keystrokes for the edit control. Alerted keys are:

TabKey	!next field
ShiftTab	!previous field
EnterKey	!complete and save
EscKey	!complete and cancel

**Tip:** Arrowup and Arrowdown keys are not alerted for a DROPLIST control because these keys are used to navigate within the DROPLIST.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts
```

See Also: Init

## SetReadOnly (set edit control to read-only:EditDropClass)

**SetReadOnly( *state* ), VIRTUAL**

<b>SetReadOnly</b>	The <b>SetReadOnly</b> method places the edit-in-place control in a read-only state.
<i>state</i>	An integer constant, variable, EQUATE, or expression that indicates whether to disable the droplist control's dropdown button. A value of one (1 or True) disables the button. A value of zero (0 or False) has no effect on the control.

Implementation: The SetReadOnly method uses PROP:ReadOnly to place the edit-in-place control in a read-only state. After the parent call in the Init method of the EditInPlace object is the recommended place to call SetReadOnly.

Example: `EditInPlace::CUS:Number.SetReadOnly( )`

See Also: [ReadOnly](#)

**TakeEvent (process edit-in-place events:EditDropList Class)****TakeEvent**( *event* ), VIRTUAL

<b>TakeEvent</b>	Processes an event for the EditDropListClass object.
<i>event</i>	An integer constant, variable, EQUATE, or expression that contains the event number (see EVENT in the <i>Language Reference</i> ).

The **TakeEvent** method processes an event for the EditDropListClass object and returns a value indicating the user requested action. Valid actions are none, complete or OK, cancel, next record, previous record, next field, and previous field.

Implementation: The TakeEvent method is called by the WindowManager.TakeEvent method. The TakeEvent method processes an EVENT:AlertKey for the edit-in-place control. TakeEvent returns a value indicating the user requested action.

Corresponding EQUATEs for the possible edit-in-place actions are declared in ABEIP.INC as follows:

```

EditAction      ITEMIZE(0),PRE
None            EQUATE                ! no action
Forward        EQUATE                ! next field
Backward       EQUATE                ! previous field
Complete       EQUATE                ! OK
Cancel         EQUATE                ! cancel
Next           EQUATE                ! next record
Previous       EQUATE                ! previous record
Ignore         EQUATE                ! no action
                END

```

Return Data Type: BYTE

Example:

```

WindowManager.TakeEvent  PROCEDURE
CODE
! Event handling code
  LOOP i=1 TO RECORDS(SELF.FileDrops)
    GET(SELF.FileDrops,i)
    ASSERT(~ERRORCODE())
    SELF.FileDrops.FileDrop.TakeEvent
  END

```

See Also: Init



---

# EditEntryClass

## EditEntryClass Overview

The EditEntryClass is an EditClass that supports an ENTRY control. The EditEntryClass lets you implement a dynamic edit-in-place ENTRY control for a column in a LIST.

## EditEntryClass Concepts

The EditEntryClass creates an ENTRY control, accepts input from the end user, then returns the input to the variable specified by the Init method, typically the variable associated with a specific LIST cell--a field in the LIST control's data source QUEUE. The EditEntryClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditEntryClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## EditEntryClass Relationship to Other Application Builder Classes

### EditClass

---

The EditEntryClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseClass

---

The EditClass is loosely integrated into the BrowseClass. The BrowseClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

**Tip:** The BrowseClass instantiates the EditEntryClass as the default edit-in-place object whenever edit-in-place is requested (when BrowseClass.AskProcedure is zero).

## EditEntryClass ABC Template Implementation

When you check the **Use EditInPlace** box and you do not set column-specific configuration, the BrowseUpdateButtons control template relies on the default BrowseBox edit-in-place behavior--which is the default BrowseClass edit-in-place implementation--which instantiates an EditEntryClass object for each BrowseBox column.

You can also use the BrowseUpdateButtons control template (**Configure EditInPlace**) to explicitly instantiate an EditEntryClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditEntryClass object's methods as needed. By explicitly requesting an EditEntryClass object, you gain access to EditEntryClass method embed points. See *Control Templates--BrowseUpdateButtons* for more information.

## EditEntryClass Source Files

The EditEntryClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditEntryClass source code and their respective components are contained in:

ABEIP.INC	EditEntryClass declarations
ABEIP.CLW	EditEntryClass method definitions

## EditEntryClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditEntryClass object and a related BrowseClass object. The example page-loads a LIST of fieldnames and associated control attributes (such as color, icon, etc.), then edits the items with an EditEntryClass object. Note that the BrowseClass object calls the EditEntryClass object's methods as needed.

```

PROGRAM
  _ABCDllMode_ EQUATE(0)
  _ABCLinkMode_ EQUATE(1)

  INCLUDE('ABWINDOW.INC')    !declare WindowManager
  INCLUDE('ABBROWSE.INC')    !declare BrowseClass
  INCLUDE('ABEIP.INC')       !declare Edit-in-place classes

MAP
END

Property FILE, DRIVER('TOPSPEED'), PRE(PR), CREATE, BINDABLE, THREAD
NameKey   KEY(PR:FieldName), NOCASE, OPT
Record    RECORD, PRE()
FieldName STRING(30)
Color     STRING(20)
Hidden    STRING(1)

```

```

IconFile      STRING(30)
ControlType   STRING(12)
              END
              END

PropView      VIEW(Property)
              END

PropQ         QUEUE
PR:FieldName  LIKE(PR:FieldName)
PR:Color      LIKE(PR:Color)
PR:ControlType LIKE(PR:ControlType)
PR:Hidden     LIKE(PR:Hidden)
PR:IconFile   LIKE(PR:IconFile)
ViewPosition  STRING(1024)
              END

PropWindow WINDOW('Browse Field Properties'),AT(, ,318,137),IMM,SYSTEM,GRAY
          LIST,AT(8,4,303,113),USE(?PropList),IMM,HVSCROLL,FROM(PropQ),|
          FORMAT( '50L(2)|_M~Field Name~@s30@[70L(2)|_M~Color~@s20@' &|
          '60L(2)|_M~Control Type~@s12@' &|
          '20L(2)|_M~Hide~L(0)@s1@/130L(2)|_M~Icon File~@s30@]|M')
          BUTTON('&Insert'),AT(169,121),USE(?Insert)
          BUTTON('&Change'),AT(218,121),USE(?Change),DEFAULT
          BUTTON('&Delete'),AT(267,121),USE(?Delete)
          END

Edit:PR:Name CLASS(EditEntryClass) !declare Edit:PR:Name-EIP ENTRY control
Init         PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,?* UseVar),VIRTUAL
              END

ThisWindow   CLASS(WindowManager)
Init         PROCEDURE(),BYTE,PROC,VIRTUAL
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL
              END

BRW1         CLASS(BrowseClass)     !declare BRW1, the BrowseClass object
Q            &PropQ                  ! that drives the EditClass objects--
              END                    ! i.e. calls Init, TakeEvent, Kill

GlobalErrors ErrorClass

Access:Property CLASS(FileManager)
Init           PROCEDURE
              END

Relate:Property CLASS(RelationManager)
Init          PROCEDURE

```

```
Kill          PROCEDURE,VIRTUAL
              END
```

```
GlobalRequest  BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest     LONG(0),THREAD
```

```
CODE
GlobalErrors.Init
Relate:Property.Init
GlobalResponse = ThisWindow.Run()
Relate:Property.Kill
GlobalErrors.Kill
```

```
ThisWindow.Init PROCEDURE()
ReturnValue     BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
SELF.FirstField = ?PropList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:Property.Open
BRW1.Init(?PropList,PropQ.ViewPosition,PropView,PropQ,Relate:Property,SELF)
OPEN(PropWindow)
SELF.Opened=True
BRW1.Q &= PropQ
BRW1.AddSortOrder(,PR:NameKey)
BRW1.AddField(PR:FieldName,BRW1.Q.PR:FieldName)      !edit with Edit:PR:Name
BRW1.AddField(PR:Color,BRW1.Q.PR:Color)              !default EditEntryClass
BRW1.AddField(PR:ControlType,BRW1.Q.PR:ControlType)!edit with default EditEntryClass
BRW1.AddField(PR:Hidden,BRW1.Q.PR:Hidden)            !edit with default EditEntryClass
BRW1.AddField(PR:IconFile,BRW1.Q.PR:IconFile)        !edit with default EditEntryClass
BRW1.AddEditControl(Edit:PR:Name,1)                  !Use Edit:PR:Name for BRW1 col 1
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change
BRW1.DeleteControl=?Delete
SELF.SetAlerts()
RETURN ReturnValue
```

```
ThisWindow.Kill PROCEDURE()
ReturnValue     BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Property.Close
RETURN ReturnValue
```

```
Edit:PR:Name.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)

CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Feq{PROP:CAP}=True           !force EIP mixed case input

Access:Property.Init PROCEDURE           !initialize FileManager
CODE
PARENT.Init(Property,GlobalErrors)
SELF.FileNameValue = 'Property'
SELF.Buffer &= PR:Record
SELF.Create = 1
SELF.AddKey(PR:NameKey,'PR:NameKey',0)

Relate:Property.Init PROCEDURE           !initialize RelationManager
CODE
Access:Property.Init
PARENT.Init(Access:Property,1)

Relate:Property.Kill PROCEDURE           !shut down RelationManager
CODE
Access:Property.Kill
PARENT.Kill
```

## EditEntryClass Properties

### **EditEntryClass Properties**

The EditEntryClass inherits all the properties of the EditClass from which it is derived. See *EditClass Properties* and *EditClass Concepts* for more information.

## EditEntryClass Methods

### EditEntryClass Methods

The EditEntryClass inherits all the methods of the EditClass from which it is derived. See *EditClass Methods* and *EditClass Concepts*.

### EditEntryClass Functional Organization--Expected Use

As an aid to understanding the EditEntryClass it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditEntryClass methods.

#### Non-Virtual Methods

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init <sub>v</sub>	initialize the EditEntryClass object
Kill <sub>v</sub>	shut down the EditEntryClass object

##### **Mainstream Use:**

TakeEvent <sub>v</sub>	handle events for the ENTRY control
------------------------	-------------------------------------

##### **Occasional Use:**

CreateContol <sub>v</sub>	create the ENTRY control
SetAlerts <sub>v</sub>	alert keystrokes for the ENTRY control

<sub>v</sub> These methods are also virtual.

<sub>i</sub> These methods are inherited from the EditClass

#### Virtual Methods

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init <sub>i</sub>	initialize the EditEntryClass object
CreateContol <sub>i</sub>	create the ENTRY control
SetAlerts <sub>i</sub>	alert keystrokes for the ENTRY control
TakeEvent <sub>i</sub>	handle events for the ENTRY control
Kill <sub>i</sub>	shut down the EditEntryClass object

## **CreateControl (create the edit-in-place ENTRY control)**

**CreateControl**, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place ENTRY control and sets the FEQ property.

Implementation: The Init method calls the CreateControl method. The CreateControl method sets the value of the FEQ property. Use the Init method or the CreateControl method to set any required properties of the ENTRY control.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
  SELF.ListBoxFeq = ListBox
  SELF.CreateControl()
  ASSERT(SELF.Feq)
  SELF.UseVar &= UseVar
  SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
  SELF.Feq{PROP:Use} = UseVar
  SELF.SetAlerts
```

See Also: FEQ, EditClass.CreateControl

---

# EditFileClass

## EditFileClass Overview

The EditFileClass is an EditClass that supports the Windows File dialog by way of a dynamic edit-in-place COMBO control.

## EditFileClass Concepts

The EditFileClass creates a COMBO control with an ellipsis button that invokes the Windows File dialog. The EditFileClass accepts a pathname selection from the end user, then returns the selection to the variable specified by the Init method, typically the variable associated with a specific LIST cell--a field in the LIST control's data source QUEUE.

The EditFileClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditFileClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## EditFileClass Relationship to Other Application Builder Classes

### EditClass

---

The EditFileClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseClass

---

The EditClass is loosely integrated into the BrowseClass. The BrowseClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

## EditFileClass ABC Template Implementation

You can use the BrowseUpdateButtons control template (**Configure EditInPlace**) to generate the code to instantiate an EditFileClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditFileClass object's methods as needed. See *Control Templates--BrowseUpdateButtons* for more information.

## EditFileClass Source Files

The EditFileClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditFileClass source code and their respective components are contained in:

ABEIP.INC	EditFileClass declarations
ABEIP.CLW	EditFileClass method definitions

## EditFileClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditFileClass object and a related BrowseClass object. The example page-loads a LIST of fieldnames and associated control attributes (such as color, icon, etc.), then edits the "IconFile" items with an EditFileClass object. Note that the BrowseClass object calls the "registered" EditFileClass object's methods as needed.

```

PROGRAM

_ABCEllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)

INCLUDE('ABWINDOW.INC')           !declare WindowManager
INCLUDE('ABBROWSE.INC')           !declare BrowseClass
INCLUDE('ABEIP.INC')              !declare Edit-in-place classes

MAP
END

Property FILE,DRIVER('TOPSPEED'),PRE(PR),CREATE,BINDABLE,THREAD
NameKey KEY(PR:FieldName),NOCASE,OPT
Record RECORD,PRE()
FieldName STRING(30)
Color STRING(20)
Hidden STRING(1)
IconFile STRING(30)
ControlType STRING(12)
END
END

PropView VIEW(Property)
END

PropQ QUEUE
PR:FieldName LIKE(PR:FieldName)
PR:Color LIKE(PR:Color)
PR:ControlType LIKE(PR:ControlType)
PR:Hidden LIKE(PR:Hidden)
PR:IconFile LIKE(PR:IconFile) !edit this LIST field with the file dialog
ViewPosition STRING(1024)
END

```

```

PropWindow WINDOW('Browse Field Properties'),AT(, ,318,137),IMM,SYSTEM,GRAY
  LIST,AT(8,4,303,113),USE(?PropList),IMM,HVSCROLL,FROM(PropQ),|
  FORMAT( '50L(2)|_M-Field Name~@s30@[70L(2)|_M-Color~@s20@' &|
    '60L(2)|_M-Control Type~@s12@' &|
    '20L(2)|_M-Hide~L(0)@s1@/130L(2)|_M-Icon File~@s30@]|M')
  BUTTON('&Insert'),AT(169,121),USE(?Insert)
  BUTTON('&Change'),AT(218,121),USE(?Change),DEFAULT
  BUTTON('&Delete'),AT(267,121),USE(?Delete)
END

Edit:PR:IconFile CLASS(EditFileClass) !declare Edit:PR:IconFile-EIP file dialog
Init
  PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
  END

ThisWindow CLASS(WindowManager)
Init
  PROCEDURE(),BYTE,PROC,VIRTUAL
Kill
  PROCEDURE(),BYTE,PROC,VIRTUAL
  END

BRW1 CLASS(BrowseClass) !declare BRW1, the BrowseClass object
Q
  &PropQ ! that drives the EditClass objects--
  END ! i.e. calls Init, TakeEvent, Kill

GlobalErrors ErrorClass
Access:Property CLASS(FileManager)
Init
  PROCEDURE
  END

Relate:Property CLASS(RelationManager)
Init
  PROCEDURE
Kill
  PROCEDURE,VIRTUAL
  END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest LONG(0),THREAD
CODE
GlobalErrors.Init
Relate:Property.Init
GlobalResponse = ThisWindow.Run()
Relate:Property.Kill
GlobalErrors.Kill

ThisWindow.Init PROCEDURE()
ReturnValue BYTE,AUTO
CODE
SELF.Request = GlobalRequest

```

```
ReturnValue = PARENT.Init()
SELF.FirstField = ?PropList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:Property.Open
BRW1.Init(?PropList,PropQ.ViewPosition,PropView,PropQ,Relate:Property,SELF)
OPEN(PropWindow)
SELF.Opened=True
  BRW1.Q &= PropQ
  BRW1.AddSortOrder(,PR:NameKey)
  BRW1.AddField(PR:FieldName,BRW1.Q.PR:FieldName)
  BRW1.AddField(PR:Color,BRW1.Q.PR:Color)
  BRW1.AddField(PR:ControlType,BRW1.Q.PR:ControlType)
  BRW1.AddField(PR:Hidden,BRW1.Q.PR:Hidden)
  BRW1.AddField(PR:IconFile,BRW1.Q.PR:IconFile)
  BRW1.AddEditControl(Edit:PR:IconFile,5)      !Use Edit:PR:IconFile to edit BRW1 col 5
  BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
  BRW1.InsertControl=?Insert
  BRW1.ChangeControl=?Change
  BRW1.DeleteControl=?Delete
  SELF.SetAlerts()
  RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
  CODE
  ReturnValue = PARENT.Kill()
  Relate:Property.Close
  RETURN ReturnValue

Edit:PR:IconFile.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*?
UseVar)
  CODE
  PARENT.Init(FieldNumber,ListBox,UseVar)
  SELF.Title='Select icon file'                !set EIP file dialog title
  SELF.FilePattern='Icon files *.ico|*.ico'    !set EIP file dialog file masks
  SELF.FileMask=FILE:KeepDir+FILE:LongName    !set EIP file dialog behavior flag
```

---

```
Access:Property.Init PROCEDURE           !initialize FileManager
CODE
PARENT.Init(Property,GlobalErrors)
SELF.FileNameValue = 'Property'
SELF.Buffer &= PR:Record
SELF.Create = 1
SELF.AddKey(PR:NameKey,'PR:NameKey',0)

Relate:Property.Init PROCEDURE           !initialize RelationManager
CODE
Access:Property.Init
PARENT.Init(Access:Property,1)

Relate:Property.Kill PROCEDURE           !shut down RelationManager
CODE
Access:Property.Kill
PARENT.Kill
```

## EditFileClass Properties

### EditFileClass Properties

The EditFileClass inherits all the properties of the EditClass from which it is derived. See *EditClass Properties* and *EditClass Concepts* for more information.

### FileMask (file dialog behavior)

**FileMask**      **BYTE**

The **FileMask** property is a bitmap that indicates the type of file action the Windows file dialog performs (select, multi-select, save directory, lock directory, suppress errors).

Implementation:      The EditFileClass (TakeEvent method) uses the FileMask property as the *flag* parameter to the FILEDIALOG procedure. See *FILEDIALOG* in the *Language Reference* for more information.

See Also:              TakeEvent

### FilePattern (file dialog filter)

**FilePattern**      **CSTRING(1024)**

The **FilePattern** property contains a text string that defines both the file masks and the file mask descriptions that appear in the file dialog's **List Files of Type** drop-down list. The first mask is the default selection in the file dialog.

The FilePattern property should contain one or more descriptions followed by their corresponding file masks in the form *description|mask|description|mask*. All elements in the string must be delimited by the vertical bar (|). For example, 'all files \*.\*|\*.\*|Clarion source \*.clw;\*.inc|\*.clw;\*.inc' defines two selections for the File dialog's **List Files of Type** drop-down list. See the *extensions* parameter to the FILEDIALOG function in the *Language Reference* for more information.

**Title (file dialog title text)****Title**    **CSTRING(256)**

The **Title** property contains a string that sets the title bar text in the Windows file dialog.

Implementation:    The EditFileClass (TakeEvent method) uses the Title property as the *title* parameter to the FILEDIALOG procedure. See *FILEDIALOG* in the *Language Reference* for more information.

See Also:    TakeEvent

## EditFileClass Methods

### EditFileClass Functional Organization--Expected Use

As an aid to understanding the EditFileClass it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditFileClass methods.

#### Non-Virtual Methods

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### Housekeeping (one-time) Use:

Init <sub>v</sub>	initialize the EditFileClass object
Kill <sub>v</sub>	shut down the EditFileClass object

##### Mainstream Use:

TakeEvent <sub>v</sub>	handle events for the edit control
------------------------	------------------------------------

##### Occasional Use:

CreateControl <sub>v</sub>	create the edit (COMBO) control
SetAlerts <sub>v</sub>	alert keystrokes for the edit control

<sub>v</sub> These methods are also virtual.

<sub>i</sub> These methods are inherited from the EditClass

#### Virtual Methods

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init <sub>i</sub>	initialize the EditFileClass object
CreateControl <sub>i</sub>	create the edit (COMBO) control
SetAlerts <sub>i</sub>	alert keystrokes for the edit control
TakeEvent <sub>i</sub>	handle events for the edit control
Kill <sub>i</sub>	shut down the EditFileClass object

## **CreateControl (create the edit-in-place control:EditFileClass)**

**CreateControl**, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place COMBO control and sets the FEQ property.

Implementation: The Init method calls the CreateControl method. The CreateControl method creates a COMBO control with an ellipsis button and sets the value of the FEQ property.

Use the Init method or the CreateControl method to set any required properties of the COMBO control.

Example:

```

EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts

```

See Also: FEQ, EditClass.CreateControl

**TakeEvent (process edit-in-place events:EditFileClass)****TakeEvent**( *event* ), VIRTUAL

<b>TakeEvent</b>	Processes an event for the EditFileClass object.
<i>event</i>	An integer constant, variable, EQUATE, or expression that contains the event number (see EVENT in the <i>Language Reference</i> ).

The **TakeEvent** method processes an event for the EditFileClass object and returns a value indicating the user requested action. Valid actions are none, complete or OK, cancel, next record, previous record, next field, and previous field.

Implementation: The BrowseClass.AskRecord method calls the TakeEvent method. The TakeEvent method processes an EVENT:AlertKey for the edit-in-place control. On EVENT:DroppingDown, TakeEvent invokes the Windows file dialog and stores the pathname selection in the edited field specified by the Init method. Finally, TakeEvent returns a value indicating the user requested action. The BrowseClass.AskRecord method carries out the user requested action.

Corresponding EQUATEs for the possible edit-in-place actions are declared in ABBROWSE.INC as follows:

```

EditAction    ITEMIZE(0),PRE
None          EQUATE          ! no action
Forward       EQUATE          ! next field
Backward      EQUATE          ! previous field
Complete      EQUATE          ! OK
Cancel        EQUATE          ! cancel
Next          EQUATE          ! next record
Previous      EQUATE          ! previous record
Ignore        EQUATE          ! no action
END

```

Return Data Type: BYTE

Example:

```
EditClassAction ROUTINE
  CASE SELF.EditList.Control.TakeEvent(EVENT())
  OF EditAction:Forward    !handle tab forward (new field, same record)
  OF EditAction:Backward  !handle tab backward (new field, same record)
  OF EditAction:Next      !handle down arrow (new record, offer to save prior record)
  OF EditAction:Previous  !handle up arrow (new record, offer to save prior record)
  OF EditAction:Complete  !handle OK or enter key (save record)
  OF EditAction:Cancel    !handle Cancel or esc key (restore record)
  END
```

See Also:           Init, BrowseClass.AskRecord

---

# EditFontClass

## EditFontClass Overview

The EditFontClass is an EditClass that supports the Windows Font dialog by way of a dynamic edit-in-place COMBO control.

## EditFontClass Concepts

The EditFontClass creates a COMBO control with an ellipsis button that invokes the Windows Font dialog. The EditFontClass accepts a font specification from the end user, then returns the specification to the variable specified by the Init method, typically the variable associated with a specific LIST cell--a field in the LIST control's data source QUEUE.

The EditFontClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditFontClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## EditFontClass Relationship to Other Application Builder Classes

### EditClass

---

The EditFontClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseClass

---

The EditClass is loosely integrated into the BrowseClass. The BrowseClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

## EditFontClass ABC Template Implementation

You can use the BrowseUpdateButtons control template (**Configure EditInPlace**) to generate the code to instantiate an EditFontClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditFontClass object's methods as needed. See *Control Templates--BrowseUpdateButtons* for more information.

## EditFontClass Source Files

The EditFontClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditFontClass source code and their respective components are contained in:

ABEIP.INC	EditFontClass declarations
ABEIP.CLW	EditFontClass method definitions

## EditFontClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditFontClass object and a related BrowseClass object. The example page-loads a LIST of fieldnames and associated control attributes (such as color, font, icon, etc.), then edits the "Font" items with an EditFontClass object. Note that the BrowseClass object calls the "registered" EditFontClass object's methods as needed.

```

PROGRAM

_ABCEllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)

INCLUDE('ABWINDOW.INC')           !declare WindowManager
INCLUDE('ABBROWSE.INC')           !declare BrowseClass
INCLUDE('ABEIP.INC')              !declare EditInPlace classes

MAP   END

Property  FILE,DRIVER('TOPSPEED'),PRE(PR),CREATE,BINDABLE,THREAD
NameKey   KEY(PR:FieldName),NOCASE,OPT
Record    RECORD,PRE()
FieldName STRING(30)
Color     STRING(20)
Hidden    STRING(1)
IconFile  STRING(30)
Font      STRING(40)
ControlType STRING(12)
ApplyTo   CSTRING(500)
          END
          END

PropView VIEW(Property)
          END

PropQ     QUEUE
PR:FieldName LIKE(PR:FieldName)
PR:Color     LIKE(PR:Color)
PR:Font      LIKE(PR:Font)
PR:ControlType LIKE(PR:ControlType)
PR:Hidden    LIKE(PR:Hidden)
PR:IconFile  LIKE(PR:IconFile)
PR:ApplyTo   LIKE(PR:ApplyTo)
ViewPosition STRING(1024)
          END

BRW1     CLASS(BrowseClass)       !declare BRW1--a BrowseClass object

```

```

Q      &PropQ      ! that drives the EditClass objects
      END

Edit:PR:Font CLASS(EditFontClass) !declare Edit:PR:Font-EIP font dialog
Init      PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
TakeEvent PROCEDURE(UNSIGNED Event),BYTE,VIRTUAL
TypeFace  CSTRING(30)           !declare font typeface property
FontSize  LONG                  !declare font size property
FontStyle LONG                  !declare font style property
FontColor LONG                  !declare font color property
      END

PropWindow WINDOW('Browse Properties'),AT(,,318,137),IMM,SYSTEM,GRAY
      LIST,AT(8,4,303,113),USE(?PropList),IMM,HVSCROLL,FROM(PropQ),|
      FORMAT( '50L(2)|_M~Field Name~@s30@[70L(2)|_M~Color~@s20@' &|
      '60L(2)|_M~Font~@s40@60L(2)|_M~Control Type~@s12@' &|
      '20L(2)|_M~Hide~L(0)@s1@/130L(2)|_M~Icon File~@s30@' &|
      '120L(2)|_M~Apply To~L(0)@s25@]|M')
      BUTTON('&Insert'),AT(169,121),USE(?Insert)
      BUTTON('&Change'),AT(218,121),USE(?Change),DEFAULT
      BUTTON('&Delete'),AT(267,121),USE(?Delete)
      END

GlobalErrors ErrorClass
Access:Property CLASS(FileManager)
Init      PROCEDURE
      END

Relate:Property CLASS(RelationManager)
Init      PROCEDURE
Kill      PROCEDURE,VIRTUAL
      END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest LONG(0),THREAD

ThisWindow CLASS(WindowManager)
Init      PROCEDURE(),BYTE,PROC,VIRTUAL
Kill      PROCEDURE(),BYTE,PROC,VIRTUAL
      END

CODE
GlobalErrors.Init
Relate:Property.Init
GlobalResponse = ThisWindow.Run()
Relate:Property.Kill
GlobalErrors.Kill

```

```

ThisWindow.Init PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
  SELF.Request = GlobalRequest
  ReturnValue = PARENT.Init()
  SELF.FirstField = ?PropList
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
  Relate:Property.Open
  BRW1.Init(?PropList,PropQ.ViewPosition,PropView,PropQ,Relate:Property,SELF)
  OPEN(PropWindow)
  SELF.Opened=True
  ?PropList{PROP:LineHeight}=12          !enlarge rows to accomodate EIP icons
  BRW1.Q &= PropQ
  BRW1.AddSortOrder(,PR:NameKey)
  BRW1.AddField(PR:FieldName,BRW1.Q.PR:FieldName)
  BRW1.AddField(PR:Color,BRW1.Q.PR:Color)
  BRW1.AddField(PR:Font,BRW1.Q.PR:Font)
  BRW1.AddField(PR:ControlType,BRW1.Q.PR:ControlType)
  BRW1.AddField(PR:Hidden,BRW1.Q.PR:Hidden)
  BRW1.AddField(PR:IconFile,BRW1.Q.PR:IconFile)
  BRW1.AddField(PR:ApplyTo,BRW1.Q.PR:ApplyTo)
  BRW1.AddEditControl(Edit:PR:Font,3)  !Use Edit:PR:Font to edit BRW1 col 3
  BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
  BRW1.InsertControl=?Insert
  BRW1.ChangeControl=?Change
  BRW1.DeleteControl=?Delete
  SELF.SetAlerts()
  RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
  ReturnValue = PARENT.Kill()
  Relate:Property.Close
  RETURN ReturnValue

Edit:PR:Font.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,?* UseVar)
Comma        BYTE(1)
SaveFont     CSTRING(100)          !indexable hold area for font spec
i            USHORT
CODE
  PARENT.Init(FieldNumber,ListBox,UseVar)
  SaveFont=SELF.UseVar             !comma separated font attributes
  IF SaveFont                      ! e.g. Arial,14,255,400
    LOOP WHILE Comma               !parse/separate the font attributes

```

```

Comma = INSTRING(',',SaveFont,1,1)
i+=1
IF Comma
  EXECUTE i
  SELF.TypeFace = SaveFont[1 : Comma-1] !get Typeface
  SELF.FontSize = SaveFont[1 : Comma-1] !get FontSize
  BEGIN
    SELF.FontColor = SaveFont[1 : Comma-1] !get FontColor & Style
    SELF.FontStyle = SaveFont[Comma+1 : LEN(SaveFont)]
  END
  END
  SaveFont=SaveFont[Comma+1 : LEN(SaveFont)]
END
END
END

Edit:PR:Font.TakeEvent PROCEDURE(UNSIGNED Event)
ReturnValue          BYTE,AUTO
CODE
CASE Event
OF EVENT:DroppingDown !call Font dialog & store result
                        ! in comma separated string
IF FONTDIALOG(SELF.Title,SELF.TypeFace,SELF.FontSize,SELF.FontColor,SELF.FontStyle)
  SELF.UseVar = SELF.TypeFace&','&SELF.FontSize&','&SELF.FontColor&','&SELF.FontStyle
  DISPLAY(SELF.Feq)
END
RETURN EditAction:Ignore          !no I/O action on DroppingDown
ELSE                               !otherwise, default I/O action:
  RETURN PARENT.TakeEvent(Event) ! save, cancel, next field, etc.
END

Access:Property.Init PROCEDURE
CODE
PARENT.Init(Property,GlobalErrors)
SELF.FileNameValue = 'Property'
SELF.Buffer &= PR:Record
SELF.Create = 1
SELF.AddKey(PR:NameKey,'PR:NameKey',0)

Relate:Property.Init PROCEDURE
CODE
Access:Property.Init
PARENT.Init(Access:Property,1)

Relate:Property.Kill PROCEDURE
CODE
Access:Property.Kill
PARENT.Kill

```

## EditFontClass Properties

### EditFontClass Properties

The EditFontClass inherits all the properties of the EditClass from which it is derived. See *EditClass Properties* and *EditClass Concepts* for more information.

In addition to the inherited properties, the EditFontClass contains the following properties:

### Title (font dialog title text)

**Title**    **CSTRING(256)**

The **Title** property contains a string that sets the title bar text in the Windows font dialog.

Implementation:    The EditFontClass (TakeEvent method) uses the Title property as the *title* parameter to the FONTDIALOG procedure. See *FONTDIALOG* in the *Language Reference* for more information.

See Also:            TakeEvent

## EditFontClass Methods

### EditFontClass Methods

The EditFontClass inherits all the methods of the EditClass from which it is derived. See *EditClass Methods* and *EditClass Concepts*.

### EditFontClass Functional Organization--Expected Use

As an aid to understanding the EditFontClass it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditFontClass methods.

#### Non-Virtual Methods

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init <sub>v</sub>	initialize the EditFontClass object
Kill <sub>v</sub>	shut down the EditFontClass object

##### **Mainstream Use:**

TakeEvent <sub>v</sub>	handle events for the edit control
------------------------	------------------------------------

##### **Occasional Use:**

CreateContol <sub>v</sub>	create the edit (COMBO) control
SetAlerts <sub>v</sub>	alert keystrokes for the edit control

<sub>v</sub> These methods are also virtual.

<sub>i</sub> These methods are inherited from the EditClass

#### Virtual Methods

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init <sub>i</sub>	initialize the EditFontClass object
CreateContol <sub>i</sub>	create the edit (COMBO) control
SetAlerts <sub>i</sub>	alert keystrokes for the edit control
TakeEvent <sub>i</sub>	handle events for the edit control
Kill <sub>i</sub>	shut down the EditFontClass object

## CreateControl (create the edit-in-place control:EditFontClass)

### CreateControl, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place COMBO control and sets the FEQ property.

Implementation: The Init method calls the CreateControl method. The CreateControl method creates a COMBO control with an ellipsis button and sets the value of the FEQ property.

Use the Init method or the CreateControl method to set any required properties of the COMBO control.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts
```

See Also: FEQ, EditClass.CreateControl

**TakeEvent (process edit-in-place events:EditFontClass)****TakeEvent**( *event* ), VIRTUAL**TakeEvent** Processes an event for the EditFontClass object.*event* An integer constant, variable, EQUATE, or expression that contains the event number (see EVENT in the *Language Reference*).

The **TakeEvent** method processes an event for the EditFontClass object and returns a value indicating the user requested action. Valid actions are none, complete or OK, cancel, next record, previous record, next field, and previous field.

Implementation: The BrowseClass.AskRecord method calls the TakeEvent method. The TakeEvent method processes an EVENT:AlertKey for the edit-in-place control. On EVENT:DroppingDown, TakeEvent invokes the Windows font dialog and stores the font specification in the edited field specified by the Init method. Finally, TakeEvent returns a value indicating the user requested action. The BrowseClass.AskRecord method carries out the user requested action.

Corresponding EQUATEs for the possible edit-in-place actions are declared in ABEIP.INC as follows:

```

EditAction      ITEMIZE(0),PRE
None            EQUATE          ! no action
Forward        EQUATE          ! next field
Backward       EQUATE          ! previous field
Complete       EQUATE          ! OK
Cancel         EQUATE          ! cancel
Next           EQUATE          ! next record
Previous       EQUATE          ! previous record
Ignore        EQUATE          ! no action
END

```

Return Data Type: BYTE

Example:

```

EditClassAction ROUTINE
CASE SELF.EditList.Control.TakeEvent(EVENT())
OF EditAction:Forward !handle tab forward (new field, same record)
OF EditAction:Backward !handle tab backward (new field, same record)
OF EditAction:Next !handle down arrow (new record, offer to save prior record)
OF EditAction:Previous !handle up arrow (new record, offer to save prior record)
OF EditAction:Complete !handle OK or enter key (save record)
OF EditAction:Cancel !handle Cancel or esc key (restore record)
END

```

See Also: Init, BrowseClass.AskRecord

# EditMultiSelectClass

## EditMultiSelectClass Overview

The EditMultiSelectClass is an EditClass that supports a MultiSelect dialog by way of a dynamic edit-in-place COMBO control.

## EditMultiSelectClass Concepts

The EditMultiSelectClass creates a COMBO control with an ellipsis button that invokes the MultiSelect dialog. The MultiSelect dialog is an interface for selecting and ordering items from a list.

The EditMultiSelectClass provides an AddValue method so you can prime the dialog's Available Items and Selected Items lists.

The EditMultiSelectClass accepts input (selection actions) from the end user, then signals the calling procedure when selection actions occur. The EditMultiSelectClass provides a virtual TakeAction method to let you take control of the end user input.

The EditMultiSelectClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, canceling the edit, or completing the edit (moving to a new record or row). The EditMultiSelectClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## EditMultiSelectClass Relationship to Other Application Builder Classes

### EditClass

---

The EditMultiSelectClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseClass

---

The EditClass is loosely integrated into the BrowseClass. The BrowseClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

## EditMultiSelectClass ABC Template Implementation

You can use the BrowseUpdateButtons control template (**Configure EditInPlace**) to generate the code to instantiate an EditMultiSelectClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditMultiSelectClass object's methods as needed. See *Control Templates--BrowseUpdateButtons* for more information.

## EditMultiSelectClass Source Files

The EditMultiSelectClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditMultiSelectClass source code and their respective components are contained in:

ABEIP.INC	EditMultiSelectClass declarations
ABEIP.CLW	EditMultiSelectClass method definitions

## EditMultiSelectClass Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditMultiSelectClass object and a related BrowseClass object. The example page-loads a LIST of fieldnames and associated control attributes (such as color, font, when-to-apply, etc.), then edits the "when-to-apply" items with an EditMultiSelectClass object. Note that the BrowseClass object calls the "registered" EditMultiSelectClass object's methods as needed.

```

PROGRAM

  _ABCDllMode_  EQUATE(0)
  _ABCLinkMode_ EQUATE(1)

  INCLUDE( 'ABWINDOW.INC' )
  INCLUDE( 'ABBROWSE.INC' )
  INCLUDE( 'ABEIP.INC' )

  MAP
  END

Property  FILE,DRIVER( 'TOPSPEED' ),PRE(PR),CREATE,BINDABLE,THREAD
NameKey    KEY(PR:FieldName),NOCASE,OPT
Record     RECORD,PRE()
FieldName  STRING(30)
Color      STRING(20)
Hidden     STRING(1)
IconFile   STRING(30)
Font       STRING(40)
ControlType STRING(12)
ApplyTo    CSTRING(500)
           END
           END

PropView   VIEW(Property)
           END

PropQ      QUEUE
PR:FieldName  LIKE(PR:FieldName)
PR:Color      LIKE(PR:Color)
PR:Font       LIKE(PR:Font)
PR:ControlType LIKE(PR:ControlType)
PR:Hidden     LIKE(PR:Hidden)
PR:IconFile   LIKE(PR:IconFile)
PR:ApplyTo    LIKE(PR:ApplyTo)
ViewPosition  STRING(1024)
           END

```

```

BRW1    CLASS(BrowseClass)
Q       &PropQ
        END
!declare Edit:PR:ApplyTo-EIP multi dialog
Edit:PR:ApplyTo CLASS(EditMultiSelectClass)
Init    PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),VIRTUAL
TakeAction PROCEDURE(BYTE Action,<STRING Item>,LONG Pos1=0,LONG Pos2=0),VIRTUAL
        END

PropWindow WINDOW('Browse Properties'),AT(,,318,137),IMM,SYSTEM,GRAY
        LIST,AT(8,4,303,113),USE(?PropList),IMM,HVSCROLL,FROM(PropQ),|
        FORMAT( '50L(2)|_M~Field Name~@s30@[70L(2)|_M~Color~@s20@' &|
        '60L(2)|_M~Font~@s40@60L(2)|_M~Control Type~@s12@' &|
        '20L(2)|_M~Hide~L(0)@s1@/130L(2)|_M~Icon File~@s30@' &|
        '120L(2)|_M~Apply To~L(0)@s25@]|M')
        BUTTON('&Insert'),AT(169,121),USE(?Insert)
        BUTTON('&Change'),AT(218,121),USE(?Change),DEFAULT
        BUTTON('&Delete'),AT(267,121),USE(?Delete)
        END

GlobalErrors ErrorClass
Access:Property CLASS(FileManager)
Init    PROCEDURE
        END

Relate:Property CLASS(RelationManager)
Init    PROCEDURE
Kill    PROCEDURE,VIRTUAL
        END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest LONG(0),THREAD

ThisWindow CLASS(WindowManager)
Init    PROCEDURE(),BYTE,PROC,VIRTUAL
Kill    PROCEDURE(),BYTE,PROC,VIRTUAL
        END

CODE
GlobalErrors.Init
Relate:Property.Init
GlobalResponse = ThisWindow.Run()
Relate:Property.Kill
GlobalErrors.Kill

```

```
ThisWindow.Init PROCEDURE()
ReturnValue BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
SELF.FirstField = ?PropList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:Property.Open
BRW1.Init(?PropList,PropQ.ViewPosition,PropView,PropQ,Relate:Property,SELF)
OPEN(PropWindow)
SELF.Opened=True
?PropList{PROP:LineHeight}=12 !enlarge rows to accomodate EIP icons
BRW1.Q &= PropQ
BRW1.AddSortOrder(,PR:NameKey)
BRW1.AddField(PR:FieldName,BRW1.Q.PR:FieldName)
BRW1.AddField(PR:Color,BRW1.Q.PR:Color)
BRW1.AddField(PR:Font,BRW1.Q.PR:Font)
BRW1.AddField(PR:ControlType,BRW1.Q.PR:ControlType)
BRW1.AddField(PR:Hidden,BRW1.Q.PR:Hidden)
BRW1.AddField(PR:IconFile,BRW1.Q.PR:IconFile)
BRW1.AddField(PR:ApplyTo,BRW1.Q.PR:ApplyTo)
BRW1.AddEditControl(Edit:PR:ApplyTo,7) !use Edit:PR:ApplyTo to edit BRW1 col 7
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change
BRW1.DeleteControl=?Delete
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Property.Close
RETURN ReturnValue

Edit:PR:ApplyTo.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Reset
SELF.AddValue('Browse',INSTRING('Browse',SELF.UseVar,1,1))!set multi-select choice
SELF.AddValue('Form',INSTRING('Form',SELF.UseVar,1,1)) !set multi-select choice
SELF.AddValue('Report',INSTRING('Report',SELF.UseVar,1,1))!set multi-select choice
SELF.AddValue('Window',INSTRING('Window',SELF.UseVar,1,1))!set multi-select choice
```

```

Edit:PR:ApplyTo.TakeAction PROCEDURE(BYTE Action,<STRING Item>,LONG Pos1=0,LONG Pos2=0)
HoldIt CSTRING(1024)          !indexable string of end user choices
Pos     USHORT                !index to parse end user selections
Comma   USHORT                !index to parse end user selections
ItemQ   QUEUE                 !Q to reorder end user selections
Item    CSTRING(100)
Ord     BYTE
      END

CODE
PARENT.TakeAction(Action,Item,Pos1,Pos2)
HoldIt=SELF.UseVar
CASE Action
OF MSAction:Add                !end user selected an Item
  IF HoldIt
    HoldIt=HoldIt&','&Item
  ELSE
    HoldIt=Item
  END
OF MSAction:Delete            !end user deselected an Item
  Pos=INSTRING(Item,HoldIt,1,1)
  CASE Pos
  OF 0
    MESSAGE(Item&' not found!')
  OF 1                          !first item
    HoldIt=HoldIt[Pos+LEN(Item)+1 : LEN(HoldIt)] !deselect first item
  ELSE
    IF Pos+LEN(Item) > LEN(HoldIt)                !last item
      HoldIt=HoldIt[1 : Pos-2]                    !deselect last item
    ELSE                                           !deselect any other item
      HoldIt=HoldIt[1 : Pos-1] & HoldIt[Pos+LEN(Item)+1 : LEN(HoldIt)]
    END
  END
OF MSAction:Move              !Selected Item moved up or down
  FREE(ItemQ)                 ! Pos1=Item's "old" position
  CLEAR(ItemQ)                ! Pos2=Item's "new" position
  Comma=1
  LOOP WHILE Comma            !build Q of Selected Items
    Comma = INSTRING(', ',HoldIt,1,1)           ! to use for repositioning
    ItemQ.Ord+=1
    IF Comma
      ItemQ.Item = HoldIt[1 : Comma-1]
      ADD(ItemQ,ItemQ.Ord)
    HoldIt=HoldIt[Comma+1 : LEN(HoldIt)]        !comma separated list of user choices
  ELSE
    ItemQ.Item = HoldIt
    ADD(ItemQ,ItemQ.Ord)
  END
END
END

```

```

ItemQ.Ord=Pos2
GET(ItemQ, ItemQ.Ord)           !get the "bumped" item
ItemQ.Ord=Pos1
PUT(ItemQ)                       !reposition the "bumped" item
ItemQ.Item=Item
GET(ItemQ, ItemQ.Item)         !get the selected item
ItemQ.Ord=Pos2
PUT(ItemQ)                       !reposition the selected item
SORT(ItemQ,ItemQ.Ord)         !reorder Q of selected items
HoldIt=''
LOOP Pos = 1 TO RECORDS(ItemQ)  !refill comma separated list
  GET(ItemQ,Pos)
  IF HoldIt
    HoldIt=HoldIt&','&ItemQ.Item
  ELSE
    HoldIt=ItemQ.Item
  END
END
OF MSAction:StartProcess       !begin AddAll (>>) or DeleteAll (<<)
  SETCURSOR(CURSORS:Wait)
OF MSAction:EndProcess        !end AddAll (>>) or DeleteAll (<<)
  SETCURSOR()
END
SELF.UseVar=HoldIt
Access:Property.Init PROCEDURE
CODE
PARENT.Init(Property,GlobalErrors)
SELF.FileNameValue = 'Property'
SELF.Buffer &= PR:Record
SELF.Create = 1
SELF.AddKey(PR:NameKey, 'PR:NameKey', 0)

Relate:Property.Init PROCEDURE
CODE
Access:Property.Init
PARENT.Init(Access:Property,1)
Relate:Property.Kill PROCEDURE
CODE
Access:Property.Kill
PARENT.Kill

```

## EditMultiSelectClass Properties

### EditMultiSelectClass Properties

The EditMultiSelectClass inherits all the properties of the EditClass from which it is derived. See *EditClass Properties* and *EditClass Concepts* for more information.

In addition to the inherited properties, the EditMultiSelectClass contains the following properties:

### Title (font dialog title text:EditMultiSelectClass)

**Title** CSTRING(256)

The **Title** property contains a string that sets the title bar text in the MultiSelect dialog.

## EditMultiSelectClass Methods

### EditMultiSelectClass Methods

The EditMultiSelectClass inherits all the methods of the EditClass from which it is derived. See *EditClass Methods* and *EditClass Concepts*.

### EditMultiSelectClass Functional Organization--Expected Use

As an aid to understanding the EditMultiSelectClass it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditMultiSelectClass methods.

#### Non-Virtual Methods

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init <sub>v</sub>	initialize the EditMultiSelectClass object
AddValue	prime the MultiSelect dialog
Kill <sub>v</sub>	shut down the EditMultiSelectClass object

##### **Mainstream Use:**

TakeAction <sub>v</sub>	handle user actions for the dialog
TakeEvent <sub>v</sub>	handle events for the edit control

##### **Occasional Use:**

CreateContol <sub>v</sub>	create the edit (COMBO) control
Reset	clear the MultiSelect dialog
SetAlerts <sub>v<sub>i</sub></sub>	alert keystrokes for the edit control

<sub>v</sub> These methods are also virtual.

<sub>i</sub> These methods are inherited from the EditClass

---

## Virtual Methods

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init	initialize the EditMultiSelectClass object
CreateControl	create the edit (COMBO) control
SetAlerts	alert keystrokes for the edit control
TakeAction	handle user actions for the dialog
TakeEvent	handle events for the edit control
Kill	shut down the EditMultiSelectClass object

## AddValue (prime the MultiSelect dialog)

**AddValue**( *item* [ ,*selected* ] )

<b>AddValue</b>	Primes the Available and Selected items lists in the MultiSelect dialog.
<i>item</i>	A string constant, variable, EQUATE, or expression that contains the value to add to the item list.
<i>selected</i>	An integer constant, variable, EQUATE, or expression that indicates which list to update. A value of zero (0 or False) adds the <i>item</i> to the Available Items list; a value of one (1 or True) adds the <i>item</i> to the Selected Items list. If omitted, <i>selected</i> defaults to zero and AddValue adds the <i>item</i> to the Available Items list.

The **AddValue** method primes the Available and Selected items lists in the MultiSelect dialog.

Example:

```
Edit:PR:ApplyTo.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Reset
SELF.AddValue('Browse',INSTRING('Browse',SELF.UseVar,1,1)) !set multi-select choice
SELF.AddValue('Form',INSTRING('Form',SELF.UseVar,1,1)) !set multi-select choice
SELF.AddValue('Report',INSTRING('Report',SELF.UseVar,1,1)) !set multi-select choice
SELF.AddValue('Window',INSTRING('Window',SELF.UseVar,1,1)) !set multi-select choice
```

**CreateControl (create the edit-in-place control:EditMultiSelectClass)****CreateControl, VIRTUAL, PROTECTED**

The **CreateControl** method creates the edit-in-place COMBO control and sets the FEQ property.

Implementation: The Init method calls the CreateControl method. The CreateControl method creates a read only COMBO control with an ellipsis button and sets the value of the FEQ property.

Use the Init method or the CreateControl method to set any required properties of the COMBO control.

Example:

```

EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts

```

See Also: FEQ, EditClass.CreateControl

**Reset (reset the EditMultiSelectClass object)****Reset**

The **Reset** method resets the EditMultiSelectClass object.

Implementation: The Reset method clears the Available and Selected items lists in the MultiSelect dialog. Use the AddValue method to refill these lists.

Example:

```

Edit:PR:ApplyTo.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
CODE
PARENT.Init(FieldNumber,ListBox,UseVar)
SELF.Reset
SELF.AddValue('Browse',INSTRING('Browse',SELF.UseVar,1,1)) !set multi-select choice
SELF.AddValue('Form',INSTRING('Form',SELF.UseVar,1,1)) !set multi-select choice
SELF.AddValue('Report',INSTRING('Report',SELF.UseVar,1,1)) !set multi-select choice
SELF.AddValue('Window',INSTRING('Window',SELF.UseVar,1,1)) !set multi-select choice

```

See Also: AddValue

**TakeAction (process MultiSelect dialog action)**

**TakeAction( *action* [, *item* ] [ ,*oldposition* ] [ ,*newposition* ] ), VIRTUAL**

<b>TakeAction</b>	Processes a MultiSelect dialog action.
<i>action</i>	An integer constant, variable, EQUATE, or expression that contains the action to process. Valid actions are add (select), delete (deselect), move, begin process, and end process.
<i>item</i>	A string constant, variable, EQUATE, or expression that contains the value of the list item affected by the <i>action</i> . If omitted, the <i>action</i> affects no <i>item</i> . For example a begin process action is not associated with a list item.
<i>oldposition</i>	An integer constant, variable, EQUATE, or expression that contains the ordinal position of the <i>item</i> (in the Selected Items list) prior to the move <i>action</i> . If omitted, <i>oldposition</i> defaults to zero (0), indicating a non-move <i>action</i> .
<i>newposition</i>	An integer constant, variable, EQUATE, or expression that contains the ordinal position of the <i>item</i> (in the Selected Items list) after the move <i>action</i> . If omitted, <i>newposition</i> defaults to zero (0), indicating a non-move <i>action</i> .

The **TakeAction** method processes a MultiSelect dialog action for the EditMultiSelectClass object. The TakeAction method is your opportunity to interpret and implement the meaning of the end user's selection.

**Tip:** The TakeAction processing is immediate and occurs while the MultiSelect dialog is open. The MultiSelect dialog does not generate an action or an event when the dialog closes.

Implementation: The TakeEvent method (indirectly) calls the TakeAction method each time the end user makes a new selection or moves a selection in the MultiSelect dialog.

Corresponding EQUATEs for the MultiSelect dialog *action* are declared in ABEIP.INC as follows:

```

MSAction      ITEMIZE(1),PRE
Add           EQUATE      !add / select
Delete       EQUATE      !delete / deselect
Move         EQUATE      !reposition a selected item
StartProcess EQUATE      !begin an add/delete series
EndProcess   EQUATE      !end an add/delete series
END

```

Example:

```

!This implementation of TakeAction converts the end user selections into
!comma separated items in a string.
Edit:PR:ApplyTo.TakeAction PROCEDURE(BYTE Action,<STRING Item>,LONG Pos1=0,LONG Pos2=0)
HoldIt CSTRING(1024)      !indexable string of end user choices
Pos    USHORT            !index to parse end user selections
Comma  USHORT            !index to parse end user selections
ItemQ  QUEUE             !Q to reorder end user selections
Item   CSTRING(100)
Ord    BYTE
END
CODE
PARENT.TakeAction(Action,Item,Pos1,Pos2)
HoldIt=SELF.UseVar
CASE Action
OF MSAction:Add           !end user selected an Item
  HoldIt=CHOOSE(HoldIt,HoldIt&','&Item,Item)
OF MSAction>Delete       !end user deselected an Item
  Pos=INSTRING(Item,HoldIt,1,1)
  IF Pos=1                !first item
    HoldIt=HoldIt[Pos+LEN(Item)+1 : LEN(HoldIt)] !deselect first item
  ELSE
    IF Pos+LEN(Item) > LEN(HoldIt)           !last item
      HoldIt=HoldIt[1 : Pos-2]               !deselect last item
    ELSE
      HoldIt=HoldIt[1 : Pos-1] & HoldIt[Pos+LEN(Item)+1 : LEN(HoldIt)]
    END
  END
END
OF MSAction:Move         !Selected Item moved up or down
  FREE(ItemQ)            ! Pos1=Item's "old" position
  CLEAR(ItemQ)           ! Pos2=Item's "new" position
  Comma=1
  LOOP WHILE Comma      !build Q of Selected Items
    Comma = INSTRING(',',HoldIt,1,1) ! to use for repositioning
    ItemQ.Ord+=1
  IF Comma
    ItemQ.Item = HoldIt[1 : Comma-1]

```

```
        ADD(ItemQ,ItemQ.Ord)
HoldIt=HoldIt[Comma+1 : LEN(HoldIt)] !comma separated list of user choices
    ELSE
        ItemQ.Item = HoldIt
        ADD(ItemQ,ItemQ.Ord)
    END
END
ItemQ.Ord=Pos2
GET(ItemQ, ItemQ.Ord)                !get the "bumped" item
ItemQ.Ord=Pos1
PUT(ItemQ)                            !reposition the "bumped" item
ItemQ.Item=Item
GET(ItemQ, ItemQ.Item)                !get the selected item
ItemQ.Ord=Pos2
PUT(ItemQ)                            !reposition the selected item
SORT(ItemQ,ItemQ.Ord)                !reorder Q of selected items
HoldIt=''

    LOOP Pos = 1 TO RECORDS(ItemQ)      !refill comma separated list
        GET(ItemQ,Pos)
        HoldIt=CHOOSE(HoldIt,HoldIt&','&ItemQ.Item,ItemQ.Item)
    END
OF MSAction:StartProcess                !begin AddAll (>>) or DeleteAll (<<)
    SETCURSOR(CURSOR:Wait)
OF MSAction:EndProcess                  !end AddAll (>>) or DeleteAll (<<)
    SETCURSOR()
END
SELF.UseVar=HoldIt
```

See Also: TakeEvent

**TakeEvent (process edit-in-place events:EditMultiSelectClass)****TakeEvent**( *event* ), VIRTUAL**TakeEvent** Processes an event for the EditMultiSelectClass object.*event* An integer constant, variable, EQUATE, or expression that contains the event number (see EVENT in the *Language Reference*).

The **TakeEvent** method processes an event for the EditMultiSelectClass object and returns a value indicating the user requested action. Valid actions are none, complete or OK, cancel, next record, previous record, next field, and previous field.

Implementation: The BrowseClass.AskRecord method calls the TakeEvent method. The TakeEvent method processes an EVENT:AlertKey for the edit-in-place control. On EVENT:DroppingDown, TakeEvent invokes the MultiSelect dialog. Finally, TakeEvent returns a value indicating the user requested action. The BrowseClass.AskRecord method carries out the user requested action.

Corresponding EQUATEs for the possible edit-in-place actions are declared in ABEIP.INC as follows:

```

EditAction      ITEMIZE(0),PRE
None            EQUATE          ! no action
Forward        EQUATE          ! next field
Backward       EQUATE          ! previous field
Complete       EQUATE          ! OK
Cancel         EQUATE          ! cancel
Next           EQUATE          ! next record
Previous       EQUATE          ! previous record
Ignore        EQUATE          ! no action
END

```

Return Data Type: BYTE

Example:

```

EditClassAction ROUTINE
CASE SELF.EditList.Control.TakeEvent(EVENT())
OF EditAction:Forward !handle tab forward (new field, same record)
OF EditAction:Backward !handle tab backward (new field, same record)
OF EditAction:Next !handle down arrow (new record, offer to save prior record)
OF EditAction:Previous !handle up arrow (new record, offer to save prior record)
OF EditAction:Complete !handle OK or enter key (save record)
OF EditAction:Cancel !handle Cancel or esc key (restore record)
END

```

See Also: Init, BrowseClass.AskRecord

---

# EditTextClass

## EditTextClass: Overview

The EditTextClass is an EditClass that supports memo and large string fields by way of an edit-in-place COMBO control.

## EditTextClass Concepts

The EditTextClass creates a COMBO control with an ellipsis button that invokes a text dialog.

The EditTextClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditTextClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

## EditTextClass: Relationship to Other Application Builder Classes

### EditClass

The EditTextClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

### BrowseEIPManagerClass

The EditClass is managed by the BrowseEIPManagerClass. The BrowseEIPManagerClass depends on the EditClass operating according to its documented specifications; however, the EditClass may be called by non-BrowseClass procedures and objects.

## ABC Template Implementation

You can use the BrowseUpdateButtons control template (**Configure EditInPlace**) to generate the code to instantiate an EditTextClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditTextClass object's methods as needed. See *Control Templates—BrowseUpdateButtons* for more information.

## EditTextClass Source Files

The EditTextClass source code is installed by default to the Clarion \LIBSRC folder. The specific EditTextClass source code and their respective components are contained in:

ABEIP.INC	EditTextClass declarations
ABEIP.CLW	EditTextClass method definitions

## EditTextClass Properties

The EditTextClass inherits all the properties of the EditClass from which it is derived. See EditClass Properties and EditClass Concepts for more information.

In addition to the inherited properties, the EditTextClass contains the following properties:

### **Title (text dialog title text)**

**Title**    **CSTRING(256)**

The **Title** property contains a string that sets the title bar text in the dialog containing the text control.

Implementation:    The EditTextClass (TakeEvent method) uses the Title property as the title text for the titlebar of the dialog containing the text control.

See Also:            TakeEvent

## EditTextClass Methods

The EditTextClass inherits all the methods of the EditClass from which it is derived. See EditClass Methods *and* EditClass Concepts.

### EditTextClass: Functional Organization—Expected Use

As an aid to understanding the EditTextClass it is useful to organize its methods into two large categories according to their expected use—the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EditTextClass methods.

#### Non-Virtual Methods

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### Housekeeping (one-time) Use:

Init <sup>v</sup>	initialize the EditTextClass object
Kill <sup>v</sup>	shut down the EditTextClass object

##### Mainstream Use:

TakeEvent <sup>v</sup>	handle events for the edit control
------------------------	------------------------------------

##### Occasional Use:

CreateContol <sup>v</sup>	create the edit (COMBO) control
SetAlerts <sup>v</sup>	alert keystrokes for the edit control

<sup>v</sup> These methods are also virtual.

<sup>i</sup> These methods are inherited from the EditClass

#### Virtual Methods

Typically you will not call these methods directly—the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init <sup>i</sup>	initialize the EditTextClass object
CreateContol <sup>i</sup>	create the edit COMBO control
SetAlerts <sup>i</sup>	alert keystrokes for the edit control
TakeEvent <sup>i</sup>	handle events for the edit control
Kill <sup>i</sup>	shut down the EditTextClass object

## CreateControl (create the edit-in-place control:EditTextClass)

### CreateControl, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place COMBO control.

Implementation: The Init method calls the CreateControl method. The CreateControl method creates a COMBO control with an ellipsis button.

Use the Init method or the CreateControl method to set any required properties of the COMBO control.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts
```

See Also: FEQ, EditClass.CreateControl

**TakeEvent (process edit-in-place events:EditTextClass)****TakeEvent( event ), VIRTUAL****TakeEvent** Processes an event for the EditTextClass object.*event* An integer constant, variable, EQUATE, or expression that contains the event number (see EVENT in the *Language Reference*).

The **TakeEvent** method processes an event for the EditTextClass object and returns a value indicating the user requested action. Valid actions are none, complete or OK, cancel, next record, previous record, next field, and previous field.

Implementation: The EIPManager.TakeFieldEvent method calls the TakeEvent method. The TakeEvent method processes an EVENT:AlertKey for the edit-in-place control. On EVENT:DroppingDown, TakeEvent invokes a Window with a text control. Finally, TakeEvent returns a value indicating the user requested action.

Corresponding EQUATEs for the possible edit-in-place actions are declared in ABEIP.INC as follows:

```

EditAction ITEMIZE(0),PRE
None      EQUATE  ! no action
Forward   EQUATE  ! next field
Backward  EQUATE  ! previous field
Complete  EQUATE  ! OK
Cancel    EQUATE  ! cancel
Next      EQUATE  ! next record
Previous  EQUATE  ! previous record
Ignore    EQUATE  ! no action
          END

```

Return Data Type: BYTE

Example:

```

EditClassAction ROUTINE
CASE SELF.EditList.Control.TakeEvent(EVENT())
OF EditAction:Forward   !handle tab forward (new field, same record)
OF EditAction:Backward !handle tab backward (new field, same record)
OF EditAction:Next      !handle down arrow (new record, offer to save prior record)
OF EditAction:Previous  !handle up arrow (new record, offer to save prior record)
OF EditAction:Complete !handle OK or enter key (save record)
OF EditAction:Cancel    !handle Cancel or esc key (restore record)
END

```

See Also: Init

---

# EIPManagerClass

## EIPManagerClass--Overview

The EIPManagerClass is a WindowManager that displays an edit-in-place dialog, and handles events for that dialog . The EIPManagerClass is an abstract class--it is not useful by itself, but serves as the foundation and framework for the BrowseEIPManagerClass. See *BrowseEIPManagerClass*.

## EIPManagerClass Concepts

Each edit-in-place control is created on top of the browse from which it is called. The EIPManager dynamically creates an edit-in-place object and control upon request (Insert, Change, or Delete) by the end user. When the end user accepts the edited record the EIPManager destroys the edit-in-place object and control.

## EIPManagerClass--Relationship to Other Application Builder Classes

---

### WindowClass

The EIPManager class is derived from the WindowManager class.

---

### BrowseClass

Each BrowseClass utilizing edit-in-place requires an BrowseEIPManager to manage the events and processes that are used by each edit-in-place field in the browse.

The BrowseClass.AskRecord method is the calling method for edit-in-place functionality.

---

### EditClasses

The EIPManager provides the basic or "under the hood" interface between the Edit classes and the Browse class. The EIPManager uses the EditQueue to keep track of the fields in the browse utilizing edit-in-place.

## EIPManagerClass--ABC Template Implementation

The Browse template declares a BrowseEIPManager when the BrowseUpdateButtons control template enables default edit-in-place support for the BrowseBox.

See *Control Templates--BrowseBox* and *BrowseUpdateButtons* for more information.

## EIPManagerClass Source Files

The EIPManagerClass source code is installed by default to the Clarion \LIBSRC folder. The specific EIPManagerClass source code and their respective components are contained in:

ABEIP.INC	EditClass declarations
ABEIP.CLW	EditClass method definitions
ABEIP.TRN	EditClass translation strings

## EIPManagerClass--Conceptual Example

The following example shows a sequence of statements to declare, and instantiate an EIPManager object. The example page-loads a LIST of actions and associated priorities, then edits the list items via edit-in-place. Note that the BrowseClass object references the BrowseEIPManager which is an EIPManager object, as referenced in ABBrowse.INC.

```

PROGRAM

_ABCEllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)

INCLUDE('ABBROWSE.INC'),ONCE
INCLUDE('ABEIP.INC'),ONCE
INCLUDE('ABWINDOW.INC'),ONCE
MAP
END

Actions          FILE,DRIVER('TOPSPEED'),PRE(ACT),CREATE,BINDABLE,THREAD
KeyAction        KEY(ACT:Action),NOCASE,OPT
Record           RECORD,PRE()
Action           STRING(20)
Priority          DECIMAL(2)
Completed        DECIMAL(1)
                END
                END

Access:Actions   &FileManager
Relate:Actions   &RelationManager
GlobalErrors     ErrorClass
GlobalRequest    BYTE(0),THREAD
ActionsView      VIEW(Actions)
                END

Queue:Browse     QUEUE
ACT:Action       LIKE(ACT:Action)
ACT:Priority      LIKE(ACT:Priority)
ViewPosition     STRING(1024)
                END

BrowseWindow     WINDOW('Browse Records'),AT(0,0,247,140),SYSTEM,GRAY
                LIST,AT(5,5,235,100),USE(?List),IMM,HVSCROLL,MSG('Browsing Records'),|
                FORMAT('80L~Action~@S20@8R~Priority~L@N2@'),FROM(Queue:Browse)
                BUTTON('&Insert'),AT(5,110,40,12),USE(?Insert),KEY(InsertKey)
                BUTTON('&Change'),AT(50,110,40,12),USE(?Change),KEY(CtrlEnter),DEFAULT
                BUTTON('&Delete'),AT(95,110,40,12),USE(?Delete),KEY(DeleteKey)
                END

```

```

ThisWindow CLASS(WindowManager)
Init          PROCEDURE(),BYTE,PROC,DERIVED
Kill         PROCEDURE(),BYTE,PROC,DERIVED
            END

BRW1 CLASS(BrowseClass)
Q           &Queue:Browse
Init PROCEDURE(SIGNED ListBox,*STRING Posit,VIEW V,QUEUE Q,RelationManager RM,WindowManager WM)
            END

BRW1::EIPManager      BrowseEIPManager      ! EIPManager for Browse using ?List

CODE
GlobalErrors.Init
Relate:Actions.Init
GlobalResponse = ThisWindow.Run()
Relate:Actions.Kill
GlobalErrors.Kill

ThisWindow.Init PROCEDURE

ReturnValue          BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue =PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?List
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(Toolbar)
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
Relate:Actions.Open
FilesOpened = True
BRW1.Init(?List,Queue:Browse.ViewPosition,BRW1::View:Browse,Queue:Browse,Relate:Actions,SELF)
OPEN(BrowseWindow)
SELF.Opened=True
BRW1.Q &= Queue:Browse
BRW1.AddSortOrder(,ACT:KeyAction)
BRW1.AddLocator(BRW1::Sort0:Locator)
BRW1::Sort0:Locator.Init(,ACT:Action,1,BRW1)
BRW1.AddField(ACT:Action,BRW1.Q.ACT:Action)
BRW1.AddField(ACT:Priority,BRW1.Q.ACT:Priority)
BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change
BRW1.DeleteControl=?Delete

```

```
BRW1.AddToolBarTarget(ToolBar)
SELF.SetAlerts()
RETURN ReturnValue
```

```
ThisWindow.Kill PROCEDURE
```

```
ReturnValue          BYTE,AUTO
CODE
ReturnValue =PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
IF FilesOpened
    Relate:Actions.Close
END
RETURN ReturnValue
```

```
BRW1.Init |
```

```
PROCEDURE(SIGNED ListBox,*STRING Posit,VIEW V,QUEUE Q,RelationManagerRM,WindowManager WM)
```

```
CODE
PARENT.Init(ListBox,Posit,V,Q,RM,WM)
SELF.EIP &= BRW1::EIPManager    ! Browse object's reference to the BrowseEIPManager
```

## EIPManagerClass Properties

### Again (column usage flag)

**Again** BYTE, PROTECTED

The **Again** property contains a value that indicates whether or not the current edit-in-place column has been selected by the user during an edit-in-place process.

The TakeEvent method is where the Again property receives a value.

### Arrow (edit-in-place action on arrow key)

**Arrow** &BYTE

The **Arrow** property is a reference to a BYTE which indicates the action to take when the end user presses the up or down arrow key during an edit-in-place process.

**Note:** The **Arrow** property should be treated as a **PROTECTED** property except during initialization.

Implementation: When the EIPManager is instantiated from a browse the Arrow property will point to the BrowseClass.ArrowAction.

See Also: BrowseClass.ArrowAction

### Column (listbox column)

**Column** UNSIGNED

The **Column** property contains a value that indicates the column number of the listbox field which currently has focus in an edit-in-place process.

## Enter (edit-in-place action on enter key)

**Enter**    &BYTE

The **Enter** property is a reference to the BrowseClass.EnterAction property, and indicates the action to take when the end user presses the ENTER key during an edit-in-place process.

**Note:** The Enter property should be treated as a PROTECTED property except during initialization.

See Also:            BrowseClass.EnterAction

## EQ (list of edit-in-place controls)

**EQ**        &EditQueue

The **EQ** property is a reference to a structure containing a list of browse list columns that will not utilize the default edit-in-place control. This list includes columns that will not utilize edit-in-place.

Implementation:    The AddControl method adds browse list columns to the EQ property. An entry without an associated control indicates a column that has been specified as non-edit-in-place.

You do not need to initialize this property to implement the default edit-in-place controls. The EQ property supports custom edit-in-place controls.

The EQ property is a reference to a QUEUE declared in ABEdit.INC as follows:

```
    EditQueue        QUEUE,TYPE
    Field            UNSIGNED
    FreeUp          BYTE
    Control         &EditClass
                    END
```

**Note:** The EQ property should be treated as a PROTECTED property except during initialization.

See Also:            AddControl

## **Fields (managed fields)**

**Fields**   **&FieldPairsClass, PROTECTED**

The **Fields** property is a reference to the FieldPairsClass object that moves and compares data between the BrowseClass object's FILE and the EditClasses.

**Note:** The **Fields** property should be treated as a **PROTECTED** property except during initialization.

See Also:        BrowseClass.TabAction

## **FocusLoss ( action on loss of focus)**

**FocusLoss**        **&BYTE**

The **FocusLoss** property is a reference to the BrowseClass.FocusLossAction property, and indicates the action to take with regard to pending changes when the edit control loses focus during an edit-in-place process.

**Note:** The **FocusLoss** property should be treated as a **PROTECTED** property except during initialization.

See Also:        BrowseClass.TabAction, BrowseClass.FocusLossAction

## Insert (placement of new record)

**Insert**            **BYTE**

The **Insert** property indicates where in the list a new record will be added when the end user inserts a new record. The default placement is below the selected record.

Implementation:    There are three places a new record can be placed in a list when using edit-in-place: above the selected record; below the selected record (the default); or appended to the bottom of the list.

**Note:** This does not change the sort order. After insertion, the list is resorted and the new record appears in the proper position within the sort sequence.

The specified placements are implemented by the BrowseEIPManager.Init method. Set the record insertion point by assigning, adding, or subtracting the following EQUATEd values to Insert. The following EQUATEs are in ABEdit.INC:

```
ITEMIZE,PRE(EIPAction)
Default EQUATE(0)
Always EQUATE(1)
Never EQUATE(2)
Prompted EQUATE(4)
Save EQUATE(7)
Remain EQUATE(8)
Before EQUATE(9) ! insert before/above selected record
Append EQUATE(10) ! insert at the bottom of the list
RetainColumn EQUATE(16)
END
```

See Also:            BrowseEIPManager.Init

### **ListControl (listbox control number)**

ListControl      SIGNED

The **ListControl** property contains the control number of the LIST control that is utilizing edit-in-place.

**Note:** The **ListControl** property should be treated as a **PROTECTED** property except during initialization.

See Also:      BrowseClass.TabAction

### **LastColumn (previous edit-in-place column)**

LastColumn      BYTE, PROTECTED

The **LastColumn** property contains the column number of the previously used edit-in-place control to facilitate the appropriate processing of a NewSelection.

Implementation:      The LastColumn method is assigned the value of the Column property in the ResetColumn method.

## **Repost (event synchronization)**

Repost      UNSIGNED, PROTECTED

The **Repost** property indicates the appropriate event to post to the edit-in-place control based on events posted from the browse procedure window.

Implementation:    The TakeEvent and TakeFieldEvent methods assign the appropriate value to the Repost property. The Kill method posts the specified event to the appropriate edit-in-place control based on the value contained in the RepostField property.

See Also:          RepostField

## **RepostField (event synchronization field)**

RepostField      UNSIGNED, PROTECTED

The **RepostField** property contains the field control number of the listbox field that is being edited.

Implementation:    The TakeFieldEvent method assigns the appropriate value to the RepostField property. The Kill method posts the specified event to the appropriate edit-in-place control based on the value contained in the RepostField property.

See Also:          Repost

### **Req (database request)**

Req                    BYTE, PROTECTED

The **Req** property indicates the database action the procedure is handling. The EIPManager uses this property to make appropriate processing decisions with regard to priming records, saving or abandoning changes, etc.

Implementation:    The Run method is passed a parameter which contains the value assigned to the Req property.

See Also:            WindowManager.Request

### **SeekForward (get next field flag)**

SeekForward        BYTE, PROTECTED

The **SeekForward** property indicates that the end user has pressed the TAB key during an edit-in-place process.

Implementation:    The TakeAction method conditionally assigns a value of one (1) to the SeekForward property based on the actions of the end user.

See Also:            Next

### **Tab (action on a tab key)**

Tab                    &BYTE

The **Tab** property is a reference to the BrowseClass.TabAction property that indicates the action to take when the end user presses the TAB key during an edit-in-place process.

**Note:** The **Tab** property should be treated as a **PROTECTED** property except during initialization.

See Also:            BrowseClass.TabAction

## EIPManagerClass Methods

### EIPManagerClass--Functional Organization--Expected Use

As an aid to understanding the EIPManagerClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the EIPManagerClass methods.

#### **Non-Virtual Methods**

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Run	run this procedure
Init <sub>d</sub>	initialize the EditClass object
InitControls	initialize edit-in-place controls
Kill <sub>d</sub>	shut down the EditClass object

##### **Mainstream Use:**

TakeAcceptAll	handle all validation settings
TakeEvent <sub>d</sub>	handle events for the edit control
TakeNewSelection <sub>d</sub>	handle Event:NewSelection

##### **Occasional Use:**

AddControl	register edit-in-place controls
ClearColumn <sub>v</sub>	reset column property values
CreateControl <sub>v</sub>	a virtual to create the edit control
GetEdit <sub>v</sub>	identify edit-in-place field
Next	get the next edit-in-place field
ResetColumn <sub>v</sub>	reset edit-in-place object to selected field
SetAlerts <sub>v</sub>	alert appropriate keystrokes for the edit control
TakeAction <sub>v</sub>	process end user actions
TakeCompleted <sub>v</sub>	process completion of edit
TakeFocusLoss <sub>v</sub>	process loss of focus
TakeFieldEvent <sub>d</sub>	handle field specific events

<sub>d</sub> These methods are also derived.

<sub>v</sub> These methods are also virtual.

---

## Virtual and Derived Methods

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are either derived or virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init <sub>D</sub>	initialize the EditClass object
Kill <sub>D</sub>	shut down the EditClass object
TakeEvent <sub>D</sub>	handle events for the edit control
TakeNewSelection <sub>D</sub>	handle Event:NewSelection
ClearColumn <sub>V</sub>	reset column property values
CreateContol <sub>V</sub>	a virtual to create the edit control
GetEdit <sub>V</sub>	identify edit-in-place field
ResetColumn <sub>V</sub>	reset edit-in-place object to selected field
SetAlerts <sub>V</sub>	alert appropriate keystrokes for the edit control
TakeAction <sub>V</sub>	process end user actions
TakeCompleted <sub>V</sub>	process completion of edit
TakeFocusLoss <sub>V</sub>	process loss of focus
TakeFieldEvent <sub>D</sub>	handle field specific events

**AddControl (register edit-in-place controls)****AddControl**([*EditClass*], *Column*, *AutoFree*)

---

<b>AddControl</b>	Specifies an edit-in-place control.
<i>EditClass</i>	The label of the EditClass. If omitted, the specified <i>column</i> is not editable.
<i>Column</i>	An integer constant, variable, EQUATE, or expression that indicates the browse list column to edit with the specified <i>editclass</i> object.
<i>AutoFree</i>	A numeric constant, variable, EQUATE, or expression that indicates whether the BrowseClass.Kill method DISPOSEs of the <i>editclass</i> object. A zero (0) value leaves the object intact. A non-zero value DISPOSEs the object.

The **AddControl** method specifies the *editclass* that defines the edit-in-place control for the browse *column*. Use *autofree* with caution; you should only DISPOSE of memory allocated with a NEW statement. See the *Language Reference* for more information on NEW and DISPOSE.

The AddControl method also registers fields which will not be editable via edit-in-place. In this instance the EditClass parameter is omitted.

Implementation: The InitControls and BrowseClass.AddEditControl methods call the AddControl method. The BrowseClass.AddEditControl method defines the *editclass* for a column not utilizing the default *editclass*.

The AddControl method ADDs a record containing the values of *EditClass*, *Column*, and *AutoFree*, to the EditQueue which is declared in ABEdit.INC as follows:

```

EditQueue QUEUE,TYPE
Field      UNSIGNED
FreeUp     BYTE
Control    &EditClass
END

```

Example:

```

BrowseClass.AddEditControl PROCEDURE(EditClass EC,UNSIGNED Id,BYTE Free)
CODE
  SELF.CheckEIP
  SELF.EIP.AddControl(EC,Id,Free)

```

See Also: EQ, InitControls, BrowseClass.AddEditControl

## ClearColumn (reset column property values:EIPManagerClass)

### ClearColumn, VIRTUAL

The **ClearColumn** method checks for a value in the LastColumn property and conditionally sets the column values to zero (0).

The TakeAction and TakeNewSelection methods call the ClearColumn method.

Example:

```
EIPManager.TakeNewSelection PROCEDURE    ! Must be overridden to handle out-of-row click
CODE
  IF FIELD() = SELF.ListControl AND KEYCODE() = MouseLeft  ! An in-row mouse click
    SELF.ClearColumn
    SELF.Column = SELF.ListControl{PROPLIST:MouseUpField}
    SELF.ResetColumn
  END
RETURN Level:Benign
```

See Also: Column, TakeAction, TakeNewSelection

## GetEdit (identify edit-in-place field)

**GetEdit, VIRTUAL, PROTECTED**

The **GetEdit** method checks for a value in the Control field of the EditQueue.

Implementation: **GetEdit** is called by the Next method, and returns one (1) if any value is in the Control field of the EditQueue, otherwise it returns zero (0).

Return Data Type: **BYTE**

Example:

```
EIPManager.Next PROCEDURE
  CODE

  GET (SELF.EQ, RECORDS (SELF.EQ))
? ASSERT (~ERRORCODE())
  LastCol=SELF.EQ.Field

  LOOP
    CLEAR (SELF.EQ)
    SELF.EQ.Field = SELF.Column
    GET (SELF.EQ, SELF.EQ.Field)
    IF ~ERRORCODE() AND SELF.GetEdit()
      BREAK
    END
  !executable code
```

See Also: **EQ, Next**

**Init (initialize the EIPManagerClass object)****Init, DERIVED, PROC**

The **Init** method initializes the EIPManagerClass object.

Implementation: The BrowseEIPManager.Init method calls the Init method. The Init method primes variables and calls the InitControls method which then initializes the appropriate edit-in-place controls.

Return Data Type: BYTE

Example:

```
BrowseEIPManager.Init      ! initialize BrowseEIPManagerClass object
!program code
RETURN PARENT.Init()      ! call to the EIPManager.Init
```

See Also: BrowseEIPManager.Init, InitControls

**InitControls (initialize edit-in-place controls)****InitControls, VIRTUAL**

The **InitControls** method registers the default edit-in-place controls with the EIPManager by calling the AddControl method, and initializes each added control.

Implementation: The Init method calls the InitControls method. The InitControls method checks for custom edit-in-place controls in the EditQueue before adding a default edit-in-place control.

Example:

```
EIPManager.Init PROCEDURE
CODE
  IF SELF.Column = 0 THEN SELF.Column = 1.
  SELF.LastColumn = 0
  SELF.Repost = 0
  SELF.RepostField = 0
  ASSERT(~SELF.EQ &= NULL)
  SELF.EQ.Field = 1

  SELF.InitControls
  SELF.ResetColumn
  RETURN Level:Benign
```

See Also: Init, EQ, AddControl

## Kill (shut down the EIPManagerClass object)

### Kill, DERIVED, PROC

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code. The Kill method must leave the object in a state in which an Init can be called.

Implementation: The BrowseEIPManager.Kill method calls the Kill method with a PARENT call. The Kill method destroys the edit-in-place controls created by the InitControls method.

Return Data Type: BYTE

Example:

```
BrowseEIPManager.Kill PROCEDURE
CODE
SELF.BC.ResetFromAsk(SELF.Req,SELF.Response)
RETURN PARENT.Kill()
```

See Also: BrowseEIPManager.Kill

## Next (get the next edit-in-place field)

### Next, PROTECTED

The **Next** method gets the next edit-in-place control in the direction specified (forward or backward) by the end user.

Implementation: The Next method loops through the EditQueue and gets the next edit-in-place control based on the RETURN value of the GetEdit method.

Example:

```
EIPManager.ResetColumn PROCEDURE
```

```
CODE
```

```
SETKEYCODE(0)
```

```
SELF.Next
```

```
IF SELF.Column <> SELF.LastColumn
```

```
    SELF.ListControl{PROP:Edit,SELF.EQ.Field} = SELF.EQ.Control.Feq
```

```
    SELECT(SELF.EQ.Control.Feq)
```

```
    SELF.LastColumn = SELF.Column
```

```
END
```

See Also: GetEdit, SeekForward, Column, EQ

**ResetColumn (reset edit-in-place object to selected field)**

**ResetColumn, VIRTUAL, PROTECTED**

The **ResetColumn** method selects the appropriate edit-in-place control based on the selected listbox field.

Implementation: The **ResetColumn** method resets the FEQ to the selected ListControl field.

Example:

```
EIPManager.TakeCompleted PROCEDURE(BYTE Force)
CODE
  SELF.Column = 1
  IF SELF.Again
    SELF.ResetColumn
  END
```

See Also: EditClass.FEQ

Init, ListControl, TakeAction, TakeCompleted, TakeNewSelection

**Run (run the EIPManager)****Run( *request* )**


---

<b>Run</b>	Run the EIPManager.
<i>request</i>	An integer constant, variable, EQUATE, or expression identifying the database action (insert, change, delete) requested.

The **Run** method assigns the passed value to the Req property and executes the EIPManager.

Implementation: Return value EQUATEs are declared in \LIBSRC\TPLEQU.CLW as follows:

<b>RequestCompleted</b>	<b>EQUATE (1)</b>	<b>!Update Completed</b>
<b>RequestCancelled</b>	<b>EQUATE (2)</b>	<b>!Update Cancelled</b>

Return Data Type: **BYTE**

Example:

```
BrowseClass.AskRecord PROCEDURE(BYTE Req)
  CODE
  SELF.CheckEIP
  RETURN SELF.EIP.Run(Req)
```

See Also: **Req**

**TakeAcceptAll (validate completed row)****TakeAcceptAll ( ), VIRTUAL**


---

**TakeAcceptAll** Processes edit-in-place validation.

When an Edit-in-Place row is completed, the **TakeAcceptAll** method will validate and reselect any column that is required and empty, or any column that returns an EditAction:Cancel equate by the EditClass:TakeAccepted method, with the exception of any cancel action performed during an Insert or Change session. If all columns are successfully validated, TakeAcceptAll returns TRUE (1).

Return Value: **BYTE**

**TakeAction (process edit-in-place action)****TakeAction( *action* ), VIRTUAL**


---

TakeAction	Processes edit-in-place action.
<i>action</i>	An integer constant, variable, EQUATE, or expression that contains the action to process. Valid EQUATEs are forward, backward, next, previous, complete, and cancel.

The **TakeAction** method processes an EIPManager dialog action. The TakeAction method is your opportunity to interpret and implement the meaning of the end user's selection.

Implementation: The TakeFieldEvent conditionally calls the TakeAction method.

Corresponding EQUATEs are declared in ABEIP.INC as follows:

```

EditAction ITEMIZE(0),PRE
None      EQUATE
Forward   EQUATE    ! Next field
Backward  EQUATE    ! Previous field
Complete  EQUATE    ! OK
Cancel    EQUATE    ! Cancel
Next      EQUATE    ! Focus moving to Next record
Previous  EQUATE    ! Focus moving to Previous record
Ignore    EQUATE
          END

```

Example:

```

EIPManager.TakeFieldEvent      PROCEDURE
I UNSIGNED(1)
CODE
IF FIELD() = SELF.ListControl THEN RETURN Level:Benign .
LOOP I = 1 TO RECORDS(SELF.EQ)+1
! Optimised to pick up subsequent events from same field
IF ~SELF.EQ.Control &= NULL AND SELF.EQ.Control.Feq = FIELD()
SELF.TakeAction(SELF.EQ.Control.TakeEvent(EVENT()))
RETURN Level:Benign
END
GET(SELF.EQ,I)
END
! Code to handle an unknown field

```

See Also: TakeFieldEvent

## **TakeCompleted (process completion of edit:EIPManagerClass)**

**TakeCompleted( *force* ), VIRTUAL**

---

TakeCompleted Determines the edit-in-place dialog's action after either a loss of focus or an end user action.

*action* An integer constant, variable, EQUATE, or expression that indicates an end user requested action.

The **TakeCompleted** method conditionally calls the ResetColumn method. The BrowseEIPManager.TakeCompleted provides the bulk of the process completion functionality, and is derived from the TakeCompleted method.

Implementation: The BrowseEIPManager.TakeCompleted method calls the TakeCompleted method via PARENT syntax. TakeFocusLoss and TakeAction also call the TakeCompleted method.

**Note: TakeCompleted does not override the WindowManager.TakeCompleted method.**

Example:

```
EIPManager.TakeFocusLoss PROCEDURE
CODE
CASE CHOOSE(SELF.FocusLoss&=NULL,EIPAction:Default,BAND(SELF.FocusLoss,EIPAction:Save)
OF EIPAction:Always OROF EIPAction:Default
    SELF.TakeCompleted(Button:Yes)
OF EIPAction:Never
    SELF.TakeCompleted(Button:No)
ELSE
    SELF.TakeCompleted(0)
END
```

See Also: BrowseEIPManager.TakeCompleted, TakeFocusLoss, TakeAction

## TakeEvent (process window specific events)

### TakeEvent, DERIVED, PROC

The **TakeEvent** method processes window specific events and returns Level:Notify for an EVENT:Size, EVENT:Iconize, or EVENT:Maximize; it returns a Level:Fatal for an EVENT:CloseDown, EVENT:CloseWindow, or EVENT:Sized; all other window events return a Level:Benign.

Implementation: The TakeFieldEvent method calls the TakeEvent method. The TakeEvent method calls the TakeFocusLoss method subsequent to returning a Level:Fatal.

Return Data Type: BYTE

Example:

```
EIPManager.TakeFieldEvent      PROCEDURE
I UNSIGNED(1)
CODE
IF FIELD() = SELF.ListControl THEN RETURN Level:Benign .
LOOP I = 1 TO RECORDS(SELF.EQ)+1
! Optimised to pick up subsequent events from same field
IF ~SELF.EQ.Control &= NULL AND SELF.EQ.Control.Feq = FIELD()
SELF.TakeAction(SELF.EQ.Control.TakeEvent(EVENT()))
RETURN Level:Benign
END
GET(SELF.EQ,I)
END
! Code to handle an unknown field
```

See Also: TakeFieldEvent, TakeFocusLoss

## TakeFieldEvent (process field specific events)

**TakeFieldEvent, DERIVED, PROC**

The **TakeFieldEvent** method processes all field-specific/control-specific events for the window. It returns a value indicating whether edit-in-place process is complete and should stop.

TakeFieldEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The WindowManager.TakeEvent method calls the TakeFieldEvent method.

Return value EQUATES are declared in ABERROR.INC.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;    RVal = SELF.TakeRejected()
    OF EVENT:Selected;    RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;   RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

## TakeFocusLoss (a virtual to process loss of focus)

### TakeFocusLoss, VIRTUAL

The **TakeFocusLoss** method determines the appropriate action to take when the EIPManager window loses focus, and calls the TakeCompleted method with the appropriate parameter.

Implementation: TakeEvent and TakeFieldEvent methods conditionally call the TakeFocusLoss method.

Example:

```
EIPManager.TakeFieldEvent      PROCEDURE
I UNSIGNED(1)
CODE
IF FIELD() = SELF.ListControl THEN RETURN Level:Benign .
LOOP I = 1 TO RECORDS(SELF.EQ)+1
! Optimized to pick up subsequent events from same field
IF ~SELF.EQ.Control &= NULL AND SELF.EQ.Control.Feq = FIELD()
SELF.TakeAction(SELF.EQ.Control.TakeEvent(EVENT()))
RETURN Level:Benign
END
GET(SELF.EQ,I)
END
! Code to handle an unknown field
```

See Also: TakeCompleted

## TakeNewSelection (reset edit-in-place column:EIPManagerClass)

**TakeNewSelection, DERIVED, PROC**

The **TakeFieldEvent** method resets the edit-in-place column selected by the end user.

Implementation: TakeNewSelection is called by the BrowseEIPManager.TakeNewSelection method.

TakeNewSelection calls ResetColumn, and returns a Level:Benign.

Return Data Type: BYTE

Example:

```
BrowseEIPManager.TakeNewSelection PROCEDURE
CODE
  IF FIELD() = SELF.ListControl
    IF CHOICE(SELF.ListControl) = SELF.BC.CurrentChoice
      RETURN PARENT.TakeNewSelection()
    ELSE
      ! Code to handle Focus change to different record
    END
  END
```

See Also: ResetColumn

# EntryLocatorClass

## EntryLocatorClass Overview

The EntryLocatorClass is a LocatorClass with an input control (ENTRY, COMBO, or SPIN). An Entry Locator is a multi-character locator that activates when the locator control is *accepted* (not upon each keystroke).

Use an Entry Locator when you want a multi-character search on numeric or alphanumeric keys and you want to delay the search until the user accepts the locator control. This delayed search reduces network traffic and provides a smoother search in a client-server environment.

## EntryLocatorClass Concepts

The EntryLocatorClass lets you specify a locator control and a sort field on which to search (the free key element) for a BrowseClass object. The BrowseClass object uses the EntryLocatorClass to locate and scroll to the nearest matching item.

When the end user places one or more characters in the locator control, then *accepts* the control by pressing TAB, pressing a locator button, or selecting another control on the screen, the EntryLocatorClass object advances the BrowseClass object's LIST to the nearest matching record.

## EntryLocatorClass Relationship to Other Application Builder Classes

The BrowseClass uses the EntryLocatorClass to locate and scroll to the nearest matching item. Therefore, if your program's BrowseClass objects use an Entry Locator, your program must instantiate the EntryLocatorClass for each use. Once you register the EntryLocatorClass object with the BrowseClass object (see BrowseClass.AddLocator), the BrowseClass object uses the EntryLocatorClass object as needed, with no other code required. See the *Conceptual Example*.

## EntryLocatorClass ABC Template Implementation

The ABC BrowseBox template generates code to instantiate the EntryLocatorClass for your BrowseBoxes. The EntryLocatorClass objects are called BRW*n*::Sort#:Locator, where *n* is the template instance number and # is the sort sequence (id) number. As this implies, you can have a different locator for each BrowseClass object sort order.

You can use the BrowseBox's **Locator Behavior** dialog (the **Locator Class** button) to derive from the EntryLocatorClass. The templates provide the derived class so you can modify the locator's behavior on an instance-by-instance basis.

## EntryLocatorClass Source Files

The EntryLocatorClass source code is installed by default to the Clarion \LIBSRC folder. The specific EntryLocatorClass source code and their respective components are contained in:

ABBROWSE.INC	EntryLocatorClass declarations
ABBROWSE.CLW	EntryLocatorClass method definitions

## EntryLocatorClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a BrowseClass object and related objects, including an EntryLocatorClass object. The example initializes and page-loads a LIST, then handles a number of associated events, including scrolling, updating, and locating records.

Note that the WindowManager and BrowseClass objects internally handle the normal events surrounding the locator.

```

PROGRAM
  INCLUDE('ABWINDOW.INC')           !declare WindowManager class
  INCLUDE('ABBROWSE.INC')          !declare BrowseClass and Locator
  MAP
  END
State      FILE,DRIVER('TOPSPEED'),PRE(ST),THREAD
StateCodeKey  KEY(ST:STATECODE),NOCASE,OPT
Record      RECORD,PRE()
STATECODE   STRING(2)
STATENAME   STRING(20)
            END
            END
StView      VIEW(State)             !declare VIEW to process
            END
StateQ      QUEUE                   !declare Q for LIST
ST:STATECODE LIKE(ST:STATECODE)
ST:STATENAME LIKE(ST:STATENAME)
ViewPosition STRING(512)
            END
Access:State CLASS(FileManager)     !declare Access:State object
Init        PROCEDURE
            END
Relate:State CLASS(RelationManager) !declare Relate:State object
Init        PROCEDURE
            END
VCRRequest  LONG(0),THREAD

```

```

StWindow WINDOW('Browse States'),AT(, ,123,152),IMM,SYSTEM,GRAY
    PROMPT('Find:'),AT(9,6)
    ENTRY(@s2),AT(29,4),USE(ST:STATECODE)
    LIST,AT(8,5,108,124),USE(?StList),IMM,HVSCROLL,FROM(StateQ),|
    FORMAT('27L(2)|M~CODE~@s2@80L(2)|M~STATENAME~@s20@')
    END

ThisWindow CLASS(WindowManager)           !declare ThisWindow object
Init      PROCEDURE(),BYTE,PROC,VIRTUAL
Kill      PROCEDURE(),BYTE,PROC,VIRTUAL
          END
BrowseSt  CLASS(BrowseClass)             !declare BrowseSt object
Q         &StateQ
          END

StLocator EntryLocatorClass              !declare StLocator object
StStep    StepStringClass                !declare StStep object

CODE
    ThisWindow.Run()                     !run the window procedure

ThisWindow.Init PROCEDURE()              !initialize things
ReturnValue     BYTE,AUTO
CODE
    ReturnValue = PARENT.Init()           !call base class init
    IF ReturnValue THEN RETURN ReturnValue.
    Relate:State.Init                     !initialize Relate:State object
    SELF.FirstField = ?ST:STATECODE       !set FirstField for ThisWindow
    SELF.VCRRequest =& VCRRequest         !VCRRequest not used
    Relate:State.Open                     !open State and related files
    !Init BrowseSt object by naming its LIST,VIEW,Q,RelationManager & WindowManager
    BrowseSt.Init(?StList,StateQ.ViewPosition,StView,StateQ,Relate:State,SELF)
    OPEN(StWindow)
    SELF.Opened=True
    BrowseSt.Q =& StateQ                  !reference the browse QUEUE
    StStep.Init(+ScrollSort:AllowAlpha,ScrollBy:Runtime)!initialize the StStep object
    BrowseSt.AddSortOrder(StStep,ST:StateCodeKey) !set the browse sort order
    BrowseSt.AddLocator(StLocator)         !plug in the browse locator
    StLocator.Init(?ST:STATECODE,ST:STATECODE,1,BrowseSt) !initialize locator object
    BrowseSt.AddField(ST:STATECODE,BrowseSt.Q.ST:STATECODE) !set a column to browse
    BrowseSt.AddField(ST:STATENAME,BrowseSt.Q.ST:STATENAME) !set a column to browse
    SELF.SetAlerts()                      !alert any keys for ThisWindow
    RETURN ReturnValue

ThisWindow.Kill PROCEDURE()              !shut down things
ReturnValue     BYTE,AUTO

```

```
CODE
ReturnValue = PARENT.Kill()           !call base class shut down
IF ReturnValue THEN RETURN ReturnValue.
Relate:State.Close                   !close State and related files
Relate:State.Kill                    !shut down Relate:State object
GlobalErrors.Kill                   !shut down GlobalErrors object
RETURN ReturnValue
```

## EntryLocatorClass Properties

### EntryLocatorClass Properties

The EntryLocatorClass inherits all the properties of the LocatorClass from which it is derived. See *LocatorClass Properties* and *LocatorClass Concepts* for more information.

In addition to the inherited properties, the EntryLocatorClass also contains the following property:

### Shadow (the search value)

#### **Shadow CSTRING(40)**

The **Shadow** property contains the search value for the entry locator.

The TakeKey method adds to the search value based on the end user's keyboard input. The BrowseClass.TakeAcceptedLocator method implements the search for the specified value.

See Also:       TakeKey, BrowseClass.TakeAcceptedLocator

## EntryLocatorClass Methods

The `EntryLocatorClass` inherits all the methods of the `LocatorClass` from which it is derived. See *LocatorClass Methods* and *LocatorClass Concepts* for more information.

### **GetShadow(return shadow value)**

GetShadow, DERIVED

The **GetShadow** method returns the value of the Shadow property. The Shadow property is set based on the users keyboard input into the entry locator field.

Return Data Type: STRING

See Also: `EntryLocatorClass.SetShadow`, `EntryLocatorClass.Shadow`

**Init (initialize the EntryLocatorClass object)**

**Init**( [*control*] , *freeelement* [,*ignorecase*] [,*browseclass*] )

---

<b>Init</b>	Initializes the EntryLocatorClass object.
<i>control</i>	An integer constant, variable, EQUATE, or expression that sets the locator control for the locator. If omitted, the control number defaults to zero (0) indicating there is no locator control.
<i>freeelement</i>	The fully qualified label of a component of the sort sequence of the searched data set. The ABC Templates further require this to be a free component of a key. A free component is one that is not range limited to a single value. Typically this is also the USE variable of the locator control.
<i>ignorecase</i>	An integer constant, variable, EQUATE, or expression that determines whether the locator does case sensitive searches or ignores case. A value of one (1) or True does case insensitive searches; a value of zero (0) or False ignores case. If omitted, nocase defaults to 0.
<i>browseclass</i>	The label of the BrowseClass object for the locator. If omitted, the LocatorClass object has no direct access to the browse QUEUE or it's underlying VIEW.

The **Init** method initializes the EntryLocatorClass object.

Implementation: The Init method sets the values of the Control, FreeElement, NoCase, and ViewManager properties. The Shadow property is the *control*'s USE variable.

By default, only the StepLocatorClass and FilterLocatorClass use the *browseclass*. The other locator classes do not.

Example:

```
BRW1::Sort1:Locator.Init(,CUST:StateCode,1)           !without locator control
BRW1::Sort2:Locator.Init(?CUST:CustMo,CUST:CustNo,1) !with locator control
```

See Also: Control, FreeElement, NoCase, ViewManager

## Set (restart the locator:EntryLocatorClass)

### Set, DERIVED

The **Set** method prepares the locator for a new search.

Implementation: The Set method clears the FreeElement property and the Shadow property.

Example:

```
MyBrowseClass.TakeScroll PROCEDURE(SIGNED Event)  !process a scroll event
CODE
!handle the scroll
SELF.PostNewSelection                !post EVENT:NewSelection to list
IF ~SELF.Sort.Locator &= NULL        !if locator is present
  SELF.Sort.Locator.Set              ! clear it
END
IF SELF.Sort.Thumb &= NULL           !if thumb is present
  SELF.UpdateThumbFixed              ! reposition it
END
```

See Also: FreeElement, Shadow

**SetShadow(set shadow value)**SetShadow(*shadow*), DERIVED**SetShadow** Set the Shadow property.*shadow* A string constant, variable, EQUATE, or expression that contains the value to give to the Shadow property.The **SetShadow** property sets the Shadow property with the passed **value**.

See Also: EntryLocatorClass.GetShadow, EntryLocatorClass.Shadow

**TakeAccepted (process an accepted locator value:EntryLocatorClass)****TakeAccepted**, DERIVEDThe **TakeAccepted** method processes the accepted locator value and returns a value indicating whether the browse list display should change.

A locator value is accepted when the end user changes the locator value, then TABS off the locator control or otherwise switches focus to another control on the same window.

Implementation: The TakeAccepted method primes the FreeElement property with the entered search value, then returns one (1 or True) if a new search is required or returns zero (0 or False) if no new search is required.

Return Data Type: BYTE

Example:

**MyBrowseClass.TakeAcceptedLocator PROCEDURE**

```

CODE
IF ~SELF.Sort.Locator &= NULL           !if locator is present
  IF SELF.Sort.Locator.TakeAccepted()    !if locator value requires a search
    SELF.Reset(1)                        !reposition the view
    SELECT(SELF.ListControl)             !focus on the list control
    SELF.ResetQueue( Reset:Done )        !reset the browse queue
    SELF.Sort.Locator.Reset              !reset the locator USE variable
  END
END

```

See Also: FreeElement

**TakeKey (process an alerted keystroke:EntryLocatorClass)****TakeKey, DERIVED**

The **TakeKey** method processes an alerted keystroke for the LIST control that displays the data to be searched and returns a value indicating whether the browse list display should change. **By default, all alphanumeric keys are alerted for LIST controls.**

Implementation: The BrowseClass.TakeKey method calls the locator TakeKey method. The TakeKey method stuffs the keystroke detected by the LIST into the locator's input control and returns zero (0 or False).

Return Data Type: BYTE

Example:

**MyBrowseClass.TakeKey PROCEDURE**

```

CODE
IF RECORDS(SELF.ListQueue)
CASE KEYCODE()
OF InsertKey OROF DeleteKey OROF CtrlEnter OROF MouseLeft2 ;!handle keys
ELSE
DO CheckLocator !handle all other keystrokes
END
END
RETURN 0

```

**CheckLocator ROUTINE**

```

IF ~(SELF.Sort.Locator &= NULL)
IF SELF.Sort.Locator.TakeKey() !add keystroke to locator input control
SELF.Reset(SELF.GetFreeElementPosition()) !and refresh browse if necessary
SELF.ResetQueue(Reset:Done)
DO HandledOut
ELSE
IF RECORDS(SELF.ListQueue)
DO HandledOut
END
END
END

```

**HandledOut ROUTINE**

```

SELF.UpdateWindow
SELF.PostNewSelection
RETURN 1

```

See Also: BrowseClass.TakeKey

## Update (update the locator control and free elements)

### Update, PROTECTED, VIRTUAL

The **Update** method redraws the locator control and updates the free key elements in the record buffer with the current locator value.

Implementation: The Update method primes the FreeElement property with the current search value (the Shadow property), then calls the UpdateWindow method to redraw the locator control.

Example:

```
MyBrowseClass.UpdateWindow PROCEDURE          !update browse related controls
CODE
  IF ~(SELF.Sort.Locator &= NULL)            !if locator is present
    SELF.Sort.Locator.UpdateWindow           !redraw locator control
  END
```

See Also: FreeElement, Shadow, UpdateWindow

## UpdateWindow (redraw the locator control)

### UpdateWindow, DERIVED

The **UpdateWindow** method redraws the locator control with the current locator value.

Implementation: The Update method calls the UpdateWindow method to redraw the locator control with the current locator contents.

Example:

```
MyBrowseClass.UpdateWindow PROCEDURE          !update browse related controls
CODE
  IF ~(SELF.Sort.Locator &= NULL)            !if locator is present
    SELF.Sort.Locator.UpdateWindow           ! redraw locator control
  END
```

See Also: Update



# ErrorClass

## ErrorClass Overview

The ErrorClass declares an error manager which consistently and flexibly handles any errors. That is, for a given program scope, you define all possible errors by ID number, severity, and message text, then when an error or other notable condition occurs, you simply pass the appropriate ID to the error manager which processes it appropriately based on its severity level.

The defined "errors" may actually include questions, warnings, notifications, messages, benign tracing calls, as well as true errors. The ErrorClass comes with about forty general purpose database errors already defined. You can expand this list to include additional general purpose errors, your own application-specific errors, or even field specific data validation errors. Your expansion of the errors list may be "permanent" or may be done dynamically at runtime.

## ErrorClass Source Files

The ErrorClass source code is installed by default to the Clarion \LIBSRC. The specific ErrorClass source code and their respective components are contained in:

ABERROR.INC	ErrorClass declarations
ABERROR.CLW	ErrorClass method definitions
ABERROR.TRN	ErrorClass default error definitions

## **Multiple Customizable Levels of Error Treatment**

### **Six Levels of Treatment**

---

By default, the error manager recognizes six different levels of error severity. The default actions for these levels range from no action for benign errors to halting the program for fatal errors. The error manager also supports the intermediate actions of simply notifying the user, or of notifying the user and letting the user decide whether to continue or abort.

### **Customizable Treatments**

---

These various levels of treatment are implemented with virtual methods so they are easy to customize. The error manager calls a different virtual method for each severity level, so you can override the default error actions with your own application specific error actions. See the various *Take* methods for examples.

The recognized severity EQUATEs are declared in ABERROR.INC. These severity levels and their default actions are:

Level:Benign	no action, returns Level:Benign
Level:User	displays message, returns Level:Benign or Level:Cancel
Level:Notify	displays message, returns Level:Benign
Level:Fatal	displays message, halts the program
Level:Program	treated as Level:Fatal
Level:Cancel	used to confirm no action taken by User
any other value	treated as Level:Program

You may define your own additional severity levels *and* their associated actions.

## **Predefined Windows and Database Errors**

A list of common database errors are defined in ABERROR.TRN for your use and for the ABC Templates. The defined "errors" include questions, warnings, messages, notifications, benign tracing calls, as well as true errors.

You may edit these error definitions to suit your own requirements. That is, you may add new error definitions, change the wording of the error message text, or even translate the English text to another language.

**Note:** If you use the ABC Templates you should not remove any of the default error definitions or change their ID numbers.

## **Dynamic Extensibility of Errors**

You may add new error definitions, override default error definitions, and modify default error definitions at runtime with the methods provided for these purposes:

AddErrors        Adds new errors, overrides errors, or both.  
RemoveErrors    Removes errors, restores overridden errors, or both.  
SetFatality      Modifies the severity level of an error.

## **ErrorClass ABC Template Implementation**

The ABC Templates instantiate a global ErrorClass object called GlobalErrors. All template recognized errors are defined at program startup and almost every generated procedure then relies on the GlobalErrors object to handle known error conditions. You can use the Application Template's Global Properties dialog to specify a different class to instantiate as GlobalErrors--providing complete flexibility for error handling in your template generated procedures.

## **ErrorClass Relationship to Other Application Builder Classes**

All the classes that access files (ASCIIFileClass, ASCIIViewerClass, FileManager, RelationManager, ViewManager, and BrowseClass) use the ErrorClass. Therefore, if your program instantiates any of these classes, it must also instantiate the ErrorClass.

## **ErrorClass Macro Expansion**

The following ErrorClass methods allow runtime customization of error message text through expansion of macro symbols:

SetField	Names the field that produced the error.
SetFile	Names the file that produced the error.
ThrowFile	Names the file that produced the error, then handles the error.
ThrowMessage	Modifies error text, then handles the error.

Each error has associated message text. The error message text may contain macro symbols recognized by the ErrorClass object. The ErrorClass object expands these macro symbols to their current runtime values before displaying the message. Supported macros and their runtime substitution values are:

%File	The ErrorClass.FileName property
%Field	The ErrorClass.FieldName property
%Message	The ErrorClass.MessageText property
%Error	Value returned by ERROR()
%ErrorCode	Value returned by ERRORCODE()
%FileError	Value returned by FILEERROR()
%FileErrorCode	Value returned by FILEERRORCODE()
%ErrorText	%Error(%ErrorCode) or %FileError(%FileErrorCode)
%Previous	Text from prior defined error with the same id

The %ErrorText macro uses %FileError(%FileErrorCode)--the more specific backend server error information--when it is available, otherwise it uses %Error(%ErrorCode).

This macro expansion capability is a feature of the ErrorClass and is not a feature of the Clarion language in general.

**Tip:** You do not need to specify two percent signs (%%) to display a percent sign (%) in your message text.

## **ErrorClass Multi-Language Capability**

Because all error message text is defined in one place (ABERROR.TRN), it is easy to implement non-English error messages. For static (permanent) language translation, simply translate the English text in ABERROR.TRN to the language of your choice. Alternatively, for dynamic language translation, you may add an error definition block to ABERROR.TRN for each supported language. For example in ABERROR.TRN declare:

```
DefaultErrors GROUP      !English error messages
              END
GermanErrors  GROUP      !German error messages
              END
```

Then at runtime, initialize the error manager with the appropriate error definition block. For example, you could override the Init method (defined in ABERROR.CLW) with something like this:

```
      INCLUDE('ABERROR.INC')                !declare ErrorClass
MyErrorClass CLASS(ErrorClass)             !declare derived class
Init          PROCEDURE(BYTE PreferredLanguage)
              END

GlobalErrors MyErrorClass                  !declare GlobalErrors object
Language     BYTE                          !Language Flag
Language:English EQUATE(0)                 !English equate
Language:German EQUATE(1)                  !German equate

CODE
Language = GETINI('Preferences','Language',0) !get language preference
GlobalErrors.Init(Language)                  !GlobalErrors initialization
                                              !with preferred language

MyErrorClass.Init PROCEDURE(BYTE PreferredLanguage) !New Init method
CODE
SELF.Errors &= NEW ErrorEntry               !allocate new Errors list
CASE PreferredLanguage                       !which language was selected
OF Language:German                           !if German
    SELF.AddErrors(GermanErrors)             !add German errors to list
ELSE                                           !otherwise...
    SELF.AddErrors(DefaultErrors)           !add default (English) errors
END
```

Alternatively, you could call the AddErrors method to define *additional* errors for the selected language as shown in the following example.



## ErrorClass Properties

### ErrorClass Properties

There are two types of ErrorClass properties, the Errors list and the macro substitution values. The most important property is the Errors list--the list of errors recognized by ErrorClass. The defined "errors" may actually include questions, warnings, notifications, benign tracing calls, as well as true errors. This list is established by the ErrorClass initialization method, ErrorClass.Init. The list may be modified thereafter by methods provided for this purpose, allowing application specific errors (such as field specific invalid data messages).

The other three ErrorClass properties support the error text "macros" recognized by the error manager. The error manager expands these macro symbols to their current runtime values before displaying the message.

### DefaultCategory (error category)

**DefaultCategory**      **ASTRING, PROTECTED**

The **DefaultCategory** is a string that is a classification of the type of error. This property is set by the SetCategory. The Init method sets the DefaultCategory to 'ABC'. When the category is changed by SetCategory, the new category becomes the default category.

See Also:

ErrorClass.Init, ErrorClass.SetCategory, ErrorClass.GetCategory

### ErrorLog (errorlog interface)

**ErrorLog**      **&ErrorLogInterface, PROTECTED**

The **ErrorLog** property is a reference to the errorlog interface that manages the error log file.

## Errors (recognized error definitions)

### Errors &ErrorEntry, PROTECTED

The **Errors** property is a reference to the data structure that holds all errors recognized by the ErrorClass. The defined "errors" may actually include questions, warnings, messages, notifications, benign tracing calls, as well as true error conditions.

The default errors are defined in ABERROR.TRN. You may edit ABERROR.TRN to customize the default error list. The Init method adds these default error definitions to the Errors property at runtime. You may also use the SetFatality method, the AddErrors method, and the RemoveErrors method to customize the Errors property at runtime.

The SetFatality method changes the severity level of a specified error.

The AddErrors method lets you add more error definitions, override existing error definitions, or both. The Errors property may have more than one error with the same ID. Error definitions added later "override" any earlier definitions with the same IDs. The "overridden" definitions are preserved for substitution into the %Previous macro symbol.

The RemoveErrors method lets you remove error definitions, restore previously overridden errors, or both.

The error message text may contain "macros" recognized by the error manager. The error manager expands these macro symbols to their current runtime values before displaying the message. See *Macro Expansion* for more information.

Implementation: Errors is a reference to a queue declared in ABERROR.INC as follows. For each recognized error, the Errors property includes an ID number, error message text, window title text, and a severity indicator.

```

ErrorEntry QUEUE,TYPE      !List of all error definitions
Id           USHORT       !Error message identifier
Message      &STRING      !Message text
Title        &STRING      !Error window caption bar text
Fatality     BYTE         !Severity of error
END

```

See Also: AddErrors, Init, RemoveErrors, SetFatality

### **FieldName (field that produced the error)**

**FieldName**      **CSTRING(MessageMaxlen), PROTECTED**

The **FieldName** property contains the name of the field that produced the error. The SetField method sets the value of the FieldName property. The FieldName value replaces any %Field symbols within the error message text.

MessageMaxlen is a constant EQUATE declared in ABERROR.INC.

See Also:          SetField

### **FileName (file that produced the error)**

**FileName**      **CSTRING(MessageMaxlen), PROTECTED**

The **FileName** property contains the name of the file that produced the error. The SetFile and ThrowFile methods both set the value of the FileName property. The FileName value then replaces any %File symbols within the error message text.

MessageMaxlen is a constant EQUATE declared in ABERROR.INC.

See Also:          SetFile, ThrowFile

### **History (error history structure)**

**History**      **&ErrorHistoryList,PROTECTED**

The **History** property is a reference to the ErrorHistoryList structure that holds the history for errors that have previously occurred. The error History is determined based on the HistoryThreshold and HistoryResetOnView properties.

### **HistoryResetOnView(clear error history log file)**

**HistoryResetOnView**      **BYTE**

The **HistoryResetOnView** property determines if the error history view structure should be cleared upon viewing an error message. If this property is set to one (1 or True), the History structure will be reset after each error is viewed. Setting this property to zero (0 or False) will cause the errors to be queued in the History structure.

### **HistoryThreshold (determine size of error history)**

**HistoryThreshold**      **LONG**

The **HistoryThreshold** property sets the number of items to store in the error log file. Setting this property to -1 keeps all errors. Setting this property to 0 switches off error history logging.

### **HistoryViewLevel (trigger error history)**

**HistoryViewLevel**      **LONG**

The **HistoryViewLevel** property sets the error level which triggers error history viewing. This property is only valid with a HistoryThreshold other than 0.

### **KeyName (key that produced the error)**

**KeyName**      **CSTRING(MessageMaxlen), PROTECTED**

The **KeyName** property contains the name of the key that produced the error. The SetKey method sets the value of the KeyName property. The KeyName value then replaces any %Key symbols within the error message text.

MessageMaxlen is a constant EQUATE declared in ABERROR.INC.

See Also: SetKey

### **LogErrors (turn on error history logging)**

**LogErrors**     **BYTE**

The **LogErrors** property turns the error history logging on or off. Setting this property to one (1 or True) turns on the error logging. Setting this property to zero (0 or False) turns off the error logging.

### **MessageText (custom error message text)**

**MessageText**     **CSTRING(MessageMaxlen), PROTECTED**

The **MessageText** property contains text to substitute for any %Message symbols within the error message text. The ThrowMessage method sets the value of the MessageText property. The MessageText value then replaces any %Message symbols within the error message text.

MessageMaxlen is a constant EQUATE declared in ABERROR.INC.

See Also:            ThrowMessage

### **Silent (silent error flag)**

**Silent**     **BYTE**

The **Silent** property determines whether an error will be displayed to the screen. If Silent is set to one (1 or True), the error message box will not be displayed to the screen; however it will be added to the error log file. If Silent is set to zero, (0 or False) the error is displayed to the screen as well as added to the error log file.

## ErrorClass Methods

### ErrorClass Functional Organization--Expected Use

As an aid to understanding the ErrorClass, it is useful to organize the various ErrorClass methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the ErrorClass methods.

#### Non-Virtual Methods

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the ErrorClass object
AddErrors	add or override recognized error definitions
SetFatality	change the severity level of a specific error
Kill	terminate the ErrorClass object

##### **Mainstream Use:**

Throw	process an error
ThrowFile	set substitution value of %File then process an error
ThrowMessage	set substitution value of %Message then process an error
Message	display an error message from the Errors list

##### **Occasional Use:**

SetField	set the substitution value of the %Field macro
SetFile	set the substitution value of the %File macro
SetErrors	save the current error state
SetId	make a selected error the current one
RemoveErrors	remove (and/or restore) error definitions
TakeError	process an error, assuming SetError has been called

---

**Virtual Methods**

---

Typically, you will not call these methods directly--the Non-Virtual methods call them. We anticipate you will want to override these methods, and because they are virtual, they are very easy to override. However they do provide reasonable default behavior in case you do not want to override them. These methods are listed functionally rather than alphabetically.

TakeBenign	process benign errors
TakeNotify	process notify errors
TakeUser	process user errors
TakeFatal	process fatal errors
TakeProgram	process program errors
TakeOther	process any other errors

**AddErrors (add or override recognized errors)****AddErrors**( *error block* ), **VIRTUAL**


---

**AddErrors** Adds entries to the Errors property from the *error block* passed to it.

*error block* A GROUP whose first component field is a USHORT containing the number of error entries in the GROUP. Subsequent component fields define the error entries.

The **AddErrors** method receives error entries and adds them to the existing Errors property. These later added Error definitions "override" any earlier definitions with the same IDs. The "overridden" definitions are preserved for substitution into the %Previous macro symbol, and may be fully restored by removing the overriding error entries with the RemoveErrors method.

Implementation: AddErrors assumes the Errors property has already been created by Init or by some other method.

Each *error block* entry consists of a USHORT containing the error ID, a BYTE containing the severity level, a PSTRING containing the title to display on the error message window, and another PSTRING containing the error message text.

Example:

```
AppErrors GROUP
Number      USHORT(2)                !number of errors in the group
            USHORT(Msg:RebuildKey)    !first error ID
            BYTE(Level:Notify)        !severity level
            PSTRING('Invalid Key')    !window title
            PSTRING('%File key is invalid.') !message text
            USHORT(Msg:RebuildFailed)  !second error ID
            BYTE(Level:Fatal)         !severity level
            PSTRING('Key was not built') !window title
            PSTRING('Repairing key for %File.')!message text
            END
GlobalErrors ErrorClass                !declare GlobalErrors object
CODE
GlobalErrors.Init                       !GlobalErrors initialization
GlobalErrors.AddErrors(AppErrors)       !add some app specific errors
Main                                     !call main procedure
GlobalErrors.Kill                       !GlobalErrors termination
```

See Also: Init, Errors, RemoveErrors

## AddHistory (update History structure)

**AddHistory**, VIRTUAL

The **AddHistory** method adds an entry to the History structure. This structure is used to display the message to the screen.

## GetCategory (retrieve error category)

**GetCategory**(*[id]*)

---

**GetCategory** Retrieves the current error category.

*id* An integer constant, variable, EQUATE, or expression that indicates the error id.

The **GetCategory** method retrieves the error category from the ErrorEntry structure. If the *id* is omitted the DefaultCategory is returned.

Return Data Type: ASTRING

## GetError (Retrieve the current error message)

**GetError**(*[errorlevel]*)

---

**GetError** Retrieves the current error message.

*errorlevel* An integer constant, variable, EQUATE, or expression that indicates the error level.

The **GetError** method retrieves the error message from the SaveError or SaveFileError property, based on the type of *errorlevel* that is detected.

Return Data Type: CSTRING

## **GetErrorCode (Retrieve the current Errorcode)**

**GetErrorCode**(*errorlevel* )

---

**GetErrorCode** Retrieves the current error code

**errorlevel** An integer constant, variable, EQUATE, or expression that indicates the error level.

The **GetErrorCode** method retrieves the error code from the SaveErrorCode or SaveFileErrorCode property, based on the type of errorlevel detected.

Return Data Type: LONG

## **GetProcedureName (return procedure name )**

**GetProcedureName**

The **GetProcedureName** method returns the name of the procedure in which it has been called.

Implementation: Returns the name of the procedure as established in the .APP file providing that the procedure name has been added to a PRIVATE queue by the SetProcedureName method. The GetProcedureName method is not called by any other methods or templates.

Return Data Type: STRING

```
MESSAGE(GlobalErrors.GetProcedureName()) ! Displays the procedure name in  
! a MESSAGE dialog
```

See Also: SetProcedureName

## HistoryMsg (initialize the message window)

**HistoryMsg**(*caption, icon, buttons, default button*), PROC, VIRTUAL

---

HistoryMsg	Initialize the message box window and controls.
<i>caption</i>	A string constant, variable, EQUATE, or expression that specifies the message box window caption.
<i>icon</i>	An integer constant, variable, EQUATE, or expression that indicates the icon to display on the message box.
<i>buttons</i>	An integer constant, variable, EQUATE, or expression that indicates which Windows standard buttons to place on the message box. This may indicate multiple buttons.
<i>default button</i>	An integer constant, variable, EQUATE, or expression that indicates the default button on the message box.

The **HistoryMsg** method initializes the error message dialog.

Return Data Type: LONG

## **Init (initialize the ErrorClass object)**

**Init**  
**Init(*errorlog*), VIRTUAL**

---

**Init** Initialize the ErrorClass object.  
*errorlog* The label of the ErrorLogInterface.

The **Init** method initializes the ErrorClass object and adds the default errors.

Implementation: Creates the Errors property and calls the AddErrors method to initialize it with the default errors defined in ABERROR.TRN. Default error ID EQUATEs are defined in ABERROR.INC.

The standard templates instantiate a single global ErrorClass object and make a single global call to Init. However, you may wish to instantiate an ErrorClass object with a separate set of errors for each base class, or for any other logical entity (for example a PayrollErrors object for the Payroll segment of your program).

Example:

```
GlobalErrors ErrorClass      !declare GlobalErrors object
CODE
GlobalErrors.Init           !GlobalErrors initialization
Main                        !call main procedure
GlobalErrors.Kill           !GlobalErrors termination
```

See Also: AddErrors, Errors, Kill

**Kill (perform any necessary termination code)****Kill**

The **Kill** method disposes any memory allocated during the object's lifetime and performs any other necessary termination code.

Implementation: Disposes the Errors queue created by the Init method.

Example:

```
GlobalErrors ErrorClass !declare GlobalErrors object
CODE
GlobalErrors.Init      !GlobalErrors initialization
Main                   !call main procedure
GlobalErrors.Kill      !GlobalErrors termination
```

See Also:        Init

**Message (display an error message)****Message**( *error id*, *buttons*, *default button* )

---

<b>Message</b>	Displays an error message dialog box and returns the button the user pressed.
<i>error id</i>	An integer constant, variable, EQUATE, or expression that indicates which ErrorClass.Errors message to show in the dialog box.
<i>buttons</i>	An integer constant, variable, EQUATE, or expression that indicates which Windows standard buttons to place on the dialog box. This may indicate multiple buttons.
<i>default button</i>	An integer constant, variable, EQUATE, or expression that indicates the default button on the dialog box.

The **Message** method displays a Windows-standard message box containing the error message text from the Errors property, and returns the number of the button the user presses to exit the dialog box. This method provides a simple, centrally maintainable, consistent way to display messages.

Implementation: Uses the MESSAGE statement to display an application modal window with a question icon, the caption defined in the Errors property, and the message text defined in the Errors property.

The ABERROR.INC file contains a list of default symbolic constants for the *error id* parameter.

The EQUATES.CLW file contains symbolic constants for the *buttons* and *default button* parameters. The EQUATEs are:

```
BUTTON:OK
BUTTON:YES
BUTTON:NO
BUTTON:ABORT
BUTTON:RETRY
BUTTON:IGNORE
BUTTON:CANCEL
BUTTON:HELP
```

Return Data Type: LONG

Example:

```
!attempted auto increment of key has failed,
!show Message box with Yes and No buttons, the default is No
GlobalErrors.SetErrors      !Set value of %ErrorText macro
IF GlobalErrors.Message(Msg:RetryAutoInc,BUTTON:Yes+BUTTON:No,BUTTON:No) = BUTTON:Yes
CYCLE
END
```

**Msg (initiate error message destination)**

**Msg**( *txt*, [*caption*], [*icon*], [*buttons*], [*default button*], [*style*]), PROC, VIRTUAL, PROTECTED

---

<b>Msg</b>	Initiates the destination of the error message.
<i>txt</i>	A string constant, variable, EQUATE, or expression that contains the error message text to display in the message box.
<i>caption</i>	A string constant, variable, EQUATE, or expression that specifies the message box window caption.
<i>icon</i>	An integer constant, variable, EQUATE, or expression that indicates the icon to display on the message box.
<i>buttons</i>	An integer constant, variable, EQUATE, or expression that indicates which Windows standard buttons to place on the message box. This may indicate multiple buttons. If omitted this is equivalent to Button:OK.
<i>default button</i>	An integer constant, variable, EQUATE, or expression that indicates the default button on the message box.

---

<i>style</i>	An integer constant, variable, EQUATE, or expression that indicates the font style to use withing the list control on the message box dialog.
--------------	---

The **Msg** method initiates the error destination. If the error is to be written to the error log file, the AddHistory method is called. The MessageBox method is called to display the error message to the window.

Return Data Type: LONG

## **MessageBox (display error message to window)**

**MessageBox**([*Level*], *txt*, [*caption*], [*icon*], *buttons*, *default button*, *style*), VIRTUAL, PROTECTED

---

**MessageBox** A simple MESSAGE() window to display error.

*level* The default error level is Level:Benign.

*txt* A string constant, variable, EQUATE, or expression that contains the error message text to display in the message box.

*caption* A string constant, variable, EQUATE, or expression that specifies the message box window caption.

*icon* An integer constant, variable, EQUATE, or expression that indicates the icon to display on the message box.

*buttons* An integer constant, variable, EQUATE, or expression that indicates which Windows standard buttons to place on the message box. This may indicate multiple buttons. If omitted this is equivalent to Button:OK.

*default button* An integer constant, variable, EQUATE, or expression that indicates the default button on the message box.

---

*style* An integer constant, variable, EQUATE, or expression that indicates the font style to use withing the list control on the message box dialog.buttons

The **MessageBox** method implements a simple MESSAGE() dialog to display the error message to the window. This can be called independantly. It is also used is the ErrorClass is configured to not have a HistoryThreshold.

Return Data Type: LONG

**RemoveErrors (remove or restore recognized errors)**

**RemoveErrors**( *error block* )

---

**RemoveErrors** Removes the entries specified in the *error block* from the Errors property.

*error block*      A GROUP whose first component field is a USHORT containing the number of error entries in the GROUP. Subsequent component fields define the error entries.

The **RemoveErrors** method receives error entries and deletes them from the existing Errors property.

The Errors property may contain more than one error with the same ID. Errors added later override earlier added errors with the same IDs. If you remove an overriding error definition, the "overridden" error is fully restored.

Implementation:      RemoveErrors assumes the Errors property has already been created by Init or by some other method.

Each *error block* entry consists of a USHORT containing the error ID, a BYTE containing the severity level, a PSTRING containing the title to display on the error message window, and another PSTRING containing the error message text. However, RemoveErrors only considers the error ID when removing errors.

Example:

```
GlobalErrors ErrorClass                   !declare GlobalErrors object
Payroll PROCEDURE
PayErrors GROUP,STATIC
Number      USHORT(2)                   !number of errors in the group
            USHORT(Msg:RebuildKey)      !first error ID
            BYTE(Level:Notify)          !severity level
            PSTRING('Invalid Key')      !window title
            PSTRING('%File key is invalid.') !message text
            USHORT(Msg:RebuildFailed)    !second error ID
            BYTE(Level:Fatal)            !severity level
            PSTRING('Key was not built')  !window title
            PSTRING('Repairing key for %File.') !message text
            END
CODE
GlobalErrors.AddErrors(PayErrors)        !add Payroll specific errors
!process payroll
GlobalErrors.RemoveErrors(PayErrors)     !remove Payroll specific errors
```

See Also:            AddErrors, Init, Errors

## ResetHistory(clear History structure)

### ResetHistory

The **ResetHistory** method resets the error History structure. This can be done after the view of each error message if the ResetHistoryOnView property is set.

## SetCategory (set error category)

### SetCategory([*id*], *category*)

---

**SetCategory** Retrieves the current error category.

***id*** An integer constant, variable, EQUATE, or expression that indicates the error id.

***category*** A string constant, variable, EQUATE, or expression that contains the category for the corresponding id

The **SetCateogry** method sets the error category in the ErrorEntry structure. If the *id* is omitted the DefaultCategory is used.

## SetErrors (save the error state)

### SetErrors

The **SetErrors** method saves the current error state for use by the ErrorClass.

Implementation: The **SetErrors** method saves the return values from `ERROR()`, `ERRORCODE()`, `FILEERROR()`, and `FILERERRORCODE()`. The saved values are used for expansion of any `%Error`, `%ErrorCode`, `%FileError`, or `%FileErrorCode` macro symbols within the error message text.

The `Throw` method calls `SetErrors` prior to handling the specified error, therefore you only need to call the `SetErrors` method when you do not use the `Throw` method.

Example:

```
!an error occurs
GlobalErrors.SetErrors           !save the error state
OPEN(LogFile)                   !open log (changes the error state)
Log:Text = FORMAT(TODAY(),@D1)&' '&FORMAT(CLOCK(),@T1)
ADD(LogFile)                    !write log (changes the error state)
RETURN GlobalErrors.TakeError(Msg:AddFailed)!process error with saved error state
```

See Also: [Throw](#)

**SetFatality (set severity level for a particular error)****SetFatality**( *error id*, *severity* )

---

<b>SetFatality</b>	Specifies the severity of a particular error in the Errors property.
<i>error id</i>	An integer constant, variable, EQUATE, or expression that indicates which error definition to modify.
<i>severity</i>	An integer constant, variable, EQUATE, or expression that indicates the severity of the error.

The **SetFatality** method specifies the severity of a particular error in the Errors property. If there is more than one error with the same *error id*, only the *last* matching error in the list is affected.

Implementation: The SetFatality method calls the SetId method to locate the specified error.

The ABERROR.INC file contains a list of default symbolic constants for the *error id* parameter. It also contains symbolic constants for the *severity* parameter. The severity EQUATEs and their default actions are:

Level:Benign	no action, returns Level:Benign
Level:User	displays message, returns Level:Benign or Level:Cancel
Level:Notify	displays message, returns Level:Benign
Level:Fatal	displays message, halts the program
Level:Program	treated as Level:Fatal
any other value	treated as Level:Program

You may define your own additional severity levels *and* their associated actions.

Example:

```
GlobalErrors ErrorClass
CODE
GlobalErrors.Init
GlobalErrors.SetFatality(Msg:CreateFailed,Level:Fatal) !change severity to fatal
CREATE(MyFile)
IF ERRORCODE( )
    GlobalErrors.SetFile('MyFile')           !specify file that failed
    GlobalErrors.Throw(Msg:CreateFailed)     !issue fatal error message
END
```

See Also: Errors, SetId

**SetField (set the substitution value of the %Field macro)****SetField**( *fieldname* )

---

<b>SetField</b>	Sets the substitution value of the %Field macro.
<i>fieldname</i>	A string constant, variable, EQUATE, or expression that indicates which field produced the error.

The **SetField** method sets the substitution value of the %Field macro. This value replaces any %Field symbols within the error message text.

Implementation: Assigns the *fieldname* parameter to the ErrorClass.FieldName property.

Example:

```
!Lookup on State Code failed
GlobalErrors.SetField('State')           !set field that failed
GlobalErrors.ThrowMessage(Msg:FieldNotInFile,'State File') !process the error
```

See Also:        [FieldName](#)

## SetFile (set the substitution value of the %File macro)

**SetFile**( *filename* )

---

<b>SetFile</b>	Sets the substitution value of the %File macro.
<i>filename</i>	A string constant, variable, EQUATE, or expression that indicates which file produced the error.

The **SetFile** method sets the substitution value of the %File macro. This value replaces any %File symbols within the error message text.

The ThrowFile method sets the %File macro before processing the specified error. That is, ThrowFile combines the functionality of SetFile and Throw into a single method.

Implementation: Assigns the *filename* parameter to the ErrorClass.FileName property.

Example:

```
CREATE(MyFile)
IF ERRORCODE( )           !if error occurred
  GlobalErrors.SetFile(NAME(MyFile)) !set file that failed
  GlobalErrors.Throw(Msg:CreateFailed) !process the error
END
```

See Also: FileName, ThrowFile

**SetKey (set the substitution value of the %Key macro)****SetKey**( *keyname* )

---

**SetKey** Sets the substitution value of the %Key macro.

*keyname* A string constant, variable, EQUATE, or expression that indicates which key produced the error.

The **SetKey** method sets the substitution value of the %Key macro. This value replaces any %Key symbols within the error message text.

Implementation: Assigns the *keyname* parameter to the ErrorClass.KeyName property.

Example:

```

CASE ERRORCODE()
OF NoError
  SELF.AutoIncDone = 0
  RETURN Level:Benign
OF DupKeyErr
  IF HandleError
    IF ~SELF.HasAutoInc
      GET(SELF.File,0)      ! Flag for DUPLICATE function
    END
    LOOP I = 1 TO RECORDS(SELF.Keys)
      GET(SELF.Keys,I)
      IF DUPLICATE(SELF.Keys.Key)
        SELF.Errors.SetKey(CHOOSE(CLIP(SELF.Keys.Description)<>' ',|
          CLIP(SELF.Keys.Description),SELF.Keys.Key{PROP:NAME}))
        SELF.ThrowMessage(Msg:DuplicateKey,SELF.Keys.Description)
        RETURN Level:Notify
      END
    END
  ELSE
    SELF.SetError(Msg:DuplicateKey)
  END
ELSE
  SELF.SetError(Msg:AddFailed)
  IF HandleError
    RETURN SELF.Throw()
  END
END
RETURN Level:Notify

```

See Also:        [KeyName](#)

**SetId (make a specific error current)****SetId**( *error id* ), PROTECTED

---

<b>SetId</b>	Makes the specified error the current one.
<i>error id</i>	An integer constant, variable, EQUATE, or expression that indicates which error definition is current.

The **SetId** method makes the specified error the current one for processing by other ErrorClass methods. If more than one error definition matches the specified *error id*, the last defined error is used. This lets errors defined later override earlier defined errors with the same ID, while preserving the earlier defined errors for substitution into the %Previous macro symbol.

This method is PROTECTED, therefore, it can only be called from an ErrorClass method, or a method in a class derived from ErrorClass.

Implementation: The ABERROR.INC file contains a list of default EQUATEs for the *error id* parameter.

Example:

```

ErrorClass.TakeError PROCEDURE(SHORT Id)
  CODE
  SELF.SetId(Id)
  CASE SELF.Errors.Fatality
  OF Level:Benign
    RETURN SELF.TakeBenign()
  OF Level:User
  OROF Level:Cancel
    RETURN SELF.TakeUser()
  OF Level:Program
    RETURN SELF.TakeProgram()
  OF Level:Fatal
    RETURN SELF.TakeFatal()
  OF Level:Notify
    SELF.TakeNotify()
    RETURN Level:Notify
  ELSE
    RETURN SELF.TakeOther()
  END

```

See Also: Errors

---

**SetProcedureName ( stores procedure names)****SetProcedureName( [*name*] )**

---

**SetProcedureName**

The **SetProcedureName** method stores the name of the procedure, as defined in the .APP file, in a PRIVATE queue.

*name* A string constant, variable or EQUATE containing the name of the procedure to add to ProcName queue. If omitted, the current procedure name is deleted from the ProcName queue.

Implementation: SetProcedureName is called by the ABWindow.tpw so that every template generated procedure utilizing a window will have an entry in the ProcName queue. SetProcedureName is inserted into the Init method of the window using the %Procedure macro as the passed parameter. It is called again in the Kill method of the window, and the *name* parameter is omitted.

The ProcName queue is a PRIVATE queue declared in ABError.clw.

Example:

```
GlobalErrors.SetProcedureName( '%Procedure' )
```

See Also:        GetProcedureName

## **SubsString (resolves error message macros)**

### **SubsString, PROTECTED**

The **SubsString** method returns the current error message text with all runtime macros resolved.

Implementation: The TakeFatal, TakeNotify, TakeUser, and Message methods call the SubsString method to resolve macros.

Return Data Type: **STRING**

#### **ErrorClass.TakeFatal PROCEDURE**

```
CODE
MESSAGE(Self.SubsString() & ' Press OK to end this application', |
  Self.Errors.Title,ICON:Exclamation,Button:OK,BUTTON:OK,0)
HALT(0,Self.Errors.Title)
RETURN Level:Fatal
```

See Also: FileName, FieldName, Message, MessageText, TakeFatal, TakeNotify, TakeUser

**TakeBenign (process benign error)****TakeBenign, PROTECTED, VIRTUAL, PROC**

The **TakeBenign** method is called when an error with Level:Benign is "Thrown" to the ErrorClass (see Throw, ThrowFile, ThrowMessage).

TakeBenign must return a severity level.

Implementation: The base class method (ErrorClass.TakeBenign) returns Level:Benign.

Return Data Type: BYTE

Example:

```

        INCLUDE( 'ABERROR.INC' )           !declare ErrorClass
MyErrorClass CLASS(ErrorClass)           !declare derived class
TakeBenign   FUNCTION,BYTE,VIRTUAL !prototype corresponding virtual
        END
GlobalErrors MyErrorClass                !declare GlobalErrors object
CODE
GlobalErrors.Init                        !GlobalErrors initialization
.
.
.
GlobalErrors.Throw(Msg:NoError) !Throw method calls SELF.TakeBenign to
                                !automatically call the derived class method
                                !rather than the base class method
.
.
.

MyErrorClass.TakeBenign FUNCTION !derived class virtual to handle benign
errors
CODE
!your custom code here
RETURN Level:Benign

```

See Also: TakeError, Throw, ThrowFile, ThrowMessage

**TakeError (process specified error)****TakeError( *error id* ), PROC**


---

<b>TakeError</b>	Locates the specified error, calls the appropriate method to handle it, then returns the severity level.
<i>error id</i>	An integer constant, variable, EQUATE, or expression that indicates which error to process.

The **TakeError** method locates the specified error, then based on its severity level calls the appropriate (*TakeLevel*) method to process the error, then returns the severity level.

TakeError assumes SetErrors has already been called to save the current error state.

Implementation: The ABERROR.INC file contains a list of default symbolic constants for the *error id* parameter.

By default, the error manager recognizes six different levels of error severity. The TakeError method calls a different virtual method (*TakeLevel*) for each severity level, which makes it easy to override the default error actions with your own application-specific error actions. The recognized severity EQUATES are declared in ABERROR.INC. These severity levels and their default actions are:

Level:Benign	no action, returns Level:Benign
Level:User	displays message, returns Level:Benign or Level:Cancel
Level:Notify	displays message, returns Level:Benign
Level:Fatal	displays message, halts the program
Level:Program	treated as Level:Fatal
any other value	treated as Level:Program

Return Data Type: BYTE

See Also: Errors, SetErrors, TakeBenign, TakeNotify, TakeUser, TakeFatal, TakeProgram, TakeOther, Throw

**TakeFatal (process fatal error)****TakeFatal, PROTECTED, VIRTUAL, PROC**

The **TakeFatal** method is called when an error with Level:Fatal is "Thrown" to the ErrorClass (see Throw, ThrowFile, ThrowMessage).

TakeFatal must return a severity level (if the program is not HALTed).

Implementation: The base class method (ErrorClass.TakeFatal) displays the error message and HALTs the program. Although this method does not actually return, the RETURN statement is required to avoid compile errors.

Return Data Type: BYTE

Example:

```

        INCLUDE('ABERROR.INC')      !declare ErrorClass
MyErrorClass CLASS(ErrorClass)    !declare derived class
TakeFatal    FUNCTION,BYTE,VIRTUAL !prototype corresponding virtual
        END
GlobalErrors MyErrorClass        !declare GlobalErrors object
CODE
GlobalErrors.Init                !GlobalErrors initialization
!program code
GlobalErrors.Throw(Msg:CreateFailed) !Throw method calls SELF.TakeFatal to
!automatically call the derived class method
!rather than the base class method

!program code

MyErrorClass.TakeFatal FUNCTION    !derived class virtual to handle fatal
errors
CODE
!your custom code here
RETURN Level:Fatal

```

See Also: TakeError, Throw, ThrowFile, ThrowMessage

**TakeNotify (process notify error)****TakeNotify, PROTECTED, VIRTUAL**

The **TakeNotify** method is called when an error with Level:Notify is "Thrown" to the ErrorClass (see Throw, ThrowFile, ThrowMessage).

Implementation: The base class method (ErrorClass.TakeNotify) displays the error message and returns nothing. Note however, that the various "Throw" methods return Level:Benign (via the TakeError method) when a Level:Notify error is "Thrown."

Example:

```

    INCLUDE('ABERROR.INC')           !declare ErrorClass
MyErrorClass CLASS(ErrorClass)      !declare derived class
TakeNotify  PROCEDURE,VIRTUAL       !prototype corresponding virtual
    END
GlobalErrors MyErrorClass           !declare GlobalErrors object
CODE
GlobalErrors.Init                   !GlobalErrors initialization
!program code
GlobalErrors.Throw(Msg:CreateFailed) !Throw method calls SELF.TakeNotify to
    !automatically call the derived class method
    !rather than the base class method
!program code

MyErrorClass.TakeNotify PROCEDURE !derived class virtual to handle notify errors
CODE
!your custom code here
RETURN

```

See Also: TakeError, Throw, ThrowFile, ThrowMessage

## TakeOther (process other error)

**TakeOther, PROTECTED, VIRTUAL, PROC**

The **TakeOther** method is called when an error with an unrecognized severity level is "Thrown" to the ErrorClass (see Throw, ThrowFile, ThrowMessage). By default, an "other" error is treated as a program error.

TakeOther must return a severity level.

Implementation: The base class method (ErrorClass.TakeOther) calls TakeProgram.

Return Data Type: BYTE

Example:

```
    INCLUDE('ABERROR.INC')           !declare ErrorClass
MyErrorClass CLASS(ErrorClass)      !declare derived class
TakeOther    FUNCTION,BYTE,VIRTUAL !prototype corresponding virtual
    END
GlobalErrors MyErrorClass           !declare GlobalErrors object
    CODE
    GlobalErrors.Init                !GlobalErrors initialization
    !program code
    GlobalErrors.Throw(Msg:CreateFailed) !Throw calls SELF.TakeOther to
    !automatically call the derived class method
    !rather than the base class method
    !program code

MyErrorClass.TakeOther FUNCTION      !derived class virtual to handle "other" errors
    CODE
    !your custom code here
    RETURN Level:Program
```

See Also: TakeError, Throw, ThrowFile, ThrowMessage

**TakeProgram (process program error)****TakeProgram, PROTECTED, VIRTUAL, PROC**

The **TakeProgram** method is called when an error with Level:Program is "Thrown" to the ErrorClass (see Throw, ThrowFile, ThrowMessage). By default, a program error is treated as a fatal error.

TakeProgram must return a severity level.

Implementation: The base class method (ErrorClass.TakeProgram) calls TakeFatal.

Return Data Type: BYTE

Example:

```

    INCLUDE('ABERROR.INC')           !declare ErrorClass
MyErrorClass CLASS(ErrorClass)      !declare derived class
TakeProgram  FUNCTION,BYTE,VIRTUAL !prototype corresponding virtual
    END
GlobalErrors MyErrorClass           !declare GlobalErrors object
    CODE
GlobalErrors.Init                   !GlobalErrors initialization
!program code
GlobalErrors.Throw(Msg:CreateFailed) !Throw calls SELF.TakeProgram to
!automatically call the derived class method
!rather than the base class method

!program code

MyErrorClass.TakeProgram FUNCTION    !derived class virtual to handle program errors
    CODE
!your custom code here
RETURN Level:Program

```

See Also: TakeError, Throw, ThrowFile, ThrowMessage

**TakeUser (process user error)****TakeUser, PROTECTED, VIRTUAL, PROC**

The **TakeUser** method is called when an error with Level:User is "Thrown" to the ErrorClass (see Throw, ThrowFile, ThrowMessage).

TakeUser must return a severity level to denote the user's response.

Implementation: The base class method (ErrorClass.TakeUser) displays the error message and returns either Level:Benign or Level:Cancel depending on the end user's response.

Return Data Type: BYTE

Example:

```

    INCLUDE('ABERROR.INC')           !declare ErrorClass
MyErrorClass CLASS(ErrorClass)     !declare derived class
TakeUser    FUNCTION,BYTE,VIRTUAL !prototype corresponding virtual
    END
GlobalErrors MyErrorClass          !declare GlobalErrors object
CODE
GlobalErrors.Init                  !GlobalErrors initialization
!program code
GlobalErrors.Throw(Msg:CreateFailed) !Throw method calls SELF.TakeUser to
    !automatically call the derived class method
.
    !rather than the base class method
!program code

MyErrorClass.TakeUser FUNCTION     !derived class virtual to handle user errors
CODE
!your custom code here
IF MESSAGE(SELF.SubsString(),SELF.Errors.Title,ICON:Question, |
    Button:Yes+Button:No,BUTTON:Yes,0) = Button:Yes
    !your custom code here
    RETURN Level:Benign
ELSE
    !your custom code here
    RETURN Level:Cancel
END

```

See Also: TakeError, Throw, ThrowFile, ThrowMessage

**Throw (process specified error)****Throw**( *error id* ), PROC

---

<b>Throw</b>	Processes the specified error then returns its severity level.
<i>error id</i>	An integer constant, variable, EQUATE, or expression that indicates which error to process.

The **Throw** method processes the specified error by calling other ErrorClass methods, then returns its severity level.

Typically, Throw is the method your program calls when it encounters a known error. That is, as your program encounters errors or other notable conditions, it simply calls the Throw method or one of its variations (ThrowFile or ThrowMessage), passing it the appropriate *error id*. Throw then calls any other ErrorClass methods required to handle the specified error.

Implementation: The Throw method saves the error state (ERROR, ERRORCODE, FILEERROR, and FILEERRORCODE), locates the specified error, calls the appropriate method to handle the error according to its severity level, then returns the severity level.

The ABERROR.INC file contains a list of default symbolic constants for the *error id* parameter.

**Note:** The Throw method may or may not RETURN to your calling program, depending on the severity of the error.

Return Data Type: BYTE

Example:

```
!user level error occurred. ask user to confirm
Severity = GlobalErrors.Throw(Msg:ConfirmCancel) !handle the error condition
IF Severity = Level:Cancel
  LocalResponse = RequestCancelled
  DO ProcedureReturn
END
```

See Also: Errors, ThrowFile, ThrowMessage

## **ThrowFile (set value of %File, then process error)**

**ThrowFile**( *error id*, *filename* ), **PROC**

---

<b>ThrowFile</b>	Sets the substitution value of %File, then processes the error.
<i>error id</i>	An integer constant, variable, EQUATE, or expression that indicates which error to process.
<i>filename</i>	A string constant, variable, EQUATE, or expression that indicates which file produced the error.

The **ThrowFile** method sets the substitution value of %File, then processes the error, and finally returns the severity level of the error.

ThrowFile combines the functionality of SetFile and Throw into a single method.

Implementation: The ABERROR.INC file contains a list of default symbolic constants for the *error id* parameter. The value of the ErrorClass.FileName property is substituted for any %File symbols in the error message text.

**Note:** The **ThrowFile** method may or may not **RETURN** to your calling program, depending on the severity of the error.

Return Data Type: BYTE

Example:

```
OPEN(MyFile)
IF ERRORCODE( )
  Severity = GlobalErrors.ThrowFile(Msg:OpenFailed, NAME(MyFile))
END
```

See Also: FileName, SetFile, Throw

## **ThrowMessage (set value of %Message, then process error)**

**ThrowMessage**( *error id*, *messagetext*), PROC

---

### **ThrowMessage**

Sets the substitution value of the %Message macro, then processes the error.

*error id* An integer constant, variable, EQUATE, or expression that indicates which error to process.

*messagetext* A string constant, variable, EQUATE, or expression to replace any %Message symbols in the message text.

The **ThrowMessage** method sets the substitution value of the %Message macro, then processes the error, and finally returns the severity level of the error.

Implementation: The ABERROR.INC file contains a list of default symbolic constants for the *error id* parameter. The value of the ErrorClass.MessageText property is substituted for any %Message symbols in the error message text.

**Note:** The **ThrowMessage** method may or may not RETURN to your calling program, depending on the severity of the error.

Return Data Type: BYTE

Example:

```
OPEN(MyFile)
IF ERRORCODE( )
    Severity = GlobalErrors.ThrowMessage(Msg:OpenFailed, NAME(MyFile))
END
```

See Also: MessageText, Throw

## **ViewHistory (initiates the view of the current errors)**

### **ViewHistory**

The **ViewHistory** method initiates the viewing of the current error History structure. This structure may contain more than one error.

---

# ErrorLogInterface

## ErrorLogInterface Concepts

The ErrorLogInterface is used to manage the update of the StdErrorFile.

## Relationship to Other Application Builder Classes

The ErrorLogInterface is implemented by the StandardErrorLogClass and used as a reference by the ErrorClass and MsgBoxClass.

## ErrorLogInterface Source Files

The ErrorLogInterface source code is installed by default to the Clarion \LIBSRC folder. The specific ErrorLogInterface source code and their respective components are contained in:

ABERROR.INC	ErrorLogInterface declaration
ABERROR.CLW	StandardErrorLogClass.ErrorLogInterface method definitions

## ErrorLogInterface Methods

### ErrorLogInterface Methods

The ErrorLogInterface defines the following methods.

#### Close (initiate close of log file)

**Close**(*force*), PROC

<b>Close</b>	Initiate the close of the StdErrorFile.
<i>force</i>	An numeric constant, variable, EQUATE, or expression that indicates whether the log file must be closed or whether it should be conditionally closed. A value of one (1 or True) unconditionally closes the errorlog file; a value of zero (0 or False) only closes the errorlog file as circumstances require.

The **Close** method closes the ErrorLog file. Level:Benign is returned from this method if no errors occur. A Level:Fatal is returned if an error occurs.

Return Data Type:     BYTE

See Also:             StandardErrorLogClass.Close

## Open (method to initiate open of log file)

**Open**(*force*), PROC, PROTECTED

<b>Open</b>	Create and open the StdErrorFile.
<i>force</i>	An numeric constant, variable, EQUATE, or expression that indicates whether the log file must be opened or whether it should be conditionally opened. A value of one (1 or True) unconditionally opens the errorlog file; a value of zero (0 or False) only opens the errorlog file as circumstances require.

The **Open** method creates and opens the ErrorLog file. Level:Benign is returned from this method. A Level:Fatal is returned if an error occurs.

Return Data Type:     BYTE

See Also:             StandardErrorLogClass.Open

## Take (update the log file)

**Take**(*errtext*), PROC

<b>Take</b>	Update the StdErrorFile.
<i>errtext</i>	The complete line of text to add to the log file.

The Take method updates the StdErrorFile with the complete line of text, including the error code, error message, date, and time.

Return Data Type:    BYTE



# FieldPairsClass

## FieldPairsClass Overview

In database oriented programs there are some fundamental operations that occur over and over again. Among these repetitive operations is the saving and restoring of field values, and comparing current field values against previous values.

The ABC Library provides two classes (FieldPairsClass and BufferedPairsClass) that supply this basic buffer management. These classes are completely generic so that they may apply to any pairs of fields, regardless of the fields' origins.

**Tip:** **The fundamental benefit of these classes is their generality; that is, they let you *move* data between pairs of structures such as FILE or QUEUE buffers, and *compare* the data, without knowing in advance what the buffer structures look like or, for that matter, without requiring that the fields even reside in conventional buffer structures.**

In some ways the FieldPairsClass is similar to Clarion's deep assignment operator (:=: see the *Language Reference* for a description of this operator). However, the FieldPairsClass has the following advantages over deep assignment:

- Field pair labels need not be an exact match
- Field pairs are not limited to GROUPs, RECORDs, and QUEUEs
- Field pairs are not restricted to a single source and a single destination
- You can compare the sets of fields for equivalence
- You can mimic a data structure where no structure exists

The FieldPairsClass has the disadvantage of not handling arrays (because the FieldPairsClass relies on the ANY datatype which only accepts references to simple datatypes). See the *Language Reference* for more information on the ANY datatype.

## FieldPairsClass Concepts

The FieldPairsClass lets you move data between field pairs, and lets you compare the field pairs to detect whether any changes occurred since the last operation.

This class provides methods that let you identify or "set up" the targeted field pairs.

Once the field pairs are identified, you call a single method to move all the fields in one direction (left to right), and another method to move all the fields in the other direction (right to left). You simply have to remember which entity (set of fields) you described as "left" and which entity you described as "right." A third method compares the two sets of fields and returns a value to indicate whether or not they are equivalent.

**Note:** The paired fields need not be contiguous in memory, nor do they need to be part of a structure. You can build a virtual structure simply by adding a series of otherwise unrelated fields to a FieldPairs object. The other FieldPairs methods then operate on this virtual structure.

## FieldPairsClass Relationship to Other Application Builder Classes

The ViewManager and the BrowseClass use the FieldPairsClass and BufferedPairsClass to accomplish various tasks.

The BufferedPairsClass is derived from the FieldPairs class, so it provides all the functionality of the FieldPairsClass; however, this class also provides a third buffer area (a "save" area), plus the ability to compare the save area with the primary buffers, and the ability to restore data from the save area to the primary buffers (to implement a standard "cancel" operation).

## FieldPairsClass ABC Template Implementation

Various ABC Library objects instantiate the FieldPairsClass as needed; therefore, the template generated code does not directly reference the FieldPairsClass (or BufferedPairsClass).

## FieldPairsClass Source Files

The FieldPairsClass source code is installed by default in the Clarion \LIBSRC folder. The specific files and their respective components are:

ABUTIL.INC	FieldPairsClass declarations
ABUTIL.CLW	FieldPairsClass method definitions

## FieldPairsClass Conceptual Example

The FieldPairs classes are very abstract, so here is a concrete example to help your understanding. The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a FieldPairsClass object.

Let's assume you have a Customer file declared as:

```
Customer FILE, DRIVER('TOPSPEED'), PRE(CUST), CREATE, BINDABLE
ByNumber KEY(CUST:CustNo), NOCASE, OPT, PRIMARY
Record RECORD, PRE()
CustNo LONG
Name STRING(30)
Phone STRING(20)
Zip DECIMAL(5)
END
END
```

And you have a Customer queue declared as:

```
CustQ QUEUE
CustNo LONG
Name STRING(30)
Phone STRING(20)
Zip DECIMAL(5)
END
```

And you want to move data between the file buffer and the queue buffer.

```
INCLUDE('ABUTIL.INC')           !declare FieldPairs Class
Fields FieldPairsClass         !declare Fields object

CODE
Fields.Init                     !initialize FieldPairs object
Fields.AddPair(CUST:CustNo, CustQ.CustNo) !establish CustNo pair
Fields.AddPair(CUST:Name, CustQ.Name)    !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone)  !establish Phone pair
Fields.AddPair(CUST:Zip, CustQ.Zip)      !establish Zip pair

Fields.AssignLeftToRight       !copy from Customer FILE to CustQ QUEUE

IF Fields.Equal()              !compare the CustQ QUEUE and Customer FILE
  MESSAGE('Buffers are equal')
ELSE
  MESSAGE('Buffers not equal')
END
Fields.AssignRightToLeft       !copy from CustQ QUEUE to Customer FILE
Fields.Kill                     !terminate FieldPairs object
```

## FieldPairsClass Properties

The FieldPairsClass contains the following properties.

### List (recognized field pairs)

#### **List      &FieldPairsQueue**

The **List** property is a reference to the structure that holds all the field pairs recognized by the FieldPairsClass object. Use the AddPair or AddItem methods to add field pairs to the List property. For each field pair, the List property includes a "Left" field and a "Right" field.

The "Left" and "Right" designations are reflected in other method names (for example, field assignments methods--AssignLeftToRight and AssignRightToLeft) so you can easily and accurately control the movement of data between the two sets of fields.

Implementation:      List is a reference to a QUEUE declared in ABUTIL.INC as follows:

```
FieldPairsQueue QUEUE,TYPE
Left                    ANY
Right                   ANY
                          END
```

The Init method creates an empty List, and the Kill method disposes of the List. AddPair and AddItem add field pairs to the List.

See Also:              AddPair, AddItem, Init

## FieldPairsClass Methods

### FieldPairsClass Functional Organization--Expected Use

As an aid to understanding the FieldPairsClass, it is useful to organize its various methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the FieldPairsClass methods.

#### Non-Virtual Methods

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the FieldPairsClass object
AddItem	add a field pair based on one source field
Kill	terminate the FieldPairsClass object

##### **Mainstream Use:**

AssignLeftToRight	assign each "left" field to its "right" counterpart
AssignRightToLeft	assign each "right" field to its "left" counterpart
Equal	return 1 if all pairs are equal, 0 if any pair is not equal

##### **Occasional Use:**

ClearLeft	CLEAR each "left" field
ClearRight	CLEAR each "right" field
EqualLeftRight	return 1 if all pairs are equal, 0 if any pair is not equal

#### Virtual Methods

---

Typically you will not call these methods directly. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

AddPair add a field pair to the List property

**AddItem (add a field pair from one source field)****AddItem**( *left* )**AddItem** Adds a field pair to the List property from one source field.*left* The address of the "left" field of the pair. The field may be any data type, but may not be an array.The **AddItem** method adds a field pair to the List property from one source field. The "right" field is supplied for you, and initially contains a copy of the data in the "left" field.

The fields need not be contiguous in memory, nor do they need to be part of a structure. Therefore you can build a virtual structure simply by adding a series of otherwise unrelated fields to a FieldPairs object. The other FieldPairs methods then operate on this virtual structure.

Implementation: AddItem assumes the List property has already been created by Init or by some other method.

By calling AddItem for a series of fields, you effectively build two virtual structures containing the fields--the "Left" is the original fields and the "Right" contains a copy of the data in the original fields at the time you call AddItem.

Example:

```

INCLUDE('ABUTIL.INC')           !declare FieldPairs Class
DKeyPair FieldPairsClass       !declare FieldPairs reference
Org FILE                       !declare a file
DptKey KEY(Dept,Grade)        !declare a multicomponent key
    RECORD
Dept SHORT
Mgr SHORT
Grade SHORT
    END
    END

CODE
DKeyPair.Init                 !initialize FieldPairs object
DKeyPair.AddItem(Org:Dept)    !add Dept (left) and a copy of Dept (right)
DKeyPair.AddItem(Org:Grade)  !add Grade (left) and a copy of Grade (right)
!some code
DKeyPair.AssignLeftToRight    !Save the current key fields' values
SET(Org:DptKey,Org:DptKey)    !position the file
NEXT(Org)                    !retrieve (hopefully) a specific record
IF ERRORCODE() OR |          !confirm retrieval of matching record by
    ~DKeyPair.Equal()        !comparing retrieved key values with saved values
    MESSAGE('Record not found!')
END

```

See Also: Init, List

**AddPair (add a field pair:FieldPairsClass)****AddPair**( *left*, *right* ), **VIRTUAL**


---

<b>AddPair</b>	Adds a field pair to the List property.
<i>left</i>	The label of the "left" field of the pair. The field may be any data type, but may not be an array.
<i>right</i>	The label of the "right" field of the pair. The field may be any data type, but may not be an array.

The **AddPair** method adds a field pair to the List property. The fields need not be contiguous in memory, nor do they need to be part of a structure. Therefore you can build a virtual structure simply by adding a series of otherwise unrelated fields to a FieldPairs object. The other FieldPairs methods then operate on this virtual structure.

Implementation: AddPair assumes the List property has already been created by Init or by some other method.

By calling AddPair for a series of fields (for example, the corresponding fields in a RECORD structure and a QUEUE structure), you effectively build two virtual structures containing the fields and a (one-to-one) relationship between the two structures.

Example:

```

INCLUDE('ABUTIL.INC')                !declare FieldPairs Class
Fields FieldPairsClass                !declare FieldPairs object
Customer FILE,DRIVER('TOPSPEED'),PRE(CUST)
ByNumber KEY(CUST:CustNo),NOCASE,OPT,PRIMARY
Record RECORD,PRE()
CustNo LONG
Name STRING(30)
Phone STRING(20)
ZIP DECIMAL(5)
END
END

CustQ QUEUE
CustNo LONG
Name STRING(30)
Phone STRING(20)
ZIP DECIMAL(5)
END

```

## CODE

```
Fields.Init                !initialize FieldPairs object
Fields.AddPair(CUST:CustNo, CustQ.CustNo) !establish CustNo pair
Fields.AddPair(CUST:Name,  CustQ.Name)    !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone)   !establish Phone pair
Fields.AddPair(CUST:ZIP,   CustQ.ZIP)     !establish ZIP pair
```

See Also:        Init, List

## AssignLeftToRight (copy from "left" fields to "right" fields)

### AssignLeftToRight

The **AssignLeftToRight** method copies the contents of each "left" field to its corresponding "right" field in the List property.

Implementation: For AddPair pairs, the "left" field is the *first* (left) parameter of the AddPair method; the "right" field is the *second* (right) parameter of the AddPair method. For AddItem pairs, the "left" field is the *only* parameter of the AddItem method. The "right" field is the FieldPairs supplied copy of the "left" field.

Example:

```
Fields.AddPair(CUST:Name, CustQ.Name) !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone)!establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP) !establish ZIP pair
!some code
IF ~Fields.Equal !compare field pairs
CASE MESSAGE('Abandon Changes?',,,BUTTON:Yes+BUTTON:No)
OF BUTTON:No
Fields.AssignRightToLeft !copy changes to CUST (write) buffer
OF BUTTON:Yes
Fields.AssignLeftToRight !restore original to CustQ (display) buffer
END
END
```

See Also: AddPair, AddItem, List

## AssignRightToLeft (copy from "right" fields to "left" fields)

### AssignRightToLeft

The **AssignRightToLeft** method copies the contents of each "right" field to its corresponding "left" field in the List property.

Implementation: For AddPair pairs, the "left" field is the *first* (left) parameter of the AddPair method; the "right" field is the *second* (right) parameter of the AddPair method. For AddItem pairs, the "left" field is the *only* parameter of the AddItem method. The "right" field is the FieldPairs supplied copy of the "left" field.

Example:

```
Fields.AddPair(CUST:Name, CustQ.Name) !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone) !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP) !establish ZIP pair
!some code
IF ~Fields.Equal !compare field pairs
CASE MESSAGE('Abandon Changes?',,,BUTTON:Yes+BUTTON:No)
OF BUTTON:No
Fields.AssignRightToLeft !copy changes to CUST (write) buffer
OF BUTTON:Yes
Fields.AssignLeftToRight !restore original to CustQ (display) buffer
END
END
```

See Also:       AddPair  
                  AddItem

## ClearLeft (clear each "left" field)

### ClearLeft

The **ClearLeft** method clears the contents of each "left" field in the List property.

Implementation: For AddPair pairs, the "left" field is the field whose label is the *first* (left) parameter of the AddPair method; the "right" field is the field whose label is the *second* (right) parameter of the AddPair method. For AddItem pairs, the "left" field is the field whose label is the *only* parameter of the AddItem method. The "right" field is the FieldPairs supplied copy of the "left" field.

The ClearLeft method CLEARS the field. See the *Language Reference* for more information on CLEAR.

Example:

```
Fields &= NEW FieldPairsClass           !instantiate FieldPairs object
Fields.Init                             !initialize FieldPairs object
Fields.AddPair(CUST:CustNo, CustQ.CustNo) !establish CustNo pair
Fields.AddPair(CUST:Name, CustQ.Name)    !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone)  !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP)      !establish ZIP pair
!some code
IF LocalRequest = InsertRecord
  Fields.ClearRight                      !clear the CustQ fields to blank or zero
END
```

See Also:        AddPair

                 AddItem

**ClearRight (clear each "right" field)****ClearRight**

The **ClearRight** method clears the contents of each "right" field in the List property.

Implementation: For AddPair pairs, the "left" field is the field whose label is the *first* (left) parameter of the AddPair method; the "right" field is the field whose label is the *second* (right) parameter of the AddPair method. For AddItem pairs, the "left" field is the field whose label is the *only* parameter of the AddItem method. The "right" field is the FieldPairs supplied copy of the "left" field.

The **ClearRight** method **CLEARs** the field. See the *Language Reference* for more information on CLEAR.

Example:

```
Fields &= NEW FieldPairsClass           !instantiate FieldPairs object
Fields.Init                             !initialize FieldPairs object
Fields.AddPair(CUST:CustNo, CustQ.CustNo) !establish CustNo pair
Fields.AddPair(CUST:Name, CustQ.Name)    !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone)  !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP)      !establish ZIP pair
!some code
IF LocalRequest = InsertRecord
  Fields.ClearRight                     !clear the CustQ fields to blank or zero
END
```

See Also:       AddPair  
                  AddItem

## Equal (return 1 if all pairs are equal)

### Equal

The **Equal** method returns one (1) if all pairs are equal and returns zero (0) if any pairs are not equal.

Implementation: The Equal method simply calls the EqualLeftRight method which does all the comparison work. Therefore, there are two different methods (Equal and EqualLeftRight) that produce exactly the same result.

This provides an alternative calling convention for the FieldPairsClass and the BufferedPairsClass. The EqualLeftRight method name is consistent with the other comparison methods in the BufferedPairsClass and is provided for that purpose. See *BufferedPairsClass Methods* for more information.

Example:

```
Fields.AddPair(CUST:Name, CustQ.Name) !establish Name pair
Fields.AddPair(CUST:Phone, CustQ.Phone) !establish Phone pair
Fields.AddPair(CUST:ZIP, CustQ.ZIP) !establish ZIP pair
!some code
IF ~Fields.Equal !compare field pairs
CASE MESSAGE('Abandon Changes?',,,BUTTON:Yes+BUTTON:No)
OF BUTTON:No
Fields.AssignRightToLeft !copy changes to CUST (write) buffer
OF BUTTON:Yes
Fields.AssignLeftToRight !restore original to CustQ (display) buffer
END
END
```

See Also: EqualLeftRight

**EqualLeftRight (return 1 if all pairs are equal)****EqualLeftRight**

The **EqualLeftRight** method returns one (1) if all pairs are equal and returns zero (0) if any pairs are not equal.

Implementation: The Equal method simply calls the EqualLeftRight method which does all the comparison work. Therefore, there are two different methods (Equal and EqualLeftRight) that produce exactly the same result.

This provides an alternative calling convention for the FieldPairsClass and the BufferedPairsClass. The EqualLeftRight method name is consistent and compatible with the other comparison methods in the BufferedPairsClass and is provided for that purpose. See *BufferedPairsClass Methods* for more information.

See Also: Equal

**Init (initialize the FieldPairsClass object)****Init**

The **Init** method initializes the FieldPairsClass object.

Implementation: The Init method creates the List property.

Example:

```

INCLUDE('ABUTIL.INC')      !declare FieldPairs Class
Fields &FieldPairsClass   !declare FieldPairs reference

CODE
Fields &= NEW FieldPairsClass !instantiate FieldPairs object
Fields.Init                !initialize FieldPairs object
.
.
.
Fields.Kill                !terminate FieldPairs object
DISPOSE(Fields)           !release memory allocated for FieldPairs object

```

See Also: Kill, List

## Kill (shut down the FieldPairsClass object)

### Kill

The **Kill** method disposes any memory allocated during the object's lifetime and performs any other necessary termination code.

Implementation: The Kill method disposes the List property created by the Init method.

Example:

```
INCLUDE('ABUTIL.INC')    !declare FieldPairs Class
Fields &FieldPairsClass !declare FieldPairs reference

CODE
Fields &= NEW FieldPairsClass !instantiate FieldPairs object
Fields.Init                !initialize FieldPairs object
.
.
.
Fields.Kill                !terminate FieldPairs object
DISPOSE(Fields)           !release memory allocated for FieldPairs object
```

See Also: Init, List



# FileDropComboClass

## Overview:FileDropComboClass

The FileDropComboClass is a FileDropClass based on a COMBO control rather than a LIST control. Therefore it supports not only the selection of existing list items but also the *selection of values not in the list*, and optionally the *addition of new values to the list*. See *Control Templates--FileDropCombo* for information on the template implementation of the FileDropCombo control.

## Future File DropCombo Classes

The current implementation of the FileDropComboClass is a place-holder implementation. In the future the FileDropComboClass, or its replacement, will be derived from the BrowseClass.

## FileDropComboClass Concepts

Based on the end user selection, you can assign one or more values from the selected item to one or more target fields. You may display one field (e.g., a description field) but assign another field (e.g., a code field) from the selected list item.

The FileDropClass also supports filters, range limits, colors, icons, sorting, and multiple item selection (marking). See *Control Templates--FileDropCombo* for information on the template implementation of these features.

## FileDropComboClass Relationship to Other Application Builder Classes

The FileDropComboClass is closely integrated with the WindowManager. These objects register their presence with each other, set each other's properties, and call each other's methods as needed to accomplish their respective tasks.

The FileDropComboClass is derived from the FileDropClass, plus it relies on several of the other Application Builder Classes to accomplish its tasks. Therefore, if your program instantiates the FileDropClass, it must also instantiate these other classes. Much of this is automatic when you INCLUDE the FileDropClass header (ABDROPS.INC) in your program's data section. See the *Conceptual Example*.

## FileDropComboClass ABC Template Implementation

The ABC Templates automatically include all the classes and generate all the code necessary to support the functionality specified in your application's FileDropCombo control templates.

The templates *derive* a class from the FileDropComboClass and instantiate an object for *each* FileDropComboControl template in the application. The derived class and object is called FDCB# where # is the FileDropCombo Control template instance number. The templates provide the derived class so you can use the FileDropComboControl template **Classes** tab to modify the FileDropCombo's behavior on an instance-by-instance basis.

The derived FileDropComboClass is local to the procedure, is specific to a single FileDropCombo and relies on the global ErrorClass object and the file-specific RelationManager and FileManager objects for the displayed lookup file.

## FileDropComboClass Source Files

The FileDropComboClass source code is installed by default to the Clarion \LIBSRC folder. The FileDropComboClass source code and their respective components are contained in:

ABDROPS.INC	FileDropComboClass declarations
ABDROPS.CLW	FileDropComboClass method definitions

## FileDropComboClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a FileDropComboClass object and related objects.

This example uses the FileDropComboClass object to let the end user select or enter a valid state code for a given client. The state code comes from the state file.

```

PROGRAM
  INCLUDE( 'ABWINDOW.INC' )
  INCLUDE( 'ABDROPS.INC' )
  MAP
  END

State      FILE, DRIVER( 'TOPSPEED' ), PRE( ST ), THREAD
StateCodeKey KEY( ST:STATECODE ), NOCASE, OPT
Record     RECORD, PRE( )
StateCode  STRING( 2 )
StateName  STRING( 20 )
          END
          END

```

```

Customer  FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,THREAD
BYNUMBER  KEY(CUS:CUSTNO),NOCASE,OPT,PRIMARY
Record    RECORD,PRE()
CUSTNO    LONG
Name      STRING(30)
State     STRING(2)
          END
          END

GlobalErrors  ErrorClass
VCRRequest   LONG(0),THREAD

Access:State  CLASS(FileManager)
Init         PROCEDURE
            END

Relate:State  CLASS(RelationManager)
Init         PROCEDURE
            END

Access:Customer CLASS(FileManager)
Init         PROCEDURE
            END

Relate:Customer CLASS(RelationManager)
Init         PROCEDURE
            END

StateQ       QUEUE
ST:STATECODE LIKE(ST:STATECODE)
ViewPosition STRING(512)
            END
StateView VIEW(State)
            END

CusWindow WINDOW('Add Customer'),AT(,,157,58),IMM,SYSTEM,GRAY
          PROMPT('Customer:'),AT(5,7),USE(?NamePrompt)
          ENTRY(@s20),AT(61,5,88,11),USE(CUS:NAME)
          PROMPT('State:'),AT(5,22),USE(?StatePrompt)
          LIST,AT(61,20,65,11),USE(CUS:State),FROM(StateQ),|
          FORMAT('8L~STATECODE~@s2@'),DROP(5)
          BUTTON('OK'),AT(60,39),USE(?OK),DEFAULT
          BUTTON('Cancel'),AT(104,39),USE(?Cancel)
          END

```

```

ThisWindow CLASS(WindowManager)
Init      PROCEDURE(),BYTE,PROC,VIRTUAL
Kill      PROCEDURE(),BYTE,PROC,VIRTUAL
          END

StateDrop CLASS(FileDropClass)
Q         &StateQ
          END

CODE
ThisWindow.Run()

ThisWindow.Init PROCEDURE()
ReturnValue      BYTE,AUTO
CODE
GlobalErrors.Init
Relate:State.Init
Relate:Customer.Init
SELF.Request = InsertRecord
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?CUS:NAME
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddUpdateFile(Access:Customer)
SELF.AddItem(?Cancel,RequestCancelled)
SELF.OkControl = ?OK
Relate:Customer.Open
Relate:State.Open
SELF.Primary &= Relate:Customer
SELF.InsertAction = Insert:Batch
IF SELF.PrimeUpdate() THEN RETURN Level:Notify.
OPEN(CusWindow)
SELF.Opened=True
!initialize the FileDropCombo Class with:
! the combo's USE variable, COMBO control, view POSITION, VIEW, combo's FROM QUEUE,
! primary file RelationManager object, WindowManager object, ErrorClass object,
! add records flag, hot fields flag, case sensitive flag
StateDrop.Init(?CUS:State,StateQ.ViewPosition,|
StateView,StateQ,Relate:State,ThisWindow,GlobalErrors,1,0,0)
StateDrop.Q &= StateQ
StateDrop.AddSortOrder()
StateDrop.AddField(ST:STATECODE,StateDrop.Q.ST:STATECODE)
StateDrop.AddUpdateField(ST:STATECODE,CUS:State)
ThisWindow.AddItem(StateDrop)
SELF.SetAlerts()
RETURN ReturnValue

```

```
ThisWindow.Kill PROCEDURE()  
ReturnValu     BYTE,AUTO  
  CODE  
  ReturnValu = PARENT.Kill()  
  IF ReturnValu THEN RETURN ReturnValu.  
  Relate:Customer.Close  
  Relate:State.Close  
  Relate:State.Kill  
  Relate:Customer.Kill  
  GlobalErrors.Kill  
  RETURN ReturnValu  
  
Access:State.Init PROCEDURE  
  CODE  
  PARENT.Init(State,GlobalErrors)  
  SELF.FileNameValue = 'State'  
  SELF.Buffer &= ST:Record  
  SELF.LazyOpen = False  
  SELF.AddKey(ST:StateCodeKey,'ST:StateCodeKey',0)  
  
Access:Customer.Init PROCEDURE  
  CODE  
  PARENT.Init(Customer,GlobalErrors)  
  SELF.FileNameValue = 'Customer'  
  SELF.Buffer &= CUS:Record  
  SELF.Create = True  
  SELF.LazyOpen = False  
  SELF.AddKey(CUS:BYNUMBER,'CUS:BYNUMBER',0)  
  
Relate:State.Init PROCEDURE  
  CODE  
  Access:State.Init  
  PARENT.Init(Access:State,1)  
  
Relate:Customer.Init PROCEDURE  
  CODE  
  Access:Customer.Init  
  PARENT.Init(Access:Customer,1)
```

## FileDropComboClass Properties

### FileDropComboClass Properties

The FileDropComboClass inherits all the properties of the FileDropClass from which it is derived. See *FileDropClass Properties* and *ViewManager Properties* for more information.

### AskProcedure (update procedure)

#### **AskProcedure USHORT**

The **AskProcedure** property is used to determine which Update Procedure to call if the FileDropCombo control allows updates.

Implementation: The FileDropCombo control template allows the filedrop queue to be updated if the entry does not exist in the queue. This property identifies the procedure to call to update the queue. The template generates the code that initiates the value of this property.

See Also: FileDropComboClass.Ask

### ECON (current state of entry completion)

#### **ECON BYTE, PROTECTED**

The **ECON** property indicates the current state of EntryCompletion. A value of one (1 or True) indicates the current state of entry completion is on; a value of zero (0 or False) indicates it is off.

See Also: EntryCompletion

## **EntryCompletion (automatic fill-ahead flag)**

**EntryCompletion**      **BYTE**

The **EntryCompletion** property indicates whether FileDropComboClass tries to automatically complete the end user selection. A value of one (1) or True enables the automatic completion; a value of zero (0) or False disables automatic completion.

When EntryCompletion is enabled, the FileDropComboClass object displays the list item that is nearest the value entered by the end user. The FileDropComboClass object reevaluates the display immediately after each end user keystroke.

Implementation:      The Init method sets the EntryCompletion property to True. The TakeEvent and TakeNewSelection methods implement the behavior specified by EntryCompletion.

See Also:              Init, TakeEvent, TakeNewSelection

## **RemoveDuplicatesFlag (remove duplicate data)**

**RemoveDuplicatesFlag**      **BYTE**

The **RemoveDuplicatesFlag** property is a flag used to determine if duplicates are not allowed in the filedrop queue. A value of one (1 or True) does not allow duplicates in the filedrop queue; a value of zero (0 or False) allows duplicates into the filedrop queue.

Implementation:      The RemoveDuplicatesFlag is set by the ABC templates when the FileDropCombo control template is used. The 'Remove Duplicates' checkbox controls this setting.

See Also:              FileDropComboClass.Init, FileDropComboClass.AddRecord

**UseField (COMBO USE variable)**

**UseField**      **ANY, PROTECTED**

The **UseField** property is a reference to the COMBO's USE variable. The FileDropComboClass uses this property to lookup the USE value in the current queue.

Implementation:    The Init method initializes the UseField property.

See Also:          Init

## FileDropComboClass Methods

### FileDropComboClass Methods

The FileDropComboClass inherits all the methods of the FileDropClass from which it is derived. See *FileDropClass Methods* and *ViewManager Methods* for more information.

### FileDropComboClass Functional Organization--Expected Use

As an aid to understanding the FileDropComboClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the FileDropComboClass methods.

#### Non-Virtual Methods

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the FileDropComboClass object
AddField <sub>i</sub>	specify display fields
AddUpdateField <sub>i</sub>	specify field assignments
AddRange <sub>ii</sub>	add a range limit to the active sort order
AppendOrder <sub>ii</sub>	refine the active sort order
Kill <sub>ii</sub>	shut down the FileDropComboClass object

##### **Mainstream Use:**

ResetQueue	refresh filedrop queue
GetQueueMatch	locate a list item
Ask <sub>v</sub>	add a record to the lookup file
TakeEvent <sub>v</sub>	process the current ACCEPT loop event
TakeNewSelection <sub>v</sub>	process the EVENT:Selected events

##### **Occasional Use:**

Open <sub>ii</sub>	open the filedrop view
PrimeRecord <sub>ii</sub>	prepare an item for adding
SetFilter <sub>ii</sub>	specify a filter for the active sort order
ApplyFilter <sub>ii</sub>	range limit and filter the result set
ApplyOrder <sub>ii</sub>	sort the result set
GetFreeElementName <sub>ii</sub>	return the free element field name
SetOrder <sub>ii</sub>	replace the active sort order
Close <sub>ii</sub>	close the filedrop view

<sub>i</sub> These methods are inherited from the FileDropClass.

<sub>ii</sub> These methods are inherited from the ViewManager.

---

## Virtual Methods

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Ask	add a record to the lookup file
SetQueueRecord <sup>i</sup>	copy data from file buffer to queue buffer
Reset <sup>ii</sup>	reset the view position
TakeEvent	process the current ACCEPT loop event
TakeNewSelection	process the EVENT:Selected events
ValidateRecord <sup>i</sup>	validate the current result set element

<sup>i</sup> These methods are inherited from the FileDropClass.

<sup>ii</sup> These methods are inherited from the ViewManager.

**AddRecord (add a record filedrop queue)****AddRecord, VIRTUAL, PROTECTED**

The **AddRecord** method adds a record to the filedropcombo's queue.

**Ask (add a record to the lookup file)****Ask, VIRTUAL, PROTECTED**

The **Ask** method adds a new record to the filedrop's lookup file and returns a value indicating its success or failure. If it succeeds it returns Level:Benign, otherwise it returns the severity level of the last error it encountered while trying to add the record. See *Error Class* for more information on severity levels.

Implementation: The TakeEvent method calls the Ask method. Return value EQUATEs are declared in ABERROR.INC (see *Error Class* for more information):

Level:Benign	EQUATE(0)
Level:User	EQUATE(1)
Level:Program	EQUATE(2)
Level:Fatal	EQUATE(3)
Level:Cancel	EQUATE(4)
Level:Notify	EQUATE(5)

Return Data Type: BYTE

Example:

```
MyFileDropComboClass.TakeEvent PROCEDURE
UserStr    CSTRING(256),AUTO
    CODE
!procedure code
IF SELF.Ask() = Level:Benign    !update lookup file
    SELF.UpdateFields.AssignLeftToRight
    SELF.Close
    SELF.ResetQueue
    SELF.ListField{PROP:Selected} = SELF.GetQueueMatch(UserStr)
    DISPLAY(SELF.ListField)
END
!procedure code
```

SeeAlso:

TakeEvent

**GetQueueMatch (locate a list item)****GetQueueMatch**( *search value* ), **PROTECTED****GetQueueMatch**      Locates the *search value* within the first field of the display queue.*search value*      A string constant, variable, EQUATE, or expression containing the value to locate.

The **GetQueueMatch** method locates a value within the first field of the display queue and returns the position of the matching item. A return value of zero (0) indicates no matching items.

The Init method *case* parameter determines the type of search (case sensitive or insensitive) performed.

Return Data Type:    LONG

Example:

```

MyFileDropComboClass.TakeEvent PROCEDURE
UserStr      CSTRING(256),AUTO
CODE
CASE EVENT()
OF EVENT:Accepted
UserStr=CLIP(SELF.UseField)
IF SELF.GetQueueMatch(UserStr) = 0      !if entered value not in
SELF.Reset                              ! lookup file / queue
IF SELF.Ask()=Level:Benign              !update the lookup file
SELF.UpdateFields.AssignLeftToRight
SELF.Close
SELF.ResetQueue
SELF.ListField{PROP:Selected}=SELF.GetQueueMatch(UserStr)!position to new item
DISPLAY(SELF.ListField)
END
!procedure code

```

See Also:            Init

**Init (initialize the FileDropComboClass object)**

**Init**( *use*, *combo*, *position*, *view*, *queue*, *relationmgr*, *windowmgr*, *errormgr* [,*add*] [,*sync*] [,*case*] )

---

<b>Init</b>	Initializes the FileDropCombClass object.
<i>use</i>	The label of the <i>combo</i> 's USE attribute variable.
<i>combo</i>	A numeric constant, variable, EQUATE, or expression containing the control number of the filedrop's COMBO control.
<i>position</i>	The label of a string variable within the <i>queue</i> containing the POSITION of the <i>view</i> .
<i>view</i>	The label of the filedrop's underlying VIEW.
<i>queue</i>	The label of the <i>combo</i> 's data source QUEUE.
<i>relationmgr</i>	The label of the filedrop's primary file RelationManager object. See <i>Relation Manager</i> for more information.
<i>windowmgr</i>	The label of the filedrop's WindowManager object. See <i>Window Manager</i> for more information.
<i>errormgr</i>	The label of the filedrop's ErrorClass object. See <i>Error Management</i> for more information.
<i>add</i>	A numeric constant, variable, EQUATE, or expression indicating whether records may be added to the lookup file. A value of zero (0 or False) prevents adds; a value of one (1 or True) allows adds. If omitted, <i>add</i> defaults to one (1).
<i>sync</i>	A numeric constant, variable, EQUATE, or expression indicating whether to reget the underlying data on a new selection (allows hot fields). A value of one (1 or True) regets the data (so it can be displayed in other controls besides the COMBO control); a value of zero (0 or False) does not. If omitted, <i>sync</i> defaults to one (1).
<i>case</i>	A numeric constant, variable, EQUATE, or expression indicating whether filedrop searches are case sensitive. A value of one (1 or True) provides case sensitive searches; a value of zero (0 or False) gives case insensitive searches. If omitted, <i>case</i> defaults to zero (0).

The **Init** method initializes the FileDropComboClass object.

Implementation: Among other things, the Init method calls the PARENT.Init (FileDropClass.Init) method. See *FileDropClass* for more information.

Example:

**ThisWindow.Init PROCEDURE**

```

CODE
!procedure code                                !init filedropcombo object
  FDBC4.Init( CLI:StateCode,                    |           ! USE variable
             ?CLI:StateCode,                   |           ! COMBO control
             Queue:FileDropCombo.ViewPosition, | ! VIEW POSITION variable
             FDCB4::View:FileDropCombo,       | ! VIEW
             Queue:FileDropCombo,             | ! QUEUE
             Relate:States,                    | ! RelationManager object
             ThisWindow,                       | ! WindowManager object
             GlobalErrors,                     | ! ErrorClass object
             1,                                | ! allow adds
             0,                                | ! refresh hot fields on new selection
             0)                                | ! case insensitive searches

  FDBC4.Q &= Queue:FileDropCombo
  FDBC4.AddSortOrder()
  FDBC4.AddField(ST:StateCode,FDBC4.Q.ST:StateCode)
  FDBC4.AddField(ST:State,FDBC4.Q.ST:State)
  FDBC4.AddUpdateField(ST:StateCode,CLI:StateCode)

```

See Also:           FileDropClass.Init

## **KeyValid (check for valid keystroke)**

### **KeyValid, VIRTUAL**

The **KeyValid** method determines if a valid keystroke is in the keyboard buffer. If LeftKey (cursor left), RightKey (cursor right), ShiftLeft (Shift-cursor left), ShiftRight (Shift-cursor right) are in the buffer a False value is returned from this method. A True value is returned if any other keystroke is pressed.

ReturnDataType: BYTE

See Also:

FileDropComboClass.TakeNewSelection

## **Kill (shut down the FileDropComboClass object)**

### **Kill, VIRTUAL**

The **Kill** method releases any memory allocated during the life of the FileDropComboClass object and performs any other required termination code.

Implementation: Among other things, the Kill method calls the PARENT.Kill (FileDropClass.Kill) method to shut down the FileDropClass object.

## **ResetFromList (reset VIEW)**

### **ResetFromList, PROTECTED**

The **ResetFromList** method resets the VIEW based on the current record in the FileDropComboClass's object.

Implementation: The ResetFromList method is called by the FileDropComboClass.TakeAccepted method and the FileDropComboClass.TakeNewSelection method.

See Also:

FileDropComboClass.TakeAccepted, FileDropComboClass.TakeNewSelection

**ResetQueue (refill the filedrop queue)****ResetQueue**( [ *force* ] ), VIRTUAL, PROC**ResetQueue** Refills the filedrop queue and the COMBO's USE variable.

*force* A numeric constant, variable, EQUATE, or expression that indicates whether to refill the queue even if the sort order did not change. A value of one (1 or True) unconditionally refills the queue; a value of zero (0 or False) only refills the queue if circumstances require it. If omitted, *force* defaults to zero.

The **ResetQueue** method refills the filedrop's display queue and the COMBO's USE variable, applying the applicable sort order, range limits, and filters, then returns a value indicating which item, if any, in the displayed lookup file already matches the *target* fields' values specified by the AddUpdateField method. A return value of zero (0) indicates no matching items; any other value indicates the position of the matching item.

For example, if the filedrop "looks up" the state code for a customer, and the current customer's state code field already contains a valid value, then the ResetQueue method positions the filedrop list to the current customer's state code value.

Implementation: The TakeEvent method calls the ResetQueue method. The ResetQueue calls the PARENT.ResetQueue method, then enables or disables the drop button depending on the presence or absence of pick list items.

Return Data Type: LONG

Example:

```
MyFileDropComboClass.TakeEvent PROCEDURE
UserStr    CSTRING(256),AUTO
CODE
CASE EVENT()
OF EVENT:Accepted
UserStr=CLIP(SELF.UseField)
IF SELF.GetQueueMatch(UserStr) = 0      !if entered value not in
SELF.Reset                             ! lookup file / queue
IF SELF.Ask()=Level:Benign             !update the lookup file
SELF.UpdateFields.AssignLeftToRight
SELF.Close
SELF.ResetQueue(1)                     !refill the updated queue
SELF.ListField{PROP:Selected}=SELF.GetQueueMatch(UserStr)!position to new item
DISPLAY(SELF.ListField)
END
!procedure code
```

See Also: TakeEvent, FileDropClass.ResetQueue

## TakeAccepted (process accepted event)

### TakeAccepted, VIRTUAL

The TakeAccepted method processes the accepted event of the entry portion of the FileDropCombo control.

## TakeEvent (process the current ACCEPT loop event:FileDropComboClass)

### TakeEvent, VIRTUAL

The **TakeEvent** method processes the current ACCEPT loop event for the FileDropComboClass object.

Implementation: The WindowManager.TakeEvent method calls the TakeEvent method. On a new item selection, the TakeEvent method calls the TakeNewSelection method.

On EVENT:Accepted for the entry portion of the COMBO, the TakeEvent method calls the GetQueueMatch method to locate the list item nearest to the entered value. If the entered value is not in the lookup file, the TakeEvent method calls the Ask method to add the new value to the lookup file. If the add is successful, TakeEvent calls the ResetQueue method to refill the display queue.

Example:

```
MyWindowManager.TakeEvent PROCEDURE
Rval BYTE(Level:Benign)
I USHORT,AUTO
CODE
!procedure code
LOOP I = 1 TO RECORDS(SELF.Browses)
GET(SELF.Browses,I)
SELF.Browses.Browse.TakeEvent
END
LOOP i=1 TO RECORDS(SELF.FileDrops)
GET(SELF.FileDrops,i)
ASSERT(~ERRORCODE())
SELF.FileDrops.FileDrop.TakeEvent
END
RETURN RVal
```

See Also: Ask, GetQueueMatch, ResetQueue, TakeNewSelection, WindowManager.TakeEvent

## TakeNewSelection (process NewSelection events:FileDropComboClass)

**TakeNewSelection**( *field* ), VIRTUAL

---

**TakeNewSelection** Processes the EVENT:NewSelection event.

*field* A numeric constant, variable, EQUATE, or expression containing the control number of the control that generated the EVENT:NewSelection event.

The **TakeNewSelection** method processes the EVENT:NewSelection event for the FileDropComboClass object.

Implementation: The ResetQueue method and the TakeEvent method call the TakeNewSelection method. If the FileDropComboClass object's LIST generated the new selection event, then the TakeNewSelection method does the field assignments specified by the AddUpdateField method or clears the target fields if there is no valid selection.

Example:

**FileDropComboClass.TakeEvent PROCEDURE**

```
CODE
CASE EVENT( )
OF EVENT:NewSelection
  SELF.TakeNewSelection(FIELD( ))
  SELF.WindowManager.Reset
END
```

See Also: AddUpdateField, ResetQueue, TakeEvent

## UniquePosition (check queue for duplicate record by key position)

### UniquePosition, Protected

The **UniquePosition** method checks the FileDropComboClass's queue for a duplicate record by checking for duplicate key values. A return value of zero (0 or False) means there was a matching value already in the queue. Any other return value indicates no match was found.

Implementation: The UniquePosition method is called by the FileDropComboClass.AddRecord method which checks for duplicate values before adding the record to the filedropcombo's queue.

Return Data Type: LONG

See Also: FileDropComboClass.AddRecord



# FileDropClass

## FileDropClass Overview

### Future FileDropClasses

The current implementation of the FileDropClass is a place-holder implementation. In the future the FileDropClass, or its replacement, will be derived from the BrowseClass.

### FileDropClass Concepts

The FileDropClass is a ViewManager that supports a file-loaded scrollable list on a window. By convention, a FileDrop provides a "pick list" for the end user. A pick list is a finite list of mutually exclusive or alternative choices--the end user may choose only one of several items, but need not memorize the choices, because all the choices are displayed.

Based on the end user selection, you can assign one or more values from the selected item to one or more target fields. You may display one field (e.g., a description field) but assign another field (e.g., a code field) from the selected list item.

The FileDropClass also supports filters, range limits, colors, icons, sorting, and multiple item selection (marking). See Control Templates--FileDrop for information on the template implementation of these features.

### FileDropClass Relationship to Other Application Builder Classes

The FileDropClass is closely integrated with the WindowManager. These objects register their presence with each other, set each other's properties, and call each other's methods as needed to accomplish their respective tasks.

The FileDropComboClass is derived from the FileDropClass, and the FileDropClass is derived from the ViewManager. The FileDropClass relies on several of the other Application Builder Classes to accomplish its tasks. Therefore, if your program instantiates the FileDropClass, it must also instantiate these other classes. Much of this is automatic when you INCLUDE the FileDropClass header (ABDROPS.INC) in your program's data section. See the *Conceptual Example*.

## FileDropClass ABC Template Implementation

The ABC Templates automatically include all the classes and generate all the code necessary to support the functionality specified in your application's FileDrop control templates.

The templates *derive* a class from the FileDropClass and instantiate an object for *each* FileDropControl template in the application. The derived class and object is called FDB# where # is the FileDrop Control template instance number. The templates provide the derived class so you can use the FileDropControl template **Classes** tab to modify the FileDrop's behavior on an instance-by-instance basis.

The derived FileDropClass is local to the procedure, is specific to a single FileDropCombo and relies on the global file-specific RelationManager and FileManager objects for the displayed lookup file.

## FileDropClass Source Files

The FileDropClass source code is installed by default to the Clarion \LIBSRC folder. The FileDropClass source code and their respective components are contained in:

ABDROPS.INC	FileDropClass declarations
ABDROPS.CLW	FileDropClass method definitions

## FileDropClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a FileDropClass object and related objects.

This example uses the FileDropClass object to let the end user select a valid state code for a given client. The state code comes from the State file. When they are initialized properly, the FileDropClass and WindowManager objects do most of the work (event handling and field assignments) internally.

```
PROGRAM

INCLUDE( 'ABWINDOW.INC' )
INCLUDE( 'ABDROPS.INC' )
MAP
END

State          FILE, DRIVER( 'TOPSPEED' ), PRE( ST ), THREAD
StateCodeKey   KEY( ST:STATECODE ), NOCASE, OPT
Record         RECORD, PRE( )
StateCode      STRING( 2 )
```

```

StateName      STRING(20)
               END
               END

Customer FILE, DRIVER('TOPSPEED'), PRE(CUS), CREATE, THREAD
BYNUMBER KEY(CUS:CUSTNO), NOCASE, OPT, PRIMARY
Record      RECORD, PRE()
CUSTNO     LONG
Name       STRING(30)
State     STRING(2)
           END
           END

GlobalErrors ErrorClass
VCRRequest  LONG(0), THREAD

Access:State CLASS(FileManager)
Init         PROCEDURE
           END

Relate:State CLASS(RelationManager)
Init         PROCEDURE
           END

Access:Customer CLASS(FileManager)
Init         PROCEDURE
           END

Relate:Customer CLASS(RelationManager)
Init         PROCEDURE
           END

StateQ        QUEUE
ST:STATECODE LIKE(ST:STATECODE)
ViewPosition  STRING(512)
           END
StateView VIEW(State)
           END
CusWindow WINDOW('Add Customer'), AT(, , 157, 58), IMM, SYSTEM, GRAY
           PROMPT('Customer: '), AT(5, 7), USE(?NamePrompt)
           ENTRY(@s20), AT(61, 5, 88, 11), USE(CUS:NAME)
           PROMPT('State: '), AT(5, 22), USE(?StatePrompt)
           LIST, AT(61, 20, 65, 11), USE(CUS:State), FROM(StateQ), |
           FORMAT(' 8L~STATECODE~@s2@'), DROP(5)
           BUTTON('OK'), AT(60, 39), USE(?OK), DEFAULT
           BUTTON('Cancel'), AT(104, 39), USE(?Cancel)
           END
ThisWindow CLASS(WindowManager)

```

```

Init          PROCEDURE(),BYTE,PROC,VIRTUAL
Kill          PROCEDURE(),BYTE,PROC,VIRTUAL
              END

StateDrop    CLASS(FileDropClass)
Q            &StateQ
              END

CODE
  ThisWindow.Run()

ThisWindow.Init PROCEDURE()
ReturnValue  BYTE,AUTO
CODE
  GlobalErrors.Init
  Relate:State.Init
  Relate:Customer.Init
  SELF.Request = InsertRecord
  ReturnValue = PARENT.Init()
  IF ReturnValue THEN RETURN ReturnValue.
  SELF.FirstField = ?CUS:NAME
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
  SELF.AddUpdateFile(Access:Customer)
  SELF.AddItem(?Cancel,RequestCancelled)
  SELF.OkControl = ?OK
  Relate:Customer.Open
  Relate:State.Open
  SELF.Primary &= Relate:Customer
  SELF.InsertAction = Insert:Batch
  IF SELF.PrimeUpdate() THEN RETURN Level:Notify.
  OPEN(CusWindow)
  SELF.Opened=True
!initialize the FileDrop Class with:
! the LISTS's USE variable, LIST control, view POSITION, VIEW, LISTS's FROM QUEUE,
! primary file RelationManager object, WindowManager object
StateDrop.Init(?CUS:State,StateQ.ViewPosition,StateView,StateQ,Relate:State,ThisWindow)
StateDrop.Q &= StateQ
StateDrop.AddSortOrder()
StateDrop.AddField(ST:STATECODE,StateDrop.Q.ST:STATECODE)
StateDrop.AddUpdateField(ST:STATECODE,CUS:State)
ThisWindow.AddItem(StateDrop)
SELF.SetAlerts()
RETURN ReturnValue

```

```
ThisWindow.Kill PROCEDURE()
ReturnValue          BYTE,AUTO
CODE
  ReturnValue = PARENT.Kill()
  IF ReturnValue THEN RETURN ReturnValue.
  Relate:Customer.Close
  Relate:State.Close
  Relate:State.Kill
  Relate:Customer.Kill
  GlobalErrors.Kill
  RETURN ReturnValue

Access:State.Init PROCEDURE
CODE
  PARENT.Init(State,GlobalErrors)
  SELF.FileNameValue = 'State'
  SELF.Buffer &= ST:Record
  SELF.LazyOpen = False
  SELF.AddKey(ST:StateCodeKey,'ST:StateCodeKey',0)

Access:Customer.Init PROCEDURE
CODE
  PARENT.Init(Customer,GlobalErrors)
  SELF.FileNameValue = 'Customer'
  SELF.Buffer &= CUS:Record
  SELF.Create = True
  SELF.LazyOpen = False
  SELF.AddKey(CUS:BYNUMBER,'CUS:BYNUMBER',0)

Relate:State.Init PROCEDURE
CODE
  Access:State.Init
  PARENT.Init(Access:State,1)

Relate:Customer.Init PROCEDURE
CODE
  Access:Customer.Init
  PARENT.Init(Access:Customer,1)
```

## FileDropClass Properties

### FileDropClass Properties

The FileDropClass inherits all the properties of the ViewManager from which it is derived. See *ViewManager Properties* for more information.

In addition to the inherited properties, the FileDropClass contains the properties listed below.

### AllowReset (allow a reset)

**AllowReset**      **BYTE**

The **AllowReset** property indicates that a reset of the object's data can occur.

### DefaultFill (initial display value)

**DefaultFill**      **BYTE**

The **DefaultFill** property indicates whether FileDropClass object's LIST displays an initial value or blank, before the end user selects a value. A value of one (1) displays an initial value; a value of zero (0) displays nothing.

Implementation:      The Init method sets the DefaultFill property to one (1). The ResetQueue method implements the behavior specified by DefaultFill.

See Also:              Init, ResetQueue

### InitSyncPair (initial list position)

**InitSyncPair**      **BYTE**

The **InitSyncPair** property controls the initial position of the droplist. A value of one (1 or True) initially positions the list closest to the value already contained in the target assignment fields. A value of zero (0 or False) positions the list to the first item in the specified sort order.

Implementation:      The Init method sets the InitSyncPair property to one (1). The ResetQueue method implements the behavior specified by the InitSyncPair property.

See Also:              Init, ResetQueue

## FileDropClass Methods

### FileDropClass Methods

The FileDropClass inherits all the methods of the ViewManager from which it is derived. See *ViewManager Methods* for more information.

### FileDropClass Functional Organization--Expected Use

As an aid to understanding the FileDropClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the FileDropClass methods.

#### Non-Virtual Methods

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the FileDropClass object
AddField	specify display fields
AddUpdateField	specify field assignments
AddRange	add a range limit to the active sort order
AppendOrder	refine the active sort order
Kill	shut down the FileDropClass object

##### **Mainstream Use:**

ResetQueue	fill or refill filedrop queue
TakeEvent <sub>v</sub>	process the current ACCEPT loop event
TakeNewSelection <sub>v</sub>	processes EVENT:Selected events

##### **Occasional Use:**

Open <sub>i</sub>	open the filedrop view
PrimeRecord <sub>i</sub>	prepare an item for adding
SetFilter <sub>i</sub>	specify a filter for the active sort order
ApplyFilter <sub>i</sub>	range limit and filter the result set
ApplyOrder <sub>i</sub>	sort the result set
GetFreeElementName <sub>i</sub>	return the free element field name
SetOrder <sub>i</sub>	replace the active sort order
Close <sub>i</sub>	close the filedrop view

<sub>i</sub> These methods are inherited from the ViewManager Class.

<sub>v</sub> These methods are also virtual.

---

## Virtual Methods

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

SetQueueRecord	copy data from file buffer to queue buffer
Reset	reset the view position
TakeEvent	process the current ACCEPT loop event
TakeNewSelection	processes EVENT:Selected events
ValidateRecord	validate the current result set element

| These methods are inherited from the ViewManager Class.

## AddField (specify display fields)

**AddField**( *filefield*, *queuefield* )

---

<b>AddField</b>	Identifies the corresponding FILE and QUEUE fields for a filedrop list column.
<i>filefield</i>	The fully qualified label of the FILE field. The <i>filefield</i> is the original source of the filedrop LIST's data.
<i>queuefield</i>	The fully qualified label of the corresponding QUEUE field. The <i>queuefield</i> is loaded from the <i>filefield</i> , and is the immediate source of the filedrop LIST's data.

The **AddField** method identifies the corresponding FILE and QUEUE fields for a filedrop list column. You must call AddField for each column displayed in the filedrop list.

You may also use the AddField method to display memory variables by specifying a variable label as the *filefield* parameter.

Implementation: The AddField method uses the FieldPairsClass to manage the specified field pairs.

Example:

```
CODE
StFD.Init(?CLI:StCode,StateQ.Pos,StateView,StateQ,Relate:States,ThisWindow)
StFD.Q &= StateQ
StFD.AddSortOrder(StCodeKey)
StFD.AddField(STFile:StCode,StFD.Q.StCode)
StFD.AddField(STFile:StName,StFD.Q.StName)
StFD.AddUpdateField(STFile:StCode,CLI:StCode)
```

**AddRecord (update filedrop queue)****AddRecord**, VIRTUAL, PROTECTED

The **AddRecord** method adds data to the filedrop's display queue.

Implementation: The ResetQueue method calls the AddRecord method to build the queue.

See Also: ResetQueue

**AddUpdateField (specify field assignments)****AddUpdateField**( *source*, *target* )

---

**AddUpdateField** Identifies a *source* field and its corresponding *target* or destination field.

*source* The fully qualified label of the field to copy from when the end user selects a filedrop list item.

*target* The fully qualified label of the field to copy to when the end user selects a filedrop list item.

The **AddUpdateField** method identifies a *source* field and its corresponding *target* or destination field that receives the *source* field's contents when the end user selects a filedrop list item.

You may call the AddUpdateField multiple times to accomplish multiple field assignments on end user selection.

Implementation: The AddUpdateField method uses the FieldPairsClass to manage the specified field pairs.

The TakeEvent method performs the specified copy.

Example:

```

CODE
StFD.Init(?CLI:StCode,StateQ.Pos,StateView,StateQ,Relate:States,ThisWindow)
StFD.Q &= StateQ
StFD.AddSortOrder(StCodeKey)
StFD.AddField(STFile:StCode,StFD.Q.StCode)
StFD.AddField(STFile:StName,StFD.Q.StName)
StFD.AddUpdateField(STFile:StCode,CLI:StCode)

```

See Also: TakeEvent

**Init (initialize the FileDropClass object)**

*Init( listcontrol, viewposition, view, listqueue, relationmanager , window manager )*

---

<b>Init</b>	Initializes the FileDropClass object.
<i>listcontrol</i>	A numeric constant, variable, EQUATE, or expression containing the control number of the filedrop's LIST control.
<i>viewposition</i>	The label of a string variable within the <i>listqueue</i> containing the POSITION of the <i>view</i> .
<i>view</i>	The label of the filedrop's underlying VIEW.
<i>listqueue</i>	The label of the <i>listcontrol</i> 's data source QUEUE.
<i>relationmanager</i>	The label of the filedrop's primary file RelationManager object. See <i>Relation Manager</i> for more information.
<i>windowmanager</i>	The label of the FileDrop object's WindowManager object. See <i>Window Manager</i> for more information.

The **Init** method initializes the FileDropClass object.

Implementation: Among other things, the Init method calls the PARENT.Init (ViewManager.Init) method to initialize the view related parts of the FileDropClass object. See *View Manager* for more information.

Example:

```
CODE
StFD.Init(?CLI:StCode,StateQ.Pos,StateView,StateQ,Relate:States,ThisWindow)
StFD.Q &= StateQ
StFD.AddSortOrder(StCodeKey)
StFD.AddField(STFile:StCode,StFD.Q.StCode)
StFD.AddField(STFile:StName,StFD.Q.StName)
StFD.AddUpdateField(STFile:StCode,CLI:StCode)
```

See Also: ViewManager.Init

## Kill (shut down the FileDropClass object)

### Kill, VIRTUAL

The **Kill** method releases any memory allocated during the life of the FileDropClass object and performs any other required termination code.

Implementation: Among other things, the Kill method calls the PARENT.Kill (ViewManager.Kill) method to shut down the initialize the view related parts of the FileDropClass object. See *View Manager* for more information.

Example:

```
CODE
StFD.Init(?CLI:StCode,StateQ.Pos,StateView,StateQ,Relate:States,ThisWindow)
StFD.Q &= StateQ
StFD.AddSortOrder(StCodeKey)
StFD.AddField(STFile:StCode,StFD.Q.StCode)
StFD.AddField(STFile:StName,StFD.Q.StName)
StFD.AddUpdateField(STFile:StCode,CLI:StCode)
!procedure code
StFD.Kill
```

See Also: ViewManager.Kill

## ResetQueue (fill filedrop queue)

**ResetQueue**( [ *force* ] ), VIRTUAL, PROC

---

**ResetQueue** Fills or refills the filedrop's display queue.

*force* A numeric constant, variable, EQUATE, or expression that indicates whether to refill the queue even if the sort order did not change. A value of one (1 or True) unconditionally refills the queue; a value of zero (0 or False) only refills the queue if circumstances require it. If omitted, *force* defaults to zero.

The **ResetQueue** method fills or refills the filedrop's display queue, applying the applicable sort order, range limits, and filters, then returns a value indicating which item, if any, in the displayed lookup file already matches the value of the *target* fields (specified by the AddUpdateField method). A return value of zero (0) indicates no matching items; any other value indicates the position of the matching item.

For example, if the filedrop "looks up" the state code for a customer, and the current customer's state code field already contains a valid value, then the ResetQueue method conditionally (based on the InitSyncPair property) positions the filedrop list to the current customer's state code value.

Return Data Type: LONG

Example:

```
ACCEPT
  IF EVENT() = EVENT:OpenWindow
    StateFileDrop.ResetQueue
  END
!program code
END
```

See Also: InitSyncPair

## **SetQueueRecord (copy data from file buffer to queue buffer:FileDropClass)**

### **SetQueueRecord, VIRTUAL**

The **SetQueueRecord** method copies corresponding data from the *filefield* fields to the *queuefield* fields specified by the AddField method. Typically these are the file buffer fields and the filedrop list's queue buffer fields so that the queue buffer matches the file buffers.

Implementation: The ResetQueue method calls the SetQueueRecord method.

Example:

#### **MyFileDropClass.SetQueueRecord PROCEDURE**

```
CODE
SELF.ViewPosition=POSITION(SELF.View)
SELF.DisplayFields.AssignLeftToRight
!Custom code here
```

See Also: ResetQueue

## **TakeAccepted (a virtual to accept data)**

### **TakeAccepted, VIRTUAL**

The **TakeAccepted** method is a virtual placeholder to accept data in a FileDropCombo control.

Implementation: The TakeAccepted method is a placeholder for the derived FileDropComboClass. The FileDropClass.TakeEvent calls the TakeAccepted method.

See Also: FileDropClass.TakeEvent, FileDropComboClass.TakeAccepted

## TakeEvent (process the current ACCEPT loop event--FileDropClass)

### TakeEvent, VIRTUAL

The **TakeEvent** method processes the current ACCEPT loop event for the FileDropClass object.

Implementation: The WindowManager.TakeEvent method calls the TakeEvent method. The TakeEvent method calls the TakeNewSelection method.

Example:

```
MyWindowManager.TakeEvent PROCEDURE
Rval BYTE(Level:Benign)
I   USHORT,AUTO
CODE
!procedure code
LOOP I = 1 TO RECORDS(SELF.Browses)
  GET(SELF.Browses,I)
  SELF.Browses.Browse.TakeEvent
END
LOOP i=1 TO RECORDS(SELF.FileDrops)
  GET(SELF.FileDrops,i)
  ASSERT(~ERRORCODE())
  SELF.FileDrops.FileDrop.TakeEvent
END
RETURN RVal
```

See Also: TakeNewSelection, WindowManager.TakeEvent

## TakeNewSelection (process EVENT:NewSelection events:FileDropClass)

**TakeNewSelection**( *field* ), VIRTUAL

---

**TakeNewSelection** Processes the EVENT:NewSelection event.

*field* A numeric constant, variable, EQUATE, or expression containing the control number of the control that generated the EVENT:NewSelection event.

The **TakeNewSelection** method processes the EVENT:NewSelection event for the FileDropClass object.

Implementation: The ResetQueue method and the TakeEvent method call the TakeNewSelection method. If the FileDropClass object's LIST generated the new selection event, then the TakeNewSelection method does the field assignments specified by the AddUpdateField method or clears the target fields if there is no valid selection.

Example:

**FileDropClass.TakeEvent PROCEDURE**

```
CODE
CASE EVENT( )
OF EVENT:NewSelection
  SELF.TakeNewSelection(FIELD( ))
END
```

See Also: AddUpdateField, ResetQueue, TakeEvent

**ValidateRecord (a virtual to validate records)****ValidateRecord, VIRTUAL**

The **ValidateRecord** method is a virtual called when the FileDropClass object fills its display QUEUE. ValidateRecord returns a value indicating whether to include the current record in the displayed list. Thus ValidateRecord provides a filtering mechanism in addition to the ViewManager.SetFilter method. Valid return values include:

Record:OK	includes the record
Record:OutOfRange	excludes the record
Record:Filtered	excludes the record

Implementation: The ResetQueue method calls the ValidateRecord method. The ValidateRecord method calls the PARENT.ValidateRecord method (ViewManager.ValidateRecord).

Return value EQUATEs are declared in \LIBSRC\TPLEQU.CLW:

Record:OK	EQUATE(0)	!Record passes range and filter
Record:OutOfRange	EQUATE(1)	!Record fails range test
Record:Filtered	EQUATE(2)	!Record fails filter tests

Return Data Type: BYTE

Example:

```
MyFileDropClass.ResetQueue PROCEDURE
i LONG
CODE
SETCURSOR(CURSOR:Wait)
FREE(SELF.ListQueue)
SELF.ApplyRange
SELF.Reset
LOOP UNTIL SELF.Next()
  IF SELF.ValidateRecord()=Record:OK      !Validate Records
    SELF.SetQueueRecord
    ADD(SELF.ListQueue)
    ASSERT(~ERRORCODE())
    IF SELF.UpdateFields.Equal()
      i=RECORDS(SELF.ListQueue)
    END
  END
END
END
!procedure code
```

See Also: ResetQueue, ViewManager.SetFilter, ViewManager.ValidateRecord



---

# FileManager

## FileManager Overview

The FileManager class declares a file manager that consistently and flexibly handles all the routine database operations for a given file. The file manager provides "setup" methods that let you describe the file and its keys, as well as other methods to open, read, write, and close the file.

The file manager automatically handles autoincrementing keys, and, as implemented by the ABC Templates, handles some of the validity checks specified in the Clarion data dictionary, and some of the file handling settings specified in the data dictionary or application generator. However, even if you don't use the data dictionary, the application generator, or if you don't specify validity checks in your dictionary, the file manager can still competently and efficiently handle routine database operations for your files.

**Note:** The FileManager class handles individual files; it does not handle referential integrity (RI) between related files. The RelationManager class enforces RI between related files.

## Dual Approach to Database Operations

The FileManager methods that do standard database operations come in two versions--the plain (or interactive) version and the "Try" (or silent) version.

### Interactive Database Operations

---

When any of these methods are called (Open, Fetch, Next, Previous, Insert, and Update), they may take several approaches and several attempts to complete the requested operation--including issuing error messages where appropriate. They may solicit information from the end user in order to proceed with the requested task. They may even terminate the application under sufficient provocation. This means the programmer can rely on the fact that if the method returned, it worked.

### Silent Database Operations

---

When any of these methods are prepended with "Try" (TryOpen, TryFetch, TryNext, TryPrevious, TryInsert, and TryUpdate), the method makes a single attempt to complete the requested operation, then returns a success or failure indicator to the calling procedure for it to handle accordingly.

## FileManager Relationship to Other Application Builder Classes

The FileManager relies on the ErrorClass for most of its error handling. Therefore, if your program instantiates the FileManager it must also instantiate the ErrorClass. See *Error Class* for more information.

Perhaps more significantly, the FileManager serves as the foundation or "errand boy" of the RelationManager. If your program instantiates the RelationManager it must also instantiate the FileManager. See *Relation Manager Class* for more information.

## FileManager and Threaded Files

FileManager objects are designed to support multiple execution threads in a way that Clarion developers will recognize. That is, several MDI procedures may access the same file at the same time, with each procedure maintaining its own file buffer and file positioning information, so there is no conflict or confusion between the procedures.

To accomplish this desirable state of independence among several MDI procedures, you only need to add the THREAD attribute to your file declaration (see the *Language Reference* for more information), then instantiate a single global FileManager object for each file. This global object automatically handles multiple execution threads, so you can use it within each procedure that accesses the file. The ABC Templates generate exactly this type of code for files with the THREAD attribute.

When you want to access a file with a single shared buffer from multiple execution threads, you simply omit the THREAD attribute from the file declaration and, again, instantiate a global file-specific FileManager object within the program. This lets all your program's procedures access the file with a single shared record buffer and a single set of positioning information.

## FileManager ABC Template Implementation

There are several important points to note regarding the ABC Template implementation of the FileManager class.

First, the ABC Templates *derive* a class from the FileManager class for *each* file the application processes. The derived classes are called Hide:Access:*filename*, but may be referenced as Access:*filename*. These derived classes and their methods are declared in the generated *appnaBC0.CLW* through *appnaBC9.CLW* files (depending on how many files your application uses). The derived class methods are specific to the file being managed, and they implement many of the file properties specified in the data dictionary such as access modes, keys, field validation and initialization, etc.

Second, the ABC Templates generate housekeeping procedures to initialize and shut down the FileManager objects. The procedures are DctInit and DctKill. These are generated into the *appnaBC.CLW* file.

Third, the derived FileManager classes are configurable with the **Global Properties** dialog. See *Template Overview--File Control Options and Classes Options* for more information.

Finally, the ABC Templates also derive a RelationManager for each file. These objects are called Hide:Relate:*filename*, but may be referenced as Relate:*filename*. The template generated code seldom calls the derived FileManager methods directly. Instead, it calls a RelationManager method that echoes the command to the appropriate (related files') FileManager methods. See *Relation Manager* for more information on the RelationManager class.

## FileManager Source Files

The FileManager source code is installed by default to the Clarion \LIBSRC folder. The specific FileManager source code and their respective components are contained in:

```
ABFILE.INC    FileManager declarations
ABFILE.CLW    FileManager method definitions
```

## FileManager Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a FileManager object.

This example uses the FileManager to insert a valid record with an auto-incrementing key.

```
PROGRAM

INCLUDE('ABFILE.INC')           !declare FileManager class
MAP                             !program map
END

GlobalErrors ErrorClass        !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init          PROCEDURE        !initialize Access:File object
PrimeRecord   PROCEDURE,BYTE,PROC,VIRTUAL !prime new record (autoinc)
ValidateField PROCEDURE(UNSIGNED Id),BYTE,VIRTUAL !validate a field
ValidateRecord PROCEDURE(<*UNSIGNED Id>),BYTE,VIRTUAL !validate all fields
                END

Client        FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
IDKey        KEY(CLI:ID),NOCASE,OPT,PRIMARY
NameKey      KEY(CLI:Name),DUP,NOCASE
Record       RECORD,PRE()
ID           LONG
Name        STRING(20)
StateCode   STRING(2)
                END
                END
```

```

InsertWindow WINDOW('Add a new Client'),AT(,,159,73),IMM,SYSTEM,GRAY
    PROMPT('&Name:'),AT(8,20),USE(?CLI:Name:Prompt)
    ENTRY(@s20),AT(61,20,84,10),USE(CLI:Name),MSG('Client Name'),REQ
    PROMPT('State Code:'),AT(8,34),USE(?CLI:StateCode:Prompt)
    ENTRY(@s2),AT(61,34,40,10),USE(CLI:StateCode),MSG('State Code')
    BUTTON('OK'),AT(12,53,45,14),USE(?OK),DEFAULT
END

CODE
GlobalErrors.Init           !initialize GlobalErrors object
Access:Client.Init         !initial Access:Client object
Access:Client.Open         !open the Client file

IF Access:Client.PrimeRecord() !prime Client record (autoinc)
    POST(Event:CloseWindow)    !if prime fails, close down
END

OPEN(InsertWindow)

ACCEPT
CASE FIELD()
OF ?OK
    IF EVENT() = Event:Accepted           !on OK button
        IF Access:Client.Insert() = Level:Benign !add the new Client record
            POST(Event:CloseWindow)         !if add succeeds, close down
        ELSE                               !if add fails
            SELECT(?CLI:Name:Prompt)        !select client name field
            CYCLE                           !and start over
        END
    END
OF ?CLI:StateCode           !on StateCode field
    IF EVENT() = EVENT:Accepted
        IF Access:Client.ValidateField(3) !validate the StateCode (3rd) field
            SELECT(?CLI:StateCode)        !if invalid, select StateCode field
            CYCLE                         !and start over
        END
    END
END
END
Access:Client.Close         !close the Client file
Access:Client.Kill         !shut down the Access:Client object
GlobalErrors.Kill         !shut down the GlobalErrors object
RETURN

Access:Client.Init PROCEDURE
CODE
PARENT.Init(Client, GlobalErrors) !call the base class Init method
SELF.FileNameValue = 'Client'    !set the file name

```

```
SELF.Buffer &= CLI:Record           !point Access:Client to Client buffer
SELF.AddKey(CLI:IDKey,'Client ID',1) !describe the primary autoinc key
SELF.AddKey(CLI:NameKey,'Client Name') !describe another key

Access:Client.PrimeRecord PROCEDURE !called by base class Insert method
Result BYTE,AUTO
CODE
Result = PARENT.PrimeRecord()       !call base class PrimeRecord method
CLI:StateCode = 'FL'                !default statecode to Florida
RETURN Result

Access:Client.ValidateField PROCEDURE(UNSIGNED Id)!called by base class ValidateFields
CODE                                 !and by this program too
IF ID = 3                            !validate the statecode (3rd) field
  GlobalErrors.SetField('StateCode') !set field in case of error
  IF ~CLI:StateCode                  !if statecode is blank
    RETURN SELF.Throw(Msg:FieldNotInList) !pass error to error handler
  END
END
RETURN Level:Benign

Access:Client.ValidateRecord PROCEDURE(<*UNSIGNED F>)!called by base class
Insert
CODE
RETURN SELF.ValidateFields(1,3,F)    !validate all 3 fields
```

## FileManager Properties

The FileManager properties include references to the specific file being managed, as well as several flags or switches that tell the FileManager how to manage the referenced file.

The references are to the file, the file name, and the file's record buffer. These references allow the otherwise generic FileManager object to process a specific file.

The processing switches include file access (sharing) mode, a create/nocreate switch, a held records mode, and a LOCK wait time parameter.

Each of these properties is fully described below.

### AliasedFile (the primary file)

#### **AliasedFile      &FileManager**

The **AliasedFile** property is a reference to the actual file's FileManager. A nonnull value for this property indicates the managed file is an alias of another file. The FileManager uses this property to synchronize commands, buffers, etc. between the alias file and its actual file.

**Tip:**    **This property should be null (uninitialized) for the actual file and initialized for any aliases.**

Implementation:    If the managed file is an alias, you should initialize the AliasedFile property after the Init method is called, or within a derived Init method specific to the managed file. See the *Conceptual Example*. The ABC Templates generate code to set this property for alias files in the *appnaBC0.CLW* file.

## Buffer (the record buffer)

### **Buffer &GROUP, PROTECTED**

The **Buffer** property is a reference to the record buffer of the managed file. You can use the property to access the buffer for the file from within a generically derived class.

Implementation: The SaveBuffer method stores a copy of the current Buffer contents into the Buffers property for subsequent retrieval by the RestoreBuffer method.

You should initialize the Buffer property after the Init method is called, or within a derived Init method specific to the managed file. See the *Conceptual Example*.

See Also: Buffers, RestoreBuffer, SaveBuffer

## Buffers (saved record buffers)

### **Buffers &BufferQueue, PROTECTED**

The **Buffers** property contains saved copies of the record buffer for the managed file. The saved record images may be used to detect changes by other workstations, to implement cancel operations, etc.

Implementation: The SaveBuffer method stores a copy of the current Buffer contents into the Buffers property and returns an ID which may subsequently be used by the RestoreBuffer method to retrieve the buffer contents.

The RestoreBuffer method releases memory allocated by the SaveBuffer method. Therefore, to prevent a memory leak, each call to SaveBuffer should be paired with a corresponding call to RestoreBuffer.

Buffers is a reference to a QUEUE declared in ABFILE.INC as follows:

```

BufferQueue  QUEUE,TYPE      !Saved records
Id           LONG           !Handle to recognize saved instance
Buffer      &STRING        !Reference to a saved record
END

```

See Also: Buffer, RestoreBuffer, SaveBuffer

## **Create (create file switch)**

**Create** BYTE

The **Create** property contains a value that tells the file manager whether or not to create the file if no file exists.

A value of one (1) creates the file; a value of zero (0) does not create the file.

Implementation: The Init method sets the Create property to a value of one (1), which invokes automatic file creation. The ABC Templates override this default with the appropriate setting from the data dictionary or application generator. See *Template Overview--File Handling* for more information.

The Open method creates the file when an attempt to open the file fails because there is no file.

See Also: Init, Open

## **Errors (the ErrorManager)**

**Error** &ErrorClass, PROTECTED

The **Error** property is a reference to the ErrorManager. The Error property simply identifies the ErrorManager for the various FileManager methods.

Implementation: The Init method sets the value of the Error property.

See Also: Init

## **File (the managed file)**

**File** &FILE

The **File** property is a reference to the managed file. The File property simply identifies the managed file for the various FileManager methods.

Implementation: The Init method sets the value of the File property.

See Also: Init



**FileNameValue (constant filename)**

**FileNameValue** STRING(File:MaxFilePath), PROTECTED

The **FileNameValue** property contains the constant value specified by the managed file's NAME attribute. The FileNameValue property supplies the managed file's DOS filename for error messages or other display purposes.

The GetName method returns the DOS file name.

Implementation: You must initialize either the FileNameValue property or the FileName property (but not both) after the Init method is called, or within a derived Init method specific to the managed file. See the *Conceptual Example*.

Example:

```
PROGRAM
  INCLUDE('ABFILE.INC')      !declare FileManager class
  MAP                        !program map
  END

GlobalErrors ErrorClass      !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init      PROCEDURE          !prototype Access:Client init
      END

Client      FILE,DRIVER('TOPSPEED'),NAME('Client.TPS')!constant filename
Record      RECORD,PRE()
ID          LONG
Name        STRING(20)
      END
      END

CODE
GlobalErrors.Init
Access:Client.Init
!program code

Access:Client.Init PROCEDURE      !initialize Access:Client object
CODE
PARENT.Init(GlobalErrors)        !call the base class Init method
SELF.File      &= Client         !point Access:Client to Client file
SELF.FileNameValue = 'Client.TPS'!set constant DOS filename
```

See Also: FileName, GetName, SetName

## **LazyOpen (delay file open until access)**

**LazyOpen**      **BYTE**

The **LazyOpen** property indicates whether to open the managed file immediately when a related file is opened, or to delay opening the file until it is actually accessed. A value of one (1 or True) delays the opening; a value of zero (0 or False) immediately opens the file.

Delaying the open can improve performance when accessing only one of a series of related files.

Implementation:      The Init method sets the LazyOpen property to True. The ABC Templates override this default if instructed. See *Template Overview--File Handling* for more information.

The various file access methods (Open, TryOpen, Fetch, TryFetch, Next, TryNext, Insert, TryInsert, etc.) use the UseFile method to implement the action specified by the LazyOpen property

See Also:              Init, Open, TryOpen, Fetch, TryFetch, Next, TryNext, Insert, TryInsert, UseFile

## **LockRecover (/RECOVER wait time parameter)**

**LockRecover**      **SHORT**

The **LockRecover** property contains the wait time parameter for the /RECOVER driver string used by the Clarion database driver. See *Database Drivers--Clarion* for more information on the /RECOVER driver string.

Implementation:      The Init method sets the LockRecover property to a value of ten (10) seconds. The ABC Templates override this default with the appropriate value from the application generator. See *Template Overview--File Handling* for more information.

The Open method implements the recovery when an attempt to open the file fails because the file is LOCKed. See the *Language Reference* for more information on LOCK.

See Also:              Init, Open

## **OpenMode (file access/sharing mode)**

**OpenMode**      **BYTE**

The **OpenMode** property contains a value that determines the level of access granted to both the user opening the file and other users in a multi-user system.

Implementation:      The Init method sets the OpenMode property to a hexadecimal value of 42h (ReadWrite/DenyNone). The ABC Templates override this default with the appropriate value from the application generator. See *Template Overview--File Handling* for more information.

The Open method uses the OpenMode property when it OPENS the file for processing. See the *Language Reference* for more information on OPEN and access modes.

See Also:              Init, Open

## **SkipHeldRecords (HELD record switch)**

**SkipHeldRecords**      **BYTE**

The **SkipHeldRecords** property contains a value that tells the file manager how to react when it encounters held records. See the *Language Reference* for more information on HOLD.

A value of one (1) skips or omits the held record and continues processing; a value of zero (0) aborts the current operation.

Implementation:      The Init method sets the SkipHeldRecords property to a value of zero (0).

The Next, TryNext, Previous, and TryPrevious methods implement the action specified by the SkipHeldRecords property when an attempt to read a record fails because the record is held.

See Also:              Init, Next, Previous, TryNext, TryPrevious

## FileManager Methods

### **Naming Conventions and Dual Approach to Database Operations**

As you study the functional organization of the FileManager methods, please keep this in mind: most of the common database operations (Open, Next, Previous, Fetch, Insert, and Update) come in two versions. The versions are easily identifiable based on their naming conventions:

<i>Operation</i>	Do <i>Operation</i> and handle any errors (automatic)
<i>TryOperation</i>	Do <i>Operation</i> but do not handle errors (manual)

### **Interactive Database Operations**

When any of these methods are called (Open, Fetch, Next, Previous, Insert, and Update), they may take several approaches and several attempts to complete the requested operation, including issuing error messages where appropriate. These methods provide automatic error handling. They may solicit information from the end user in order to proceed with the requested task. They may even terminate the application under sufficient provocation. This means the programmer can rely on the fact that if the method returned, it worked.

### **Silent Database Operations**

When any of these methods prepend "Try" (TryOpen, TryFetch, TryNext, TryPrevious, TryInsert, and TryUpdate), the method makes a single attempt to complete the requested operation, then returns a success or failure indicator to the calling procedure for it to handle accordingly. These methods require manual error handling.

## **FileManager Functional Organization--Expected Use**

As an aid to understanding the FileManager class, it is useful to organize the various FileManager methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the FileManager methods.

### **Non-Virtual Methods**

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

#### **Housekeeping (one-time) Use:**

Init	initialize the FileManager object
Kill	terminate the FileManager object

#### **Mainstream Use:**

Open <sub>v</sub>	open the file
TryOpen	open the file
Next	get the next record in sequence
TryNext	get the next record in sequence
Previous	get the previous record in sequence
TryPrevious	get the previous record in sequence
Fetch	get a specific record by key value
TryFetch	get a specific record by key value
Position	return the unique position of the current record
TryReget	get a specific record by unique position
PrimeAutoInc <sub>v</sub>	prepare an autoincremented record for adding
Insert	add a new record
TryInsert	add a new record
CancelAutoInc <sub>v</sub>	restore file to its pre-PrimeAutoInc state
Update	change the current record
TryUpdate	change the current record
Close <sub>v</sub>	close the file

<sub>v</sub> These methods are also Virtual.

#### **Occasional Use:**

ClearKey	clear a range of key component fields
SetKey	make a specific key current for other methods
KeyToOrder	return ORDER expression equal to specified key
GetComponents	return the number of components of a key
GetField	return a reference to a key component
GetFieldName	return the field name of a key component
GetEOF	return current end of file status
GetError	return the current error ID
SetError	save the current error state

---

GetName	return the name of the file
SetName	set the file name
SaveBuffer	save the current record buffer contents
RestoreBuffer	restore previously saved buffer contents
SaveFile	save the current file state
RestoreFile	restore a previously saved file state
UseFile	open a LazyOpen file
AddKey	describe the soft KEYs

### Virtual Methods

---

Typically, with the possible exception of Open and Close, you will not call these methods directly- the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Open	open the file
BindFields	BIND all the file's fields
PrimeAutoInc	prepare an autoincremented record for adding
TryPrimeAutoInc	prepare an autoincremented record for adding
CancelAutoInc	restore file to its pre-PrimeAutoInc state
EqualBuffer	detect record buffer changes
PrimeFields	prepare record fields for adding
PrimeRecord	prepare a record for adding
Throw	process an error
ThrowMessage	set custom message text then process an error
ValidateField	validate a specific field in the current buffer
ValidateFields	validate a range of fields in the current buffer
ValidateRecord	validate all fields in the current buffer
Close	close the file

**AddField(track fields in a structure)**

**AddField**(*field tag*, *field*, *field type*, [*field picture*]), **PROC**

---

<b>AddField</b>	Updates internal queue to track fields in a structure.
<i>field tag</i>	A variable length string that represents the actual field name.
<i>field</i>	The field represents the field name used in a queue structure.
<i>field type</i>	A variable length string that represents the field type.
<i>field picture</i>	A variable length string that represents the field picture.

The **AddField** method updates an internal queue which keeps track of the relationship of fields in a file structure to their equivalent field in a queue structure. An internal queue tracks the field name, structure field name and field type.

Implementation: The AddField method returns Level:Benign if no error occurs when the field information is added to the internal tracking queue, otherwise Level:Notify is returned.

Return Data Type: BYTE

**AddKey (set the file's keys)**

**AddKey** ( *key*, *description* [,*autoincrement*] )

---

<b>AddKey</b>	Describes a KEY or static INDEX of the managed file.
<i>key</i>	The label of the KEY or static INDEX.
<i>description</i>	A string constant, variable, EQUATE, or expression describing the key.
<i>autoincrement</i>	An integer constant, variable, EQUATE, or expression that indicates whether the FileManager automatically generates incrementing numeric values for the key when inserting new records. A value of one (1 or True) automatically increments the key; a value of zero (0 or False) does not increment the key. If omitted, <i>autoincrement</i> defaults to zero.

The **AddKey** method describes a KEY or static INDEX of the managed file so that other FileManager methods can process it. You should typically call AddKey after the Init method is called (or within your derived Init method).

Implementation: The *description* appears at runtime on certain key related error messages.

Example:

```
Access:Client.Init  PROCEDURE
CODE
PARENT.Init(Client, GlobalErrors)      !call the base class Init method
SELF.FileNameValue = 'Client'          !set the file name
SELF.Buffer &= CLI:Record              !point Access:Client to Client buffer
SELF.AddKey(CLI:IDKey,'Client ID',1)   !describe the primary key
SELF.AddKey(CLI:NameKey,'Client Name') !describe another key
```

See Also:        Init

**BindFields (bind fields when file is opened)****BindFields, VIRTUAL**

The **BindFields** method BINDs the fields when the file is opened. See the *Language Reference* for more information on BIND.

Implementation: The Open method calls the BindFields method.

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP                             !program map
  END

GlobalErrors ErrorClass          !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
BindFields  PROCEDURE,VIRTUAL    !prep fields for dynamic use
  END

Client      FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
IDKey       KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record      RECORD,PRE()
ID          LONG
Name        STRING(20)
StateCode   STRING(2)
  END
  END

CODE
!program code

Access:Client.BindFields PROCEDURE !called by the base class Open method
CODE
  BIND(CLI:RECORD)                !bind all fields for dynamic use

```

See Also: Open

**CancelAutoInc (undo PrimeAutoInc)**

**CancelAutoInc**( [*relation manager*] ), **VIRTUAL, PROC**

**CancelAutoInc** Undoes any PrimeAutoInc action.

*relation manager*

The label of the managed file's RelationManager object. If present, the "undo" action cascades to any related files. If omitted, the "undo" action does not cascade to related files.

The **CancelAutoInc** method restores the managed file, and optionally any related files, to their pre-PrimeAutoInc state, typically when an insert operation is cancelled. CancelAutoInc returns a value indicating its success or failure. A return value of zero (0 or Level:Benign) indicates success; any other return value indicates a problem.

Implementation: The PrimeAutoInc method adds a "dummy" record when inserting records with autoincrementing keys. CancelAutoInc deletes this "dummy" record, and, if the *relation manager* parameter is present, CancelAutoInc deletes any children of the "dummy" record as well.

If CancelAutoInc succeeds, it returns Level:Benign (declared in ABERROR.INC). If it ultimately fails, it returns the severity level of the error it encountered while trying to restore the files. See *Error Class* for more information on severity levels.

Return Data Type: **BYTE**

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP                             !program map
  END

GlobalErrors ErrorClass          !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init      PROCEDURE              !prototype Access:File init
CancelAutoInc PROCEDURE,VIRTUAL  !prototype CancelAutoInc
      END

Client      FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
IDKey      KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record     RECORD,PRE()
ID         LONG
Name       STRING(20)
StateCode  STRING(2)

```

```

        END
    END
InsertWindow WINDOW('Add a new Client'),AT(,,159,73),IMM,SYSTEM,GRAY
    PROMPT('&Name:'),AT(8,20),USE(?CLI:Name:Prompt)
    ENTRY(@s20),AT(61,20,84,10),USE(CLI:Name),MSG('Client Name'),REQ
    PROMPT('State Code:'),AT(8,34),USE(?CLI:StateCode:Prompt)
    ENTRY(@s2),AT(61,34,40,10),USE(CLI:StateCode),MSG('State Code')
    BUTTON('OK'),AT(12,53,45,14),USE(?OK),DEFAULT
    BUTTON('Cancel'),AT(82,53,45,14),USE(?Cancel)
    END

CODE
GlobalErrors.Init          !initialize GlobalErrors object
Access:Client.Init        !initialize Access:Client object
Access:Client.Open        !open the Client file
IF Access:Client.PrimeRecord() !prime Client record (autoinc)
    POST(Event:CloseWindow)    !if prime fails, close down
END

OPEN(InsertWindow)

ACCEPT
CASE FIELD()
OF ?OK
    IF EVENT() = Event:Accepted    !on OK button
        IF Access:Client.Insert() = Level:Benign!finish adding the new Client record
            POST(Event:CloseWindow)    !if add succeeds, close down
        ELSE
            !if add fails
            SELECT(?CLI:Name:Prompt)    !select client name field
            CYCLE                        !and start over
        END
    END
OF ?Cancel
    IF EVENT() = EVENT:Accepted    !on Cancel button
        Access:Client.CancelAutoInc    !restore Client to pre-PrimeRecord
        POST(Event:CloseWindow)    !close down
    END
END
EMD
END

```

---

```
Access:Client.Close           !close the Client file
Access:Client.Kill           !shut down the Access:Client object
GlobalErrors.Kill           !shut down the GlobalErrors object
RETURN
```

```
Access:Client.CancelAutoInc PROCEDURE      !restore file to pre-PrimeAutoInc
CODE
!your custom code here
PARENT.CancelAutoInc                    !call the base class method
!your custom code here
```

See Also:        PrimeAutoInc

**ClearKey (clear specified key components)**

**ClearKey** ( *key* [, *firstcomponent*] [, *lastcomponent*] [, *highvalue*] )

---

<b>ClearKey</b>	Clears or (re)initializes the specified range of key component fields.
<i>key</i>	The label of the KEY.
<i>firstcomponent</i>	A numeric constant, variable, EQUATE, or expression that indicates the first component to clear. If omitted, <i>firstcomponent</i> defaults to one (1).
<i>lastcomponent</i>	A numeric constant, variable, EQUATE, or expression that indicates the last component to clear. If omitted, <i>lastcomponent</i> defaults to twenty-two (22).
<i>highvalue</i>	An integer constant, variable, EQUATE, or expression that indicates whether to clear the components to zero (or spaces for string fields) or to their highest possible values. A value of one (1) applies the highest possible value; a value of zero (0) applies spaces for strings and zeros for numerics. If omitted, <i>highvalue</i> defaults to zero (0).

The **ClearKey** method clears or (re)initializes the specified range of key component fields.

Implementation: ClearKey is useful for range limiting to the first instance of the first "free" key component. By retaining higher order key component values and clearing lower order key component values, you can fetch the first (or last) record that matches the retained higher order component values; for example, the first order (lower order key component) for a customer (higher order key component).

The value ClearKey assigns depends on three things: the data type of the component field (numeric or string), the sort direction of the component (ascending or descending), and the value of the *highvalue* parameter (True or False). The following table shows the values ClearKey assigns for each combination of data type, sort direction, and *highvalue*.

<u><i>highvalue</i></u>	<b>Numeric Fields</b>		<b>String Fields</b>	
	<u>Ascending</u>	<u>Descending</u>	<u>Ascending</u>	<u>Descending</u>
True (1)	High Values	zero	High Values	spaces
False (0)	zero	High Values	spaces	High Values

Example:

```
PROGRAM
  INCLUDE('ABFILE.INC')
  MAP
  END
GlobalErrors ErrorClass
Access:Order CLASS(FileManager)
END
```

```
!declare FileManager class
!program map
!declare GlobalErrors object
!derive Access:Order object
```

```
Order      FILE, DRIVER('TOPSPEED'), PRE(ORD), CREATE, BINDABLE, THREAD
IDKey      KEY(Ord:Cust, Ord:ID, Ord:Date), NOCASE, OPT, PRIMARY
Record     RECORD, PRE()
Cust       LONG
ID         LONG
Date       LONG
           END
           END

CODE
!program code
!find first order for current customer by clearing all components except Ord:Cust
Access:Order.ClearKey( ORD:IDKey, 2 ) !clear Ord:ID and Ord:Date
Access:Order.Fetch                !get the next record by key
```

**Close (close the file)****Close, VIRTUAL, PROC**

The **Close** method tells the FileManager the calling procedure is done with the file, then closes the file if no other procedure is using it. The Close method handles any errors that occur while closing the file.

Implementation: The Close method returns a value of Level:Benign (EQUATE declared in ABERROR.INC). See *Error Class* for more information on Level:Benign and other severity levels.

Return Data Type: BYTE

Example:

```
PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP                             !program map
  END

GlobalErrors ErrorClass         !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init      PROCEDURE             !prototype Access:File init
      END

Client      FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
      !file declaration
      END

CODE
GlobalErrors.Init               !initialize GlobalErrors object
Access:Client.Init              !initialize Access:Client object
Access:Client.Open              !open the Client file

!program code

Access:Client.Close             !close the Client file
Access:Client.Kill              !shut down the Access:Client object
GlobalErrors.Kill               !shut down the GlobalErrors object
```

**Deleted (return record status)****Deleted,VIRTUAL**

The **Deleted** method returns Level:Benign (declared in ABERROR.INC) if the current record is active, i.e., if the record has not been identified in some way as deleted. In cases where the DeleteRecord method has been derived to say, flag deleted records rather than physically delete them, deriving a corresponding Deleted method allows these records to be identified.

The standard Deleted method always returns Level:Benign.

Return Data Type: BYTE

Example:

```

PROGRAM
INCLUDE('ABFILE.INC')           !declare FileManager class
MAP . !program map
GlobalErrors ErrorClass        !declare GlobalErrors object
Access:Client CLASS(FileManager) . !derive Access:Client object
InsertWindow WINDOW('Add a new Client'),AT(,,159,73),IMM,SYSTEM,GRAY
    PROMPT('&Name:'),AT(8,20),USE(?CLI:Name:Prompt)
    ENTRY(@s20),AT(61,20,84,10),USE(CLI:Name),MSG('Client Name'),REQ
    PROMPT('State Code:'),AT(8,34),USE(?CLI:StateCode:Prompt)
    ENTRY(@s2),AT(61,34,40,10),USE(CLI:StateCode),MSG('State Code')
    BUTTON('OK'),AT(12,53,45,14),USE(?OK),DEFAULT
END
CODE
!program code
ACCEPT
CASE FIELD()
OF ?OK
IF EVENT() = Event:Accepted           !on OK button
    IF Access:Client.Deleted() = Level:Benign !if the record is not already deleted
        IF Access:Client.DeleteRecord = Level:Benign !delete it
            POST(Event:CloseWindow)           !if add succeeds, close down
        ELSE                                   !if add fails
            Access:Client.CancelPrimeAutoInc !restore the file
        CYCLE                                 !and start over
    END
END
END
!more code

```

See Also: DeleteRecord

**DeleteRecord (delete a record)**

**DeleteRecord**(*< query >*),PROC,VIRTUAL

---

**DeleteRecord** Deletes a record from a file.

*query* A numeric constant, variable, EQUATE, or expression that determines whether DeleteRecord offers to confirm the delete with the end user. A value of one (1 or True) deletes only on confirmation from the end user; a value of zero (0 or False) delete without confirmation. If omitted, query defaults to 1.

The **DeleteRecord** method deletes a record from the file and returns Level:Benign (declared in ABERROR.INC). The primary purpose of this method is to permit the redefinition of "Delete" where appropriate. Possible uses include flagging a record as opposed to physically deleting it or logging delete transactions as they occur. The Query parameter is provided for compatibility with the Relation Manager's Delete method (See...) RelationManager.Delete calls FileManager.DeleteRecord with Query = 1 in all cases except that where a delete has been requested without notifying the user. This allows derived versions of FileManager.DeleteRecord to physically delete a record after, say, the cancellation of autoincremented update.

The standard DeleteRecord method physically deletes the record and always returns Level:Benign.

Return Data Type: BYTE

Example:

```

PROGRAM
INCLUDE('ABFILE.INC')           !declare FileManager class
MAP
END
GlobalErrors ErrorClass        !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
END
InsertWindow WINDOW('Add a new Client'),AT(,,159,73),IMM,SYSTEM,GRAY
    PROMPT('&Name:'),AT(8,20),USE(?CLI:Name:Prompt)
    ENTRY(@s20),AT(61,20,84,10),USE(CLI:Name),MSG('Client Name'),REQ
    PROMPT('State Code:'),AT(8,34),USE(?CLI:StateCode:Prompt)
    ENTRY(@s2),AT(61,34,40,10),USE(CLI:StateCode),MSG('State Code')
    BUTTON('OK'),AT(12,53,45,14),USE(?OK),DEFAULT
END
CODE
!program code
ACCEPT
    CASE FIELD()
    OF ?OK
    IF EVENT() = Event:Accepted           !on OK button
    IF Access:Client.DeleteRedord() = Level:Benign !delete the new Client record
        POST(Event:CloseWindow)         !if add succeeds, close down
    ELSE                                  !if add fails

```

```
Access:Client.CancelPrimeAutoInc      !restore the file
CYCLE                                  !and start over
END
END
!more code
```

See Also: Deleted

## Destruct (automatic destructor)

### Destruct, VIRTUAL

The **Destruct** method is an automatic destructor that is called when the object is removed from memory. This ensures that all data allocated by the object is removed from memory.

**EqualBuffer (detect record buffer changes)****EqualBuffer**( *buffer id* ), VIRTUAL

---

**EqualBuffer** Compares the managed file's record buffer with the specified buffer and returns a value indicating whether the buffers are equal.

*buffer id* An integer constant, variable, EQUATE, or expression that identifies the buffer contents to compare--typically a value returned by the SaveBuffer method.

The **EqualBuffer** method compares the managed file's record buffer, including any MEMOs or BLOBs, with the specified buffer and returns a value indicating whether the buffers are equal. A return value of one (1 or True) indicates the buffers are equal; a return value of zero (0 or False) indicates the buffers are not equal. Assigning PROP:Handle on a BLOB field constitutes a change to the BLOB and will cause EqualBuffer() to return False.

Return Data Type: BYTE

Example:

**MyWindowManager.TakeCloseEvent** PROCEDURE

CODE

```
IF SELF.Response = RequestCancelled           !if end user cancelled the form
  IF ~SELF.Primary.Me.EqualBuffer(SELF.Saved)  !check for any pending changes
    !handle cancel of pending changes
```

END

END

See Also: SaveBuffer

**Fetch (get a specific record by key value)**Fetch( *key* ), VIRTUAL, PROC**Fetch** Gets a specific record by its key value and handles any errors.*key* The label of the primed KEY.

The **Fetch** method gets a specific record by its key value and handles any errors. You must prime the key before calling Fetch. If the key is not unique, Fetch gets the first record with the specified key value.

The TryFetch method provides a slightly different (manual) alternative for fetching specific records.

Implementation: Fetch tries to get the specified record. If it succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns Level:Notify (also declared in ABERROR.INC) and *clears the record buffer*. See *Error Class* for more information on severity levels.

Return Data Type: BYTE

Example:

```

PROGRAM
  INCLUDE( 'ABFILE.INC' )                !declare FileManager class
  MAP                                     !program map
  END
GlobalErrors  ErrorClass                !declare GlobalErrors object
Access:States CLASS(FileManager)        !declare Access:States object
  END

States        FILE, DRIVER( 'TOPSPEED' ), PRE( ST ), CREATE, BINDABLE, THREAD
StateCodeKey  KEY( ST:StateCode ), NOCASE, OPT, PRIMARY
Record        RECORD, PRE( )
StateCode     STRING( 2 )
State         STRING( 20 )
  END
  END

CODE
!program code
!get the state record for Florida
ST:StateCode = 'FL'                      !prime the state key for the fetch
Access:States.Fetch(ST:StateCodeKey)!fetch the record and handle any errors

```

See Also: TryFetch

**GetComponent** (return the number of key components)**GetComponent**( *key* )**GetComponent** Returns the number of components in the specified key.*key* The label of the KEY.The **GetComponent** method returns the number of components in the specified key.

Return Data Type: BYTE

Example:

```

PROGRAM
  INCLUDE( 'ABFILE.INC' )           !declare FileManager
  MAP                               !program map
  END
GlobalErrors ErrorClass           !declare GlobalErrors objec
Access:Order CLASS(FileManager)   !derive Access:Order object
  END
I BYTE
Order FILE,DRIVER('TOPSPEED'),PRE(ORD),THREAD !declare order file
IDKey KEY(Ord:Cust,Ord:ID,Ord:Date),NOCASE,OPT,PRIMARY
Record RECORD,PRE()
Cust LONG
ID LONG
Date LONG
  END
  END
KeyQueue QUEUE,PRE(KeyQ)          !a list of key components
Field ANY                         !component field reference
FieldName STRING(12)              !component field name
  END
CODE
!program code
LOOP Access:Order.GetComponent( ORD:IDKey ) TIMES !step thru key components
  I += 1                             !increment counter
  KeyQ.Field = Access:Order.GetField(ORD:IDKey,I)!get component reference
  KeyQ.FieldName = Access:Order.GetFieldName(ORD:IDKey,I)!get component name
END

```

## GetEOF (return end of file status)

### GetEOF

The **GetEOF** method returns the current end of file status for the managed file.

**Tip:** **GetEOF** is designed to be used after a call to the **Next** or **Previous** method. The **GetEOF** return value is undefined prior to the call to **Next** or **Previous**.

Implementation: **GetEOF** returns one (1 or True) if the last record in a **Next/Previous** series was read; otherwise it returns zero (0 or False).

Return Data Type: **BYTE**

Example:

```
PROGRAM
INCLUDE('ABFILE.INC')           !declare FileManager class
MAP                             !program map
END
GlobalErrors ErrorClass        !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
END
CODE
!program code
LOOP                             !loop through client file
CASE Access:Client.Next()       !get next record in sequence
OF Level:Notify OROF Level:Fatal!if error occurred
  POST(Event:CloseWindow)       !shut down
  BREAK
ELSE                             !otherwise
  PRINT(Rpt:Detail)             !print the record
END
UNTIL Access:Client.GetEOF()     !stop looping at end of file
```

See Also: **Next**, **TryNext**, **Previous**, **TryPrevious**

**GetError (return the current error ID)****GetError**

The **GetError** method returns the current error ID for the managed file. See *Error Class* for more information on error IDs.

Return Data Type: SIGNED

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP                             !program map
  LogError(STRING filename, SHORT error) !prototype LogError procedure
  END

GlobalErrors ErrorClass          !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
  END

ErrorLog FILE,DRIVER('TopSpeed'),PRE(LOG),CREATE,THREAD!declare log file
Record RECORD
Date LONG
Time LONG
File STRING(20)
ErrorId SHORT
  END
  END

CODE
!program code
IF Access:Client.Open()           !if error occurs
  LogError(Access:Client.GetName(),Access:Client.GetError())!log name and error id
END
!program code

LogError PROCEDURE(STRING filename, SHORT error)
CODE
LOG:Date = TODAY()               !store date
LOG:Time = CLOCK()               !store time
LOG:File = filename              !store filename
LOG:ErrorId = error              !store error id
ADD(ErrorLog)                    !write logfile

```

**GetField (return a reference to a key component)**

<b>GetField</b> (  <i>key</i> , <i>component</i>  )	
<i>field tag</i>	
<i>index</i> , <i>field tag</i> , <i>field</i>	

---

<b>GetField</b>	Returns a reference to the specified key component or field.
<i>key</i>	The label of the KEY.
<i>component</i>	A numeric constant, variable, EQUATE, or expression that indicates the component field to reference. A value of one (1) specifies the first component; two (2) specifies the second component, etc.
<i>field tag</i>	A variable length string that represents the actual field name.
<i>index</i>	An integer constant, variable, EQUATE or expression that contains the field number.
<i>field</i>	Represents the field name used in a queue structure.

The **GetField** method returns a reference to the specified key component or field.

**GetField**(key, component)  
Returns a reference to the specified key component.

**GetField**(field tag)  
Returns a reference to the field based on the specified field tag.

**GetField**(index, field tag, field)  
Returns Level:Notify if no field exists at the specified index position.  
Returns Level:Benign if a field is successfully retrieved at the specified index position.

Return Data Type:     \*?, (untyped variable parameter)  
                          where prototype is GetField(key, component)

                          \*?, (untyped variable parameter)  
                          where prototype is GetField(field tag)

                          BYTE  
                          where prototype is GetField(index, field tag, field)

Example:

```
PROGRAM
  INCLUDE('ABFILE.INC')
  MAP
  END
GlobalErrors ErrorClass
!declare FileManager
!program map
!declare GlobalErrors objec
```

```

Access:Order CLASS(FileManager)      !derive Access:Order object
      END
I      BYTE

Order      FILE,DRIVER('TOPSPEED'),PRE(ORD),THREAD !declare order file
IDKey      KEY(Ord:Cust,Ord:ID,Ord:Date),NOCASE,OPT,PRIMARY
Record     RECORD,PRE()
Cust       LONG
ID         LONG
Date      LONG
      END
      END
KeyQueue   QUEUE,PRE(KeyQ)           !a list of key components
Field      ANY                       !component field reference
FieldName  STRING(12)                !component field name
      END

CODE
!program code
LOOP Access:Order.GetComponents( ORD:IDKey ) TIMES      !step thru key components
  I += 1                                               !increment counter
  KeyQ.Field      = Access:Order.GetField(ORD:IDKey,I)  !get component reference
  KeyQ.FieldName = Access:Order.GetFieldName(ORD:IDKey,I)!get component name
END

```



---

```
FieldName  STRING(12)           !component field name
      END
CODE
!program code
LOOP Access:Order.GetComponents( ORD:IDKey ) TIMES      !step thru key components
  I += 1                                               !increment counter
  KeyQ.Field      = Access:Order.GetField(ORD:IDKey,I)  !get component reference
  KeyQ.FieldName = Access:Order.GetFieldName(ORD:IDKey,I)!get component name
END
```

## GetFields(get number of fields)

### **GetFields**

The **GetFields** method returns the number of fields in the file's record structure.

Return Data Type:       LONG

## GetFieldPicture(get field picture)

### **GetFieldPicture(*field tag*)**

---

**GetFieldPicture**       Returns the field type of the specified field.

*field tag*       A variable length string that represents the field picture.

The **GetFieldPicture** method returns the field picture based on the specified field.

Return Data Type:    ASTRING

## GetFieldType(get field type)

### **GetFieldType(*field*)**

---

**GetFieldType**       Returns the field type of the specified field.

*field*       A variable length string that represents the field name to be queried.

The **GetFieldType** method returns the field type based on the specified field.

Return Data Type:       ASTRING

**GetName (return the filename)****GetName**

The **GetName** method returns the filename of the managed file for display in error messages, etc.

The SetName method sets the (variable) filename of the managed file.

Implementation:    **GetName** returns the value of the FileNameValue property if it has a value; otherwise, it returns the value of the FileName property.

Return Data Type:   **STRING**

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')                !declare FileManager class
  MAP                                  !program map
  LogError (STRING filename, SHORT error)!prototype LogError procedure
  END

GlobalErrors ErrorClass                !declare GlobalErrors object
Access:Client CLASS(FileManager)      !derive Access:Client object
  END

ErrorLog FILE,DRIVER('TopSpeed'),PRE(LOG),CREATE,THREAD!declare log file
Record RECORD
Date LONG
Time LONG
File STRING(20)
ErrorId SHORT
  . .
CODE
  !program code
  IF Access:Client.Open()                !if error occurs
    LogError(Access:Client.GetName(),Access:Client.GetError())!log name and error id
  END
  !program code

LogError PROCEDURE(STRING filename, SHORT error)
CODE
LOG:Date = TODAY()                      !store date
LOG:Time = CLOCK()                      !store time
LOG:File = filename                     !store filename
LOG:ErrorId = error                     !store error id
ADD(ErrorLog)                            !write logfile

```

See Also:            FileName, FileNameValue, SetName

**Init (initialize the FileManager object)****Init**( *file*, *error handler* ), **VIRTUAL****Init** Initializes the FileManager object.*file* The label of the managed file.*error handler* The label of an ErrorClass object. See *Error Class* for more information.The **Init** method initializes the FileManager object.

Implementation: The Init method does not initialize some file specific properties (Buffer, FileName, and FileNameValue). You should explicitly initialize these properties after the Init method is called (or within your derived Init method). See the Conceptual Example.

Example:

```

PROGRAM
  INCLUDE( 'ABFILE.INC' )           !declare FileManager class
  MAP                               !program map
  END

GlobalErrors ErrorClass           !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init      PROCEDURE               !initialize Access:File object
      END

Client      FILE,DRIVER( 'TOPSPEED' ),PRE(CLI),CREATE,BINDABLE,THREAD
IDKey      KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record     RECORD,PRE()
ID         LONG
Name       STRING(20)
StateCode  STRING(2)
      END
      END

CODE
GlobalErrors.Init           !initialize the GlobalErrors object
Access:Client.Init         !initialize Access:Client object
!program code
Access:Client.Kill         !shut down the Access:Client object
GlobalErrors.Kill         !shut down the GlobalErrors object

```

Access:Client.Init PROCEDURE

CODE

```
PARENT.Init(Client, GlobalErrors)!call the base class Init method
SELF.FileNameValue = 'Client'    !set the file name
SELF.Buffer &= CLI:Record        !point Access:Client to Client buffer
SELF.AddKey(CLI:IDKey,'Client ID',1) !describe the primary key
```

See Also: Buffer, File, FileName, FileNameValue

**Insert (add a new record)****Insert, PROC**

The **Insert** method adds a new record to the file, making sure the record is valid, and automatically incrementing key values as required. The Insert method handles any errors that occur while adding the record.

The TryInsert method provides a slightly different (manual) alternative for adding new records.

Implementation: If Insert succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns the severity level of the last error it encountered while trying to add the record. See Error Class for more information on severity levels.

Return Data Type: BYTE

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP .                            !program map
GlobalErrors ErrorClass          !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
  END
InsertWindow WINDOW('Add a new Client'),AT(,,159,73),IMM,SYSTEM,GRAY
  PROMPT('&Name:'),AT(8,20),USE(?CLI:Name:Prompt)
  ENTRY(@s20),AT(61,20,84,10),USE(CLI:Name),MSG('Client Name'),REQ
  PROMPT('State Code:'),AT(8,34),USE(?CLI:StateCode:Prompt)
  ENTRY(@s2),AT(61,34,40,10),USE(CLI:StateCode),MSG('State Code')
  BUTTON('OK'),AT(12,53,45,14),USE(?OK),DEFAULT
  END CODE
!program code
ACCEPT
CASE FIELD()
OF ?OK
  IF EVENT() = Event:Accepted      !on OK button
  IF Access:Client.Insert() = Level:Benign !add the new Client record
  POST(Event:CloseWindow)         !if add succeeds, close down
  ELSE                             !if add fails
  Access:Client.CancelPrimeAutoInc !restore the file
  CYCLE                            !and start over
. .
!more code

```

See Also: TryInsert, PrimeRecord



---

```
ClientView VIEW(Order)           !declare order view
        PROJECT(ORD:Cust,ORD:ID,ORD:Date)
        END

CODE
!program code
ClientView{PROP:Order}=Access:Order.KeyToOrder(ORD:IDKey,2)!set runtime view order
!ClientView{PROP:Order}='ORD:ID,ORD:Date'           !equivalent to this
OPEN(ClientView)
SET(ClientView)
```

**Kill (shutdown the FileManager object)****Kill, VIRTUAL**

The **Kill** method disposes any memory allocated during the object's lifetime and performs any other necessary termination code.

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP                             !program map
  END

GlobalErrors ErrorClass          !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
  END

Client   FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
IDKey    KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record   RECORD,PRE()
ID       LONG
Name     STRING(20)
StateCode STRING(2)
  END
  END

CODE
GlobalErrors.Init               !initialize the GlobalErrors object
Access:Client.Init              !initialize Access:Client object
!program code
Access:Client.Kill              !shut down the Access:Client object
GlobalErrors.Kill

```

## Next (get next record in sequence)

### Next, PROC

The **Next** method gets the next record in sequence. The Next method handles any errors, except end of file, that occur while getting the record.

The TryNext method provides slightly different (manual) alternative for getting records in sequence.

Implementation: If Next succeeds, it returns Level:Benign (declared in ABERROR.INC). If it ultimately fails, it returns the severity level of the last error it encountered while trying to get the next record. See Error Class for more information on severity levels.

Return Data Type: BYTE

Example:

```
PROGRAM
INCLUDE('ABFILE.INC')           !declare FileManager class

Access:Client CLASS(FileManager) !derive Access:Client object
END

CODE
!program code
LOOP                             !loop through client file
CASE Access:Client.Next()        !get next record in sequence
OF Level:Notify OROF Level:Fatal !if error occurred
  POST(Event:CloseWindow)       !shut down
  BREAK
ELSE                             !otherwise
  PRINT(Rpt:Detail)             !print the record
END
END
```

See Also: TryNext

**Open (open the file)****Open, VIRTUAL, PROC**

The **Open** method tells the FileManager the calling procedure is using the file, then OPENS the file if it is not already open. The Open method handles any errors that occur while opening the file, including creating the file and rebuilding keys if necessary.

The TryOpen method provides slightly different (manual) alternative for opening files.

Implementation: If the file does not exist and the Create property is not zero, Open tries to create the file. If Open succeeds, it returns Level:Benign (declared in ABERROR.INC). If it ultimately fails, it returns the severity level of the last error it encountered while trying to open the file. See *Error Class* for more information on severity levels.

Return Data Type: BYTE

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class

GlobalErrors ErrorClass          !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init      PROCEDURE              !prototype Access:File init
      END

Client FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
      !file declaration
      END

CODE
GlobalErrors.Init                !initialize GlobalErrors object
Access:Client.Init              !initialize Access:Client object
Access:Client.Open              !open the Client file

!program code

Access:Client.Close             !close the Client file
Access:Client.Kill              !shut down the Access:Client object
GlobalErrors.Kill               !shut down the GlobalErrors object

```

See Also: Create, TryOpen

## Position (return the current record position)

### Position

The **Position** method returns the unique position of the current record.

The TryReget method retrieves a record based on the value returned by Position.

Implementation: Position returns the POSITION of the primary key if there is one; otherwise it returns the file POSITION. See the *Language Reference* for more information on POSITION.

Return Data Type: STRING

Example:

```
Hold = SELF.Position()
PUT( SELF.File )
CASE ERRORCODE()
OF NoError
OF RecordChangedErr
  SELF.SetError(Msg:ConcurrencyFailedFromForm)
  SELF.Throw
WATCH( SELF.File )
  SELF.TryReget(Hold)
ELSE
  SELF.SetError(Msg:PutFailed)
  RETURN SELF.Throw()
END
```

See Also: TryReget

**PostDelete(trigger delete action post-processing)****PostDelete**( *ErrCode*, *ErrMsg* ), *returncode*

**PostDelete** Returns confirmation that valid dictionary trigger activity has occurred after delete action, and optionally sets an error code and message to be processed.

*ErrCode* A string constant, variable, EQUATE, or expression that represents an error code.

*ErrMsg* A string constant, variable, EQUATE, or expression that represents an error message.

*returncode* Indicates if an error occurs.

The **PostDelete** method is a virtual method that returns a TRUE value by default if post delete trigger activity was processed normally. The developer must set the return level to FALSE if any problem occurred in the post delete trigger activity code. When PostDelete returns FALSE, an ERRORCODE 100 (trigger error) is posted. If *ErrCode* is set, then FILEERRORCODE will be set to the contents of *ErrCode* when the error is processed by the ErrorClass object for the associated file (table). Similarly, FILEERROR will be set to *ErrMsg*.

The PostDelete method is accessible from a table's trigger properties located in the Dictionary Editor, or, in the Global Embeds of a target application

Implementation: PostDelete is implemented using the file driver callback mechanism, therefore, this method will have access to all variables that the File Manager has access to. These variables and the ones added in the Data section of the method will allow the developer to insert code that will be executed after a DELETE action for a file.

Return Data Type: BYTE

Example:

```

CODE
!Push any pending errors on stack, to allow trigger error detection
PUSHERRORS()
ReturnValue = PARENT.PostDelete(ErrCode,ErrMsg) !returns TRUE by default
!trigger processing here - optionally set ErrCode, ErrMsg and ReturnValue
POPERRORS()
!restore errors saved on the stack
RETURN ReturnValue

```

See Also: PreDelete

PostInsert

PostUpdate

**PostInsert(trigger insert action post-processing)****PostInsert**( *ErrCode*, *ErrMsg* ), *returncode*


---

<b>PostInsert</b>	Returns confirmation that valid dictionary trigger activity has occurred after an insert action, and optionally sets an error code and message to be processed.
<i>ErrCode</i>	A string constant, variable, EQUATE, or expression that represents an error code.
<i>ErrMsg</i>	A string constant, variable, EQUATE, or expression that represents an error message.
<i>returncode</i>	Indicates if an error occurs.

The **PostInsert** method is a virtual method that returns a TRUE value by default if post insert trigger activity was processed normally. The developer must set the return level to FALSE if any problems occurred in the post insert trigger activity code. When **PostInsert** returns FALSE, an ERRORCODE 100 (trigger error) is posted. If *ErrCode* is set, then FILEERRORCODE will be set to the contents of *ErrCode* when the error is processed by the ErrorClass object for the associated file (table). Similarly, FILEERROR will be set to *ErrMsg*.

The PostInsert method is accessible from a table's trigger properties located in the Dictionary Editor, or, in the Global Embeds of a target application

Implementation: PostInsert is implemented using the file driver callback mechanism, therefore, this method will have access to all variables that the File Manager has access to. These variables and the ones added in the Data section of the method will allow the developer to insert code that will be executed after an INSERT action for a file.

Return Data Type: BYTE

Example:

```

CODE
!Push any pending errors on stack, to allow trigger error detection
PUSHERRORS()
ReturnValue = PARENT.PostInsert(ErrCode,ErrMsg) !returns TRUE by default
!trigger processing here - optionally set ErrCode, ErrMsg and ReturnValue
POPERRORS()
!restore errors saved on the stack
RETURN ReturnValue

```

See Also:       PreDelete  
                   PostDelete  
                   PostUpdate

**PostUpdate(trigger update action post-processing)****PostUpdate**( *ErrCode*, *ErrMsg* ), *returncode*


---

<b>PostUpdate</b>	Returns confirmation that valid dictionary trigger activity has occurred after an insert action, and optionally sets an error code and message to be processed.
<i>ErrCode</i>	A string constant, variable, EQUATE, or expression that represents an error code.
<i>ErrMsg</i>	A string constant, variable, EQUATE, or expression that represents an error message.
<i>returncode</i>	Indicates if an error occurs.

The **PostUpdate** method is a virtual method that returns a TRUE value by default if post update trigger activity was processed normally. The developer must set the return level to FALSE if any problems occurred in the post update trigger activity code. When PostUpdate returns FALSE, an ERRORCODE 100 (trigger error) is posted. If *ErrCode* is set, then FILEERRORCODE will be set to the contents of *ErrCode* when the error is processed by the ErrorClass object for the associated file (table). Similarly, FILEERROR will be set to *ErrMsg*.

The PostUpdate method is accessible from a table's trigger properties located in the Dictionary Editor, or, in the Global Embeds of a target application

Implementation: PostUpdate is implemented using the file driver callback mechanism, therefore, this method will have access to all variables that the File Manager has access to. These variables and the ones added in the Data section of the method will allow the developer to insert code that will be executed after a CHANGE action for a file.

Return Data Type: BYTE

Example:

```

CODE
!Push any pending errors on stack, to allow trigger error detection
PUSHERRORS()
ReturnValue = PARENT.PostUpdate(ErrCode,ErrMsg) !returns TRUE by default
!trigger processing here - optionally set ErrCode, ErrMsg and ReturnValue
POPERRORS()
!restore errors saved on the stack
RETURN ReturnValue

```

See Also:       PreDelete  
                   PostDelete  
                   PostInsert

**PreDelete(trigger delete action pre-processing)****PreDelete**( *ErrCode*, *ErrMsg* ), *returncode*


---

<b>PreDelete</b>	Returns confirmation that valid dictionary trigger activity has occurred before a delete action is executed, and optionally sets an error code and message to be processed.
<i>ErrCode</i>	A string constant, variable, EQUATE, or expression that represents an error code.
<i>ErrMsg</i>	A string constant, variable, EQUATE, or expression that represents an error message.
<i>returncode</i>	Indicates if an error occurs.

The **PreDelete** method is a virtual method that returns a TRUE value by default if pre-delete trigger activity was processed normally. The developer must set the return level to FALSE if any problems occurred in the pre-delete trigger activity code. When **PreDelete** returns FALSE, an ERRORCODE 100 (trigger error) is posted. If *ErrCode* is set, then FILEERRORCODE will be set to the contents of *ErrCode* when the error is processed by the ErrorClass object for the associated file (table). Similarly, FILEERROR will be set to *ErrMsg*.

The **PreDelete** method is accessible from a table's trigger properties located in the Dictionary Editor, or, in the Global Embeds of a target application

Implementation: **PreDelete** is implemented using the file driver callback mechanism, therefore, this method will have access to all variables that the File Manager has access to. These variables and the ones added in the Data section of the method will allow the developer to insert code that will be executed after a CHANGE action for a file.

Return Data Type: BYTE

Example:

```

CODE
!Push any pending errors on stack, to allow trigger error detection
PUSHERRORS()
ReturnValue = PARENT.PreDelete(ErrCode,ErrMsg) !returns TRUE by default
!trigger processing here - optionally set ErrCode, ErrMsg and ReturnValue
POPERRORS()
!restore errors saved on the stack
RETURN ReturnValue

```

See Also: PostDelete

PreInsert

PreUpdate

**PreInsert(trigger insert action pre-processing)**

**PreInsert**(*OpCode, AddLength ErrCode, ErrMsg*), *returncode*

---

<b>PreInsert</b>	Returns confirmation that valid dictionary trigger activity has occurred before an insert action, and optionally sets an error code and message to be processed.
<i>OpCode</i>	A SIGNED integer that indicates the type of ADD that will be attempted.
<i>AddLength</i>	An UNSIGNED integer that indicates the record length about to be added when the ADD ( File, Length) mode is active.
<i>ErrCode</i>	A string constant, variable, EQUATE, or expression that represents an error code.
<i>ErrMsg</i>	A string constant, variable, EQUATE, or expression that represents an error message.
<i>returncode</i>	Indicates if an error occurs.

The **PreInsert** method is a virtual method that returns a TRUE value by default if pre-insert trigger activity was processed normally. The developer must set the return level to FALSE if any problems occurred in the pre-insert trigger activity code. When **PreInsert** returns FALSE, an ERRORCODE 100 (trigger error) is posted. If *ErrCode* is set, then FILEERRORCODE will be set to the contents of *ErrCode* when the error is processed by the ErrorClass object for the associated file (table). Similarly, FILEERROR will be set to *ErrMsg*.

The *OpCode* and *AddLength* parameters can be used in your pre-processing trigger code.

DriverOp:ADD	ADD(FILE)
DriverOp:Append	APPEND(FILE)
DriverOP:AddLen	ADD(FILE,LENGTH)
DriverOp:AppendLen	APPEND(FILE,LENGTH)

Use the Equates provided in EQUATES.CLW (shown above) to test the *OpCode*. *AddLength* is used to return the value of the length parameter if used with ADD.

The **PreInsert** method is accessible from a table's trigger properties located in the Dictionary Editor, or, in the Global Embeds of a target application

Implementation: PreInsert is implemented using the file driver callback mechanism, therefore, this method will have access to all variables that the File Manager has access to. These variables and the ones added in the Data section of the method will allow the developer to insert code that will be executed before an INSERT action for a file.

Return Data Type: BYTE

Example:

```
CODE
PUSHERRORS()
ReturnValue = PARENT.PreInsert(OpCode,AddLen,ErrCode,ErrMsg)
  MESSAGE('Trigger Test Before Insert')
  !Trigger code entered here
POPERRORS()
RETURN ReturnValue
```

See Also:            PostInsert, PreDelete, PreUpdate

**PreUpdate(trigger update action pre-processing)**

**PreUpdate**( *Pointer*, *PutLength*, *ErrCode*, *ErrMsg* ), *returncode*

**PreUpdate** Returns confirmation that valid dictionary trigger activity has occurred before an attempted action, and optionally sets an error code and message to be processed.

*Pointer* A LONG that represents the file pointer to be written if PUT ( File, Pointer) or PUT (File, Pointer, Length) is used.

*PutLength* An UNSIGNED integer that represents the number of bytes to write to the file when PUT (File, Pointer, Length) is used.

*ErrCode* A string constant, variable, EQUATE, or expression that represents an error code.

*ErrMsg* A string constant, variable, EQUATE, or expression that represents an error message.

*returncode* Indicates if an error occurs.

The **PreUpdate** method is a virtual method that returns a TRUE value by default if pre-update trigger activity was processed normally. The developer must set the return level to FALSE if any problems occurred in the pre-update trigger activity code. When **PreUpdate** returns FALSE, an ERRORCODE 100 (trigger error) is posted. If *ErrCode* is set, then FILEERRORCODE will be set to the contents of *ErrCode* when the error is processed by the ErrorClass object for the associated file (table). Similarly, FILEERROR will be set to *ErrMsg*.

The **PreUpdate** method is accessible from a table's trigger properties located in the Dictionary Editor, or, in the Global Embeds of a target application

Implementation: PreUpdate is implemented using the file driver callback mechanism, therefore, this method will have access to all variables that the File Manager has access to. These variables and the ones added in the Data section of the method will allow the developer to insert code that will be executed after a CHANGE action for a file.

Return Data Type: BYTE

Example:

```
CODE
!Push any pending errors on stack, to allow trigger error detection
PUSHERRORS()
  ReturnValue = PARENT.PreUpdate(ErrCode,ErrMsg) !returns TRUE by default
  !trigger processing here - optionally set ErrCode, ErrMsg and ReturnValue
POPERRORS()
!restore errors saved on the stack
RETURN ReturnValue
```

See Also:       PreDelete  
                  PostDelete  
                  PostInsert

**Previous (get previous record in sequence)****Previous, PROC**

The **Previous** method gets the previous record in sequence. The Previous method handles any errors that occur while getting the record.

The TryPrevious method provides a slightly different (manual) alternative for getting records in sequence.

Implementation: If Previous succeeds, it returns Level:Benign (declared in ABERROR.INC). If it ultimately fails, it returns the severity level of the last error it encountered while trying to get the previous record. See *Error Class* for more information on severity levels.

Return Data Type: BYTE

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')                !declare FileManager class

Access:Client CLASS(FileManager)      !derive Access:Client object
  END

CODE
  !program code
LOOP                                  !loop through client file
  CASE Access:Client.Previous()        !get previous record in sequence
  OF Level:Notify OROF Level:Fatal     !if error occurred
    POST(Event:CloseWindow)           !shut down
    BREAK
  ELSE                                  !otherwise
    PRINT(Rpt:Detail)                 !print the record
  END
END

```

See Also: TryPrevious

**PrimeAutoInc (prepare an autoincremented record for adding)****PrimeAutoInc, VIRTUAL, PROC**

When a record is inserted, the **PrimeAutoInc** method prepares an autoincremented record for adding to the managed file and handles any errors it encounters. If you want to provide an update form that displays the auto-incremented record ID or where RI is used to keep track of children, then you should use the PrimeAutoInc method to prepare the record buffer.

The TryPrimeAutoInc method provides a slightly different (manual) alternative for preparing autoincremented records.

The CancelAutoInc method restores the managed file to its pre-PrimeAutoInc state.

Implementation: The PrimeRecord method calls PrimeAutoInc if the file contains an autoincrementing key.

If PrimeAutoInc succeeds, it returns Level:Benign (declared in ABERROR.INC). If it ultimately fails, it returns the severity level of the error it encountered while trying to prime the record. See Error Class for more information on severity levels.

Return Data Type: BYTE

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP                             !program map
  END
GlobalErrors ErrorClass          !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init PROCEDURE                   !initialize Access:File object
PrimeAutoInc PROCEDURE,VIRTUAL  !prepare new record for adding
  END

Client FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
IDKey KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record RECORD,PRE()
ID LONG
Name STRING(20)
StateCode STRING(2)
  END
  END

InsertWindow WINDOW('Add a new Client'),AT(, ,159,73),IMM,SYSTEM,GRAY
  PROMPT('&Name: '),AT(8,20),USE(?CLI:Name:Prompt)

```

```

ENTRY(@s20),AT(61,20,84,10),USE(CLI:Name),MSG('Client Name'),REQ
PROMPT('State Code:'),AT(8,34),USE(?CLI:StateCode:Prompt)
ENTRY(@s2),AT(61,34,40,10),USE(CLI:StateCode),MSG('State Code')
BUTTON('OK'),AT(12,53,45,14),USE(?OK),DEFAULT
BUTTON('Cancel'),AT(82,53,45,14),USE(?Cancel)
END

CODE
GlobalErrors.Init           !initialize GlobalErrors object
Access:Client.Init          !initialize Access:Client object
Access:Client.Open          !open the Client file
IF Access:Client.PrimeAutoInc() !prime Client record
  POST(Event:CloseWindow)    !if prime fails, close down
END

OPEN(InsertWindow)
ACCEPT
CASE FIELD()
OF ?OK
  IF EVENT() = Event:Accepted !on OK button
    IF Access:Client.Insert() = Level:Benign !finish adding the new Client record
      POST(Event:CloseWindow) !if add succeeds, close down
    ELSE
      SELECT(?CLI:Name:Prompt) !select client name field
      CYCLE !and start over
    END
  END
OF ?Cancel
  IF EVENT() = EVENT:Accepted !on Cancel button
    Access:Client.CancelAutoInc !restore Client to pre-PrimeRecord
    POST(Event:CloseWindow) !close down
  END
END
END

Access:Client.Close          !close the Client file
Access:Client.Kill          !shut down the Access:Client object
GlobalErrors.Kill          !shut down the GlobalErrors object
RETURN

Access:Client.PrimeAutoInc PROCEDURE
CODE
!your custom code here
PARENT.PrimeAutoInc         !call the base class method
!your custom code here

```

See Also:           CancelAutoInc, PrimeRecord, TryPrimeAutoInc

## PrimeFields (a virtual to prime fields)

### PrimeFields, VIRTUAL

The **PrimeFields** method is a virtual placeholder method to prime fields before adding a record.

Implementation: The ABC Templates use the PrimeFields method to implement field priming specified in the Data Dictionary.

The PrimeRecord method calls the PrimeFields method before calling the PrimeAutoInc method. You can use the PrimeRecord method to prime the nonincrementing components of an autoincrementing key.

Example:

```
Access:Customer.PrimeFields PROCEDURE  
CODE  
CLI:StateCode = 'FL'
```

**PrimeRecord (prepare a record for adding:FileManager)**

**PrimeRecord**( [*suppress clear*] ), VIRTUAL, PROC

**PrimeRecord** Prepares a record for adding to the managed file.

*suppress clear* An integer constant, variable, EQUATE, or expression that indicates whether or not to clear the record buffer. A value of zero (0 or False) clears the buffer; a value of one (1 or True) does not clear the buffer. If omitted, *suppress clear* defaults to zero (0).

The **PrimeRecord** method prepares a record for adding to the managed file and returns a value indicating success or failure. A return value of Level:Benign indicates success; any other return value indicates a problem.

Implementation: PrimeRecord prepares the record by optionally clearing the record buffer, then calling the PrimeFields method to prime field values, and the PrimeAutoInc method to increment autoincrementing key values. If it succeeds, it returns Level:Benign (declared in ABERROR.INC), otherwise it returns the severity level of the last error it encountered. See *Error Class* for more information on severity levels. The *suppress clear* parameter lets you clear or retain any other values in the record buffer.

Return Data Type: BYTE

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP                             !program map
  END

GlobalErrors ErrorClass          !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init      PROCEDURE              !initialize Access:File object
PrimeAutoInc PROCEDURE,VIRTUAL   !prepare new record for adding
      END

Client      FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
IDKey      KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record     RECORD,PRE()
ID         LONG
Name       STRING(20)
StateCode  STRING(2)
      END
      END

```

```

InsertWindow WINDOW('Add a new Client'),AT(,,159,73),IMM,SYSTEM,GRAY
    PROMPT('&Name:'),AT(8,20),USE(?CLI:Name:Prompt)
    ENTRY(@s20),AT(61,20,84,10),USE(CLI:Name),MSG('Client Name'),REQ
    PROMPT('State Code:'),AT(8,34),USE(?CLI:StateCode:Prompt)
    ENTRY(@s2),AT(61,34,40,10),USE(CLI:StateCode),MSG('State Code')
    BUTTON('OK'),AT(12,53,45,14),USE(?OK),DEFAULT
    BUTTON('Cancel'),AT(82,53,45,14),USE(?Cancel)
END

CODE
GlobalErrors.Init           !initialize GlobalErrors object
Access:Client.Init         !initialize Access:Client object
Access:Client.Open         !open the Client file
IF Access:Client.PrimeRecord() !prime Client record
    POST(Event:CloseWindow)   !if prime fails, close down
END
OPEN(InsertWindow)
ACCEPT
CASE FIELD()
OF ?OK
    IF EVENT() = Event:Accepted !on OK button
        IF Access:Client.Insert() = Level:Benign !finish adding the new Client record
            POST(Event:CloseWindow) !if add succeeds, close down
        ELSE !if add fails
            SELECT(?CLI:Name:Prompt) !select client name field
            CYCLE !and start over
        END
    END
OF ?Cancel
    IF EVENT() = EVENT:Accepted !on Cancel button
        Access:Client.CancelAutoInc !restore Client to pre-PrimeRecord
        POST(Event:CloseWindow) !close down
    END
END
EMD
END

Access:Client.Close         !close the Client file
Access:Client.Kill         !shut down the Access:Client object
GlobalErrors.Kill         !shut down the GlobalErrors object
RETURN

Access:Client.PrimeAutoInc PROCEDURE
CODE
!your custom code here
PARENT.PrimeAutoInc         !call the base class method
!your custom code here

```

See Also: PrimeAutoInc, CancelAutoInc

**RestoreBuffer (restore a previously saved record buffer)**

**RestoreBuffer**( *buffer id* [, *restore*] )

**RestoreBuffer** Restores previously saved record buffer contents.

*buffer id* An integer constant, variable, EQUATE, or expression that identifies the buffer contents to restore--this is a value returned by the SaveBuffer method.

*restore* An integer constant, variable, EQUATE, or expression that indicates whether to restore the managed file's buffer contents, or simply DISPOSE of the specified buffer. A value of one (1 or True) updates the file's Buffer; a value of zero (0 or False) does not update the file's Buffer. If omitted, *restore* defaults to True.

The **RestoreBuffer** method restores record buffer contents to the managed file's record buffer (the Buffer property). RestoreBuffer restores values previously saved by the SaveBuffer method, including MEMO fields.

Implementation: The RestoreBuffer method releases memory allocated by the SaveBuffer method. Therefore, to prevent a memory leak, each call to SaveBuffer should be paired with a corresponding call to RestoreBuffer. The RestoreBuffer method retrieves and DISPOSEs the specified contents from the Buffers property.

Example:

```
FileManager.RestoreFile PROCEDURE(*USHORT Id)
  CODE
  IF ~SELF.UseFile()
    SELF.Saved.Id = Id
    GET(SELF.Saved,SELF.Saved.Id)
    ASSERT(~ERRORCODE())
    IF SELF.Saved.Key &= NULL
      RESET(SELF.File,SELF.Saved.Pos)
    ELSE
      RESET(SELF.Saved.Key,SELF.Saved.Pos)
    END
  END
  IF SELF.Saved.WHeld
    HOLD(SELF.File)
  END
  IF SELF.Saved.WWatch
    WATCH(SELF.File)
  END
  NEXT(SELF.File)
  SELF.RestoreBuffer(SELF.Saved.Buffer)
  DELETE(SELF.Saved)
  Id = 0
END
```

See Also: Buffer, Buffers, SaveBuffer

## RestoreFile (restore a previously saved file state)

**RestoreFile**( *filestateid*, *dorestore* )

---

<b>RestoreFile</b>	Restores a previously saved file state.
<i>filestateid</i>	A USHORT returned by the SaveFile method that identifies the file state to restore.
<i>dorestore</i>	An boolean integer constant, variable, EQUATE, or expression to indicate whether or not to perform the restoration. If omitted, the default of TRUE (1) is implied.

The **RestoreFile** method restores the specified file state for the managed file. RestoreFile restores from states previously saved by the SaveFile method.

Implementation: The RestoreFile method restores file position, as well as any active HOLD or WATCH. RestoreFile calls the RestoreBuffer method to restore the managed file's record buffer contents.

Example:

```
SaveState USHORT           !must be a USHORT
CODE
SaveState = Access:MyFile.SaveFile() !save the file state
SET(MyKey,MyKey)           !access the file (change the file state)
LOOP UNTIL Access:MyFile.Next()
    !Check range limits here
    !Process the record here
END
Access:MyFile.RestoreFile(SaveState) !restore the previously saved file state
```

See Also: SaveFile, RestoreBuffer

**SaveBuffer (save a copy of the record buffer)****SaveBuffer**

The **SaveBuffer** method saves a copy of the managed file's record buffer contents (the Buffer property) and returns a number that uniquely identifies the saved record. SaveBuffer stores buffer contents for subsequent retrieval by the RestoreBuffer method.

Implementation: SaveBuffer saves MEMO contents as well as other fields.

SaveBuffer allocates memory which is subsequently released by the RestoreBuffer method. Therefore, to prevent a memory leak, each call to SaveBuffer should be paired with a corresponding call to RestoreBuffer.

Return Data Type: USHORT

Example:

```

FileManager.SaveFile PROCEDURE
Id LONG,AUTO
I SHORT,AUTO
CODE
  Id = RECORDS(SELF.Saved)
  IF Id
    GET(SELF.Saved,Id)
    ASSERT(~ERRORCODE())
    Id = SELF.Saved.Id + 1
  ELSE
    Id = 1
  END
  SELF.Saved.Id = Id
  SELF.Saved.Buffer = SELF.SaveBuffer()
  SELF.Saved.Key &= SELF.File{PROP:CurrentKey}
  SELF.Saved.WHeld = SELF.File{PROP:Held}
  SELF.Saved.WWatch = SELF.File{PROP:Watched}
  IF SELF.Saved.Key &= NULL
    SELF.Saved.Pos = POSITION(SELF.File)
  ELSE
    SELF.Saved.Pos = POSITION(SELF.Saved.Key)
  END
  ADD(SELF.Saved)
  RETURN Id

```

See Also: Buffer, Buffers, RestoreBuffer

## SaveFile (save the current file state)

### SaveFile

The **SaveFile** method saves the managed file's current state and returns a number that uniquely identifies the saved state. SaveFile saves the managed file's state for subsequent restoration by the RestoreFile method.

Implementation: The SaveFile method saves file position, as well as any active HOLD or WATCH. SaveFile calls the SaveBuffer method to save a copy of the managed file's record buffer contents.

Return Data Type: USHORT

Example:

```
SaveState USHORT           !must be a USHORT
CODE
SaveState = Access:MyFile.SaveFile() !save the file state
SET(MyKey,MyKey)           !access the file (change the file state)
LOOP UNTIL Access:MyFile.Next()
    !Check range limits here
    !Process the record here
END
Access:MyFile.RestoreFile(SaveState) !restore the previously saved file state
```

See Also: RestoreFile, SaveBuffer

**SetError (save the specified error and underlying error state)****SetError**( *error id* )

---

**SetError** Saves the specified error and the underlying error state for use by the Throw method, etc.

*error id* A numeric constant, variable, EQUATE, or expression that identifies the error. See *Error Class* for more information on error id.

The **SetError** method saves the specified error and underlying error state for use by the Throw method, etc.

Example:

```
Access:Client.Next FUNCTION(BYTE HandleError) !Next function
CODE                                     !with alternative error handling
LOOP
NEXT( SELF.File )                       !get the next record
CASE ERRORCODE()                         !check for error conditions
OF BadRecErr OROF NoError
RETURN Level:Benign
OF IsHeldErr                             !if record is HELD by another
SELF.SetError(Msg:RecordHeld)           !make RecordHeld the current error
IF HandleError                           !if interactive error handling
RETURN SELF.Throw()                     !pass current error to error handler
ELSE                                      !otherwise (silent error handling)
RETURN Level:Notify                      !return error code to caller
END
END
END
```

See Also:        Throw

**SetErrors (set the error class used)****SetErrors**( *ErrorClass* )**SetErrors**      Sets the Error Class that the file manager uses.*ErrorClass*      The label of the ErrorClass object.

The new SetErrors() method of the file manager allows you to set the error class that the file manager uses after it has been initialized.

Example:

```

DLLInitializer.Construct PROCEDURE

CODE
!Initialize the local INI manager to use windows INI file
LocalINIMgr.Init('allfiles.INI', NVD_INI)

INIMgr &= LocalINIMgr

FuzzyMatcher.Init                                    ! Init the browse 'fuzzy matcher'
FuzzyMatcher.SetOption(MatchOption:NoCase, 1) ! Configure case matching
FuzzyMatcher.SetOption(MatchOption:WordOnly, 0)! Configure 'word only' matching

!These procedures are used to initialize the DLL.
!It must be called by the main executable when it starts up

allfiles:Init PROCEDURE(<ErrorClass curGlobalErrors>, <INIClass curINIMgr>)

CODE
IF ~curGlobalErrors &= NULL
  GlobalErrors &= curGlobalErrors
END
IF ~curINIMgr &= NULL
  INIMgr &= curINIMgr
END
Access:Customer.SetErrors(GlobalErrors)
Access:Orders.SetErrors(GlobalErrors)
Access:Detail.SetErrors(GlobalErrors)
Access:Products.SetErrors(GlobalErrors)
Access:Phones.SetErrors(GlobalErrors)

```

## SetKey (set current key)

**SetKey**( *key* ), PROTECTED

---

**SetKey**            Makes the specified key current for use by other FileManager methods.

*key*                The label of the KEY.

The **SetKey** method makes the specified key the current one for use by other FileManager methods.

Example:

```
FileManager.GetComponents FUNCTION(KEY K) !returns the number of key components
CODE
  SELF.SetKey(K)                            !locate the specified key
RETURN RECORDS( SELF.Keys.Fields )        !count the components
```

**SetName (set current filename)**

**SetName**( *filename* )

---

<b>SetName</b>	Sets the variable filename of the managed file.
<i>filename</i>	A string constant, variable, EQUATE, or expression that contains the filename of the managed file.

The **SetName** method sets the variable filename (NAME attribute) of the managed file. This value determines which file is actually opened and processed by the FileManager object. The filename is also displayed in error messages, etc.

The GetName method returns the name of the managed file.

Implementation: SetName assumes the FileName property is a reference to the file's NAME attribute variable.

Example:

```

PROGRAM
  INCLUDE( 'ABFILE.INC' )           !declare FileManager class
  MAP .                             !program map
  ClientFile STRING(8)              !client filename variable
  GlobalErrors ErrorClass           !declare GlobalErrors object
  Access:Client CLASS(FileManager) !derive Access:Client object
  Init      PROCEDURE               !initialize Access:File object
  END

Client      FILE,DRIVER('TOPSPEED'),PRE(CLI),THREAD,NAME(ClientFile)
IDKey       KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record      RECORD,PE()
ID          LONG
Name        STRING(20)
StateCode   STRING(2)
            END
            END

CODE
GlobalErrors.Init      !initialize the GlobalErrors object
Access:Client.Init     !initialize the Access:Client object
LOOP I# = 1 TO 12      !step through 12 monthly files
  Access:Client.SetName('Client'&I#) !set the filename variable
  Access:Client.Open   !open the monthly file
  !process the file
  Access:Client.Close  !close the monthly file
END

```

Access:Client.Init PROCEDURE

CODE

```
PARENT.Init(GlobalErrors)    !call the base class Init method
SELF.File &= Client          !point Access:Client to Client file
SELF.Buffer &= CLI:Record    !point Access:Client to Client buffer
SELF.FileName &= ClientFile  !point Access:Client to the filename variable
```

See Also:            FileName, FileNameValue, GetName

**Throw (pass an error to the error handler for processing)****Throw**( [error id] ), VIRTUAL, PROC

---

**Throw** Passes the specified error to the error handler object for processing.

*error id* A numeric constant, variable, EQUATE, or expression that indicates the error to process. If omitted, Throw processes the current error--that is, the error identified by the previous call to SetError or Throw.

The **Throw** method passes the current (last encountered) error to the nominated error handler for processing, including FILEERROR() and FILEERRORCODE() values, then returns the severity level of the error.

Implementation: The SetError method saves the specified error and underlying error state for use by the Throw method. See *Error Class* for more information on error ids and severity levels.

The Init method receives and sets the error handler object.

Return Data Type: BYTE

Example:

```
Access:Client.Next FUNCTION(BYTE HandleError)!Next function
CODE                                     !with alternative error handling
LOOP
  NEXT( SELF.File )                     !get the next record
  CASE ERRORCOD()                       !check for error conditions
  OF BadRecErr OROF NoError
    RETURN Level:Benign
  OF IsHeldErr                           !if record is HELD by another
    SELF.SetError(Msg:RecordHeld)        !make RecordHeld the current error
  IF HandleError                         !if interactive error handling
    RETURN SELF.Throw()                 !pass current error to error handler
  ELSE                                    !otherwise (silent error handling)
    RETURN Level:Notify                 !return error code to caller
  END
END
END
```

See Also: Init, SetError

**ThrowMessage (pass an error and text to the error handler)****ThrowMessage( *error id*, *text* ), VIRTUAL, PROC****ThrowMessage**

Passes the specified error and text to the error handler object for processing.

*error id* A numeric constant, variable, EQUATE, or expression that indicates the error to process.*text* A string constant, variable, EQUATE, or expression to include in the error message.

The **ThrowMessage** method passes the specified error, including FILEERROR() and FILEERRORCODE() values, and text to the error handler object for processing, then returns the severity level of the error. See *Error Class* for more about error ids and severity levels.

Implementation: The Init method receives and sets the error handler. The incorporation of the *text* into the error message depends on the error handler. See *Error Class*.

Return Data Type: BYTE

Example:

```
GlobalErrors ErrorClass                                !declare GlobalErrors object

Access:Client CLASS(FileManager)                      !derive Access:Client object
ValidateField FUNCTION(UNSIGNED Id),BYTE,VIRTUAL !prototype Access:File validation
END

Client FILE,DRIVER('TOPSPEED'),PRE(CLI),THREAD
IDKey KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record RECORD,PRE()
ID LONG
Name STRING(20)
StateCode STRING(2)
END
END
```

```
CODE
!program code

Access:Client.ValidateField FUNCTION(UNSIGNED Id)
CODE
IF ID = 3                !validate statecode (3rd) field
  IF ~CLI:StateCode      !if no statecode,
    !pass error & text to error handler
    RETURN Access:Client.ThrowMessage(Msg:RequiredField,'StateCode')
  END
END
RETURN(LEVEL:Notify)
```

See Also:        Init

**TryFetch (try to get a specific record by key value)**

**TryFetch**( *key* ), VIRTUAL, PROC

---

**TryFetch** Gets a specific record by its key value.

*key* The label of the primed KEY.

The **TryFetch** method gets a specific record by its key value. You must prime the key before calling TryFetch. If the key is not unique, TryFetch gets the first record with the specified key value.

The Fetch method provides a slightly different (automatic) alternative for fetching specific records.

Implementation: Fetch tries to get the specified record. If it succeeds, it returns Level:Benign. If it fails, it returns Level:Notify and does not clear the record buffer. See *Error Class* for more information on Level:Benign and Level:Notify.

Return Data Type: BYTE

Example:

```

PROGRAM
  INCLUDE( 'ABFILE.INC' )           !declare FileManager class
  MAP                               !program map
  END
GlobalErrors  ErrorClass           !declare GlobalErrors object
Access:States CLASS(FileManager)  !declare Access:States object
  END

States        FILE,DRIVER( 'TOPSPEED' ),PRE( ST ),CREATE,BINDABLE,TREAD
StateCodeKey  KEY( ST:StateCode ),NOCASE,OPT,PRIMARY
Record        RECORD,PRE( )
StateCode     STRING( 2 )
State         STRING( 20 )
  END
  END

CODE
!program code
!get the state record for Florida
ST:StateCode = 'FL'                !prime the state key for the fetch
IF Access:States.TryFetch(ST:StateCodeKey)!fetch the record
  GlobalErrors.Throw(Msg:FieldNotInFile) !handle any errors yourself
END

```

See Also: Fetch

**TryInsert (try to add a new record)****TryInsert, PROC**

The **TryInsert** method adds a new record to the file, making sure the record is valid, and automatically incrementing key values as required. The TryInsert method does not attempt to handle errors.

The Insert method provides a slightly different (automatic) alternative for adding records.

Implementation: TryInsert tries to add the record. If it succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns the severity level of the error it encountered while trying to add the record. See *Error Class* for more information on severity levels.

Return Data Type: BYTE

Example:

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP .                             !program map
  GlobalErrors ErrorClass          !declare GlobalErrors object
  Access:Client CLASS(FileManager) !derive Access:Client object
  END
  InsertWindow WINDOW('Add a new Client'),AT(,,159,73),IMM,SYSTEM,GRAY
    PROMPT('&Name:'),AT(8,20),USE(?CLI:Name:Prompt)
    ENTRY(@s0),AT(61,20,84,10),USE(CLI:Name),MSG('Client Name'),REQ
    PROMPT('State Code:'),AT(8,34),USE(?CLI:StateCode:Prompt)
    ENTRY(@s2),AT(61,34,40,10),USE(CLI:StateCode),MSG('State Code')
    BUTTON('OK'),AT(12,53,45,14),USE(?OK),DEFAULT
  END
CODE
!program code
ACCEPT
CASE FIELD()
OF ?OK
  IF EVENT() = Event:Accepted      !on OK button
    IF Access:Client.TryInsert() = Level:Benign !add the new Client record
      POST(Event:CloseWindow)        !if add succeeds, close down
    ELSE                             !if add fails
      Access:Client.Throw(Msg:InsertFailed) !handle the error
      Access:Client.CancelPrimeAutoInc !restore the file
    CYCLE                             !and start over
  END
END
!more code

```

See Also: Insert, PrimeRecord

**TryNext (try to get next record in sequence)****TryNext, PROC**

The **TryNext** method gets the next record in sequence. The TryNext method does not attempt to handle errors that occur while getting the next record.

The Next method provides a slightly different (automatic) alternative for getting records in sequence.

Implementation: TryNext tries to get the next record. If it succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns the severity level of the error it encountered while trying to get the record. See *Error Class* for more information on severity levels.

Return Data Type: BYTE

Example:

```
PROGRAM BatchReport                !batch process--don't display errors
  INCLUDE('ABFILE.INC')            !declare FileManager class
  MAP                               !program map
  END

GlobalErrors ErrorClass            !declare GlobalErrors object
Access:Client CLASS(FileManager)  !derive Access:Client object
  END

CODE
!program code
LOOP                               !loop through client file
  CASE Access:Cliet.TryNext()      !get next record in sequence
  OF Level:Notify OROF Level:Fatal !if error occurred
    POST(Event:CloseWindow)       !shut down
    BREAK
  ELSE                              !otherwise
    PRINT(Rpt:Detail)              !print the record
  END
END
```

See Also: Next

**TryOpen (try to open the file)****TryOpen, PROC**

The **TryOpen** method tells the FileManager the calling procedure is using the file, then OPENS the file if it is not already open. The TryOpen method does not attempt to handle errors that occur while opening the file.

The Open method provides a slightly different (automatic) alternative for opening files.

Implementation: TryOpen tries to open the file. If it succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns the severity level of the error it encountered while trying to open the file. See *Error Class* for more information on severity levels.

Return Data Type: BYTE

Example:

```
PROGRAM
  INCLUDE('ABFILE.INC')           !declare FileManager class
  MAP                             !program map
  END

GlobalErrors ErrorClass          !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init      PROCEDURE              !prototype Access:File init
      END

Client      FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
      END

CODE
GlobalErrors.Init                !initialize GlobalErrors object
Access:Client.Init              !initialize Access:Client object
IF Access:Client.TryOpen        !try to open the Client file
  MESSAGE('Could not open the Client file') !handle the error yourself
  RETURN
END

!program code

Access:Client.Close             !close the Client file
Access:Client.Kill              !shut down the Access:Client object
GlobalErrors.Kill               !shut down the GlobalErrors object
```

See Also: Open

**TryPrevious (try to get previous record in sequence)****TryPrevious, PROC**

The **TryPrevious** method gets the previous record in sequence. The TryPrevious method does not attempt to handle errors that occur while getting the previous record.

The Previous method provides a slightly different (automatic) alternative for getting records in sequence.

Implementation: TryPrevious tries to get the previous record. If it succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns the severity level of the error it encountered while trying to get the record. See *Error Class* for more information on severity levels.

Return Data Type: BYTE

Example:

```
PROGRAM BatchReport           !batch report--don't display errors
  INCLUDE('ABFILE.INC')       !declare FileManager class
  MAP                          !program map
  END

GlobalErrors ErrorClass      !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
  END

  CODE
  !program code
  LOOP                          !loop through client file
    CASE Access:Client.TryPrevious() !get previous record in sequence
    OF Level:Notify OROF Level:Fatal !if error occurred
      POST(Event:CloseWindow)      !shut down
      BREAK
    ELSE                            !otherwise
      PRINT(Rpt:Detail)            !print the record
    END
  END
END
```

See Also: Previous

**TryPrimeAutoInc (try to prepare an autoincremented record for adding)****TryPrimeAutoInc, VIRTUAL, PROC**

When a record is Inserted, the **TryPrimeAutoInc** method prepares an autoincremented record for adding to the managed file. The TryPrimeAutoInc method does not handle any errors it encounters.

The PrimeAutoInc method provides a slightly different (automatic) alternative for preparing autoincremented records.

The CancelAutoInc method restores the managed file to its pre-TryPrimeAutoInc state.

Implementation: TryPrimeAutoInc tries to prime the record. If it succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns the severity level of the error it encountered while trying to prime the record. See *Error Class* for more information on severity levels.

Return Data Type: BYTE

Example:

```

PROGRAM
INCLUDE('ABFILE.INC')           !declare FileManager class
MAP                             !program map
END

GlobalErrors ErrorClass        !declare GlobalErrors object
Access:Client CLASS(FileManager) !derive Access:Client object
Init      PROCEDURE            !initialize Access:File object
PrimeAutoInc PROCEDURE,VIRTUAL !prepare new record for adding
      END

Client      FILE,DRIVER('TOPSPEED'),PRE(CLI),CREATE,BINDABLE,THREAD
IDKey      KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record     RECORD,PRE()
ID         LONG
Name       STRING(20)
StateCode  STRING(2)
      END
      END

InsertWindow WINDOW('Add a new Client'),AT(,,159,73),IMM,SYSTEM,GRAY
      PROMPT('&Name:'),AT(8,20),USE(?CLI:Name:Prompt)
      ENTRY(@s20),AT(61,20,84,10),USE(CLI:Name),MSG('Client Name'),REQ
      PROMPT('State Code:'),AT(8,34),USE(?CLI:StateCode:Prompt)

```

```

ENTRY(@s2),AT(61,34,40,10),USE(CLI:StateCode),MSG('State Code')
BUTTON('OK'),AT(12,53,45,14),USE(?OK),DEFAULT
BUTTON('Cancel'),AT(82,53,45,14),USE(?Cancel)
END
CODE
GlobalErrors.Init                !initialize GlobalErrors object
Access:Client.Init               !initialize Access:Client object
Access:Client.Open               !open the Client file
IF Access:Client.TryPrimeAutoInc() !prime Client record
  POST(Event:CloseWindow)        !if prime fails, close down
END

OPEN(InsertWindow)
ACCEPT
CASE FIELD()
OF ?OK
  IF EVENT() = Event:Accepted     !on OK button
    IF Access:Client.Insert() = Level:Benign
      !finish adding the new Client record
      POST(Event:CloseWindow)     !if add succeeds, close down
    ELSE
      !if add fails
      SELECT(?CLI:Name:Prompt)    !select client name field
      CYCLE                       !and start over
    END
  END
OF ?Cancel
  IF EVENT() = EVENT:Accepted     !on Cancel button
    Access:Client.CancelAutoInc   !restore Client to pre-PrimeRecord
    POST(Event:CloseWindow)       !close down
  END
END
END

Access:Client.Close              !close the Client file
Access:Client.Kill               !shut down the Access:Client object
GlobalErrors.Kill                !shut down the GlobalErrors object
RETURN

Access:Client.PrimeAutoInc PROCEDURE
CODE
!your custom code here
PARENT.PrimeAutoInc              !call the base class method
!your custom code here

```

See Also:           CancelAutoInc, PrimeAutoInc

## **TryReget (try to get a specific record by position)**

**TryReget**( *position* ), PROC

---

**TryReget** Gets a specific record by position.  
*position* A string constant, variable, EQUATE, or expression that indicates the position of the record to retrieve--typically the value returned by the Position method.

The **TryReget** method retrieves a specific record based its position and returns a success or failure indicator.

Implementation: The TryReget method tries to retrieve the specified record. If it succeeds, it returns Level:Benign; otherwise it returns the severity level of the last error encountered. See Error Class for more information on severity levels.

Return Data Type: BYTE

See Also: Position

## **TryUpdate (try to change the current record)**

**TryUpdate**, PROC

The **TryUpdate** method changes (rewrites) the current record. The TryUpdate method does not attempt to handle errors that occur while changing the record.

The Update method provides a slightly different (automatic) alternative for changing records.

Implementation: TryUpdate tries to change the record. If it succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns the severity level of the error it encountered while trying to change the record. See *Error Class* for more information on severity levels.

**Note:** This method does not handle referential integrity (RI) between related files. The RelationManager class enforces RI between related files.

Return Data Type: BYTE

See Also: Update

## TryValidateField(validate field contents)

**TryValidateField**(*fieldid*), PROC, VIRTUAL

---

**TryValidateField** Validates the current record buffer value of the specified field and returns a success or failure indicator.

*fieldid* A numeric constant, variable, EQUATE, or expression that identifies the field to validate. The field is identified by its position in the FILE declaration. A value of one (1) indicates the first field, two (2) indicates the second field, etc.

The **TryValidateField** method initiates the validation of the field's buffer and requests that no errors be reported to the user. Level:Benign is returned if no errors occur.

Implementation: The TryValidateField method calls the ValidateFieldServer method to validate the field's contents.

Return Data Type: BYTE

See Also: FileManager.ValidateFieldServer, FileManager.ValidateField

## Update (change the current record)

### Update, PROC

The **Update** method changes (rewrites) the current record. The Update method handles any errors that occur while changing the record.

The TryUpdate method provides a slightly different (manual) alternative for changing records.

Implementation: If Update succeeds, it returns Level:Benign (declared in ABERROR.INC). If it ultimately fails, it returns the severity level of the last error it encountered while trying to change the record. See *Error Class* for more information on severity levels.

**Note:** This method does not handle referential integrity (RI) between related files. The RelationManager class enforces RI between related files.

Return Data Type: BYTE

See Also: TryUpdate

**UseFile (use LazyOpen file)****UseFile, PROC**


---

<b>UseFile</b>	The <b>UseFile</b> method notifies ABC Library objects that the managed file whose opening was delayed by the LazyOpen property is about to be used. UseFile returns a value indicating whether the file is ready for use. A return value of Level:Benign indicates the file is ready; any other return value indicates a problem.
<i>usetype</i>	<p>A numeric constant, variable, EQUATE, or expression that determines how UseFile handles the file buffer.</p> <p>A value of <b>UseType:Corrupts</b> changes the value in the file buffer but does not rely on the new value.</p> <p>A value of <b>UseType:Uses</b> changes the value of the file buffer and then uses that value.</p> <p>A value of <b>UseType&gt;Returns</b> holds a value from the file buffer to return it to the parent. This mode is useful if you have a form record split between two windows and need to preserve the values from one to the next.</p> <p>A value of <b>UseType:Benign</b> indicates that no special file buffer handling is requested.</p>

## Implementation:

UseFile return values are declared in ABERROR.INC. See Error Class for more information on these severity levels. The UseType EQUATEs are declared in ABFILE.INC as follows:

```

UseType  ITEMIZE(1),PRE
Corrupts EQUATE
Uses     EQUATE
Returns  EQUATE
Benign   EQUATE
        END

```

Return Data Type: BYTE

**Example:**

```
FileManager.TryFetch PROCEDURE(KEY Key)
CODE
  IF SELF.UseFile() THEN RETURN Level:Fatal. !really open the file
  GET(SELF.File,Key)
  IF ERRORCODE()
    RETURN Level:Notify
  ELSE
    RETURN Level:Benign
  END
```

See Also: LazyOpen

**ValidateField (validate a field)****ValidateField**( *field id* ), VIRTUAL, PROC

---

**ValidateField** Validates the current record buffer value of the specified field and returns a success or failure indicator.

*field id* A numeric constant, variable, EQUATE, or expression that identifies the field to validate. The field is identified by its position in the FILE declaration. A value of one (1) indicates the first field, two (2) indicates the second field, etc.

The **ValidateField** method initiates the validation of the field's buffer and requests that any errors be reported to the user. Level:Benign is returned if no errors occur.

Implementation: The ValidateField method calls the ValidateFieldServer method to validate the field's contents.

Return Data Type: BYTE

Example:

```

MyFile  FILE, DRIVER('TOPSPEED'), THREAD
Record  RECORD, PRE()
TGroup  GROUP                !field id 1
Name    STRING(20)           !field id 2
Name2   STRING(20)           !field id 3
FirstName STRING(10), OVER(Name2) !field id 4
      END
Another  STRING(10)           !field id 5
      END
      END
CODE
!program code
Access:MyFile.ValidateField(4)    !validate FirstName

```

See Also: ValidateFields, ValidateFieldServer

## ValidateFields (validate a range of fields)

**ValidateFields**( *firstfield*, *lastfield* [,*failed*] ), VIRTUAL, PROTECTED, PROC

---

<b>ValidateField</b>	Validates the specified range of fields in the current record buffer and returns a success or failure indicator.
<i>firstfield</i>	A numeric constant, variable, EQUATE, or expression that identifies the first field to validate by its position in the FILE declaration. A value of one (1) indicates the first field, two (2) indicates the second field, etc.
<i>lastfield</i>	A numeric constant, variable, EQUATE, or expression that identifies the last field to validate by its position in the FILE declaration. A value of one (1) indicates the first field, two (2) indicates the second field, etc.
<i>failed</i>	A signed numeric variable that receives the identifier of the field that failed the validation process. A value of one (1) indicates the first field, two (2) indicates the second field, etc. If omitted, the calling procedure gets no indication of which field failed the validation process.

The **ValidateField** method validates the specified range of fields in the current record buffer and returns a success or failure indicator, and optionally identifies the field that failed the validation process.

Implementation: The **ValidateFields** method invokes the **ValidateField** method for each field in the range *firstfield* to *lastfield*.

Return Data Type: BYTE

See Also: **ValidateField**

## ValidateFieldServer(validate field contents)

**ValidateFieldServer**(*field id*, *handle errors*), PROC, VIRTUAL, PROTECTED

---

### **ValidateFieldServer**

Validates the current record buffer value of the specified field and returns a success or failure indicator.

*field id* A numeric constant, variable, EQUATE, or expression that identifies the field to validate. The field is identified by its position in the FILE declaration. A value of one (1) indicates the first field, two (2) indicates the second field, etc.

*handle errors* An integer constant, variable, EQUATE, or expression that indicates an error has occurred when validating the field id.

The **ValidateFieldServer** method validates the specified field in the current record buffer and returns a success or failure indicator. If an error occurs when the field's buffer is validated an error message (Msg:FieldNotInFile) is indicated to the user.

Implementation: The ValidateFieldServer method simply returns a zero (0). By convention a return value of zero (0) indicates a valid field and any other value indicates a problem. The ABC Templates derive a file-specific ValidateFieldServer method for each file that implements Validity Checks specified in the Clarion data dictionary.

The ValidateField and TryValidateField methods both call ValidateFieldServer to do their work, any global editing functions added by deriving the FileManager Class should be placed in the ValidateFieldServer method to assure that they will be executed by both the Validate and TryValidate methods.

Return Data Type: BYTE

See Also: FileManager.TryValidateField, FileManager.ValidateField

## ValidateRecord (validate all fields)

ValidateRecord( [*failed* ]), VIRTUAL

---

### ValidateRecord

Validates all the fields in the current record buffer and returns a success or failure indicator.

*failed* A signed numeric variable that receives the identifier of the field that failed the validation process. A value of one (1) indicates the first field, two (2) indicates the second field, etc. If omitted, the calling procedure gets no indication of which field failed the validation process.

The **ValidateRecord** method validates all the fields in the current record buffer and returns a success or failure indicator, and optionally identifies the field that failed the validation process.

Implementation: The ValidateRecord method invokes the ValidateField method for each field in the record.

Return Data Type: BYTE

See Also: FileManager.ValidateField



# FilterLocatorClass

## FilterLocatorClass Overview

The FilterLocatorClass is an IncrementalLocatorClass that filters or limits the result set of the BrowseClass object's underlying view. That is, it not only locates matching items in the result set, but it limits the result set to only those items.

Use a Filter Locator when you want a multi-character search on alphanumeric keys and you want to *minimize network traffic*.

## FilterLocatorClass Concepts

A Filter Locator is a multi-character locator, with no locator control required (but strongly recommended). The FilterLocatorClass lets you specify a locator control and a field on which to search for a BrowseClass object. The locator control accepts a search value which the FilterLocatorClass applies to the search field. The search can match the search value beginning with the first position of the search field ("begins with" search), or it can match the search value anywhere within the search field ("contains" search).

When the end user places one or more characters in the locator control, then *accepts* the control by pressing TAB, pressing a locator button, or selecting another control on the screen, the FilterLocatorClass creates a filter expression based on the input search value and applies the filter. Each additional (incremental) search character supplied results in a smaller, more refined result set. For example, a search value of 'A' returns all records from 'AA' to 'Az'; a search value of 'AB' returns all records from 'ABA' to 'ABz', and so on.

The Filter Locator determines the boundaries for the search based on the user specified search value. The implementation of the boundaries depends on the database--for SQL databases, the Filter Locator uses a LIKE; for ISAM databases it supplies upper and lower bounds.

**Tip:** The Filter Locator performs very well on SQL databases and on high order key component fields; however, performance may suffer if applied to non-key fields or low order key fields of non-SQL databases.

## FilterLocatorClass Relationship to Other Application Builder Classes

The BrowseClass optionally uses the FilterLocatorClass. Therefore, if your BrowseClass objects use a FilterLocator, then your program must instantiate the FilterLocatorClass for each use. Once you register the FilterLocatorClass object with the BrowseClass object (see BrowseClass.AddLocator), the BrowseClass object uses the FilterLocatorClass object as needed, with no other code required. See the Conceptual Example.

## FilterLocatorClass ABC Template Implementation

The ABC BrowseBox template generates code to instantiate the FilterLocatorClass for your BrowseBoxes. The FilterLocatorClass objects are called BRW*n*::Sort#:Locator, where *n* is the template instance number and # is the sort sequence (id) number. As this implies, you can have a different locator for each BrowseClass object sort order.

You can use the BrowseBox's **Locator Behavior** dialog (the **Locator Class** button) to derive from the EntryLocatorClass. The templates provide the derived class so you can modify the locator's behavior on an instance-by-instance basis.

## FilterLocatorClass Source Files

The FilterLocatorClass source code is installed by default to the Clarion \LIBSRC folder. The specific FilterLocatorClass source code and its respective components are contained in:

ABBROWSE.INC	FilterLocatorClass declarations
ABBROWSE.CLW	FilterLocatorClass method definitions

## FilterLocatorClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a BrowseClass object and related objects, including a Locator object. The example initializes and page-loads a LIST, then handles a number of associated events, including scrolling, updating, and locating records.

Note that the WindowManager and BrowseClass objects internally handle the normal events surrounding the locator.

```

PROGRAM
  INCLUDE('ABWINDOW.INC')           !declare WindowManager class
  INCLUDE('ABBROWSE.INC')          !declare BrowseClass and Locator
  MAP
  END

State      FILE,DRIVER('TOPSPEED'),PRE(ST),THREAD
StateCodeKey  KEY(ST:STATECODE),NOCASE,OPT
Record      RECORD,PRE()
STATECODE   STRING(2)
STATENAME   STRING(20)
            END
            END
StView      VIEW(State)             !declare VIEW to process
            END
StateQ      QUEUE                   !declare Q for LIST
ST:STATECODE  LIKE(ST:STATECODE)

```

```

ST:STATENAME LIKE(ST:STATENAME)
ViewPosition STRING(512)
    END

Access:State CLASS(FileManager)      !declare Access:State object
Init          PROCEDURE
    END
Relate:State CLASS(RelationManager) !declare Relate:State object
Init          PROCEDURE
    END
VCRRequest LONG(0),THREAD

StWindow WINDOW('Browse States'),AT(, ,123,152),IMM,SYSTEM,GRAY
    PROMPT('Find:'),AT(9,6)
    ENTRY(@s2),AT(29,4),USE(ST:STATECODE)
    LIST,AT(8,5,108,124),USE(?StList),IMM,HVSCROLL,FROM(StateQ),|
    FORMAT(' 27L(2)|M~CODE~@s2@80L(2)|M~STATENAME~@s20@')
    END

ThisWindow CLASS(WindowManager)      !declare ThisWindow object
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL
    END
BrowseSt    CLASS(BrowseClass)       !declare BrowseSt object
Q           &StateQ
    END

StLocator   FilterLocatorClass        !declare StLocator object
StStep      StepStringClass           !declare StStep object

CODE
ThisWindow.Run()                      !run the window procedure

ThisWindow.Init PROCEDURE()           !initialize things
ReturnValue     BYTE,AUTO
CODE
ReturnValue = PARENT.Init()           !call base class init
IF ReturnValue THEN RETURN ReturnValue.
Relate:State.Init                !initialize Relate:State object
SELF.FirstField = ?ST:STATECODE     !set FirstField for ThisWindow
SELF.VCRRequest &= VCRRequest       !VCRRequest not used
Relate:State.Open                 !open State and related files
!Init BrowseSt object by naming its LIST,VIEW,Q,RelationManager & WindowManager
BrowseSt.Init(?StList,StateQ.ViewPosition,StView,StateQ,Relate:State,SELF)
OPEN(StWindow)
SELF.Opened=True
BrowseSt.Q &= StateQ                !reference the browse QUEUE
StStep.Init(+ScrollSort:AllowAlpha,ScrollBy:Runtime) !initialize the StStep object

```

```
BrowseSt.AddSortOrder(StStep,ST:StateCodeKey)      !set the browse sort order
BrowseSt.AddLocator(StLocator)                    !plug in the browse locator
StLocator.Init(?ST:STATECODE,ST:STATECODE,1,BrowseSt) !initialize the locator object
BrowseSt.AddField(ST:STATECODE,BrowseSt.Q.ST:STATECODE) !set a column to browse
BrowseSt.AddField(ST:STATENAME,BrowseSt.Q.ST:STATENAME) !set a column to browse
SELF.SetAlerts()                                  !alert any keys for ThisWindow
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()                        !shut down things
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()                        !call base class shut down
IF ReturnValue THEN RETURN ReturnValue.
Relate:State.Close                               !close State and related files
Relate:State.Kill                                !shut down Relate:State object
GlobalErrors.Kill                                !shut down GlobalErrors object
RETURN ReturnValue
```

## FilterLocatorClass Properties

### FilterLocatorClass Properties

The FilterLocatorClass inherits all the properties of the IncrementalLocatorClass from which it is derived. See *IncrementalLocatorClass Properties* and *LocatorClass Concepts* for more information.

In addition to the inherited properties, the FilterLocatorClass also contains the following property:

### FloatRight ("contains" or "begins with" flag)

**FloatRight**      **BYTE**

The **FloatRight** property determines whether the FilterLocator applies the search value to the entire field (field *contains* search value) or only to the leftmost field positions (field *begins with* search value). A value of one (1 or True) applies the "contains" test; a value of zero (0 or False) applies the "begins with" test.

The FilterLocatorClass does not initialize the FloatRight property, therefore FloatRight defaults to zero.

Implementation:      The UpdateWindow method implements the action specified by the FloatRight property.

Example:              A FilterLocator searching for "ba" returns:

<b>FloatRight=False</b>	<b>FloatRight=True</b>
Bain	Bain
Barber	Barber
Bayert	Bayert
	Dunbar
	Suba

See Also:              UpdateWindow

## FilterLocatorClass Methods

### FilterLocatorClass Methods

The FilterLocatorClass inherits all the methods of the IncrementalLocatorClass from which it is derived. See *IncrementalLocatorClass Methods* and *LocatorClass Concepts* for more information.

### TakeAccepted (process an accepted locator value:FilterLocatorClass)

#### TakeAccepted, VIRTUAL

The **TakeAccepted** method processes the accepted locator value and returns a value indicating whether the BrowseClass list display should be updated. A return value of one (1 or True) indicates the list should be refreshed; a return value of zero (0 or False) indicates no refresh is needed.

This method is only appropriate for LocatorClass objects with locator controls that accept user input; for example, entry controls, combo controls, or spin controls.

A locator value is accepted when the end user changes the locator value, then TABS off the locator control or otherwise switches focus to another control on the same window.

Implementation: The TakeAccepted method primes the FreeElement property with the appropriate search value. If there is a search value, TakeAccepted calls the UpdateWindow method to apply the search value.

Return Data Type: BYTE

Example:

```
BrowseClass.TakeAcceptedLocator PROCEDURE !process an accepted locator entry
CODE
IF ~SELF.Sort.Locator &= NULL AND ACCEPTED() = SELF.Sort.Locator.Control
  IF SELF.Sort.Locator.TakeAccepted() !call locator take accepted method
    SELF.Reset(1) !if search needed, reset the view
    SELECT(SELF.ListControl) !focus on the browse list control
    SELF.ResetQueue( Reset:Done ) !reload the browse queue
  IF ~SELF.Sort.Locator &= NULL !if locator is present
    SELF.Sort.Locator.Reset ! match search value to actual record
  END
END
END
```

See Also: FreeElement

## UpdateWindow (apply the search criteria)

### UpdateWindow, VIRTUAL

The **UpdateWindow** method applies the search criteria and redraws the locator control with its current value.

Implementation: The UpdateWindow method refilters the underlying view, primes the FreeElement property with the current search value (the Shadow property), then redraws the locator control.

Example:

```
MyBrowseClass.UpdateWindow PROCEDURE          !update browse related controls
CODE
IF ~(SELF.Sort.Locator &= NULL)             !if locator is present
    SELF.Sort.Locator.UpdateWindow           ! redraw locator control
END
```



---

# FuzzyClass

## FuzzyClass Overview

The FuzzyClass supports the BrowseFuzzyMatching control template. These classes provide the searching and weighting algorithms.

FuzzyMatching provides a way to search for a value and get all records that have that value somewhere in the record's columns. The data returned is weighted based on where in the record the value exists.

Lets see this in an example. Using a database of Books, we might have some fields such as Title, Author, and ISBN. If we choose to search for the value of 'Potter', we will get all records which have 'Potter' in the Title (Harry Potter and the Goblet of Fire) or in the Author (Beatrix Potter).

## Relationship to Other Application Builder Classes

The FuzzyClass is completely independent of other Application Builder Classes.

## FuzzyClass ABC Template Implementation

The FuzzyClass is instantiated globally in any application that has the global 'Enable Fuzzy Matching' box turned on. The global settings also have two options, Ignore Case and Word Only that can optionally be set for Fuzzy Matching.

To use FuzzyMatching within a procedure, add the control template to the window. This will add a GROUP control that contains an ENTRY control and 2 BUTTON controls.

## FuzzyClass Source Files

The FuzzyClass source code is installed by default to the Clarion \LIBSRC folder. The specific FuzzyClass source code and its respective components are contained in:

ABFUZZY.INC	FuzzyClass declarations
ABFUZZY.CLW	FuzzyClass method definitions

## FuzzyClass Properties

The FuzzyClass contains no public properties.

## FuzzyClass Methods

### **Construct (initialize FuzzyClass object)**

#### **Construct**

The **Construct** method performs the necessary code to initialize the FuzzyClass object.

Implementation:       The Construct method is automatically called when the object is instantiated.

### **Init (initialize FuzzyClass object)**

#### **Init**

The **Init** method performs the necessary code to initialize the FuzzyClass object and its default settings.

Implementation:       The Init method is called globally in the start of an application.

### **Kill (shutdown FuzzyClass object)**

#### **Kill**

The **Kill** method performs the necessary code to initialize the FuzzyClass object and its default settings.

Implementation:       The Kill method is called globally at the end of the application.

**Match (find query matches)**

**Match**(*document, query*)

<b>Match</b>	Finds matching records based on the query.
<i>document</i>	A string constant, variable, EQUATE or expression that is compared against when matching records.
<i>query</i>	A string constant, variable, EQUATE, or expression that contains the value to search on.

The **Match** method returns a value based on where an instance of the query is found within the *document*.

Return Data Type:     **BYTE**

**SetOption (set fuzzymatch options)****SetOption**(*whichoption*, *value*)

<b>SetOption</b>	Set the Ignore Case and Word Only options.
<i>whichoption</i>	An integer constant, variable, EQUATE, or expression that specifies which option to set. The equates for these options are located in ABFUZZY.INC. MatchOption:NoCase sets the Ignore Case option. MatchOption:WordOnly sets the Word Only option.
<i>value</i>	An integer constant, variable, EQUATE, or expression that specifies the value for the option. A value of one (1 or True) will set the option on; a value of zero (0 or False) will turn the option off. The default value is True.

The **SetOption** method logically sets one of the two options available for FuzzyMatching. These are *Ignore Case* and *Word Only*. When *Ignore Case* is set the query is case insensitive. *Word Only* finds the query value only if it is a separate word (denoted by a space directly before and directly after the text).

Return Data Type:     **BYTE**

Example:

```
FuzzyMatcher.SetOption(MatchOption:NoCase, 1)           !set for case insensitive search
FuzzyMatcher.SetOption(MatchOption:WordOnly, 0)       !turn off word only search
```

---

# FormVCRClass

## FormVCRClass Overview

The FormVCRClass is a special class that uses a group control populated with scrolling and update buttons. It is designed as an accessory to a Form procedure that is designed to function independently from a standard Browse procedure.

Use the FormVCRClass template when you want to call a form directly from a menu item or button and use the FormVCRClass to navigate through a primary file and perform standard update actions.

## FormVCRClass Concepts

The FormVCRClass lets you specify a group control with navigation and standard update buttons.

As the form is first opened, internal properties are set that control what buttons are disabled and what database operations are allowed.

## FormVCRClass Relationship to Other Application Builder Classes

The FormVCRClass is closely integrated with several other ABC Library objects--in particular the WindowManager and ToolbarClass objects. These objects register their presence with each other, set each other's properties, and call each other's methods as needed to accomplish their respective tasks.

The FormVCRClass is derived from the ViewManager, plus it relies on many of the other Application Builder Classes (RelationManager, ToolbarClass, etc.) to accomplish its tasks. Therefore, if your program instantiates the FormVCRClass, it must also instantiate these other classes. Much of this is automatic when you INCLUDE the FormVCRClass header (ABVCRFRM.INC) in your program's data section.

## FormVCRClass ABC Template Implementation

The ABC Templates declare a local FormVCR class *and* object for each instance of the FormVCRControl template. The ABC Templates automatically include all the classes necessary to support the functionality specified in the FormVCRControl template.

The FormVCR Control Template requires the use of the SaveButton Control Template, which is the framework of the Form template.

## FormVCRClass Source Files

The FormVCRClass source code is installed by default to the Clarion \LIBSRC folder. The specific FormVCRClass source code and their respective components are contained in:

ABVCRFRM.INC	FormVCRClass declarations
ABVCRFRM.CLW	FormVCRClass method definitions

## FormVCRClass Properties

### **QuickScan (buffered reads flag)**

**QuickScan**     **BYTE**

The **QuickScan** property contains a value that tells the FormVCRClass whether or not to quickscan when paging up or down within the form. Quick scanning only affects file systems that use multi-record buffers. See *Database Drivers* for more information.

A value of zero (0) disables quick scanning; a non-zero value enables quick scanning. Quick scanning is the normal way to read records for browsing. However, rereading the buffer may provide slightly improved data integrity in some multi-user circumstances at the cost of substantially slower reads.

Implementation:     The **QuickScan** property sets SetQuickScan method, which SENDs the QUICKSCAN driver string to the file driver for each specified file. The QUICKSCAN driver string is supported by the ASCII, BASIC, and DOS drivers.

## **Toolbar (FormVCR Toolbar object)**

### **Toolbar &ToolbarClass**

The **Toolbar** property is a reference to the ToolbarClass for this FormVCRClass object. The ToolbarClass object collects toolbar events and passes them on to the active ToolbarTarget object for processing.

The AddToolbarTarget method registers a ToolbarTarget, such as a ToolbarListBoxClass object, as a potential target of a ToolbarClass object.

The ToolbarClass.SetTarget method sets the active target for a ToolbarClass object.

Implementation: The ToolbarClass object for FormVCR controls is the object that detects toolbar events, such as scroll down or page down, and passes them on to the *active* ToolbarListBoxClass (ToolbarTarget) object. In the standard template implementation, there is a single global toolbar, and a ToolbarClass object per procedure that may drive several different browses and forms, each of which is a ToolbarTarget. Only one ToolbarTarget is active at a time.

## **ToolbarItem (FormVCR ToolbarTarget object)**

### **ToolbarItem &ToolbarListBoxClass**

The **ToolbarItem** property is a reference to the ToolbarListBoxClass for this FormVCRClass object. The ToolbarListBoxClass (ToolbarTarget) object receives toolbar events (from a ToolbarClass object) and processes them.

The AddToolbarTarget method registers a ToolbarTarget, such as a ToolbarListBoxClass object, as a potential target of a ToolbarClass object.

The ToolbarClass.SetTarget method sets the active target for a ToolbarClass object.

Implementation: The ToolbarClass object for the Form VCR controls is the object that detects toolbar events, such as scroll down or page down, and passes them on to the *active* ToolbarListBoxClass (ToolbarTarget) object. In the standard template implementation, there is a single global toolbar, and a ToolbarClass object per procedure that may drive several different browses and forms, each of which is a ToolbarTarget. Only one ToolbarTarget is active at a time.

See Also: [Toolbar](#), [AddToolbarTarget](#), [ToolbarClass.SetTarget](#)

**ViewPosition (store the current record position)****ViewPosition**    **STRING(1024)**

The **ViewPosition** property stores the unique position of the current record.

Implementation:    Position returns the POSITION of the primary key if there is one; otherwise it returns the file POSITION. See the *Language Reference* for more information on POSITION.

Example:

```

FormVCRClass.TakeEvent                    PROCEDURE
VSP BYTE,AUTO
CODE
  SELF.Window.Update()
  IF EVENT()=EVENT:Accepted THEN
    CASE ACCEPTED()
      OF SELF.Window.OkControl
      OROF SELF.Window.SaveControl
        SELF.ViewPosition=POSITION(SELF.View)
          SELF.SaveRequired = True
          IF SELF.OnFirstRecord THEN
            SELF.MoveDirection = Event.ScrollDown
          END
    ELSE
      IF NOT 0{PROP:AcceptAll} THEN
        SELF.TakeAcceptedLocator()
      END
    END
  END

```

## FormVCRClass Methods:

**AddToolBarTarget (set the FormVCR toolbar)**AddToolBarTarget( *toolbar* )

---

**AddToolBarTarget** Registers the FormVCR object as a potential target of the specified *toolbar*.

*toolbar* The label of the ToolbarClass object that directs toolbar events to this FormVCRClass object.

The **AddToolBarTarget** method registers the FormVCRClass object as a potential target of the specified *toolbar*. The ToolbarClass.SetTarget method sets the active target for a ToolbarClass object.

Implementation: The Toolbar object for a FormVCR is the object that detects toolbar events, such as scroll down or page down, and passes them on to the *active* ToolbarTarget object. In the standard template implementation, there is a single global toolbar, and a Toolbar object per procedure that may drive several different browses and forms, each of which is a ToolbarTarget. Only one ToolbarTarget is active at a time.

Example:

```

OPEN(QuickWindow)                                ! Open window
SELF.Opened=True
FormVCR6.Init|
(?FormVCRControls,10,FormVCR6::View,Relate:people,SELF) ! Init FormVCR manager
FormVCR6.InsertWhenNoRecords = True
FormVCR6.SetVCRControls|
(?FormVCR:Top,?FormVCR:PageUp,?FormVCR:Up,?FormVCR:Down,?FormVCR:PageDown,|
?FormVCR:Bottom)
FormVCR6.SetRequestControl|
(?FormVCR:Request,?FormVCR:Request:View,?FormVCR:Request:Insert,?FormVCR:Request:Change,|
?FormVCR:Request:Delete)
Do DefineListBoxStyle
FormVCR6.AddSortOrder()                            !Add the sort order for sort order 1
! Controls like list boxes will resize, whilst controls like buttons will move
Resizer.Init(AppStrategy:Surface,Resize:SetMinSize)
SELF.AddItem(Resizer)                             ! Add resizer to window manager
SELF.AddItem(ToolBarForm)
FormVCR6.AddToolBarTarget(ToolBar)                 ! Browse accepts toolbar control
FormVCR6.Reset
SELF.SetAlerts()
RETURN ReturnValue

```

See Also: Toolbar, ToolbarItem, ToolbarClass.SetTarget

**Init (initialize the FormVCR object)**

**Init** (*listcontrol, pagesize, view, relationmanager, windowmanager*)

<b>Init</b>	Initializes the FormVCR object.
<i>listcontrol</i>	A numeric constant, variable, EQUATE, or expression containing the control number of the FormVCR GROUP control.
<i>pagesize</i>	A numeric constant, variable, EQUATE, or expression containing the number of records to page in the FormVCR object.
<i>view</i>	The label of the FormVCR's underlying VIEW.
<i>relationmanager</i>	The label of the FormVCR primary file RelationManager object. See <i>Relation Manager</i> for more information.
<i>windowmanager</i>	The label of the FormVCR WindowManager object. See <i>Window Manager</i> for more information.

The **Init** method initializes the FormVCR object.

Implementation: In addition to other things (like initialization of properties), the **Init** method calls the PARENT.Init method to initialize the FormVCR ViewManager object.

Example:

```

OPEN(QuickWindow)                ! Open window
    SELF.Opened=True
! Initialize the FormVCR manager
FormVCR6.Init(?FormVCRControls,10,FormVCR6::View,Relate:people,SELF)

```

**InitSort (initialize locator values)**

**InitSort** (*neworder*), **VIRTUAL**

The **InitSort** method initializes locator values when a new sort order is applied to a browse list.

See Also: **SetSort**

## **Kill (shut down the FormVCR object)**

### **Kill, VIRTUAL**

The **Kill** method shuts down the FormVCR object.

Implementation: Among other things, the FormVCR.Kill method calls the PARENT.Kill (ViewManager.Kill) method to shut down the browser's ViewManager object. See *View Manager* for more information.

## **CheckBorders (check for existence of records)**

### **CheckBorders( ), VIRTUAL**

---

**CheckBorders** Checks for the existence of records.

The **CheckBorders** method is a virtual method used to check for the existence of records after the FormVCR class has completed an update.

Implementation: The CheckBorders method is called from the UpdateWindow, ResetSort, TakeRequestChanged methods, and set a number of private properties used to control the display state of the FormVCR controls.

Example:

```
FormVCRClass.UpdateWindow PROCEDURE
CODE
  IF ~(SELF.Sort.Locator &= NULL)
    SELF.Sort.Locator.UpdateWindow
  END
  SELF.CheckBorders
  SELF.UpdateButtons
  IF ~SELF.Toolbar &= NULL
    SELF.Toolbar.DisplayButtons
  END
```

**GetAction (return FormVCR action)****GetAction( ), BYTE, VIRTUAL****GetAction** Detect that an action has taken place.

The **GetAction** method is a virtual method used by the `ToolBarClass` to detect and process actions posted by the `FormVCR` class.

Implementation: The `GetAction` method is called from the `ToolBarFormVCRClass` to detect an action and enable/disable buttons where appropriate.

Example:

```

ToolBarFormVCRClass.DisplayButtons PROCEDURE
CODE
  IF SELF.FormVCR.GetAction( )=InsertRecord THEN
    ENABLE(ToolBar:History)
  ELSE
    DISABLE(ToolBar:History)
  END
  Toolbar:Bottom{PROP:DISABLE} =|
  CHOOSE(SELF.FormVCR.GetActionAllowed(EVENT:ScrollBottom,0),False,True)
  Toolbar:Top{PROP:DISABLE} =|
  CHOOSE(SELF.FormVCR.GetActionAllowed(EVENT:ScrollTop,0),False,True)
  Toolbar:PageDown{PROP:DISABLE} =|
  CHOOSE(SELF.FormVCR.GetActionAllowed(EVENT:PageDown,0),False,True)
  Toolbar:PageUp{PROP:DISABLE} =|
  CHOOSE(SELF.FormVCR.GetActionAllowed(EVENT:PageUp,0),False,True)
  Toolbar:Down{PROP:DISABLE} =|
  CHOOSE(SELF.FormVCR.GetActionAllowed(EVENT:ScrollDown,0),False,True)
  Toolbar:Up{PROP:DISABLE} =|
  CHOOSE(SELF.FormVCR.GetActionAllowed(EVENT:ScrollUp,0),False,True)

  DISABLE(ToolBar:Locate)
  PARENT.DisplayButtons

```

## GetActionAllowed (validate a requested FormVCR action)

`GetActionAllowed( event, action ), VIRTUAL`

---

**GetActionAllowed** Checks for a valid FormVCR action.

The **GetActionAllowed** method is a virtual method used to validate a FormVCR action.

Implementation: The `GetActionAllowed` method is called by the `TakeEvent` method and is used to synchronize toolbar buttons with the appropriate FormVCR action..

Example:

```
CASE ACCEPTED()
OF SELF.VCRRequest
  IF SELF.GetActionAllowed(EVENT:Accepted,SELF.Window.Request) THEN
    CHANGE(SELF.VCRRequest,SELF.Window.Request)
    SELF.Window.OriginalRequest = SELF.Window.Request
    SELF.TakeRequestChanged(SELF.VCRPrevRequest,SELF.Window.Request)
    SELF.VCRPrevRequest = SELF.Window.Request
  ELSE
    IF SELF.NoRecords THEN
      SELF.NoRecords = RECORDS(SELF.View)
      IF NOT SELF.GetActionAllowed(EVENT:Accepted,SELF.Window.Request) THEN
        SELF.Window.Request = SELF.VCRPrevRequest
        CHANGE(SELF.VCRRequest,SELF.Window.Request)
        SELF.UpdateWindow
      ELSE
        CHANGE(SELF.VCRRequest,SELF.Window.Request)
        SELF.Window.OriginalRequest = SELF.Window.Request
        SELF.TakeRequestChanged(SELF.VCRPrevRequest,SELF.Window.Request)
        SELF.VCRPrevRequest = SELF.Window.Request
      END
    END
  END
END
```

## **Next (get the next FormVCR item)**

**Next( ), BYTE, VIRTUAL**

The **Next** method gets the next record from the FormVCR view and returns a value indicating its success or failure.

Next returns Level:Benign if successful, Level:Notify if it reached the end of the file, and Level:Fatal if it encountered a fatal error.

Implementation: Corresponding return value EQUATEs are declared in ABERROR.INC. See *Error Class* for more information on these severity level EQUATEs.

<b>Level:Benign</b>	<b>EQUATE(0)</b>
<b>Level:User</b>	<b>EQUATE(1)</b>
<b>Level:Program</b>	<b>EQUATE(2)</b>
<b>Level:Fatal</b>	<b>EQUATE(3)</b>
<b>Level:Cancel</b>	<b>EQUATE(4)</b>
<b>Level:Notify</b>	<b>EQUATE(5)</b>

The Next method is called by the Fetch method. Among other things, Next calls the PARENT.Next (ViewManager.Next) method. See *ViewManager* for more information.

Return Data Type: **BYTE**

Example:

**Previous (get the previous FormVCR item)****Previous, VIRTUAL**

The **Previous** method gets the previous record from the FormVCR view and returns a value indicating its success or failure.

Implementation: Returns Level:Benign if successful, Level:Notify if it reached the end of the file, and Level:Fatal if it encountered a fatal error. Corresponding severity level EQUATEs are declared in ABERROR.INC. See *Error Class* for more information on error severity levels.

Level:Benign	EQUATE(0)
Level:User	EQUATE(1)
Level:Program	EQUATE(2)
Level:Fatal	EQUATE(3)
Level:Cancel	EQUATE(4)
Level:Notify	EQUATE(5)

The Previous method is called by the Fetch method. Among other things, Previous calls the PARENT.Previous (ViewManager.Previous) method. See *ViewManager* for more information.

Return Data Type: BYTE

**ResetSort (apply sort order to FormVCR)****ResetSort( *force* ), VIRTUAL, PROC**


---

**ResetSort** Reapplies the active sort order to the FormVCR controls.

*force* A numeric constant, variable, EQUATE, or expression that indicates whether to reset the FormVCR object conditionally or unconditionally. A value of one (1 or True) unconditionally resets the FormVCR; a value of zero (0 or False) only resets the FormVCR as circumstances require (sort order changed, reset fields changed, first loading, etc.).

The **ResetSort** method reapplies the active sort order to the FormVCR object and returns one (1) if the sort order changed; it returns zero (0) if the order did not change. Any range limits, locators, or reset fields associated with the sort order are enabled.

Implementation: The ResetSort method calls the SetSort method to apply the current sort order.

Return Data Type: BYTE

## **SetAlerts (alert keystrokes for FormVCR controls)**

### **SetAlerts, VIRTUAL**

The **SetAlerts** method alerts standard keystrokes for the FormVCR controls.

The FormVCR.TakeEvent method processes the alerted keystrokes.

Implementation: The BrowseClass.SetAlerts method alerts the INSERT, DELETE and CTRL+ENTER keys for the browse's list control and calls the LocatorClass.SetAlerts method for each associated locator control. Corresponding keycode EQUATEs are declared in KEYCODES.CLW.

## **SetRequestControl (assign field equates to FormVCR actions)**

### **SetRequestControl**

*(pVCRRequest, pVCRViewRecord, pVCRInsertRecord, pVCRChangeRecord, pVCRDeleteRecord)*

The **SetRequestControl** method assigns the related field equate of a control to a particular FormVCR function. Each parameter specified is a SIGNED integer.

Implementation:

The SetRequestControl method is used to map an appropriate control to a particular database action. If the parameter is not set by the SetRequestControl method, than the action is not supported.

Example:

```

OPEN(QuickWindow)                                ! Open window
    SELF.Opened=True
! Initialize the FormVCR manager
FormVCR6.Init(?FormVCRControls,10,FormVCR6::View,Relate:people,SELF)
FormVCR6.InsertWhenNoRecords = True
FormVCR6.SetVCRControls|
(?FormVCR:Top,?FormVCR:PageUp,?FormVCR:Up,?FormVCR:Down,|
?FormVCR:PageDown,?FormVCR:Bottom)
FormVCR6.SetRequestControl|
(?FormVCR:Request,?FormVCR:Request:View,?FormVCR:Request:Insert,|
?FormVCR:Request:Change,?FormVCR:Request>Delete)

```

## **SetVCRControls (assign field equates to FormVCR scrolling)**

### **SetVCRControls**

*(pVCRTop, pVCRUp, pVCRPageUp, pVCRPageDown, pVCRDown, pVCRBottom, pVCRNewRecord=0)*

The **SetVCRControls** method assigns the related field equate of a control to a particular FormVCR scrolling function. Each parameter specified is a SIGNED integer.

Implementation:

The SetVCRControls method is used to map an appropriate control to a particular scrolling action. If the parameter is not set by the SetVCRControls method, than the action is not supported.

### **Example:**

```
OPEN(QuickWindow)                                ! Open window
  SELF.Opened=True
! Initialize the FormVCR manager
FormVCR6.Init(?FormVCRControls,10,FormVCR6::View,Relate:people,SELF)
FormVCR6.InsertWhenNoRecords = True
FormVCR6.SetVCRControls |
(?FormVCR:Top,?FormVCR:PageUp,?FormVCR:Up,?FormVCR:Down, |
?FormVCR:PageDown,?FormVCR:Bottom)
FormVCR6.SetRequestControl |
(?FormVCR:Request,?FormVCR:Request:View,?FormVCR:Request:Insert, |
?FormVCR:Request:Change,?FormVCR:Request:Delete)
```

**SetSort (apply a sort order to the FormVCR group)****SetSort**( *order*, *force reset* ), VIRTUAL, PROC

---

<b>SetSort</b>	Applies a specified sort order to the FormVCR group.
<i>order</i>	An integer constant, variable, EQUATE, or expression that specifies the sort order to apply.
<i>force reset</i>	A numeric constant, variable, EQUATE, or expression that tells the method whether to reset the FormVCR conditionally or unconditionally. A value of zero (0 or False) resets the FormVCR only if circumstances require (sort order changed, reset fields changed, first time loading); a value of one (1 or True) unconditionally resets the FormVCR.

The **SetSort** method applies the specified sort *order* to the FormVCR group and returns one (1) if the sort order changed; it returns zero (0) if the sort order did not change. Any range limits, locators, and reset fields associated with the sort order are enabled and applied.

The *order* value is typically a value returned by the ViewManager's AddSortOrder method which identifies the particular sort order. Since AddSortOrder returns sequence numbers, a value of one (1) applies the sort order specified by the first call to AddSortOrder; two (2) applies the sort order specified by the next call to AddSortOrder; etc. A value of zero (0) applies the default sort order.

Implementation: The ResetSort method calls the SetSort method.

Return Data Type: BYTE

## **TakeAcceptedLocator (apply an accepted FormVCR locator value)**

### **TakeAcceptedLocator, VIRTUAL**

The **TakeAcceptedLocator** method applies an accepted locator value to the FormVCR group--the FormVCR object loads the requested item.

Locators with entry controls are the only locators whose values are accepted. Other types of locators are invoked in other ways, for example, with alerted keys. Locator values are accepted when the end user TABS off or otherwise switches focus away from the locator's entry control.

The AddLocator method establishes locators for the browse.

Implementation: The TakeAcceptedLocator method calls the appropriate LocatorClass.TakeAccepted method.

## **TakeEvent (process the current ACCEPT loop event)**

### **TakeEvent, VIRTUAL**

The **TakeEvent** method processes the current ACCEPT loop event for the FormVCR object. The TakeEvent method handles all events associated with the FormVCR group.

Implementation: The WindowManager.TakeEvent method calls the TakeEvent method. The TakeEvent method calls the TakeScroll or TakeKey method as appropriate.

## **TakeLocate (a FormVCR virtual to process each sort)**

### **TakeLocate, VIRTUAL**

The **TakeLocate** method is a virtual method that sets the active sort and filter criteria, and enables any necessary toolbar activity.

Implementation: The FormVCR.TakeEvent method calls the TakeLocate method for each Locate event.

## **TakeScroll (process a FormVCR scroll event)**

### **TakeScroll( [*scrollevent*] ), VIRTUAL**

---

**TakeScroll** Processes a scroll event for the FormVCR group.

*scrollevent* An integer constant, variable, EQUATE, or expression that specifies the scroll event. Valid scroll events are back one item, forward one item, forward one page, back one page, back to the first item, and ahead to the last item. If omitted, no scrolling occurs.

The **TakeScroll** method processes a scroll event for the FormVCR group.

Implementation: A *scrollevent* value of EVENT:ScrollUp scrolls back one item; EVENT:ScrollDown scrolls ahead one item; EVENT:PageUp scrolls ahead one page; EVENT:PageDown scrolls back one page; EVENT:ScrollTop scrolls to the first item; EVENT:ScrollBottom scrolls to the last item. Corresponding *scrollevent* EQUATES are declared in EQUATES.CLW.

```
EVENT:ScrollUp      EQUATE (03H)
EVENT:ScrollDown    EQUATE (04H)
EVENT:PageUp        EQUATE (05H)
EVENT:PageDown      EQUATE (06H)
EVENT:ScrollTop     EQUATE (07H)
EVENT:ScrollBottom  EQUATE (08H)
```

The TakeScroll method calls the ScrollEnd, ScrollOne, or ScrollPage method as needed.

**UpdateWindow (update display variables to match FormVCR action)****UpdateWindow, VIRTUAL**

The **UpdateWindow** method updates display variables to match the current state of the FormVCR group.

Implementation: The FormVCRClass.UpdateWindow method calls the appropriate LocatorClass.UpdateWindow method, which ensures the locator field contains the current search value.



# GraphClass

## GraphClass Overview

The GraphClass is used to manage and control the SVgraph group control available on windows and reports. This control is created and initialize by the SVGGraph control template. At runtime, the group control reads from initial settings of the GraphClass, and arranges the data points into a default graph type format. The data included in this graph includes grouping of data, legends, labels, popup menus, and default print capabilities.

The SVGGraph template is most useful for displaying graphical representation of data for nearly every type of business application.

## Relationship to Other Application Builder Classes

The GraphClass is derived from the GraphAxisClass . It also is affected by a number of other Graph Classes.

The GraphClass is the only class that will be visible to the developer. All other classes are used internally by the Graph Class and are not documented.

## GraphClass ABC Template Implementation

The SVGGraph template generates code to instantiate the GraphClass for your graph control. The GraphClass objects are called, by default, "GRP*n*", where *n* is the template instance number. Multiple graph controls may be used in a single procedure.

## GraphClass Source Files

The GraphClass source code is installed by default to the Clarion \LIBSRC folder. The specific GraphClass source code and its respective components are contained in:

SVGRAPH.INC	GraphClass declarations
SVGRAPH.CLW	GraphClass method definitions
SVGRAPH.EQU	Graph Property Equates and Events

## GraphClass Properties

### **eShowSBonFirstThread (display on base status bar)**

**eShowSBonFirstThread**      **BOOL**

**eShowSBonFirstThread** is a Boolean value that determines if status bar information from the graph object needs to be redirected to the application's first thread.

If the current WINDOW that populated the graph control does not have a Status bar attribute, then **eShowSBonFirstThread** set to TRUE will show graph control information in the status bar of the APPLICATION (first thread). Set this property to FALSE to suppress (hide) this information.

See Also: ShowOnStatusBar

### **eSumYMax (calculated maximum node value)**

**eSumYMax**      **REAL**

**eSumYMax** is a property that is used to calculate the maximum sum of node values for the accumulation graph types (Bar and Column). This property is used for drawing the diagram (e.g., It is necessary for calculating the height of the highest column).

See Also: CalcGraph

**gShowDiagramName (show diagram name on target)****gShowDiagramName****LIKE(gShowToType)**

**gShowDiagramName** is used to determine where the diagram name information will be displayed in the graph control. Diagram name (series name) is the label of each set of data points that you have defined for the graph control, with an added 'Diagram:' prompt.

**gShowDiagramName** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also: DiagramText, ShowOnField

**gShowDiagramNameV (show diagram value on target)****gShowDiagramNameV****LIKE(gShowToType)**

**gShowDiagramNameV** is used to determine where the diagram name text information will be displayed in the graph control. Diagram name text (series name) is the label of each set of data points that you have defined for the graph control, without an added 'Diagram:' prompt.

**gShowDiagramNameV** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also:

DiagramNameText, ShowOnField

**gShowMouse (show mouse coordinates on target)**

**gShowMouse**                      **LIKE(gShowToType)**

**gShowMouse** is used to determine where mouse X and Y coordinates will be displayed in the graph control.

**gShowMouse** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also:  
ToShowValues, ToolTip

**gShowMouseX (show mouse X coordinate on target)**

**gShowMouseX**      **LIKE(gShowToType)**

**gShowMouseX** is used to determine where mouse X coordinate information will be displayed in the graph control.

**gShowMouseX** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also:  
ToShowValues, ToolTip

**gShowMouseY (show mouse Y coordinate on target)**

**gShowMouseY**      **LIKE(gShowToType)**

**gShowMouseY** is used to determine where mouse Y coordinate information will be displayed in the graph control.

**gShowMouseY** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also:  
ToShowValues, ToolTip

**gShowNodeName (show node name on target)****gShowNodeName****LIKE(gShowToType)**

**gShowNodeName** is used to determine if and where the node name information will be displayed in the graph control. Node name is the full text of the active X and Y Node name, less the "Node:" prompt and values. The text is formed by the NodeText method. Node Text is returned as the full form as follows:

"Node: **name** (X, Y)"

**gShowNodeName** determines if and where the *name* part of the above node text will be displayed.

**gShowNodeName** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also:  
NodeText

**gShowNodeNameV (show node name value on target)**

gShowNodeNameV

LIKE(gShowToType)

**gShowNodeNameV** is used to determine where the node name value information will be displayed in the graph control. Node name value is the actual node *Name* identifier.

**gShowNodeNameV** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also:  
NodeNameText

**gShowNodeValue (show node axis values on target)****gShowNodeValue****LIKE(gShowToType)**

**gShowNodeValue** is used to determine where the node value information will be displayed in the graph control. Node name value is part of the full text of the active X and Y Node name and values the label of each set of data points that you have defined for the graph control, without an added 'Node:' prompt. The text is formed by the NodeText method. Node Text is returned as the full form as follows:

"Node: name (**X, Y**)"

**gShowNodeValue** determines if and where the (X, Y) part of the above node text will be displayed

**gShowNodeValue** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also:  
NodeValueText

**gShowNodeValueX (show node x-axis value on target)****gShowNodeValueX****LIKE(gShowToType)**

**gShowNodeValueX** is used to determine where the node X value information will be displayed in the graph control. The Node X value is the full text of the active X Node.

**gShowNodeValueX** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also:  
NodeXText

**gShowNodeValueY (show node y-axis value on target)****gShowNodeValueY****LIKE(gShowToType)**

**gShowNodeValueY** is used to determine where the node Y value information will be displayed in the graph control. The Node Y value is the full text of the active Y Node.

**gShowNodeValueY** uses the gShowToType GROUP. Each group element is defined as follows:

eOnT	A BYTE value, when set to TRUE, shows information on graph object's ToolTip
eOnW	A BYTE value, when set to TRUE, shows target information on the WINDOW title
eOnF	A LONG integer that identifies the field equate to show the target information. The target field can be any field capable of displaying the designated information.
eOnS	A LONG integer Number of a zone of a status bar of a condition to show the text.

See Also:  
NodeYText

## GraphClass Methods

### **AllText (return full graph text information)**

**AllText**

**AllText**          Process and return full graph text to tool tip.

The **AllText** method reads Mouse, Diagram and Node text information from the MouseText, DiagramText and NodeText methods respectively, and returns a formatted string to use in the graph object's tool tip.

Return Data Type:    **STRING**

Implementation:      The AllText method can be called to query the graph object's tool tip text as needed.

See Also:             MouseText, DiagramText  
                             NodeText

### **BeginRefresh (prepare drawing of graph class object)**

**BeginRefresh**

**BeginRefresh**      Force a redraw of graph object

The **BeginRefresh** method carries out the role of a callback function. This method is called at the beginning of the Refresh method, and used for the purpose of preparing the diagram list and their respective nodes to draw.

The default return value is FALSE. If the method returns TRUE, the graph object is not redrawn.

Return Data Type:    **BOOL**

Implementation:      The BeginRefresh method is called from the graph object's Refresh method

See Also:             Refresh

## **CalcBestPositionNodeText (calculate graph text best fit position)**

### **CalcBestPositionNodeText**

**CalcBestPositionNodeText** Calculates the best position of the text used to describe the selected node.

The **CalcBestPositionNodeText** method sets the best position of the node text to be displayed.

Implementation: The method is called if the `eBestPositionNodeText` property is set to `TRUE`. This property is set by graph control's procedure template interface. Internally, each node in the graph stores the X and Y position in a local queue.

See Also: `CalcGraph`

## **CalcCurrentGraph (calculates values for current graph type)**

### **CalcCurrentGraph**

**CalcCurrentGraph** Calculates the new parameters for a designated graph type.

The **CalcCurrentGraph** method is used to set the parameters for a designated graph type. In the Graph Control template, a user can dynamically switch graph types at runtime. This method determines which graph type was specified and instantiates a new object used to set parameters specific to that graph type.

Implementation: The method is called from the `CalcGraph` method each time a graph type changes or the window is initially loaded.

See Also: `CalcGraph`

## **CalcCurrentNode (calculates values of current node)**

### **CalcCurrentNode**

**CalcCurrentNode** Calculates all pertinent values for a selected node.

The **CalcCurrentNode** method calculates all attributes of a current node, including fill color, shape, hide property, value, background, min/max values, radius, etc.

Implementation: The method is called from the CalcGraph method., for each node in the current graph type.

See Also: CalcGraph

## **CalcGraph (calculates all graph object values)**

### **CalcGraph**

**CalcGraph** Calculates all positions, node values and cosmetic settings for a graph control.

The **CalcGraph** method is used to calculate positions and values for a graph's title, legend, axis, and nodes for a specified graph control.

Implementation: The CalcGraph method is called prior to the DrawGraph method.

See Also: CalcCurrentGraph  
CalcCurrentNode  
CalcBestPositionNodeText

**CalcPopup (create popup menu for graph object)****CalcPopup (PopupClass PopupMgr)**

**CalcPopup**            Creates a popup menu control for a selected graph object.

*PopupMgr*            The reference to an object of a PopupClass type.

The **CalcPopup** method creates a popup menu for specific use with the Graph Class object. It also creates standard popup menu items that are used with all graph types (i.e., Zoom settings).

Implementation:        The **CalcPopup** method is called from the GraphClass Popup method, initializing standard menu items for use with all graph types.

See Also:              Popup CalcPopupAdd2

**CalcPopupAdd2 (create popup menu item text for graph object)**

**CalcPopupAdd2**(*PopupMgr, Text, Name, Items, ItemMask, Check, CheckValue*)

<b>CalcPopupAdd2</b>	Create popup menu items specific to a selected graph type
<i>PopupMgr</i>	The reference to an object of a <i>PopupClass</i> type.
<i>Text</i>	A string constant, variable, EQUATE, or expression containing the text of the menu item. A single hyphen (-) creates a non-selectable separator (a 3D horizontal bar) on the menu. An ampersand (&) designates the next character as the menu item's hot key.
<i>Name</i>	A string constant, variable, EQUATE, or expression containing the menu item name. Other methods refer to the menu item by its <i>name</i> , not by its <i>text</i> . This lets you apply runtime translation or dynamic reordering of menus without changing your code.
<i>Items</i>	A LONG integer that is bit-mapped and used with <i>ItemMask</i> to determine the availability of a popup menu item based on graph type and area.
<i>ItemMask</i>	A LONG integer that is used with <i>Items</i> to determine a popup items visibility.
<i>Check</i>	A LONG integer compared with <i>CheckValue</i> to determine a popup items initial state (checked/unchecked, enabled/disabled)
<i>CheckValue</i>	A LONG integer that is used with <i>Check</i> to determine a popup items current state.

The **CalcPopupAdd2** method adds conditional popup menu items to the graph control popup based on graph type and other factors. If an item has been added correctly, a non-zero value is returned by the method.

Return Data Type: LONG

Implementation: The **CalcPopupAdd2** method is called from the *CalcPopup* method. A set of properties and equates (see GRAPH.EQU in the \LIBSRC folder) determine the availability of menu items and their initial state.

See Also: *CalcPopup*

### **DiagramNameText (create diagram name text)**

**DiagramNameText**( *showtext* )

<b>DiagramNameText</b>	Forms the text of diagram's name
<i>showtext</i>	Enable or suppress text to be returned. Default is TRUE.

The **DiagramNameText** method forms the text of a diagram's name. A diagram name is the text name assigned to a given set of the graph control's data points.

The *showtext* parameter defaults to returning the diagram name text. If the *showtext* Boolean value is set to FALSE, an empty string value is returned.

Return Data Type: STRING

Implementation: The **DiagramNameText** method is called from any method that needs to update or refresh a graph control's textual elements.

See Also: ToShowValues, AllText

### **DiagramText (create diagram name text with prompts)**

**DiagramText**( *showtext* )

<b>DiagramText</b>	Forms the full text of a diagram's name
<i>showtext</i>	Enable or suppress text to be returned. Default is TRUE.

The **DiagramText** method forms the full text of a diagram's name. This includes any special prompt information attached to the diagram name. A diagram name is the text name assigned to a given set of the graph control's data points.

The *showtext* parameter defaults to returning the diagram name text. If the *showtext* Boolean value is set to FALSE, an empty string value is returned.

Return Data Type: STRING

Implementation: The **DiagramText** method is called from any method that needs to update or refresh a graph control's textual elements.

See Also: ToShowValues, AllText  
DiagramNameText

## **DiagramNameText (create diagram name text)**

**DiagramNameText**( *showtext* )

**DiagramNameText** Forms the text of diagram's name

*showtext* Enable or suppress text to be returned. Default is TRUE.

The **DiagramNameText** method forms the text of a diagram's name. A diagram name is the text name assigned to a given set of the graph control's data points.

The *showtext* parameter defaults to returning the diagram name text. If the *showtext* Boolean value is set to FALSE, an empty string value is returned.

Return Data Type: STRING

Implementation: The **DiagramNameText** method is called from any method that needs to update or refresh a graph control's textual elements.

See Also: ToShowValues, AllText

## **DiagramText (create diagram name text with prompts)**

**DiagramText**( *showtext* )

**DiagramText** Forms the full text of a diagram's name

*showtext* Enable or suppress text to be returned. Default is TRUE.

The **DiagramText** method forms the full text of a diagram's name. This includes any special prompt information attached to the diagram name. A diagram name is the text name assigned to a given set of the graph control's data points.

The *showtext* parameter defaults to returning the diagram name text. If the *showtext* Boolean value is set to FALSE, an empty string value is returned.

Return Data Type: STRING

Implementation: The **DiagramText** method is called from any method that needs to update or refresh a graph control's textual elements.

See Also: ToShowValues, AllText  
DiagramNameText

## **Draw (calculate and draw GraphClass object)**

### **Draw**

**Draw** Calculates and draws the graph object

The **Draw** method calculates and draws the graph object. It is a parent method that calls other processing methods and divides the individual drawing tasks.

Implementation: The **Draw** method is called from the Resize method which is called from the Refresh method

See Also:       Resize  
                  Refresh  
                  CalcGraph  
                  DrawGraph

## **DrawGraph (draws calculated values)**

### **DrawGraph**

**DrawGraph**       Draws the graph object

The **DrawGraph** method draws the graph object, based on calculated values. Its primary purpose is to determine which parts of the graph need to be drawn based on graph type and values.

Implementation: The **DrawGraph** method is called after the CalcGraph method has set the appropriate graph control's values.

See Also:       Draw CalcGraph

## **DrawReport (draw graph object on report)**

**DrawReport** (*parReport*, *parBand*, *parQueue* , [*parBestFit*])

<b>DrawReport</b>	Draws a copy of the graph object into a REPORT structure
<i>parReport</i>	A reference to the target REPORT structure
<i>parBand</i>	A LONG that references the FEQ of the target detail band.
<i>parQueue</i>	The label of the report's print preview queue.
<i>parBestFit</i>	A BYTE value that calculates a best fit when TRUE (Default is FALSE)

The **DrawReport** method draws a copy of the graph object into a REPORT structure and prepares WMF-files for subsequent viewing and printing.

Implementation: The **DrawReport** method is called from the PrintGraph method, prior to the Previewer object is initialized.

See Also: PrintGraph

## **DrawWallpaper (draw background wallpaper for graph object)**

**DrawWallpaper**

**DrawWallpaper** Draws the graph object background wallpaper

The **DrawWallpaper** method draws the graph object background wallpaper. The name of the image file should be set in the *eWallpaperFile* property.

Implementation: The **DrawWallpaper** method is called from the DrawGraph method.

See Also: DrawGraph

**DrillDown (transfer control to new graph object)**

**Drilldown** (| *parGraphNpp*, *graphName*, *NodeID*, *NodeName*, *Xnode*, *Ynode* |)

<b>Drilldown</b>	Unhides, calculates and draws the target graph object
<i>parGraphNpp</i>	A LONG integer that identifies the GraphID FEQ to process
<i>graphName</i>	String text that identifies the graph name.
<i>NodeID</i>	The positional node value of the active graph control that is passed to the drilldown graph control.
<i>NodeName</i>	The positional node name of the active graph control that is passed to the drilldown graph control.
<i>Xnode</i>	The positional X Node coordinate (in Physical Units) of the active graph control that is passed to the drilldown graph control.
<i>Ynode</i>	The positional Y Node coordinate (in Physical Units) of the active graph control that is passed to the drilldown graph control.

The **Drilldown** method hides the current active graph control, and unhides the target (drilldown) graph control. The target graph object's data points are calculated and drawn.

Optional parameters can be processed by the target drilldown graph control. This allows the target control to be related to the original graph object.

Implementation: The **Drilldown** method is called from the TakeEventOfParent method, when a Drilldown event is posted.

See Also: TakeEventOfParent

ReturnFromDrillDown

**FindNearbyNodes (locate nodes based on mouse position)****FindNearbyNodes**( *Xpos*, *Ypos*)**FindNearbyNodes** Finds the closest node based on X and Y Mouse coordinates*Xpos* The current X mouse coordinate position*Ypos* The current Y mouse coordinate position

The **FindNearbyNodes** method returns the closest node id based on the x and y mouse coordinates.

Return Data Type: LONG

Implementation: The **FindNearbyNodes** method is called from the Interactivity method. It is also called in the Drilldown method to accurately return the proper node id when the mouse is clicked.

See Also: Drilldown, Interactivity

## **GetMouse (get mouse coordinates in all formats)**

### **GetMouse**

The GetMouse method is used to retrieve mouse coordinates each time a mouse event has occurred on the graph control.

Implementation: The **GetMouse** method is called from the TakeEventOfParent method, and sets the following properties:

! Coordinates of the mouse in DC units

self.eMouseX (see mouseX())

self.eMouseY (see mouseY())

! Coordinates of the mouse in logic (independent) units

self.eMouseXI

self.eMouseYI

! Coordinates of the mouse in physical (node)units

self.eMouseXa

self.eMouseYa

See Also: TakeEventOfParent

### **GetValueFromField (get contents of specified field)**

**GetValueFromField** ( *fieldeq* )

**GetValueFromField** Retrieve text from a specified control.

*fieldeq* An LONG integer that indicates the field equate label of the control the text is to be retrieved.

The **GetValueFromField** method is used to return the contents of a specified field for use in other areas of the graph control (i.e., tool tips).

Return Data Type: STRING

Implementation: The **GetValueFromField** method is called from the ToShowValues method.

See Also: AllText, ToShowValues

### **GetValueFromStatusBar (return status bar zone contents)**

**GetValueFromStatusBar**( *statuszone* )

**GetValueFromStatusBar** Retrieve text from a window status bar.

*statuszone* An LONG integer that indicates which status bar section the text is to be retrieved from.

The **GetValueFromStatusBar** method is used to return the contents of a selected window's status bar zone for use in other areas of the graph control (i.e., tool tips).

Return Data Type: STRING

Implementation: The **GetValueFromStatusBar** method is called from the ToShowValues method.

See Also: AllText, ToShowValues

## **ImageToWMF (Save object and return WMF file name)**

### **ImagetoWMF**

The **ImagetoWMF** method is used to transfer a graph object's current state to a WMF file. This file can be used later for printing or additional processing. **ImagetoWMF** returns the name of the WMF file that the graph image is transferred to.

Return Data Type:            **STRING**

Implementation:            The **ImagetoWMF** method is called from the SaveGraph method.

See Also:                 **SaveGraph**

## **Init (Initialize the graph object)**

**Init** (*parWin, parFParent, parL=0, parT=0, parR=0, parB=0*)

**Init**            Initializes the GraphClass object

*parWin*        The reference to a window where the object is located.

*parFParent*    A LONG identifying the parent id of the graph object.

*parL, parT, parR, parB*

                  Indents from borders of the parent for drawing object.

The **Init** method is used to initialize the graph object prior to display. It calls all of the necessary methods used to set default values and position the graph object.

Implementation:            Typically, the **Init** method is paired with the Kill method, performing the converse of the Kill method tasks.

See Also:                 **Kill**

## **Interactivity (process mouse location data to tool tip or control)**

### **Interactivity**

The **Interactivity** method processes the mouse coordinates of a graph control, and finds corresponding nodes of diagrams and displays it on a ToolTip or other appropriate target. It is used to keep the mouse and node data in sync.

Implementation: The **Interactivity** method calls several methods (FindNearbyNodes, SetSelectedNode and ToShowValues) to gather node data. It is called during a MouseMove event.

See Also: TakeEventOfParent

## **IsOverNode ( is mouse over node location)**

### **IsOverNode( )**

The **IsOverNode** method is used to detect a node location to pass to the drilldown procedure. A drilldown graph control is normally selected by doubleclicking on a selected node, but the **IsOverNode** method allows node data to be passed when selecting a drilldown graph from the popup menu.

**IsOverNode** returns TRUE when a valid node was in range when the mouse right-click was pressed. If the user right-clicks over an area where no node is present, the "Drilldown" or "Return from DrillDown" menu options will be disabled.

Return Data Type: BOOL

Implementation: The **IsOverNode** method is called from the CalcPopup method.

See Also: CalcPopup

## **Kill (shut down the GraphClass object)**

### **Kill**

The **Kill** method frees any memory allocated during the life of the GraphClass object and performs any other required termination code.

## **MouseEvent (creates text and mouse coordinate information)**

### **MouseEvent( *parShow* )**

**MouseEvent** Returns text of the active X and Y mouse coordinates.

*parShow* Flag to conditionally suppress return value.

The **MouseEvent** method returns the text of the active X and Y mouse coordinates. If *parShow* is set to FALSE, the text is suppressed, and an empty string is returned.

Return Data Type: STRING

Implementation: The **MouseEvent** method is called from the ToShowValues and AllText method. It is also used to form the text for the ToolTip method.

See Also: AllText, ToShowValues

## **MouseXText (generate X coordinate text only)**

### **MouseXText( *parShow* )**

**MouseXText** Returns text of the active X mouse coordinate.

*parShow* Flag to conditionally suppress return value.

The **MouseXText** method returns the text of the active X mouse coordinate. If *parShow* is set to FALSE, the node text is suppressed, and an empty string is returned.

Return Data Type: STRING

Implementation: The **MouseXText** method is called from the MouseEvent method.

See Also: AllText, MouseEvent

### **MouseYText (generate Y coordinate text only)**

**MouseYText**( *parShow* )

**MouseYText** Returns text of the active Y mouse coordinate.

*parShow* Flag to conditionally suppress return value.

The **MouseYText** method returns the text of the active Y mouse coordinate. If *parShow* is set to FALSE, the node text is suppressed, and an empty string is returned.

Return Data Type: STRING

Implementation: The **MouseYText** method is called from the MouseText method.

See Also: AllText, MouseText

### **NodeNameText (generate current node name identifier)**

**NodeNameText**( *parShow* )

**NodeNameText** Returns text of the active X and Y Node names.

*parShow* Flag to conditionally suppress return value.

The **NodeNameText** method returns the text of the active X and Y Node names. If *parShow* is set to FALSE, the node text is suppressed, and an empty string is returned.

Return Data Type: STRING

Implementation: The **NodeNameText** method is called from the NodeText method.

See Also: AllText, NodeText

### **NodeText (generate label, name, and node value)**

**NodeText**( *parShow* )

**NodeText** Returns text of the active X and Y Node name and values.

*parShow* Flag to conditionally suppress return value.

The **NodeText** method returns the full text of the active X and Y Node name and values. If *parShow* is set to FALSE, the node text is suppressed, and an empty string is returned.

Return Data Type: STRING

Implementation: The **NodeText** method is called from the ToShowValues and AllText methods.

See Also: AllText

### **NodeTipText (generate node information for tool tip)**

**NodeTipText**( *parShow* )

**NodeTipText** Returns text of the active node for the tool tip.

*parShow* Flag to conditionally suppress return value.

The **NodeTipText** method returns the tool tip text of the active X and Y Node values. If *parShow* is set to FALSE, the return text is suppressed, and an empty string is returned.

Return Data Type: STRING

Implementation: The **NodeTipText** method is called from the graph object's ToolTip method.

See Also: AllText, ToolTip

### **NodeValueText (generate current node value text)**

**NodeValueText**( *parShow* )

**NodeValueText** Returns text of the active X and Y Node values.

*parShow* Flag to conditionally suppress return value.

The **NodeValueText** method returns the text of the active X and Y Node values. If *parShow* is set to FALSE, the node text is suppressed, and an empty string is returned.

Return Data Type: STRING

Implementation: The **NodeValueText** method is called from the method.

See Also: AllText

### **NodeXText (generate X node text value)**

**NodeXText**( *parShow* )

**NodeXText** Returns text of the active X Node value.

*parShow* Flag to conditionally suppress return value.

The **NodeXText** method returns the text of the active X Node value. If *parShow* is set to FALSE, the node text is suppressed, and an empty string is returned.

Return Data Type: STRING

Implementation: The **NodeXText** method is called from the NodeValueText method.

See Also: AllText, NodeValueText

## **NodeYText (generate Y node text value)**

**NodeYText**( *parShow* )

**NodeYText**

Returns text of the active Y Node value.

*parShow*

Flag to conditionally suppress return value.

The **NodeYText** method returns the text of the active Y Node value. If *parShow* is set to FALSE, the node text is suppressed, and an empty string is returned.

Return Data Type: STRING

Implementation: The **NodeYText** method is called from the NodeValueText method.

See Also: AllText, NodeValueText

## **Popup (GraphClass object popup menu manager)**

**Popup**

**Popup**

Creates, displays, and processes the graph object popup menu

The **Popup** method is used to create, display and process the graph object's popup menu (when enabled).

Implementation: The **Popup** method is called if the user right-clicks on a graph control, and the *self.ePopUp* property is set to TRUE.

See Also: CalcPopup

**PopupAsk (Display popup menu for graph object)****PopupAsk**( *PopupMgr* )**PopupAsk**

Returns the selected popup menu item name

*PopupMgr*

The reference to an object of a PopupClass type.

The **PopupAsk** method is used to display and return the graph object popup menu item selected.

Implementation: The **PopupAsk** method is called from the Popup method, and invokes the Ask method references by the Popup Class.

See Also: [PopupMgr.Ask](#)

**PostEvent (send an event to the GraphClass object)****PostEvent** ( *eventnumber* )**PostEvent**

Post event to graph object

*eventnumber*

A LONG value that passes the event number to the graph object

The **PostEvent** method posts an event to the graph object. The graph object event is processed by the appropriate TakeEvent method.

Graph Events are template-defined events, and listed in the SVGRAPH.EQU source file:

```

itemize(3000) ! Events numbered from 3000
event:TitleON      equate ! To show title
event:TitleOFF     equate ! To not show title
event:WallpaperON  equate ! To show Wallpaper
event:WallpaperOFF equate ! To not show Wallpaper
event:3DON         equate ! To draw in 3D mode (if accessible)
event:3DOFF       equate ! To switch off 3D a mode (if accessible)
event:GridON      equate ! To draw a grid (if accessible)
event:GridOFF     equate ! To hide a grid (if accessible)
event:GridXON    equate
event:GridXOFF   equate
event:GridYON    equate
event:GridYOFF   equate
event:AxisScaleMinMaxON  equate
event:AxisScaleMinMaxOFF equate
event:AxisNameON  equate ! To show the names of Axis
event:AxisNameOFF equate ! To not show the names of Axis
event:GradientON  equate
event:GradientOFF equate
event:NodeMinMaxON  equate ! To show units of a minimum/maximum
event:NodeMinMaxOFF equate ! To not show units of a minimum/maximum
event:NodeLabelON  equate ! To show the names of nodes
event:NodeLabelOFF equate ! To not show the names of nodes
event:NodeValueON  equate ! To show the values of nodes
event:NodeValueOFF equate ! To not show the values of nodes
event:NodeBgrON    equate
event:NodeBgrOFF   equate
event:LegendBoxON  equate
event:LegendBoxOFF equate
event:ToolTipON    equate
event:ToolTipOFF   equate
event:Zoom         equate
event:Zoom500      equate
event:Zoom300      equate
event:Zoom200      equate

```

```

event:Zoom100          equate
event:Zoom50           equate
event:Zoom25           equate
event:GraphTypeLine   equate
event:GraphTypeScatterGraph equate
event:GraphTypeAreaGraph equate
event:GraphTypeFloatingArea equate
event:GraphTypeColumnChart equate
event:GraphTypeColumnWithAccumulation equate
event:GraphTypeBarChart equate
event:GraphTypeBarWithAccumulation equate
event:GraphTypeFloatingColumn equate
event:GraphTypeFloatingBar equate
event:GraphTypePieChart equate
event:GraphSubTypeSimple equate
event:GraphSubTypeNormalized equate
event:FigureTypeBar equate
event:FigureTypeCylinder equate
event:LegendPosition:None equate      ! To not show legend
event:LegendPosition:Left equate
event:LegendPosition:Right equate
event:LegendPosition:Top equate
event:LegendPosition:Bottom equate
event:AxisStyle:None equate          ! To not show Axis
event:AxisStyle:Standard equate
event:AxisStyle:Long equate
event:AxisScale equate
event:AxisScale:Linear equate
event:AxisScale:AsMSWord equate
event:NodeType:None equate          ! To not show nodes
event:NodeType:Square equate        ! To set a type of node as a square
event:NodeType:Triangle equate      ! To set a type of node as a triangle
event:NodeType:Circle equate        ! To set a type of node as a circle
event:Refresh equate               ! To refresh object
event:Draw equate                  ! To draw object
event:Hide equate                  ! To hide object
event:UnHide equate                ! To unhide object
event:Print equate                 ! To print diagram
event:PrintBestFit equate
event:Save equate                  ! To save diagram
event:SaveAs equate               ! To save the diagram under a new name
event:DrillDown equate
event:ReturnFromDrillDown equate
end

```

See Also:      TakeEvent

**PrintGraph (send graph object to printer)****PrintGraph**( *bestfit* )**PrintGraph**

Pre-processes and draws graph object

*bestfit*

A Boolean value that determines best fit or relative printing

The method allows to choose the printer, draw the graph object (using the DrawReport method), Preview the report, and send pages to the printer.

If the *bestfit* flag is set to FALSE (default), the graph object will print "as is", anchoring the object at the top left corner of the paper.

If the *bestfit* flag is set to TRUE, the graph object will resize to a best fit within the band that it is populated.

Implementation:       The **PrintGraph** method is called when graph events Event:Print, or Event:PrintBestFit are posted.

See Also:            DrawReport  
                      TakeEventOfParent

## Refresh (refresh drawing of GraphClass object)

**Refresh**( *parRefresh* )

**Refresh**                      Pre-processes and draws graph object

*parRefresh*                      A Boolean value that determines refresh is active

The **Refresh** method is used to redraw and resize the graph object. This method is called throughout the graph control template when any element of the graph has changed. You can also use the **Refresh** method when data has changed, or initialization of a drilldown graph object is needed.

If the *parRefresh* flag is FALSE (default), the graph object will not be refresh. The BeginRefresh method is used to pre-process the Refresh method, and set the *parRefresh* flag to TRUE if a refresh is ready to execute.

Implementation:                      The **Refresh** method is called when a graph event (Event:Refresh) is posted.

See Also:                      BeginRefresh

  Resize

## Resize (conditional refresh when size changed)

**Resize**( *redraw* )

**Resize**                      Redraws graph object

*redraw*                      A Boolean value that determines conditional redraw

The **Resize** method is used to reposition and redraw the graph object.

If the *redraw* flag is FALSE (default), the graph object will only redraw if the size of the parent field has changed. If the redraw flag is set to TRUE, a redraw is always executed.

Implementation:                      The **Resize** method is called from the Refresh method when the parRefresh property is set to TRUE.

See Also:                      Refresh

  Draw

**ReturnFromDrillDown ( transfer control to graph object after drilldown)****ReturnFromDrillDown**

(| *parGraphNpp* ,*graphName* ,*NodeID* ,*NodeName* ,*Xnode* ,*Ynode* |)

**ReturnFromDrilldown**

Unhides, calculates and draws the original graph object

<i>parGraphNpp</i>	A LONG integer that identifies the GraphID FEQ to process
<i>graphName</i>	String text that identifies the original graph name.
<i>NodeID</i>	The positional node value of the drilldown graph control that is passed to the active graph control.
<i>NodeName</i>	The positional node name of the drilldown graph control that is passed to the active graph control.
<i>Xnode</i>	The positional X Node coordinate of the drilldown graph control that is passed to the active graph control.
<i>Ynode</i>	The positional Y Node coordinate of the drilldown graph control that is passed to the active graph control.

The **ReturnFromDrilldown** method is used to "drill back" to the original graph object. It is designed to restore the original graph object's state prior to calling the Drilldown method.

Optional parameters can be processed by the target "drill back" graph control. This allows the target control to be related to the drilldown graph object.

Implementation: The **ReturnFromDrilldown** method is called from the TakeEventOfParent method when the graph event Event:ReturnFromDrillDown is posted. This event is triggered when the user double-clicks on a valid graph node, or selects the "Return from DrillDown" popup menu item.

See Also: DrillDown  
TakeEventOfParent

## **SaveAsGraph (save graph to WMF file selected)**

### **SaveAsGraph**

**SaveAsGraph** Save graph to WMF file.

The **SaveAsGraph** method writes the current state of the graph object to a WMF file. The user is prompted to enter a valid WMF file name.

Implementation: The **SaveAsGraph** method calls the SaveGraph method, forcing the *askSave* property to TRUE. The SaveAsGraph method is called when a graph event (Event:SaveAs) is posted.

See Also: SaveGraph

## **SaveGraph (auto-save graph to WMF file)**

### **SaveGraph( *askSave* )**

**SaveGraph** Save graph to WMF file.

*askSave* A Boolean value that determines whether or not to prompt the user to save. The default value is FALSE (save without asking)

The **SaveGraph** method writes the current state of the graph object to a WMF file. If the *askSave* parameter is set to TRUE, the user is first prompted to enter a valid WMF file name.

Implementation: The **SaveGraph** method is called when a graph event (Event:Save) is posted.

See Also: ImageToWMF

## **SetDefault (initialize selected graph properties)**

### **SetDefault**

**SetDefault** Set initial values on the graph control.

The **SetDefault** method initializes critical properties of the graph object prior to its data point calculations and display. Most of these properties are found in the Initial Settings section of the Graph Control template.

Implementation: The **SetDefault** method is called from the graph object's Init method.

See Also: Init

## **ShowOnField (show text contents to specified field)**

### **ShowOnField** (*text, fieldequ*)

**ShowOnField** Show text on a selected control.

*text* A string constant or variable that contains the text to display on the control.

*fieldequ* An LONG integer that indicates the field equate label of the control the text is to be displayed.

The **ShowOnField** method displays a *text* value into a given control, specified by its field equate (*fieldequ*). The control must be a valid target (string, entry, text, or combo). Other types of controls are assigned the text value using the PROP:Text property.

Implementation: The **ShowOnField** method is called from the ToShowValues method.

See Also: ToShowValues

## **ShowOnStatusBar (show text to status bar zone)**

**ShowOnStatusBar**( *text*, *zone* )

**ShowOnStatusBar** Show text on the window status bar.

*text* A string constant or variable that contains the text to display on the status bar.

*zone* An LONG integer that indicates which status bar section the text is to be displayed.

The **ShowOnStatusBar** method displays a *text* value into a given status bar zone, specified by *zone*.

Implementation: The **ShowOnStatusBar** method is called from the ToShowValues method. If a window has a separate status bar defined, the *eShowSBonFirstThread* property determines the target.

See Also: ToShowValues  
eShowSBonFirstThread

## **TakeEvent (process graph control events)**

**TakeEvent**

**TakeEvent** Processes the events of the graph object.

The **TakeEvent** method is used to process events returned by the graph object. These events are returned by the parent control of the graph object (by default, a GROUP) and to other support controls of the object. In general, all events should be posted to the parent field of the graph object.

Implementation: The **TakeEvent** method is called within the graph object procedure's ACCEPT event handler. It provides a virtual method in which the developer can use to trap and process all graph control events.

See Also: TakeEventOfParent

**TakeEventofParent (process all graph events)****TakeEventOfParent****TakeEventOfParent** Processes the events of the graph object's parent control.

By default, all graph events are processed by the graph object's parent control (by default, a GROUP). This method is the control center from which other methods are executed and properties are set.

Implementation: The **TakeEventOfParent** method is called from the TakeEvent method.

See Also: TakeEvent

**ToolTip (show all text to tool tips)****ToolTip****ToolTip** Show tool tip text.

The **ToolTip** method displays all text that is directed to the graph object's tool tip box.

Implementation: The **ToolTip** method is called from the Interactivity method.

See Also: ToShowValues

AllText

Interactivity

**ToShowValues (show all composite text to all graph targets)****ToShowValues****ToShowValues**      Control what text is shown where.

The **ToShowValues** method checks a variety of properties that are set by the graph control template and popup menu, and filter what specific text should be displayed on which target. For example, you may have omitted mouse text from the graph control's tool tip, or limit a control to only displaying the X node values.

Implementation:      The **ToShowValues** method is called from the Interactivity method.

See Also:      [Interactivity](#)



# GridClass

## GridClass Overview

The GridClass is used to manage and control BrowseGrid controls. These combined controls make up the BrowseGrid control template. At runtime, the BrowseGrid replaces the underlying BrowseBox and arranges the data in as groups within a matrix format. The data included in this matrix includes all controls populated within the BrowseGrid GROUP control.

The BrowseGrid template is most useful for displaying product lists on a web page, but it may be used on desktop applications as well.

## Relationship to Other Application Builder Classes

The GridClass is derived from the BrowseClass. It also implements the IListControl interface which defines the set of behaviors that relate to a LIST control.

## GridClass ABC Template Implementation

The ABC BrowseGrid template generates code to instantiate the GridClass for your BrowseBoxes. The GridClass objects are called BRW $n$ , where  $n$  is the template instance number. Multiple grids may be used in a single procedure.

## GridClass Source Files

The GridClass source code is installed by default to the Clarion \LIBSRC folder. The specific GridClass source code and its respective components are contained in:

ABGRID.INC	GridClass declarations
ABGRID.CLW	GridClass method definitions

## GridClass Properties

### GridClass Properties

The GridClass inherits all the properties of the BrowseClass from which it is derived.

### Children (reference to child group controls)

**Children**      **&ChildQueue, PROTECTED**

The **Children** property is a reference to a structure containing a list of child controls of the basic GROUP grid control. This queue also contains the position of each of the child controls.

Implementation:      This property is initialized in the GridClass.Init method. The QUEUE is loaded with data in **GridClass.CheckChildren** which is a **PRIVATE** method.

The Children property refers to a QUEUE declared in ABGRID.INC.

```
ChildQueue QUEUE, TYPE
Feq            SIGNED
XD            SIGNED
YD            SIGNED
HE            SIGNED
WI            SIGNED
Use            ANY
                  END
```

See Also:              GridClass.Init

### Chosen (current browse queue element)

**Chosen**            **SIGNED, PROTECTED**

The **Chosen** property contains the current browse queue element number.

See Also:              GridClass.TakeEvent

## ClickPress (forward control)

**ClickPress**      **SIGNED, PROTECTED**

The **ClickPress** property specifies a control number in which an EVENT:Accepted will be posted to when a mouse click occurs. In a Windows application the mouse click is a DOUBLE-CLICK. An application running in an internet browser will accept the ClickPress control on a single click.

Implementation:      The BrowseGrid template assigns the ClickPress control based on the control supplied to the template. The ClickPress action occurs in the GridClass.TakeEvent method.

See Also:              GridClass.TakeEvent

## ControlBase (base control number)

**ControlBase**      **SIGNED, PROTECTED**

The **ControlBase** property specifies a base control number that other child control feq values are based on.

Implementation:      This property is initialized to 1000 in the GridClass.CheckChildren method.

See Also:              GridClass.CheckChildren

## ControlNumber (number of controls)

**ControlNumber**      **SIGNED, PROTECTED**

The **ControlNumber** property specifies the number of controls used per grid element.

Implementation:      This property is calculated in the GridClass.CheckChildren method.

See Also:              GridClass.CheckChildren

### **GroupColor (background color of group fields)**

**GroupColor**      **LONG, PROTECTED**

The **GroupColor** property specifies the background color of the GROUP control. This is used as the background color for all fields displayed in the GROUP as well as for the Group Title. The GROUP text color is also used for these fields.

### **GroupControl (GROUP control number)**

**GroupControl**      **SIGNED, PROTECTED**

The **GroupControl** property specifies the control number for the GROUP control used by the BrowseGrid. The GROUP control is automatically populated when the BrowseGrid control template is used.

Implementation:      The ABC Templates automatically assigns this property its value. If the template is not being used this value should be initializes in the Init method.

### **GroupTitle (title of group element)**

**GroupTitle**      **ASTRING, PROTECTED**

The **GroupTitle** property contains a string to use as the title for the group box that outlines each entry in the grid.

Implementation:      The ABC Templates automatically assigns this property its value. If the template is not being used this value should be initializes in the Init method.

### **SelColor (color of selected element)**

**SelColor**      **LONG, PROTECTED**

The **SelColor** property specifies the color of the currently selected element of the grid.

Implementation:      The ABC Templates automatically assigns this property its value. If the template is not being used this value should be initializes in the Init method.

### **Selectable (element selectable flag)**

**Selectable**      **BYTE**

The **Selectable** property indicates that an element (group) in the grid is selectable. Clicking on any elemen in the group will make it the currently selected element of the queue.

Implementation:      The ABC Templates automatically assigns this property its value. If the template is not being used this value should be initializes in the Init method.

### **UpdateControl (file update trigger)**

**UpdateControl**      **SIGNED**

The **UpdateControl** property specifies a contol that when accepted will trigger a file update.

Implementation:      The ABC Templates automatically assigns this property its value. If the template is not being used this value should be initializes in the Init method.

### **UpdateControlEvents**

**UpdateControlEvents**      **SIGNED**

The **UpdateControlEvents** property is not currently implemented.

## GridClass Methods

The GridClass inherits all the methods of the BrowseClass from which it is derived.

### AddLocator (specify a locator)

**AddLocator**(*locator*)

<b>AddLocator</b>	Specifies a locator object for a specific sort order.
<i>locator</i>	The label of the locator object.

The **AddLocator** method specifies a locator object for the sort order defined by the preceding call to the AddSortOrder or SetSort method.

Implementation: The specified *locator* is sort order specific. It is enabled only when the associated sort order is active.

See Also: BrowseClass.AddSortOrder, BrowseClass.AddLocator

### FetchRecord (retrieve selected record)

**FetchRecord**(*record*)

<b>FetchRecord</b>	Retrieve selected record.
<i>record</i>	An integer constant, variable, EQUATE, or expression that indicates which entry number in the LIST to read.

The **FetchRecord** method is used in conjunction with the WebBuilder WebGridExtension. This method is used to read the currently selected record into memory.

Implementation: This method is called by the WbGridHTMLProperties.SetProperty method.

## **GetAcross (number of horizontal grids)**

### **GetAcross**

The **GetAcross** method returns the number of GROUPs that fit into the designed listbox.

Implementation: The GetAcross method returns a value that is determined in the GridClass.GetDimensions PRIVATE method.

Return Data Type: SIGNED

## **GetClickPress (forward click control)**

### **GetClickPress**

The **GetClickPress** method returns the control number of the control to be accepted. This control is specified in the BrowseGrid template by the *Forward other control clicks to:* prompt.

Implementation: The GridClass.TakeEvent method executes code that will post an EVENT:Accepted to SELF.ClickPress, which is the value that this method returns.

Return Data Type: SIGNED

See Also: GridClass.TakeEvent

## **GetDown (number of vertical grids )**

### **GetDown( )**

The **GetDown** method returns the number of GROUPs that fit into the designed listbox.

Implementation: The GetDown method returns a value that is determined in the GridClass.GetDimensions PRIVATE method.

Return Data Type: SIGNED

**GetPosition (retrieve group control position)****GetPosition**(*instance*, *XPos*, *Ypos*), **VIRTUAL, PROTECTED**

<b>GetPosition</b>	Sets the XPosition and YPosition for the control <i>instance</i> .
<i>instance</i>	A numeric constant, variable, EQUATE, or expression containing the control number.
<i>XPos</i>	An integer constant, variable, EQUATE, or expression that indicates the X position of the specific GROUP control instance.
<i>YPos</i>	An integer constant, variable, EQUATE, or expression that indicates the Yposition of the specific GROUP control instance.

Implementation: The GetPosition method is called to set the X and Y positions of the GROUP control is created to simulated the grid appearance. This control is created in the GridClass.CreateControls PRIVATE method.

**IfGroupField (determine if current control is a GROUP)****IfGroupField**(*field*)

<b>IfGroupField</b>	Determines if the control in focus is a GROUP control.
<i>field</i>	A numeric constant, variable, EQUATE, or expression containing the control number of the control that currently has focus.

The **IfGroupField** method determines if the control currently in focus is a GROUP control. A one (1 or True) is returned if the control is GROUP control, otherwise a zero (0 or False) is returned.

Implementation: This method is called by the GridClass.TakeEvent method when an event occurs on any control defined as part of the BrowseGrid.

See Also: GridClass.TakeEvent

Return Data Type: **BYTE**

## Init (initialize the GridClass object)

Init(listcontrol, viewposit, view listqueue, relationmanager, windowmanager)

<b>Init</b>	Initializes the GridClass object.
<i>listcontrol</i>	A numeric constant, variable, EQUATE, or expression containing the control number of the LIST control.
<i>viewposition</i>	The label of a string field withing the <i>listqueue</i> containing the POSITION of the view.
<i>view</i>	The label of the browse's underlying VIEW.
<i>listqueue</i>	The label of the <i>listcontrol</i> 's data source QUEUE.
<i>relationmanager</i>	The label of the browse's primary file RelationManager object. See <i>Relation Manager</i> for more information.
<i>windowmanager</i>	The label of the browse'WindowManager object. See <i>Window Manager</i> for more information.

The **Init** method initializes the GridClass object.

Implementation: In addition to other things, the Init method calls the PARENT.Init method to initialize the browse's ViewManager object.

See Also: BrowseClass.Init

## IsSkelActive

IsSkelActive, VIRTUAL

The **IsSkelActive** method is not implemented at this time.

Return Data Type: BYTE

### **Kill (shutdown the GridClass object)**

Kill, DERIVED

The **Kill** method shuts down the GridClass object. It frees any memory allocated during the life of the objects and performs any other necessary cleanup code.

### **SetAlerts (initialize and create child controls)**

SetAlerts, DERIVED

The **SetAlerts** method initiates the managing and storage of child controls to the BrowseGrid.

### **SyncGroup (initialize GROUP field properties)**

**SyncGroup**(*record*)

<b>SyncGroup</b>	Initialize the GROUP field properties.
<i>record</i>	An integer constant, variable, EQUATE, or expression that indicates which entry number in the LIST to read.

The **SyncGroup** method is used in conjunction with the WebBuilder WebGridExtension. This method is used to initialize the GROUP control properties such as Group Title, Group Box, and Group color.

Implementation: This method is called by the WbGridHTMLProperties.SetProperty method.

## TakeEvent (process the current ACCEPT loop event)

### TakeEvent, DERIVED

The **TakeEvent** method processes the current ACCEPT loop for the GridClass object. The method handles all events associated with the Grid controls including the parent GROUP control and all of its children.

Implementation: When this method completes its PARENT method is called.

See Also: BrowseClass.TakeEvent

## UpdateRecord (refresh BrowseGrid)

### UpdateRecord (*record*), VIRTUAL

<b>UpdateRecord</b>	Refresh the data in the BrowseGrid.
<i>record</i>	A numeric constant, variable, EQUATE, or expression that indicates which entry number in the LIST to update.

The **UpdateRecord** method refreshes the data shown in the BrowseGrid by calling the FileManager to post data updates. The BrowseClass.ResetFromBuffer is then called to update the queue that is used for the LIST and then the BrowseClass.UpdateViewRecord to update the VIEW.

Implementation: This method is called by the GridClass.TakeEvent method.

See Also: GridClass.TakeEvent, FileManager.Update, BrowseClass.ResetFromBuffer, BrowseClass.UpdateViewRecord

## UpdateWindow (refresh window display)

### UpdateWindow, DERIVED

The **UpdateWindow** method calls the PARENT UpdateWindow method to update all display variables and refreshes the BrowseGrid control values and properties.

Implementation: The UpdateWindow method is called by the GridClass.TakeEvent method.

See Also: BrowseClass.UpdateWindow. GridClass.TakeEvent



# HistHandlerClass

## HistHandlerClass Source Files

The HistHandlerClass source code is installed by default to the Clarion \LIBSRC. The specific ErrorClass source code and their respective components are contained in:

ABERROR.INC HistHandlerClass declarations  
ABERROR.CLW HistHandlerClass method definitions  
ABERROR.TRN HistHandlerClass default error definitions

## HistHandlerClass Properties

The HistHandlerClass contains the following properties:

### Err (errorclass object)

**Err**      **&ErrorClass, PROTECTED**

The **Err** property is a reference to the ErrorClass object that handles any runtime errors.

### History (error history structure)

**History**      **&ErrorHistoryList,PROTECTED**

The **History** property is a reference to the ErrorHistoryList structure that holds the history for errors that have previously occurred. The error History is determined based on the HistoryThreshold and HistoryResetOnView properties.

**LBColumns (number of listbox columns)****LBColumns      SHORT, PROTECTED**

The **LBColumns** property represents the number of columns in the list box on the MsgBox window.

**Win (reference to window)****Win              &Window, PROTECTED**

The **Win** property is a reference to the message box window, which the HistHandler uses.

## HistHandlerClass Methods

The HistHandlerClass contains the following methods:

### Init (initialize the HistHandlerClass object)

**Init**(*win, err, history*)

<b>Init</b>	Initialize the HistHandlerClass object.
<i>win</i>	A reference to the message box window which the HistHandler uses.
<i>err</i>	The label of the ErrorClass object.
<i>history</i>	A reference to the ErrorHistoryList structure that holds the history for errors that have previously occurred.

The **Init** method initializes the HistHandlerClass object. This object is initialized from the ErrorClass.HistoryMsg method.

See Also:           ErrorClass.HistoryMsg

### TakeEvent (process window events)

**TakeEvent**, VIRTUAL

The **TakeEvent** method processes the MsgBox window events. In particular the OpenWindow event is processed. Upon the EVENT:OpenWindow, the LIST control is interrogated. A Level:Benign is returned.

Return Data Type:    BYTE

**VLBProc (retrieve LIST and error history information.)****VLBProc**(*rowindex*, *colindex*), VIRTUAL, PROTECTED

<b>VLBProc</b>	Retrieve LIST and error history information.
<i>rowindex</i>	An index used to determine information regarding the listbox and associated history data.
<i>colindex</i>	An index used to determine information regarding an actual error history record.

The **VLBProc** method returns various information regarding the LIST control and its data.

Implementation: If passed a *rowindex* of -1, the VLBProc returns the number of history error records that are contained in the error history queue. If passed a *rowindex* of -2, the method returns the number of listbox columns.

Any positive integer passed as the *rowindex* will cause the VLBProc to lookup the error history record at the relative *rowindex* position. The *colindex* parameter is used in association with a positive integer *rowindex*. A *colindex* value of 1 will return the error text. A *colindex* of 2 will return the error category.

Return Data Type: STRING

# IDbChangeAudit Interface

## IDbChangeAudit Concepts

The IDbChangeAudit interface defines a set of common methods the DbAuditManager object must implement in order for the object to handle additions, updates and deletions to the audit files.

## Relationship to Other Application Builder Classes

The DbAuditManager implements the IDbChangeAudit interface.

## IDbChangeAudit Source Files

The IDbChangeAudit source code is installed by default to the Clarion \LIBSRC folder. The specific IDbChangeAudit source code and their respective components are contained in:

ABFILE.INC	IDbChangeAudit interface declaration
ABFILE.CLW	DbAuditManager.IDbChangeAudit method definitions

## IDbChangeAudit Methods

The IDbChangeAudit interface defines the following methods.

### **BeforeChange (update audit log file before file change)**

**BeforeChange**(*filename*, *BFP*), VIRTUAL

#### **BeforeChange**

*filename*            A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.

*BFP*                    The label of aBufferedPairsClass object. See *BufferedPairsClass* for more information.

Implementation:      BeforeChange is a VIRTUAL method that lets you easily implement your own custom version of this method. This method can be called to process code before the record buffer is saved. This method calls the DbAuditManager.BeforeChange method.

See Also:              DbAuditManager.BeforeChange

## ChangeField (virtual method for managing field changes)

**ChangeField**(*left, right, fieldname, filename*), **VIRTUAL**

<b>ChangeField</b>	Manage field changes.
<i>left</i>	The label of the "left" field of the pair that contains the original value of the field being updated. The field may be any data type, but may not be an array.
<i>right</i>	The label of the "right" field of the pair that contains the new value of the field being updated. The field may be any data type, but may not be an array.
<i>fieldname</i>	A string constant, variable, EQUATE, or expression containing the label of the field that is to be audited.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is audited.

The **ChangeField** method is called for each field in the record that has changed. The before and after values are passed to this method. This method calls DbAuditManager.OnFieldChange.

Implementation: **ChangeField** is a VIRTUAL method so that other base class methods can directly call the OnFieldChange virtual method in a derived class. This lets you easily implement your own custom version of this method.

See Also: DbAuditManager.OnFieldChange

**OnChange (update audit log file after a record change)**

**OnChange**(*filename*, *file*), VIRTUAL

<b>OnChange</b>	Initiates an update to the audit log file after a Change to the file.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the label of the file that is to be audited.
<i>file</i>	The label of the FILE being audited.

The **OnChange** method initiates the update to the audit log file after a Change action.

Implementation: The OnChange method is a VIRTUAL method so that other base class methods can directly call the OnChange virtual method in a derived class. This lets you easily implement your own custom version of this method.

See Also: DbAuditManager.OnChange



# IListControl Interface

## IListControl Concepts

The IListControl interface is a defined set of behaviors that relate to a LIST control.

## Relationship to Other Application Builder Classes

The StandardBehavior class implements the IListControl interface.

## IListControl Source Files

The IListControl source code is installed by default to the Clarion \LIBSRC folder. The specific IListControl source code and their respective components are contained in:

ABBROWSE.INC	IListControl interface declaration
ABBROWSE.CLW	IListControl method definitions

## IListControl Methods

The IListControl interface defines the following methods.

### **Choice(returns current selection number)**

#### **Choice**

The **Choice** method returns the entry number of the highlighted record in a LIST control.

Return Data Type:      SIGNED

Example:

```
Self.CurrentChoice = SELF.ILC.Choice()  
!Returns 2 if second record in LIST is highlighted
```

### GetControl(returns control number)

#### **GetControl**

The **GetControl** method returns the control number (field equate) for a LIST control.

Return Data Type:      SIGNED

Example:

```
SELF.ListControl = Li.GetControl()
```

### GetItems(returns number of entries)

#### **GetItems**

The **GetItems** method returns the number of entries visible in a LIST control.

Return Data Type:      SIGNED

Example:

```
LI = SELF.ILC.GetItems()
```

### GetVisible(returns visibility of control)

#### **GetVisible**

The **GetVisible** method returns a value indicating whether the control is currently visible on the window. An empty string is returned if the control is not showing.

Return Data Type:      BYTE

Example:

```
IF ~SELF.ILC.GetVisible()            !If control is not visible  
    MESSAGE('Control is not visible')  
END
```

## SetChoice(change selected entry)

**SetChoice**(*newchoice*)

<b>SetChoice</b>	Changes the selected entry in a LIST control.
<i>newchoice</i>	An integer constant, variable, EQUATE, or expression that specifies the new entry to select in a LIST control.

The **SetChoice** method changes the selected entry in a LIST control.

Example:

```
SELF.ILC.SetChoice(SELF.CurrentChoice) !Set new selected entry to  
Self.CurrentChoice
```

## SetControl(change selected entry)

**SetControl**(*newcontrol*)

<b>SetControl</b>	Changes the selected control.
<i>newcontrol</i>	An integer constant, variable, EQUATE, or expression that specifies the new control to select.

The **SetControl** method changes the selected control.



# IncrementalLocatorClass

## IncrementalLocatorClass Overview

The IncrementalLocatorClass is an EntryLocatorClass that activates on each additional search character added to the search value (not when the locator control is accepted).

Use an Incremental locator when you want a multi-character search on numeric or alphanumeric keys and you want the search to take place immediately upon the end user's keystroke.

## IncrementalLocatorClass Concepts

An IncrementalLocator is a multi-character locator, with no locator control required (but strongly recommended).

The locator control may be a STRING, ENTRY, COMBO, or SPIN, however, any control other than a STRING causes the Incremental locator to behave like an Entry locator--the search is delayed until the control is accepted.

With a STRING control (or no control), when the BrowseClass LIST has focus, keyboard input characters are automatically added to the locator's search value string for each keystroke, and the BrowseClass *immediately* advances to the nearest matching record. The Backspace key removes characters from the locator's search value string.

We strongly recommend using a STRING control as the Incremental Locator control for the following reasons:

- So the search occurs *immediately* with each keystroke, and

- So the user can see the value for which the BrowseClass object is searching.

## IncrementalLocatorClass Relationship to Other Application Builder Classes

The BrowseClass uses the IncrementalLocatorClass to locate and scroll to the nearest matching item. Therefore, if your program's BrowseClass objects use an Incremental Locator, your program must instantiate the IncrementalLocatorClass for each use. Once you register the IncrementalLocatorClass object with the BrowseClass object (see BrowseClass.AddLocator), the BrowseClass object uses the IncrementalLocatorClass object as needed, with no other code required. See the Conceptual Example.

## IncrementalLocatorClass ABC Template Implementation

The ABC BrowseBox template generates code to instantiate the IncrementalLocatorClass for your BrowseBoxes. The IncrementalLocatorClass objects are called BRW*n*::Sort#:Locator, where *n* is the template instance number and # is the sort sequence (id) number. As this implies, you can have a different locator for each BrowseClass object sort order.

You can use the BrowseBox's **Locator Behavior** dialog (the **Locator Class** button) to derive from the EntryLocatorClass. The templates provide the derived class so you can modify the locator's behavior on an instance-by-instance basis.

## IncrementalLocatorClass Source Files

The IncrementalLocatorClass source code is installed by default to the Clarion \LIBSRC folder. The IncrementalLocatorClass source code and its respective components are contained in:

ABBROWSE.INC	IncrementalLocatorClass declarations
ABBROWSE.CLW	IncrementalLocatorClass method definitions

## IncrementalLocatorClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a BrowseClass object and related objects, including a IncrementalLocatorClass object. The example initializes and page-loads a LIST, then handles a number of associated events, including scrolling, updating, and locating records.

Note that the WindowManager and BrowseClass objects internally handle the normal events surrounding the locator.

```

PROGRAM
  INCLUDE('ABWINDOW.INC')           !declare WindowManager class
  INCLUDE('ABBROWSE.INC')          !declare BrowseClass and Locator
  MAP
  END

State      FILE,DRIVER('TOPSPEED'),PRE(ST),THREAD
StateCodeKey  KEY(ST:STATECODE),NOCASE,OPT
Record      RECORD,PRE()
STATECODE   STRING(2)
STATENAME   STRING(20)
            END
            END

```

```

StView    VIEW(State)                !declare VIEW to process
        END
StateQ    QUEUE                      !declare Q for LIST
ST:STATECODE LIKE(ST:STATECODE)
ST:STATENAME LIKE(ST:STATENAME)
ViewPosition STRING(512)
        END

Access:State CLASS(FileManager)      !declare Access:State object
Init       PROCEDURE
        END
Relate:State CLASS(RelationManager)  !declare Relate:State object
Init       PROCEDURE
        END
VCRRequest LONG(0),THREAD

StWindow WINDOW('Browse States'),AT(, ,123,152),IMM,SYSTEM,GRAY
        PROMPT('Find:'),AT(9,6)
        STRING(@s2),AT(29,4),USE(ST:STATECODE) !locator control
        LIST,AT(8,5,108,124),USE(?StList),IMM,HVSCROLL,FROM(StateQ),|
        FORMAT('27L(2)|M~CODE~@s2@80L(2)|M~STATENAME~@s20@')
        END

ThisWindow CLASS(WindowManager)      !declare ThisWindow object
Init       PROCEDURE(),BYTE,PROC,VIRTUAL
Kill       PROCEDURE(),BYTE,PROC,VIRTUAL
        END
BrowseSt   CLASS(BrowseClass)        !declare BrowseSt object
Q          &StateQ
        END

StLocator  IncrementalLocatorClass   !declare StLocator object
StStep     StepStringClass           !declare StStep object

CODE
ThisWindow.Run()                     !run the window procedure

ThisWindow.Init PROCEDURE()          !initialize things
ReturnValue     BYTE,AUTO
CODE
ReturnValue = PARENT.Init()          !call base class init
IF ReturnValue THEN RETURN ReturnValue.
Relate:State.Init                     !initialize Relate:State object
SELF.FirstField = ?StList             !set FirstField for ThisWindow
SELF.VCRRequest &= VCRRequest        !VCRRequest not used
Relate:State.Open                     !open State and related files

```

```

!Init BrowseSt object by naming its LIST,VIEW,Q,RelationManager & WindowManager
BrowseSt.Init(?StList,StateQ.ViewPosition,StView,StateQ,Relate:State,SELF)
OPEN(StWindow)
SELF.Opened=True
BrowseSt.Q &= StateQ           !reference the browse QUEUE
StStep.Init(+ScrollSort:AllowAlpha,ScrollBy:Runtime) !initialize the StStep object
BrowseSt.AddSortOrder(StStep,ST:StateCodeKey)!set the browse sort order
BrowseSt.AddLocator(StLocator) !plug in the browse locator
StLocator.Init(?ST:STATECODE,ST:STATECODE,1,BrowseSt) !initialize the locator object
BrowseSt.AddField(ST:STATECODE,BrowseSt.Q.ST:STATECODE)!set a column to browse
BrowseSt.AddField(ST:STATENAME,BrowseSt.Q.ST:STATENAME)!set a column to browse
SELF.SetAlerts()             !alert any keys for ThisWindow
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()      !shut down things
ReturnValue BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()      !call base class shut down
IF ReturnValue THEN RETURN ReturnValue.
Relate:State.Close             !close State and related files
Relate:State.Kill              !shut down Relate:State object
GlobalErrors.Kill              !shut down GlobalErrors object
RETURN ReturnValue

```

## IncrementalLocatorClass Properties

### **IncrementalLocatorClass Properties**

The IncrementalLocatorClass inherits all the properties of the EntryLocatorClass from which it is derived. See *EntryLocatorClass Properties* and *LocatorClass Properties* for more information.

## IncrementalLocatorClass Methods

### IncrementalLocatorClass Methods

The IncrementalLocatorClass inherits all the methods of the EntryLocatorClass from which it is derived. See *EntryLocatorClass Methods* and *LocatorClass Methods* for more information.

### SetAlerts (alert keystrokes for the LIST control:IncrementalLocatorClass)

**SetAlerts**( *control* ), VIRTUAL

---

**SetAlerts** Alerts appropriate keystrokes for the specified LIST control.

*control* An integer constant, variable, EQUATE, or expression that resolves to the control number of the LIST or COMBO control displaying the data to be searched.

The **SetAlerts** method alerts appropriate keystrokes for the specified LIST control.

Implementation: The SetAlerts method alerts the backspace key and the space key.

Example:

```
MyBrowseClass.SetAlerts PROCEDURE                !alert keys for browse object
I BYTE,AUTO
CODE
LOOP I = 1 TO RECORDS( SELF.Sort )              !for each sort order
  GET( SELF.Sort, I )
  IF ~ ( SELF.Sort.Locator &= NULL )           !if locator is present
    SELF.Sort.Locator.SetAlerts( SELF.ListControl )! call Locator.SetAlerts method
  END
END
```

## TakeKey (process an alerted keystroke:IncrementalLocatorClass)

### TakeKey, VIRTUAL

The **TakeKey** method processes an alerted locator keystroke for the LIST control that displays the data to be searched, and returns a value indicating whether the browse display should change.

**Tip:** By default, all alphanumeric keys are alerted for LIST controls.

Implementation: The TakeKey method adds to or subtracts from the search value (the Shadow property) based on the end user's keystrokes, then returns one (1) if a new search is required or returns zero (0) if no new search is required. A search is required only if the keystroke is a valid search character.

Return Data Type: BYTE

Example:

```
CheckLocator ROUTINE
  IF SELF.Sort.Locator.TakeKey()           !handle locator alerted keys
    SELF.Reset(1)                          !if search needed, reset view
    SELF.ResetQueue(Reset:Done)            ! and relead queue
  ELSE                                       !if no search needed
    SELF.ListControl{PROP:Selected}=SELF.CurrentChoice !highlight selected list item
  END
```

See Also: EntryLocatorClass.Shadow



# INIClass

## INIClass Overview

The INIClass provides a simple interface to different methods of non-volatile storage (e.g., storage that is persistent beyond the life cycle of your programs). By default, the INIClass object centrally handles reads and writes for a given configuration (.INI) file. It also supports other methods of non-volatile storage by allowing read and write access to the Windows system registry or a local table. The INIClass Init method controls which access method is used.

## INIClass Concepts

By convention an INI file is an ASCII text file that stores information between computing sessions and contains entries of the form:

```
[SECTION1]
ENTRY1=value
ENTRYn=value
[SECTIONn]
ENTRY1=value
ENTRYn=value
```

The INIClass automatically creates INI files and the sections and entries within them. The INI class also updates and deletes sections and entries. In particular, the INIClass makes it very easy to save and restore Window sizes and positions between sessions; plus it provides a single repository for INI file code, so you only need to specify the INI file name in one place.

## INIClass Relationship to Other Application Builder Classes

The PopupClass and the PrintPreviewClass optionally use the INIClass; otherwise, it is completely independent of other Application Builder Classes.

## INIClass ABC Template Implementation

The ABC Templates generate code to instantiate a global INIClass object called INIMgr. If you request to **Use INI file to save and restore program settings** in the **Global Properties** dialog, then each procedure based on the Window procedure template (Frame, Browse, and Form) calls the INIMgr to save and restore its WINDOW's position and size.

## **INIClass Source Files**

The INIClass source code is installed by default to the Clarion \LIBSRC folder. The INIClass source code and its respective components are contained in:

ABUTIL.INC	INIClass declarations
ABUTIL.CLW	INIClass method definitions

## INIClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate an INIClass object.

```

PROGRAM

    INCLUDE('ABUTIL.INC')           !declare INIClass class
    MAP
    END

INIMgr  INIClass                    !declare INIMgr object
Sound   STRING('ON ')              !user's sound preference
Volume  BYTE(3)                    !user's volume preference

PWindow WINDOW('Preferences'),AT(,,89,34),MAX,RESIZE
        CHECK('&Sound'),AT(8,6),USE(Sound),VALUE('ON','OFF')
        PROMPT('&Volume'),AT(31,19),USE(?VolumePrompt)
        SPIN(@s20),AT(8,20,21,7),USE(Volume),HVSCROLL,RANGE(0,9),STEP(1)
        BUTTON('OK'),AT(57,3,30,10),USE(?OK)
    END

CODE
INIMgr.Init('.\MyApp.INI')          !initialize the INIMgr object
INIMgr.Fetch('Preferences','Sound',Sound) !get sound, default 'ON'
Volume=INIMgr.TryFetch('Preferences','Volume') !get volume, no default
IF Volume
    Sound=INIMgr.FetchField('Preferences','Sound&Vol',1) !get comma delimited sound
    Volume=INIMgr.FetchField('Preferences','Sound&Vol',2) !get comma delimited volume
END
OPEN(PWindow)
INIMgr.Fetch('Preferences',PWindow) !restore window size & pos
ACCEPT
IF EVENT() = EVENT:Accepted
    IF FIELD() = ?OK
        INIMgr.Update('Preferences','Sound',Sound) !store sound
        INIMgr.Update('Preferences','Volume',Volume) !store volume
        INIMgr.Update('Preferences','Sound&Vol',| !store comma delimited values
        CLIP(Sound)('&', '&Volume) !e.g., Sound&Vol=ON,3
        POST(EVENT:CloseWindow)
    END
END
END
INIMgr.Update('Preferences',PWindow) !store window size & pos

```

## INIClass Properties

### INIClass Properties

The INIClass contains the following properties.

#### FileName

**FileName**      **CSTRING(File:MaxFilePath)**

The **FileName** property contains the name of the managed storage medium (INI file or system registry key). The INIClass methods use the FileName property to identify the storage medium.

If a full path is specified, and the INIClass Init Method has specified INI file storage, the INIClass looks for the file in the specified path. If no path is specified, the INIClass looks for the file in the Windows directory. If no name is specified (""), the INIClass uses the WIN.INI file. For example:

<u>FileName Property</u>	<u>Resulting INI File</u>
''	c:\Windows\WIN.INI
'invoice.cfg'	c:\Windows\invoice.cfg
'.\invoice.cfg'	<i>current directory</i> \invoice.cfg
'c:\invoice\invoice.cfg'	c:\invoice\invoice.cfg

If the INIClass Init Method has specified system registry storage, the INIClass looks for the specified registry key.

The Init method sets the contents of the FileName property.

Implementation:    The INIClass methods use the FileName property as the file parameter in GETINI and PUTINI statements. See the *Language Reference* for more information.

See Also:            Init

## INIClass Methods

The INIClass contains the following methods.

### Fetch (get INI file entries)

```
Fetch( section, | entry, value, filename | ),VIRTUAL, PROTECTED
      | entry [, value] |
      | window |
```

---

<b>Fetch</b>	Gets or returns values from the INI file.
<i>section</i>	A string constant, variable, EQUATE, or expression containing the INI file section name.
<i>entry</i>	A string constant, variable, EQUATE, or expression containing the INI file entry name.
<i>value</i>	The label of a variable that contains the default fetched value and receives the actual fetched value. If omitted, there must be a matching <i>section</i> and <i>entry</i> in the INI file for the Fetch method to return.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the INI file name. If <i>filename</i> specifies a full path, the Fetch method looks for the file in the specified path. If no path is specified, the Fetch method looks for the file in the Windows directory. If <i>filename</i> is specified as null (""), the Fetch method uses the WIN.INI file.
<i>window</i>	The label of the WINDOW or APPLICATION to restore to its previously stored position and size. If this parameter is present, Fetch does not return a value, but restores the <i>window's</i> position and size.

The **Fetch** method gets or returns values from the INI file.

**Fetch**(*section,entry[,value]*)

Retrieves a single value specified by *section* and *entry* from the INI file specified in the INIClass.FileName property. If a *value* parameter is present, the Fetch method updates it with the requested value and returns nothing. If no *value* parameter is present the Fetch method returns the requested value.

**Fetch**(*section,entry,value,filename*)

Retrieves a single value specified by *section* and *entry* from the INI file specified by *filename*. The *value* parameter is updated with the requested value. If the *entry* or *filename* does not exist in the INI file, the Fetch method returns an empty string.

**Fetch**(*section>window*)

Retrieves and restores several WINDOW attributes saved by a prior corresponding call to Update(*section>window*) from the INI file specified in the INIClass.FileName property. Restoring the values returns the specified WINDOW to its saved position and size.

Implementation: If a *window* is present, the Fetch method gets five entries from the specified INI file *section*: Maximize, XPos, YPos, Height, and Width. Then it applies the retrieved values to the specified WINDOW or APPLICATION.

Return Data Type: **STRING** where prototype is Fetch(*section, name, value, filename*)  
**STRING** where prototype is Fetch(*section, name*)

Example:

```

Sound  STRING('ON ')
PWindow WINDOW('Preferences'),AT(,,89,34),IMM,MAX,RESIZE
        CHECK('&Sound'),AT(8,6),USE(Sound),VALUE('ON','OFF')
        BUTTON('OK'),AT(57,3,30,10),USE(?OK)
        END
CODE
INIMgr.Fetch('Preferences','Sound',Sound)      !get 'Sound', default ON
Sound=INIMgr.Fetch('Preferences','Sound')      !return 'Sound', no default
Sound=INIMgr.Fetch('Preferences','Sound',Sound,'MYAPP.INI')
!get 'Sound', from MYAPP.INI
OPEN(PWindow)
INIMgr.Fetch('Preferences',PWindow)            !restore PWindow size & position

```

See Also: Update

**FetchField (return comma delimited INI file value)**

**FetchField**( *section*, *entry*, *field*)

---

<b>FetchField</b>	Returns a comma delimited value from the INI file.
<i>section</i>	A string constant, variable, EQUATE, or expression containing the INI file section name.
<i>entry</i>	A string constant, variable, EQUATE, or expression containing the INI file entry name.
<i>field</i>	An integer constant, variable, EQUATE, or expression identifying the comma delimited value to return.

The **FetchField** method returns one of several comma delimited values from the INI file. **FetchField** assumes the value for the *entry* is one of several comma delimited values of the form V1,V2,...,Vn. For example:

```
[MySection]
MyEntry=M,35,Blue,Brown,160
```

A *field* value of one (1) returns the value prior to the first comma in the string; a value of two (2) returns the value between the first and second commas; a three (3) returns the value between the second and third commas, etc.

Return Data Type:   STRING

Example:

```
Sound     STRING('ON ')
Volume    BYTE(3)
          CODE
INIMgr.Update('Preferences','Sound&Volume', |                   !create INI entry like
CLIP(Sound)&'','&Volume)                   !Sound&Volume=ON,3
!program code
Sound=INIMgr.FetchField('Preferences','Sound&Volume',1) !get 1st value - 'ON'
Volume=INIMgr.FetchField('Preferences','Sound&Volume',2) !get 2nd value - 3
```

**FetchQueue (get INI file queue entries)**

**FetchQueue**( *section*, *entry*, *queue*, *field* [,*field*] [,*field*])

---

<b>FetchQueue</b>	Adds a series of values from the INI file to a QUEUE.
<i>section</i>	A string constant, variable, EQUATE, or expression containing the INI file section name.
<i>entry</i>	A string constant, variable, EQUATE, or expression containing the INI file entry name.
<i>queue</i>	The label of the QUEUE to receive the values.
<i>field</i>	The label of the field in the QUEUE to receive the value. You must specify at least one field, and you may specify up to three fields.

The **FetchQueue** method adds a series of values from the INI file into the specified *fields* in the specified *queue*.

Implementation: FetchQueue assumes multiple *entry* values of the form:

```
[section]
entry=ItemsInQueue
entry_n=value,optionalvalue,optionalvalue
```

for example:

```
[Users]
User=3
User_1=Fred,1
User_2=Barney,0
User_3=Wilma,1
```

Example:

```
UserQ  QUEUE
Name   STRING(20)
Auth   BYTE
      END
CODE
INIMgr.FetchQueue('Users','User',UserQ,UserQ.Name,UserQ.Auth)!get UserQ
!program code
INIMgr.Update('Users','User',RECORDS(UserQ))           !put UserQ count
LOOP i# = 1 TO RECORDS(UserQ)                           !put UserQ entries
  GET(UserQ,i#)
  INIMgr.Update('Users','User_'&i#,CLIP(UserQ.Name)&','&UserQ.Auth)
END
```

**Init (initialize the INIClass object)**

**Init** (*FileName*, [*NvType*], [*ExtraData*])

---

<b>Init</b>	Initializes the INIClass object.
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the INI file name. If <i>filename</i> specifies a full path, the INIClass looks for the file in the specified path. If no path is specified, the INIClass looks for the file in the Windows directory. If <i>filename</i> is specified as null ("), the INIClass uses the WIN.INI file.
<i>NvType</i>	An UNSIGNED integer that conforms to the following equates:  NVD_INI        use named INI file. NVD_Registry use the system registry (ignores the filename parameter) NVD_Table     use a local table – NOT SUPPORTED YET
<i>ExtraData</i>	A LONG integer. When the registry is used in the second parameter, <i>ExtraData</i> is used to specify the part of the registry to use as the root of all entries. Valid values are:  REG_CLASSES_ROOT REG_CURRENT_USER REG_LOCAL_MACHINE REG_USERS REG_PERFORMANCE_DATA REG_CURRENT_CONFIG REG_DYN_DATA

The **Init** method initializes the INIClass object.

Implementation:    The Init method assigns *filename* to the FileName property.

Example:

```
    INCLUDE('ABUTIL.INC')
INIMgr      INIClass
CODE
INIMgr.Init('c:\MyApp\MyApp.INI')    !read & write from c:\MyApp\MyApp.INI
INIMgr.Init('.\MyApp.INI')          !read & write from currentdirectory\MyApp.INI
INIMgr.Init('')                     !read & write from c:\Windows\WIN.INI
INIMgr.Init('MyApp.INI')            !read & write from c:\Windows\MyApp.INI

INIMgr.Init('', NVD_Registry, REG_LOCAL_MACHINE)
!read & write from Windows system registry, using LOCAL_MACHINE as the root

INIMgr.Init('CONFIG.TPS',NVD_Table)
!read & write from a local table
```

See Also:        FileName

**TryFetch (get a value from the INI file)**

**TryFetch**( *section*, *entry* )

---

<b>TryFetch</b>	Returns a value from the INI file.
<i>section</i>	A string constant, variable, EQUATE, or expression containing the INI file section name.
<i>entry</i>	A string constant, variable, EQUATE, or expression containing the INI file entry name.

The **TryFetch** method returns a value from the INI file. If the specified section and entry do not exist, TryFetch returns an empty string. This allows you to check the return value and take appropriate action when the INI file entry is missing.

Return Data Type:   STRING

Example:

```
Color        BYTE
DefaultColor EQUATE(5)
CODE
Color=INIMgr.TryFetch('Preferences','Color')     !return 'Color', no default
IF NOT Color
  Color=DefaultColor
END
```

**TryFetchField (return comma delimited INI file value)**

**TryFetchField**( *section, entry, field* )

---

**TryFetchField** Returns a comma delimited value from the INI file.

*section*        A string constant, variable, EQUATE, or expression containing the INI file section name.

*entry*            A string constant, variable, EQUATE, or expression containing the INI file entry name.

*field*            An integer constant, variable, EQUATE, or expression identifying the comma delimited value to return.

The **TryFetchField** method returns one of several comma delimited values from the INI file. If the specified section and entry do not exist, TryFetchField returns an empty string. This allows you to check the return value and take appropriate action when the INI file entry is missing.

TryFetchField assumes the *entry* value is a comma delimited string of the form V1,V2,...,Vn. A *field* value of one (1) returns the value prior to the first comma in the string; a value of two (2) returns the value between the first and second commas; a three (3) returns the value between the second and third commas, etc.

Return Data Type:    **STRING**

Example:

```

Sound    STRING(3)
Volume    BYTE
          CODE
          Sound=INIMgr.TryFetchField('Preferences','Sound&Volume',1)
          !get Sound value
          IF NOT Sound                   !if not present
          Sound='ON'                   !default to on
          END
          Volume=INIMgr.TryFetchField('Preferences','Sound&Volume',2)
!get Volume value
          IF NOT Volume                 !if not present
          Volume=3                     !default to 3
          END
!program code
INIMgr.Update('Preferences','Sound&Volume', | !create INI entry like
CLIP(Sound)&', '&Volume)                   !Sound&Volume=ON,3

```

**Update (write INI file entries)**

```

Update( section, | entry, value
        | entry, value, filename
        | window
        | entry, queue, field, [field], [field]
        ), VIRTUAL

```

---

<b>Update</b>	Writes entries to the INI file.
<i>section</i>	A string constant, variable, EQUATE, or expression containing the INI file section name.
<i>entry</i>	A string constant, variable, EQUATE, or expression containing the INI file entry name.
<i>value</i>	A constant, variable, EQUATE, or expression containing the value to store for the <i>section</i> and <i>entry</i> .
<i>filename</i>	A string constant, variable, EQUATE, or expression containing the INI file name. If <i>filename</i> specifies a full path, the Update method looks for the file in the specified path. If no path is specified, the Update method looks for the file in the Windows directory. If <i>filename</i> is specified as null (""), the Fetch method uses the WIN.INI file.
<i>window</i>	The label of a WINDOW or APPLICATION whose position and size parameters the Update method stores.
<i>queue</i>	The label of a QUEUE.
<i>field</i>	The label of a FIELD within the QUEUE.

The **Update** method writes entries to the INI file. If the specified value is null ("), the existing entry is deleted.

**Update**(*section,entry,value*)

Writes a single value specified by *section* and *entry* to the INI file specified in the INIClass.FileName property.

**Update**(*section,entry,value,filename*)

Writes a single value specified by *section* and *entry* to the INI file specified by *filename*.

**Update**(*section>window*)

Writes several WINDOW position and size attributes to the INI file specified in the INIClass.FileName property, for retrieval by a subsequent corresponding call to Fetch(*section>window*). Restoring the values returns the specified WINDOW to its saved position and size.

**Update**(*queue,field,[field],[field]*)

Writes the records of a QUEUE with a maximum of three fields to the INI file specified in the INIClass.FileName property.

Implementation: If a *window* is present, the Update method writes five entries to the specified INI file *section*: Maximize, XPos, YPos, Height, and Width. These entries are retrieved and applied by the Fetch method to restore the window's position and size.

Example:

```
Sound    STRING('ON ')
PWindow  WINDOW('Preferences'),AT(,,89,34),IMM,MAX,RESIZE
         CHECK('&Sound'),AT(8,6),USE(Sound),VALUE('ON','OFF')
         BUTTON('OK'),AT(57,3,30,10),USE(?OK)
         END
CODE
OPEN(PWindow)
INIMgr.Fetch('Preferences',PWindow)           !restore PWindow size & position
INIMgr.Fetch('Preferences','Sound',Sound)     !get 'Sound' entry
!program code
INIMgr.Update('Preferences','Sound',Sound)    !save 'Sound' entry
INIMgr.Update('Preferences','Sound',Sound,'MYAPP.INI') !save 'Sound' entry to INI
INIMgr.Update('Preferences',PWindow)         !save PWindow size & position
```

See Also:        Fetch

# IReportGenerator Interface

## IReportGenerator Interface

### IReportGenerator Concepts

The IReportGenerator interface is a defined set of commands that are implemented by various report output generators. This interface is not intended to be called by any code other than the WMFParse.

### IReportGenerator Methods

#### **AskProperties (pop up window to set properties)**

**AskProperties( ), BYTE**

---

**AskProperties** Allow the user to set properties before printing.

The **AskProperties** method is used to pop up a window that allows a user to set properties defining how the output is to be generated. The **AskProperties** method returns one (1) if the OK button is pressed and printing will proceed. If the Cancel button is pressed, the **AskProperties** method returns zero (0) and printing will not proceed.

Return Data Type: BYTE

**CloseDocument (end document printing)****CloseDocument( ), BYTE**

---

**CloseDocument**      Called after the document is printed, and returns an appropriate error level.

The **CloseDocument** method is used to detect any error level state that may exist after the document has been printed. **CloseDocument** returns Level:Benign if the document printed normally.

Implementation:      Called immediately after the document has been printed.

Return Data Type:    BYTE

**ClosePage (end a page print)****ClosePage( ), BYTE**

---

**ClosePage**      Called after each page is printed to detect a possible error condition.

The **ClosePage** method is used to detect any error state that may exist after each page (WMF file) has been printed. **ClosePage** returns Level:Benign if the page printed successfully.

Implementation:      Called immediately after each WMF page has been printed.

Return Data Type:    BYTE

**GetProperty (get a property value)****GetProperty ( *property* ), STRING**

---

**GetProperty** Returns the value of a specified property.

*property* A string constant, variable, EQUATE, or expression containing a valid document property name.

The **GetProperty** method returns a value of the current document property. These properties are defined in the documentation for each ReportGenerator object (i.e., the HTML report generator contains a list of supported properties).

Return Data Type: STRING

**Init (initialize error class before printing)****Init(ErrorClass EC)**

---

**Init** Initializes the ErrorClass object used to detect document errors..

**EC** The name of the ErrorClass object used to report any errors encountered during report generation.

The **Init** method is used to initialize the ErrorClass object that is used with the ReportGenerator Class.

### **OpenDocument (begin document printing)**

**OpenDocument( ), BYTE**

---

**OpenDocument** Called before the document is printed, and posts any error level code that may have occurred during initialization.

The **OpenDocument** method is used to detect any error state that may exist before the document has been printed. If the document is ready to print, **OpenDocument** returns a Level:Benign

Implementation: Called immediately before the document begins to print.

Return Data Type: BYTE

### **OpenPage (begin a page print)**

**OpenPage( ), BYTE**

---

**OpenPage** Called before each page is printed to detect if an error has occurred.

The **OpenPage** method is used to detect an error level state that may exist before each page (WMF file) has been printed. If the page is ready to be printed, **OpenPage** returns Level:Benign.

Implementation: Called immediately before each WMF page is printed.

Return Data Type: BYTE

### **Opened (file opened flag)**

**Opened** BYTE

The **Opened** property indicates whether the DbLogFileManager's FILE (the log file) has been opened. A value of one (1 or True) indicates the FILE is open; a value of zero (0 or False) indicates the FILE is not opened.

## **ProcessArc (print an arc)**

**ProcessArc**( \*ArcFormatGrp *arc*, *comment* )

---

**ProcessArc** Prints an ARC of an ellipse to a target output document.

*arc* A TYPEd GROUP structure that holds all of the properties of the target arc graphic.

*comment* A string constant, variable, EQUATE, or expression containing information necessary for the arc of an ellipse to be properly rendered to the target document.

The **ProcessArc** method prints an arc of an ellipse to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *arc* group contains the position and style of the arc from the contents of the passed *ArcFormatGrp*.

See Also: ARC

**ProcessBand (begin/end report band processing)**

**ProcessBand** ( *bandtype, position* )

---

**ProcessBand** Processes each report band

*bandtype* A string constant, variable, EQUATE, or expression containing the type of band to process. Valid band types are HEADER, FOOTER, DETAIL, BREAK and FORM.

*position* A BYTE, variable, EQUATE, or expression that identifies the start or end of the band.

The **ProcessBand** method is used to process all report bands and redirect to an alternative document where appropriate. If the position attribute is TRUE (1), the start of each band is processed. If the position attribute is FALSE (0), the start of each band is processed.

**ProcessCheck (print a checkbox)**

**ProcessCheck**( \*CheckFormatGrp *check, text, comment* )

---

**ProcessCheck** Prints a CHECK control to the output document.

*check* A TYPEd GROUP that holds all of the properties of the target check box.

*text* A string constant, variable, EQUATE, or expression containing the check box prompt contents.

*comment* A string constant, variable, EQUATE, or expression containing information necessary for the check box to be properly rendered to the target document.

The **ProcessCheck** method prints a check box to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *check* group contains the position, style, text, and check box state from the contents of the passed CheckFormatGrp.

## **ProcessChord (print a section of an ellipse)**

**ProcessChord**( \*ChordFormatGrp *chord*, *comment* )

---

**ProcessChord** Prints a section of an ellipse to a target output document.

*chord* A TYPEd GROUP structure that holds all of the properties of the target chord graphic.

*comment* A string constant, variable, EQUATE, or expression containing information necessary for the section of an ellipse to be properly rendered to the target document.

The **ProcessChord** method prints a section of an ellipse to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *chord* group contains the position and style of the chord from the contents of the passed *ChordFormatGrp*.

See Also: CHORD

**ProcessEllipse (print an ellipse)**

**ProcessEllipse**( \*EllipseFormatGrp *ellipse*, *comment* )

---

<b>ProcessEllipse</b>	Prints an ELLIPSE control to a target output document.
<i>ellipse</i>	A TYPEd GROUP structure that holds all of the properties of the target ellipse control.
<i>comment</i>	A string constant, variable, EQUATE, or expression containing information necessary for the ellipse to be properly rendered to the target document.

The **ProcessEllipse** method prints an ellipse control to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *ellipse* group contains the position and style of the ellipse control from the contents of the passed *EllipseFormatGrp*.

**ProcessImage (print an image)**

**ProcessImage** ( \*ImageFormatGrp *image*, *iName*, *comment* )

---

<b>ProcessImage</b>	Prints an IMAGE control to a target output document.
<i>image</i>	A TYPEd GROUP structure that holds all of the properties of the target image control.
<i>iname</i>	A string constant, variable, EQUATE, or expression containing the file name containing the image to be printed.
<i>comment</i>	A string constant, variable, EQUATE, or expression containing information necessary for the image to be properly rendered to the target document.

The **ProcessImage** method prints an image control to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *image* group contains the position and stretch mode (centered, tiled, stretched) of the image control from the contents of the passed *ImageFormatGrp*.

### **ProcessLine (print a line)**

**ProcessLine** (\*LineFormatGrp *line*, *comment* )

---

**ProcessLine** Prints a LINE control to a target output document.

*line* A TYPED GROUP structure that holds all of the properties of the target LINE control.

*comment* A string constant, variable, EQUATE, or expression containing information necessary for the LINE to be properly rendered to the target document.

The **ProcessLine** method prints a LINE control to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *name* parameter accepts up to 1024 characters.

### **ProcessOption (print an option control)**

**ProcessOption** (\*OptionFormatGrp *option*, *text*, *comment* )

---

**ProcessOption** Prints an OPTION control to a target output document.

*option* A TYPED GROUP structure that holds all of the properties of the target OPTION control.

*text* A string constant, variable, EQUATE, or expression containing the OPTION text contents.

*comment* A string constant, variable, EQUATE, or expression containing information necessary for the OPTION to be properly rendered to the target document.

The **ProcessOption** method prints an option control to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *option* group contains the position and other attributes of the option control from the contents of the passed *OptionFormatGrp*.

**ProcessRadio (print a radio button)**

**ProcessRadio** (\*RadioFormatGrp *radio*, *text*, *comment* )

---

**ProcessRadio** Prints a RADIO control to a target output document.

<i>radio</i>	A TYPEd GROUP structure that holds all of the properties of the target RADIO control.
<i>text</i>	A string constant, variable, EQUATE, or expression containing the RADIO text contents.
<i>comment</i>	A string constant, variable, EQUATE, or expression containing information necessary for the RADIO to be properly rendered to the target document.

The **ProcessRadio** method prints a RADIO control to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *radio* group contains the contents, position (outer, inner) and radio state of the radio control from the contents of the passed *RadioFormatGrp*.

**ProcessRectangle (print a box control)**

**ProcessRectangle** ( \*RectFormatGrp *rect*, *comment* )

---

**ProcessRectangle** Prints a BOX control to a target output document.

<i>rect</i>	A TYPEd GROUP structure that holds all of the properties of the target BOX control.
<i>comment</i>	A string constant, variable, EQUATE, or expression containing information necessary for the BOX to be properly rendered to the target document.

The **ProcessRectangle** method prints a BOX control to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *rect* group contains the position, style, and other attributes of the BOX control from the contents of the passed *RectFormatGrp*.

**ProcessText (print a string or text control)**

**ProcessText** (\*TextFormatGrp *txtgrp*, *text*, *comment* )

---

<b>ProcessText</b>	Prints a TEXT or STRING control to a target output document.
<i>txtgrp</i>	A TYPEd GROUP structure that holds all of the properties of the target TEXT or STRING control.
<i>text</i>	A string constant, variable, EQUATE, or expression containing the TEXT or STRING contents.
<i>comment</i>	A string constant, variable, EQUATE, or expression containing information necessary for the TEXT or STRING control to be properly rendered to the target document.

The **ProcessText** method prints a TEXT or STRING control to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: The *txtgrp* group contains the position, alignment, styles, and other attributes of a TEXT or STRING control from the contents of the passed *TextFormatGrp*.

### **SetProperty (set a property value)**

**SetProperty ( *property* , *value* ), STRING**

---

**SetProperty** Sets the value of a specified property.

*property* A string constant, variable, EQUATE, or expression containing a valid document property name.

*value* A string constant, variable, EQUATE, or expression containing a value for the specified property name.

The  **SetProperty** method sets a value of the named document *property*. These properties are defined in the documentation for each ReportGenerator object (i.e., the HTML report generator contains a list of supported properties).

### **WhoAml (identify the report generator type)**

**WhoAml ( ), STRING**

---

**WhoAml** Identifies the type of report generator.

The  **WhoAml** method returns a string used to identify the type of report generator. For example, the HTML report generator returns 'HTML'.

Implementation: Returns the value of the *IAm* property initialized in the ReportGenerator object's constructor.

Return Data Type: STRING

# LocatorClass

## LocatorClass Overview

The LocatorClass is an abstract class--it is not useful by itself. However, other useful classes are derived from it and other structures (such as the BrowseClass) use it to reference its derived classes.

## LocatorClass Concepts

The classes derived from LocatorClass let you specify a locator control and a sort field on which to search for each sort order of a BrowseClass object. These LocatorClass objects help the BrowseClass locate and scroll to the requested items.

LocatorClass objects implement some of the common variations in locator controls (none, STRING, ENTRY), locator invocation (keystroke, ENTER key, TAB key), and search methods (single character search starting from current item, incremental character, exclusive search) that occur in the browse context.

## LocatorClass Relationship to Other Application Builder Classes

The BrowseClass optionally uses the classes derived from the LocatorClass. Therefore, if your BrowseClass objects use a locator, then your program must instantiate a LocatorClass for each use.

The StepLocatorClass, EntryLocatorClass, IncrementalLocatorClass, and FilterLocatorClass are all derived (directly or indirectly) from the LocatorClass. Each of these derived classes provides slightly different search behaviors and characteristics.

### Step Locator

Use a Step Locator when the search field is a STRING, CSTRING, or PSTRING, a single character search is sufficient (a step locator is not appropriate when there are many key values that begin with the same character), and you want the search to take place immediately upon the end user's keystroke. Step Locators are not appropriate for numeric keys.

### Entry Locator

Use an Entry Locator when you want a multi-character search (more precise) on numeric or alphanumeric keys and you want to delay the search until the user accepts the locator control. The delayed search reduces network traffic and provides a smoother search in a client-server environment.

**Incremental Locator**

Use an Incremental locator when you want a multi-character search (more precise) on numeric or alphanumeric keys and you want the search to take place immediately upon the end user's keystroke.

**Filter Locator**

Use a Filter Locator when you want a multi-character search (more precise) on alphanumeric keys and you want to *minimize network traffic*.

**LocatorClass ABC Template Implementation**

Because the LocatorClass is abstract, the ABC Template generated code does not directly reference the LocatorClass.

**LocatorClass Source Files**

The LocatorClass source code is installed by default to the Clarion \LIBSRC folder. The LocatorClass source code and its respective components are contained in:

ABBROWSE.INC	LocatorClass declarations
ABBROWSE.CLW	LocatorClass method definitions

## LocatorClass Properties

The LocatorClass has the several properties described below. These properties are inherited by classes derived from the LocatorClass.

### **Control (the locator control number)**

#### **Control SIGNED**

The **Control** property contains the locator control number if there is a locator control. If there is no locator control, it contains zero (0). The LocatorClass uses the Control property to refresh the control or change its properties.

The Init method sets the value of the Control property.

See Also:           Init

### **FreeElement (the locator's first free key element)**

#### **FreeElement    ANY**

The **FreeElement** property contains a reference to a component of the sort sequence of the searched data set. The ABC Templates further require this to be a free component of a key. A free component is one that is not range limited to a single value. Typically this is also the USE variable of the locator control. The LocatorClass uses the FreeElement property to prime the free component with the appropriate search value.

The Init method sets the value of the FreeElement property.

See Also:           Init

## **NoCase (case sensitivity flag)**

### **NoCase BYTE**

The **NoCase** property determines whether the LocatorClass object performs case sensitive searches or case insensitive searches.

The Init method sets the value of the NoCase property.

Implementation: If NoCase contains a non-zero value, the search is not case sensitive. That is, searches for "Tx," "tx," or "TX" all produce the same result. If NoCase contains a value of zero (0), the search is case sensitive.

See Also: Init

## **ViewManager (the locator's ViewManager object)**

### **ViewManager &BrowseClass**

The **ViewManager** property is a reference to the BrowseClass object that the LocatorClass object is working for. See *ViewManager* and *BrowseClass* for more information. The LocatorClass uses this property to manipulate the searched data set as well as the displayed LIST.

The Init method sets the value of the ViewManager property.

See Also: Init

## LocatorClass Methods

The LocatorClass contains the following methods.

### **GetShadow(return shadow value)**

GetShadow, VIRTUAL

---

GetShadow Is a virtual placeholder method for the EntryLocatorClass.SetShadow method.

The **GetShadow** method returns the value of the Shadow property found in the child CLASS. The Shadow property is set based on the users keyboard input into the entry locator field.

Implementation: The GetShadow method is a placeholder method for the EntryLocatorClass which is derived from LocatorClass.

Return Data Type: **STRING**

See Also: LocatorClass.SetShadow, EntryLocatorClass.Shadow

**Init (initialize the LocatorClass object)**

**Init**( [*control*] , *freeelement*, *nocase* [,*browseclass*] )

---

<b>Init</b>	Initializes the LocatorClass object.
<i>control</i>	An integer constant, variable, EQUATE, or expression that sets the locator control number for the LocatorClass object. If omitted, the control number defaults to zero (0) indicating there is no locator control.
<i>freeelement</i>	The fully qualified label of a component of the sort sequence of the searched data set. The ABC Templates further require this to be a free component of a key. A free component is one that is not range limited to a single value. Typically this is also the USE variable of the locator control.
<i>nocase</i>	An integer constant, variable, EQUATE, or expression that determines whether the LocatorClass object performs case sensitive searches or case insensitive searches.
<i>browseclass</i>	The label of the BrowseClass object for the locator. If omitted, the LocatorClass object has no direct access to the browse QUEUE or it's underlying VIEW.

The **Init** method initializes the LocatorClass object.

Implementation: The Init method sets the values of the Control, FreeElement, NoCase, and ViewManager properties.

A *nocase* value of zero (0 or False) produces case sensitive searches; a value of one (1 or True) produces case insensitive searches.

By default, only the StepLocatorClass and FilterLocatorClass use the *browseclass*. The other locator classes do not.

Example:

```
BRW1::Sort1:Locator.Init(,CUST:StateCode,1)           !without locator control
BRW1::Sort2:Locator.Init(?CUST:CustMo,CUST:CustNo,1)!with locator control
```

See Also: Control, FreeElement, NoCase, ViewManager

## Reset (reset the locator for next search)

### Reset, VIRTUAL

The **Reset** method is a virtual placeholder method to reset the locator for the next search.

Implementation: The BrowseClass.TakeAcceptedLocator method calls the Reset method.

Example:

```
BrowseClass.TakeAcceptedLocator PROCEDURE      !process an accepted locator entry
CODE
IF ~SELF.Sort.Locator &= NULL AND ACCEPTED() = SELF.Sort.Locator.Control
  IF SELF.Sort.Locator.TakeAccepted()          !call locator take accepted method
    SELF.Reset(1)                               !if search needed, reset the view
    SELECT(SELF.ListControl)                    !focus on the browse list control
    SELF.ResetQueue( Reset:Done )              !reload the browse queue
    SELF.Sort.Locator.Reset                    !reset the locator
    SELF.UpdateWindow                           !update (redraw) the window
  END
END
```

See Also: BrowseClass.TakeAcceptedLocator

**Set (restart the locator:LocatorClass)****Set, VIRTUAL**

The **Set** method prepares the locator for a new search.

Implementation: The Set method clears the FreeElement property.

Example:

```
MyBrowseClass.TakeScroll PROCEDURE( SIGNED Event )
CODE
CASE Event
OF Event:ScrollUp OROF Event:ScrollDown
  SELF.ScrollOne( Event )
OF Event:PageUp OROF Event:PageDown
  SELF.ScrollPage( Event )
OF Event:ScrollTop OROF Event:ScrollBottom
  SELF.ScrollEnd( Event )
END                                     !after a scroll event
IF ~SELF.Sort.Locator &= NULL THEN    !if locator is present
  SELF.Sort.Locator.Set               !set it to blank
END
```

**SetAlerts (alert keystrokes for the LIST control:LocatorClass)****SetAlerts( control ), VIRTUAL**


---

**SetAlerts** Alerts appropriate keystrokes for the specified control.

*control* An integer constant, variable, EQUATE, or expression containing the control number of the control displaying the data to search.

The **SetAlerts** method alerts appropriate keystrokes for the specified control, typically a LIST or COMBO.

The SetAlerts method is a placeholder method for classes derived from LocatorClass--IncrementalLocatorClass, etc.

See Also: IncrementalLocatorClass.SetAlerts

## SetEnabled (enable or disable the locator control)

**SetEnabled**( *enabled* )

---

**SetEnabled** Enables or disables the locator control.

*enabled* An integer constant, variable, EQUATE, or expression that enables or disables the locator control. A value of zero (0 or False) disables the control; a value of one (1 or True) enables the control.

The **SetEnabled** method enables or disables the locator control for this LocatorClass object. See *ENABLE* and *DISABLE* in the *Language Reference*.

Example:

```
MyBrowseClass.Enable PROCEDURE
CODE
IF ~SELF.Sort.Locator &= NULL                !if locator is present
    SELF.Sort.Locator.SetEnabled(RECORDS(SELF.ListQueue)) !disable locator if 0 items
END
```

## SetShadow (update shadow value)

**SetShadow**( *value* ),VIRTUAL

---

**SetShadow** Is a virtual placeholder method for the EntryLocatorClass.SetShadow method.

*value* A string constant, variable, EQUATE, or expression that is assigned to the Shadow property.

The **SetShadow** method places a value in the EntryLocatorClass.Shadow property.

See Also: EntryLocatorClass.SetShadow, EntryLocatorClass.Shadow

## **TakeAccepted (process an accepted locator value:LocatorClass)**

### **TakeAccepted, VIRTUAL**

The **TakeAccepted** method processes the accepted locator value and returns a value indicating whether the browse list display must change. The TakeAccepted method is only a placeholder method for classes derived from LocatorClass--EntryLocatorClass, FilterLocatorClass, etc.

This method is only appropriate for LocatorClass objects with locator controls that accept user input; for example, entry controls, combo controls, or spin controls.

A locator value is accepted when the end user changes the locator value, then TABS off the locator control or otherwise switches focus to another control on the same window.

Return Data Type: BYTE

See Also: EntryLocatorClass.TakeAccepted, FilterLocatorClass.TakeAccepted

## **TakeKey (process an alerted keystroke:LocatorClass)**

### **TakeKey, VIRTUAL**

The **TakeKey** method processes an alerted keystroke for the LIST control and returns a value indicating whether the browse list display must change.

**Tip:** By default, all alphanumeric keys are alerted for LIST controls.

The TakeKey method is only a placeholder method for classes derived from LocatorClass--StepLocatorClass, EntryLocatorClass, IncrementalLocatorClass, etc.

Return Data Type: BYTE

See Also: StepLocatorClass.TakeKey, EntryLocatorClass.TakeKey, IncrementalLocatorClass.TakeKey

**UpdateWindow (redraw the locator control with its current value)****UpdateWindow, VIRTUAL**

The **UpdateWindow** method redraws the locator control with its current value.

The UpdateWindow method is only a placeholder method for classes derived from LocatorClass--EntryLocatorClass, FilterLocatorClass, etc.

See Also:       EntryLocatorClass.UpdateWindow, FilterLocatorClass.UpdateWindow



---

# MsgBoxClass

## MsgBoxClass Overview

The MsgBoxClass provides a message window to the ErrorClass. This class manages the display of the current error.

## MsgBoxClass Source Files

The MsgBoxClass source code is installed by default to the Clarion \LIBSRC. The specific MsgBoxClass source code and their respective components are contained in:

ABERROR.INC	MsgBoxClass declarations
ABERROR.CLW	MsgBoxClass method definitions
ABERROR.TRN	MsgBoxClass default error definitions

## MsgBoxClass Properties

The The MsgBoxClass inherits all the properties of the WindowManager class from which it is derived. In addition to the inherited properties, the MsgBoxClass contains the following properties:

### ButtonTypes (standard windows buttons)

**ButtonTypes**                      **LONG, PROTECTED**

The **ButtonTypes** property is used to indicate which Windows standard buttons to place on the message box dialog. This may indicate multiple buttons.

### Caption (window title)

**Caption**                              **&STRING, PROTECTED**

The **Caption** property is a string that specifies the message box window caption.

### Err (errorclass object)

**Err**                                      **&ErrorClass, PROTECTED**

The **Err** property is a reference to the ErrorClass object.

### Icon (icon for image control)

**Icon**                                      **LONG, PROTECTED**

The **Icon** property is a long value that specifies the icon to use on the message box window.

### HistoryHandler (windowcomponent interface)

**HistoryHandler**            **&WindowComponent**

The **HistoryHandler** property is a reference to the WindowComponent interface.

### MsgRVal (message box return value)

**MsgRVal**                    **LONG**

The **MsgRVal** property is a long value that is the return value from the message box. The MsgBox.Init method sets this property to the Default window button.

### Style (font style)

**Style**                        **LONG, PROTECTED**

The **Style** property indicates the font style to use withing the list control on the message box dialog.

### Win (reference to window)

**Win**                          **&Window, PROTECTED**

The **Win** property is a reference to the message box window.

## MsgBoxClass Methods

The MsgBoxClass inherits all the methods of the WindowManager from which it is derived. In addition to the inherited methods, the MsgBoxClass contains the methods listed below.

### FetchFeq (retrieve button feq)

**FetchFeq( *btn* ), VIRTUAL**

<b>FetchFeq</b>	Retrieve button FEQ.
<i>btn</i>	An integer constant, variable, EQUATE, or expression that indicates the button number.

The **FetchFeq** method determines the FEQ for the specified button number.

Return Data Value:     SHORT

### FetchStdButton (determine button pressed)

**FetchStdButton(feq)**

The **FetchStdButton** determines which of the windows standard buttons was pressed. The FEQ of the button is returned.

Return Data Value:     BYTE

## Init (initialize the MsgBoxClass object)

**Init, PROC, DERIVED**

**Init**(*win, err, [caption], icon, [buttons], default button, style*)

<b>Init</b>	Initialize the MsgBoxClass object.
<i>win</i>	Reference to the MsgBox window.
<i>err</i>	Reference to the ErrorClass object.
<i>caption</i>	A string constant, variable, EQUATE, or expression that specifies the message box window caption.
<i>icon</i>	An integer constant, variable, EQUATE, or expression that indicates the icon to display on the message box.
<i>buttons</i>	An integer constant, variable, EQUATE, or expression that indicates which Windows standard buttons to place on the message box. This may indicate multiple buttons. If omitted this is equivalent to Button:OK.
<i>default button</i>	An integer constant, variable, EQUATE, or expression that indicates the default button on the message box.
<i>style</i>	An integer constant, variable, EQUATE, or expression that indicates the font style to use withing the list control on the message box dialog.

The **Init** method initializes the MsgBoClass object.

Return Data Type:     BYTE

## Kill (perform any necessary termination code)

**Kill**

The **Kill** method disposes any memory allocated during the object's lifetime and performs any other necessary termination code.

## **SetupAdditionalFeqs (initialize additional control properties)**

**SetupAdditionalFeqs, VIRTUAL, PROTECTED**

The **SetupAdditionalFeqs** method initializes additional properties for the controls on the window.

## **TakeAccepted (process accepted event)**

**TakeAccepted, PROC, DERIVED**

The **TakeAccepted** method processes EVENT:Accepted events for the message box dialog's controls, and returns a value indicating whether ACCEPT loop processing is complete and should stop. TakeAccepted returns Level:Fatal to indicate a standard button on the dialog was pressed and ACCEPT loop should BREAK.

Return Data Type:      **BYTE**

---

# PopupClass

## PopupClass Overview

The PopupClass object defines and manages a full featured popup (context) menu. The PopupClass object makes it easy to add fully functional popup menus to your procedures.

## PopupClass Concepts

You can set the popup menu items to mimic existing buttons on a window, so that associated menu item text matches the *button* text, is enabled only when the *button* is enabled, and, when selected, invokes the *button* action.

Alternatively, you can set the popup menu item to POST a particular event or simply return its ID so you can trap it and custom code the item's functionality.

## PopupClass Relationship to Other Application Builder Classes

The PopupClass optionally uses the TranslatorClass so you can translate menu text to other languages without changing your popup menu code. The PopupClass optionally uses the INIClass to save and restore menu definitions to a configuration (.INI) file. Neither class is required by the PopupClass; however, if you use either facility, you must instantiate them in your program. See the Conceptual Example.

The ASCIIViewerClass, BrowseClass, and PrintPreviewClass all use the PopupClass to manage their popup menus. This PopupClass use is automatic when you INCLUDE the class header (ABASCII.INC, ABBROWSE.INC, or ABPRINT.INC) in your program's data section.

## PopupClass ABC Template Implementation

The ABC Templates declare a local PopupClass class *and* object for each instance of the Popup code template.

The class is named PopupMgr# where # is the instance number of the Popup code template. The templates provide the derived class so you can use the Popup code template **Classes** tab to easily modify the popup menu behavior on an instance-by-instance basis.

The template generated code does not reference the PopupClass objects encapsulated within the ASCIIViewerClass, BrowseClass, and PrintPreviewClass.

## PopupClass Source Files

The PopupClass source code is installed by default to the Clarion \LIBSRC folder. The PopupClass source code and its respective components are contained in:

ABPOPOP.INC	PopupClass declarations
ABPOPOP.CLW	PopupClass method definitions
ABPOPOP.TRN	PopupClass translation strings

## PopupClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a PopupClass object.

This example displays a dialog with a right-click popup menu that mimics the dialog buttons with three different PopupClass techniques. The dialog buttons demonstrate the PopupClass' ability to save and restore menus to and from an INI file.

```

PROGRAM
MAP
END
INCLUDE('ABPOPUP.INC')           !declare PopupClass
INCLUDE('ABUTIL.INC')           !declare INIClass & Translator
INCLUDE('KEYCODES.CLW')        !declare right-click EQUATE

PopupString  STRING(20)         !to receive menu selection
PopupMgr     PopupClass         !declare PopupMgr object
Translator   TranslatorClass    !declare Translator object
INIMgr       INIClass           !declare INIMgr object
INIFile      EQUATE('.\Popup.ini') !declare INI pathname EQUATE

PopupWin WINDOW('Popup Demo'),AT(,,184,50),ALRT(MouseRight),GRAY
        BUTTON('&Save Popup'),AT(17,16),USE(?Save)
        BUTTON('&Restore Popup'),AT(74,16),USE(?Restore),DISABLE
        BUTTON('Close'),AT(140,16),USE(?Close)
        END

CODE
OPEN(PopupWin)
    Translator.Init              !initialize Translator object
    INIMgr.Init(INIFile)        !initialize INIMgr object
    PopupMgr.Init(INIMgr)       !initialize PopupMgr object
    PopupMgr.AddItemMimic('Save',?Save) !Save item mimics ?Save button
    PopupMgr.AddItem('Restore Popup','Restore') !add menu item: Restore
    PopupMgr.SetItemEnable('Restore',False) !initially disable Restore item
    PopupMgr.AddItem('-', 'Separator1') !add a menu item separator
    PopupMgr.AddItem('Disable Save','Disable') !add a menu item: Disable
    PopupMgr.AddItem('-', 'Separator2') !add a menu item separator
    PopupMgr.AddItem('Close (EVENT:Accepted)','Close')!add a menu item: Close
    PopupMgr.AddItemEvent('Close',EVENT:Accepted,?Close)!Close POSTs event to a control
    PopupMgr.AddItem('Close (EVENT:CloseWindow)','Close2')!add a menu item: Close2
    PopupMgr.AddItemEvent('Close2',EVENT:CloseWindow,0) !Close2 POSTs independent event
    PopupMgr.SetTranslator(Translator) !enable popup text translation

ACCEPT
CASE EVENT()
OF EVENT:AlertKey
    IF KEYCODE() = MouseRight !trap for alerted keys
        !if right-click
    
```

```

PopupString=PopupMgr.Ask()           !display popup menu
CASE PopupString                     !check for selected item
OF 'Disable'                          !if Disable item selected
  IF PopupMgr.GetItemChecked('Disable')
    PopupMgr.SetItemCheck('Disable',False) !toggle the menu check mark
    ENABLE(?Save)                       !toggle ?Save button state
  ELSE                                  !which automatically toggles
    PopupMgr.SetItemCheck('Disable',True) !the Save menu item, because
    DISABLE(?Save)                      !it mimics the ?Save button
  END
OF 'Restore'                          !if Restore item selected
  POST(EVENT:Accepted,?Restore)        !code your own functionality
ELSE                                    !if any other item selected
  END                                    !Ask automatically handled it
END
END
CASE FIELD()
OF ?Save                               !Save button mimiced by Save item
  CASE EVENT()
  OF EVENT:Accepted
    PopupMgr.Save('MyPopup')          !save menu definition to INI
    RUN('Notepad '&INIFile)          !display/edit menu definition
    ENABLE(?Restore)                   !enable the Restore button
    PopupMgr.SetItemEnable('Restore',True) !enable the Restore item
  END
OF ?Restore
  CASE EVENT()
  OF EVENT:Accepted
    PopupMgr.Restore('MyPopup')       !restore/define menu from INI
  END
OF ?Close                               !Close btn Accepted by Close item
  CASE EVENT()
  OF EVENT:Accepted
    POST(Event:CloseWindow)
    END
  END
END
END
PopupMgr.Kill

```

## PopupClass Properties

The PopupClass contains the properties described below.

### ClearKeycode (clear KEYCODE character)

**ClearKeycode** BYTE

The **ClearKeycode** property determines whether the PopupClass object clears the (MouseRight) value from the KEYCODE() "buffer" before invoking the selected menu item's action. A value of one (1 or True) sets the KEYCODE() "buffer" to zero; a value of zero (0 or False) leaves the KEYCODE() "buffer" intact. See *KEYCODE* and *SETKEYCODE* in the *Language Reference* for more information.

**Tip:** The uncleared KEYCODE() value can cause the popup menu to reappear in some circumstances; therefore we recommend setting the ClearKeycode property to True.

**Implementation:** The ABC Templates set the ClearKeycode property to True by default. The Ask method implements the action specified by the ClearKeycode property.

**See Also:** Ask, Init

## PopupClass Methods

The PopupClass contains the methods listed below.

### **PopupClass Functional Organization--Expected Use**

As an aid to understanding the PopupClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the PopupClass methods.

#### **Non-Virtual Methods**

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the PopupClass object
AddMenu	add a menu
AddItem	add menu item
AddItemEvent	set menu item action
AddItemMimic	tie menu item to a button
AddSubMenu	add submenu
Kill	shut down the PopupClass object

##### **Mainstream Use:**

Ask	display and process the popup menu
GetItemChecked	return toggle item status
GetItemEnabled	return item status
SetItemCheck	set toggle item status
SetItemEnable	set item status

##### **Occasional Use:**

DeleteItem	remove menu item
GetLastSelection	return last selected item
SetTranslator	set run-time translator
Save	save a menu for restoration
SetLevel	set menu item hierarchy level
SetText	set menu item text
Restore	restore a saved menu

#### **Virtual Methods**

---

The PopupClass has no virtual methods.

**AddItem (add menu item)**

```
AddItem( text | [,name]           | )
          | name, position, level |
```

---

<b>AddItem</b>	Adds an item to the popup menu.
<i>text</i>	A string constant, variable, EQUATE, or expression containing the text of the menu item. A single hyphen (-) creates a non-selectable separator (a 3D horizontal bar) on the menu. An ampersand (&) designates the next character as the menu item's hot key.
<i>name</i>	A string constant, variable, EQUATE, or expression containing the menu item name. Other PopupClass methods refer to the menu item by its <i>name</i> , not by its <i>text</i> . This lets you apply runtime translation or dynamic reordering of menus without changing your code. If omitted, AddItem derives the <i>name</i> from the <i>text</i> .
<i>position</i>	A string constant, variable, EQUATE, or expression containing the name after which to add the new menu item.
<i>level</i>	An integer constant, variable, EQUATE, or expression containing the nesting level or depth of the new menu item.

The **AddItem** method adds an item to the popup menu.

You set the action taken for each menu item with the AddItemMimic or AddItemEvent methods, or with your own custom code. These methods (and your code) must refer to the menu items by name (not by text).

```
AddItem(text)
  Adds a single menu item at the end of the menu. The item name
  is derived.

AddItem(text, name)
  Adds a single menu item at the end of the menu with the name
  specified.

AddItem(text, name, position, level)
  Adds a single menu item following item position, at level level,
  with name specified.
```

Implementation: The *text* and *name* parameters accept up to 1024 characters.

Each derived menu item name is the same as its *text* minus any special characters. That is, the name contains only characters 'A-Z', 'a-z', and '0-9'. If the resulting name is not unique, the PopupClass appends a sequence number to the name to make it unique.

Example:

```
PopupMgr.AddItem('Save Popup')           !add menu item named SavePopup
PopupMgr.AddItem('Save Popup','Save')    !add menu item named Save
PopupMgr.AddItem('-', 'Separator')       !add a separator
PopupMgr.AddItem('Restore Popup','Restore','Save',1)!add Restore item after Save item
```

See Also:        AddItemEvent, AddItemMimic, SetText

**AddItemEvent (set menu item action)**

**AddItemEvent( *name*, *event* [,*control*] ), PROC**

---

**AddItemEvent** Associates an event with a menu item.

- name*            A string constant, variable, EQUATE, or expression containing the name of the menu item associated with the *event*. If the named item does not exist, AddItemEvent adds it at the bottom of the popup menu.
- event*            An integer constant, variable, EQUATE, or expression containing the event number to POST when the end user selects the menu item.
- control*          An integer constant, variable, EQUATE, or expression containing the control number to POST the *event* to when the end user selects the menu item. To post a field-independent event, use a *control* value of zero (0). If omitted, *control* defaults to zero (0).

The **AddItemEvent** method associates an *event* with a menu item and returns the name of the item. When the end user selects the menu item, the PopupClass object POSTs the *event* to the *control*.

Implementation:    The Ask method traps the selected item and POSTs the *event*.

                      The *name* parameter accepts up to 1024 characters.

Return Data Type:    STRING

Example:

```

PopupMgr.AddItem('Close (control event)','Close')    !add a menu item: Close
PopupMgr.AddItemEvent('Close',EVENT:Accepted,?Close)!Close POSTs event to a control
PopupMgr.AddItem('Close (window event)','Close2')    !add a menu item: Close2
PopupMgr.AddItemEvent('Close2',EVENT:CloseWindow,0) !Close2 POSTs independent event
    
```

See Also:            AddItem, AddItemMimic, AddMenu, Ask

**AddItemMimic (tie menu item to a button)**

**AddItemMimic**( *name*, *button* [, *text* ] ), PROC

---

**AddItemMimic** Associates a menu item with a BUTTON.

*name*            A string constant, variable, EQUATE, or expression containing the menu item name to associate with the *button*. If the named item does not exist, AddItemMimic adds it at the bottom of the popup menu. To add a new item, the *button* must have text, or you must supply the *text* parameter.

*button*           A numeric constant, variable, EQUATE, or expression containing the associated BUTTON's control number. If the button has no text, you should supply the *text* parameter.

*text*              A string constant, variable, EQUATE, or expression containing the text of the menu item. Other PopupClass methods refer to the menu item by its *name*, not by its *text*. This lets you apply runtime translation or dynamic reordering of menus without changing your code. If omitted, AddItemMimic uses the button text as the text of the menu item.

The **AddItemMimic** method associates a menu item with a *button* and returns the name of the item. AddItemMimic can add a *new* menu item, or add an *association* to an *existing* menu item. The associated menu item text matches the *button* text, is enabled only when the *button* is enabled, and, when selected, invokes the *button* action.

Implementation:    The Ask method traps the selected item and POSTs an EVENT:Accepted to the *button*.

If *button* does not represent a BUTTON, AddItemMimic does nothing.

The *text* and *name* parameters accept up to 1024 characters.

Return Data Type:    STRING

Example:

```

PopupMgr.AddItem('Save Popup','Save')            !add menu item: Save
PopupMgr.AddItemMimic('Save',?Save)            !Save item mimics ?Save button
PopupMgr.AddItemMimic('Insert',?Insert)        !add Insert item & mimic ?Insert button

```

See Also:            AddItem, AddMenu, Ask, SetText

## AddMenu (add a menu)

**AddMenu**( *selections* [, *position* ] )

---

<b>AddMenu</b>	Adds a popup menu.
<i>selections</i>	A string constant, variable, EQUATE, or expression containing the text for the popup menu choices.
<i>position</i>	An integer constant, variable, EQUATE, or expression containing the position within the PopupClass' existing menu at which to add the <i>selections</i> . If omitted or zero (0), AddMenu clears any existing menu selections.

The **AddMenu** method adds an entire popup menu or adds additional selections to an existing menu. The AddMenu method creates a popup menu item with a unique name for each text specified by the *selections* parameter. The *selections* parameter is identical to the *selections* parameter for the POPUP command. See *POPUP* in the *Language Reference* for more information.

You set the action taken for each menu item with the AddItemMimic or AddItemEvent methods, or with your own custom code. These methods (and your code) must refer to the menu items by name (not by text).

Implementation: The AddMenu method optionally replaces any previously defined menu for this PopupClass object. The Ask method displays the popup menu and returns the selected item's name.

The Popup class object derives the menu item name from its text. Each derived item name is the same as its text minus any special characters. That is, the name contains only characters 'A-Z', 'a-z', and '0-9'. If the resulting name is not unique, the PopupClass appends a sequence number to the name to make it unique.

The *selections* parameter accepts up to 10,000 characters.

Example:

```
MenuChoices EQUATE('&Save Menu|&Restore Menu|-|&Close')!declare menu definition string
CODE
  PopupMgr.AddMenu(MenuChoices)                !add Popup menu
  PopupMgr.AddItemMimic('SaveMenu',?Save)      !SaveMenu mimics ?Save button
  PopupMgr.AddItemEvent('Close',EVENT:Accepted,?Close) !Close POSTs event to a control
!program code
  IF PopupMgr.Ask() = 'RestoreMenu'            !if RestoreMenu item selected
    PopupMgr.Restore('MyMenu')                 !code your own functionality
  ELSE                                         !if any other item selected
  END                                           !Ask automatically handled it
```

See Also: AddItemEvent, AddItemMimic, Ask

**AddSubMenu (add submenu)**

**AddSubMenu**( [*text*] ,*selections*, *name to follow* )

**AddSubMenu** Adds a submenu to an existing menu.

- text* A string constant, variable, EQUATE, or expression containing the submenu text. If omitted, the submenu text must be prepended to the *selections* parameter.
- selections* A string constant, variable, EQUATE, or expression containing the text for the submenu items. The submenu items must be preceded by a double open curly brace ({{) and followed by a single close curly brace (}).
- name to follow* A string constant, variable, EQUATE, or expression containing the menu name or item name after which to insert the submenu.

The **AddSubMenu** method adds a submenu to an existing menu. The AddSubMenu method adds a submenu and its items, including a unique name for each item specified by the *selections* parameter. The *selections* parameter is identical to the submenu section of the *selections* parameter for the POPUP command. See *POPUP* in the *Language Reference* for more information. Set the action taken for each menu item with the AddItemMimic or AddItemEvent methods, or with your own custom code. These methods (and your code) must refer to the menu items by name (not by text).

Implementation: The Ask method displays the popup menu and returns the selected item's name.

The Popup class object derives the menu item name from its text. Each derived item name is the same as its text minus any special characters. That is, the name contains only characters 'A-Z', 'a-z', and '0-9'. If the resulting name is not unique, the PopupClass appends a sequence number to the name to make it unique. The *text* parameter accepts up to 1,024 characters; the *selections* parameter accepts up to 10,000 characters.

Example:

```
MenuChoices EQUATE('&Insert|&Change|&Delete')    !declare menu definition string
SubChoices EQUATE('{{by &name|by &ZIP code}}')    !declare submenu definition
CODE
  PopupMgr.AddMenu(MenuChoices)                  !add Popup menu
  PopupMgr.AddSubMenu('&Print',SubChoices,'Delete')!add Print submenu after delete
CASE PopupMgr.Ask()
  OF ('Insert')    ;DO Update(1)                  !process end user choice
  OF ('Change')    ;DO Update(2)                  !process end user choice
  OF ('Delete')    ;DO Update(3)                  !process end user choice
  OF ('byname')    ;DO PrintByName                !process end user choice
  OF ('byZIPcode') ;DO PrintByZIP                 !process end user choice
END
```

See Also: AddItemEvent, AddItemMimic, AddMenu, Ask

**Ask (display the popup menu)**

**Ask**( [x] [,y] ), PROC

---

<b>Ask</b>	Returns the selected popup menu item name.
x	An integer constant, variable, EQUATE, or expression that specifies the horizontal position of the top left corner of the menu. If omitted, the menu appears at the current cursor position.
y	An integer constant, variable, EQUATE, or expression that specifies the vertical position of the top left corner of the menu. If omitted, the menu appears at the current cursor position.

The **Ask** method displays the popup menu, performs any action set by AddItemEvent or AddItemMimic for the selected item, then returns the selected item's name. The AddItem, AddItemMimic, or AddMenu method sets the item name.

Return Data Type:   STRING

Example:

```
MenuChoices EQUATE('&Save Menu|&Restore Menu|-|&Close')!declare menu definition string
CODE
  PopupMgr.AddMenu(MenuChoices)                                !add Popup menu
  PopupMgr.AddItemMimic('SaveMenu',?Save)                    !SaveMenu mimics ?Save button
  PopupMgr.AddItemEvent('Close',EVENT:Accepted,?Close)      !Close POSTs event to a control
!program code
  IF PopupMgr.Ask() = 'RestoreMenu'                            !if RestoreMenu item selected
    PopupMgr.Restore('MyMenu')                                !code your own functionality
  ELSE                                                         !if any other item selected
    END                                                         !Ask automatically handled it
```

See Also:           AddItem, AddItemMimic, AddMenu

**Deleteltem (remove menu item)****Deleteltem**( *name* )**Deleteltem** Deletes a popup menu item.*name* A string constant, variable, EQUATE, or expression containing the menu item name. The AddItem, AddItemMimic, or AddMenu methods set the item name.The **Deleteltem** method deletes a popup menu item and any associated submenu items.Implementation: The *name* parameter accepts up to 1024 characters.

Example:

```

PopupMgr.AddItem('&Insert','Insert')           !Insert item
PopupMgr.AddItem('&Change','Change')          !Change item
PopupMgr.AddItem('&Delete','Delete')          !Delete item
PopupMgr.AddItem('&Select','Select')          !Select item
IF No_Records_Found
  PopupMgr.DeleteItem('Change')                !remove change item
  PopupMgr.DeleteItem('Delete')                !remove delete item
  PopupMgr.DeleteItem('Select')                !remove select item
END

```

See Also: AddItem, AddItemMimic, AddMenu

## DeleteMenu (remove a popup submenu)

**DeleteMenu**( *name* )

---

**DeleteMenu** Deletes a popup submenu item, and all associated child items.

*name* A string constant, variable, EQUATE, or expression containing the submenu item name. The AddMenu method set the item name.

The **DeleteMenu** method deletes a popup submenu item and any associated child items.

Implementation: The *name* parameter accepts up to 1024 characters.

Example:

```
PopupMgr.AddItem('&Insert','Insert')           !Insert item
PopupMgr.AddItem('&Change','Change')           !Change item
PopupMgr.AddItem('&Delete','Delete')           !Delete item
PopupMgr.AddItem('&Select','Select')           !Select item
IF No_Records_Found
  PopupMgr.DeleteItem('Change')                 !remove change item
  PopupMgr.DeleteItem('Delete')                 !remove delete item
  PopupMgr.DeleteItem('Select')                 !remove select item
  PopupMgr.DeleteMenu('Format List')            !remove select sub menu
END
```

See Also: AddItem, AddItemMimic, AddMenu

## GetItemChecked (return toggle item status)

**GetItemChecked**( *name* )

---

**GetItemChecked** Returns the status of a toggle menu item.

*name* A string constant, variable, EQUATE, or expression containing the menu item name. The AddItem, AddItemMimic, or AddMenu methods set the item name.

The **GetItemChecked** method returns one (1) if the item is checked (on) and zero (0) if the item is not checked (off). The SetItemCheck method sets the state of a toggle menu item.

Implementation: The *name* parameter accepts up to 1024 characters.

Return Data Type: BYTE

Example:

```
IF PopupMgr.Ask() = 'Disable'           !if Disable item selected
IF PopupMgr.GetItemChecked('Disable')  !if item is checked/on
  PopupMgr.SetItemCheck('Disable',False) ! toggle it off
  ENABLE(?Save)                        ! take appropriate action
ELSE
  PopupMgr.SetItemCheck('Disable',True) ! toggle it on
  DISABLE(?Save)                       ! take appropriate action
END
END
```

See Also: AddItem, AddItemMimic, AddMenu, SetItemCheck

**GetItemEnabled (return item status)**

**GetItemEnabled**( *name* )

---

**GetItemEnabled** Returns the enabled/disabled status of a menu item.

*name* A string constant, variable, EQUATE, or expression containing the menu item name. The AddItem, AddItemMimic, or AddMenu methods set the item name.

The **GetItemEnabled** method returns one (1) if the item is enabled and zero (0) if the item is disabled. The SetItemEnable method sets the enabled/disabled state of a menu item.

Implementation: The *name* parameter accepts up to 1024 characters.

Return Data Type: BYTE

Example:

```
IF PopupMgr.GetItemEnabled('Save')      !if item is enabled
  PopupMgr.SetItemEnable('Save',False) ! disable it
ELSE                                     !if item is disabled
  PopupMgr.SetItemEnable('Save',True)  ! enable it
END
```

See Also: AddItem, AddItemMimic, AddMenu, SetItemEnable

## **GetItems(returns number of popup entries)**

### **GetItems(*onlyitems*)**

The **GetItems** method returns the number of entries that exist in the current active popup menu.

The *onlyitems* parameter (DEFAULT=0) specifies that only the number of popup items is returned. If non-zero, **GetItems** also returns the depth (nested) levels of the popup menu.

Return Data Type:      SHORT

Example:

```
IF Popup.GetItems() ! Only if there are previous items.  
  Popup.AddItem( '-', 'Separator1', Popup.GetItems(), 1)  
END
```

## **GetLastNumberSelection (get last menu item number selected)**

### **GetLastNumberSelection**

The GetLastNumberSelection method returns the numeric order of the last selected menu item. This is also the last number returned by the POPUP() function

The ExecutePopup method (a PRIVATE method) sets the menu item number.

This method is equivalent to:  
**Return SELF.LastNumberSelection**

Return Data Type:    USHORT

## GetLastSelection (return selected item)

### GetLastSelection

The **GetLastSelection** method returns the name of the last selected item.

The AddItem, AddItemMimic, AddMenu, or AddSubMenu method sets the item name.

Return Data Type: STRING

Example:

```
MenuChoices EQUATE('Fred|Barney|Wilma')      !declare menu definition string
CODE
PopupMgr.AddMenu(MenuChoices)                 !add Popup menu
!program code
PopupMgr.Ask()                                !display menu
MESSAGE('Thank you for choosing '&PopupMgr.GetLastSelection)
```

See Also: AddItem, AddItemMimic, AddMenu, AddSubMenu

**Init (initialize the PopupClass object)**`Init([INIClass])`


---

<b>Init</b>	Initializes the PopupClass object.
<i>INIClass</i>	The label of the INIClass object for this PopupClass object. The Save method uses the INIClass object to save menu definitions to an INI file; the Restore method uses it to restore the saved menu definitions. If omitted, the Save and Restore methods do nothing.

The **Init** method initializes the PopupClass object.

Example:

```

PopupMgr  PopupClass          !declare PopupMgr object
INIMgr    INIClass            !declare INIMgr object
CODE
  PopupMgr.Init(INIMgr)       !initialize PopupMgr object
  PopupMgr.AddItem('Save Popup','Save') !add menu item: Save
  PopupMgr.AddItemMimic('Save',?Save)  !Save item mimics ?Save button

```

See Also:           Restore, Save

**Kill (shut down the PopupClass object)****Kill**

The **Kill** method frees any memory allocated during the life of the PopupClass object and performs any other required termination code.

Example:

```

PopupMgr.Init                !initialize PopupMgr object
!program code
PopupMgr.Kill                 !shut down PopupMgr object

```

**Restore (restore a saved menu)**

**Restore**( *menu* )

---

<b>Restore</b>	Restores a menu saved by the PopupClass.Save method.
<i>menu</i>	A string constant, variable, EQUATE, or expression containing the name of the menu to restore.

The **Restore** method restores a menu saved by the Save method. The Restore method restores all menu attributes that the PopupClass object knows about, including associated menu actions.

Implementation: The Restore method requires an INIClass object. The Init method specifies the INIClass object.

Example:

```

PopupMgr  PopupClass          !declare PopupMgr object
INIMgr    INIClass           !declare INIMgr object
MenuChoices  EQUATE('&Save Menu|&Restore Menu|-|&Close')!declare menu definition

CODE
PopupMgr.Init(INIMgr)          !initialize PopupMgr object
PopupMgr.AddMenu(MenuChoices) !add Popup menu
ACCEPT
CASE FIELD()
OF ?Save
CASE EVENT()
OF EVENT:Accepted
  PopupMgr.Save('MyPopup')    !save menu definition to INI
END
OF ?Restore
CASE EVENT()
OF EVENT:Accepted
  PopupMgr.Restore('MyPopup') !restore menu from INI
END
END
END
END
  
```

See Also: Init, Save

**Save (save a menu for restoration)****Save**( *menu* )

---

<b>Save</b>	Saves a menu for restoration by the PopupClass.Restore method.
<i>menu</i>	A string constant, variable, EQUATE, or expression containing the name of the menu to save.

The **Save** method saves a menu for restoration by the Restore method. The Save method saves all menu attributes that the PopupClass object knows about, including associated menu actions.

Implementation: The Save method requires an INIClass object. The Init method specifies the INIClass object.

Example:

```

PopupMgr   PopupClass           !declare PopupMgr object
INIMgr     INIClass             !declare INIMgr object
MenuChoices   EQUATE('&Save Menu|&Restore Menu|-|&Close')!declare menu definition

CODE
PopupMgr.Init(INIMgr)           !initialize PopupMgr object
PopupMgr.AddMenu(MenuChoices)  !add Popup menu
ACCEPT
  CASE FIELD()
  OF ?Save
    CASE EVENT()
    OF EVENT:Accepted
      PopupMgr.Save('MyPopup')   !save menu definition to INI
    END
  OF ?Restore
    CASE EVENT()
    OF EVENT:Accepted
      PopupMgr.Restore('MyPopup') !restore menu from INI
    END
  END
END
END

```

See Also:        Init, Restore

## SetIcon (set icon name for popup menu item)

**SetIcon**( *name*, *iconname* )

---

**SetIcon** Sets the icon name of a popup menu item.

*name* A string constant, variable, EQUATE, or expression containing the menu item name. The AddItem, AddItemMimic, or AddMenu methods set the item name.

*iconname* A string constant, variable, EQUATE, or expression containing the name of the icon to attach to the menu item.

The **SetIcon** method sets the icon of a popup menu item. The AddItem or AddSubItem methods adds the menu item to the popup.

Implementation: The *iconname* parameter accepts up to 255 characters.

Example:

```
LOOP sm = 1 to RECORDS(QQ)
  GET(QQ,sm)
  pID = Popup.AddItem(CLIP(QQ.Item) & ' ',Clip(QQ.Item),pID,2)
  Popup.SetIcon(pID,SELF.QkIcon)
  Popup.AddItemEvent(pID,EVENT:NewSelection,QueryControl)
  SELF.PopupList.PopupID = pID
  SELF.PopupList.QueryName = QQ:Item
  ADD(SELF.PopupList)
END
```

**SetItemCheck (set toggle item status)****SetItemCheck**( *name*, *status* )**SetItemCheck** Sets the status of a toggle menu item.

*name*            A string constant, variable, EQUATE, or expression containing the menu item name. The AddItem, AddItemMimic, or AddMenu methods set the item name.

*status*           A Boolean constant, variable, EQUATE, or expression containing the status to which to set the toggle item. A *status* value of one (1) indicates a checked (on) item; zero (0) indicates an unchecked (off) item.

The **SetItemCheck** method sets the status of a toggle menu item. The GetItemChecked method returns the status of a toggle menu item.

Implementation:    The *name* parameter accepts up to 1024 characters.

Example:

```

IF PopupMgr.Ask() = 'Disable'            !if Disable item selected
  IF PopupMgr.GetItemChecked('Disable') !if item is checked/on
    PopupMgr.SetItemCheck('Disable',False) ! toggle it off
    ENABLE(?Save)                        ! take appropriate action
  ELSE                                    !if item is not checked/off
    PopupMgr.SetItemCheck('Disable',True) ! toggle it on
    DISABLE(?Save)                       ! take appropriate action
  END
END

```

See Also:            AddItem, AddItemMimic, AddMenu, GetItemChecked

## SetItemEnable (set item status)

**SetItemEnable**( *name* )

---

**SetItemEnable** Sets the enabled/disabled status of a menu item.

*name*            A string constant, variable, EQUATE, or expression containing the menu item name. The AddItem, AddItemMimic, or AddMenu methods set the item name.

*status*          A Boolean constant, variable, EQUATE, or expression containing the status to which to set the item. A *status* value of one (1) indicates an enabled item; zero (0) indicates a disabled item.

The **SetItemEnable** method sets the enabled/disabled status of a menu item. The GetItemEnabled method returns the enabled/disabled status of a menu item.

Implementation:    The *name* parameter accepts up to 1024 characters.

Example:

```
IF PopupMgr.GetItemEnabled('Save')            !if item is enabled
  PopupMgr.SetItemEnable('Save',False)       ! disable it
ELSE                                            !if item is disabled
  PopupMgr.SetItemEnable('Save',True)        ! enable it
END
```

See Also:            AddItem, AddItemMimic, AddMenu, GetItemEnabled

**SetLevel (set menu item level)**

**SetLevel**( *name*, *level* )

---

<b>SetLevel</b>	Sets the menu item hierarchy level.
<i>name</i>	A string constant, variable, EQUATE, or expression containing the menu item name. The AddItem, AddItemMimic, or AddMenu methods set the item name.
<i>level</i>	An integer constant, variable, a EQUATE, or expression containing the level of the menu item.

The **SetLevel** method sets the menu item hierarchy (nesting) level.

Implementation: The *name* parameter accepts up to 1024 characters.

Example:

```
PopupMgr.SetLevel( 'Save' , 2 )
```

See Also: AddItem, AddItemMimic, AddMenu

**SetText (set menu item text)**

**SetText**( *name*, *text* )

---

<b>SetText</b>	Sets the menu item text.
<i>name</i>	A string constant, variable, EQUATE, or expression containing the menu item name. The AddItem, AddItemMimic, or AddMenu methods set the item name.
<i>text</i>	A string constant, variable, EQUATE, or expression containing the text of the menu item. A single hyphen creates a non-selectable separator (a 3D horizontal bar) on the menu.

The **SetText** method sets the text for a menu item.

Implementation: The *name* and *text* parameters accept up to 1024 characters.

Example:

```
PopupMgr.SetText( 'Save', '&Save' )
```

See Also: AddItem, AddItemMimic, AddMenu

## **SetToolbox (set menu item toolbox status)**

**SetToolbox**( *name*, *showflag* )

---

**SetToolbox** Sets the menu item toolbox appearance status.

*name* A string constant, variable, EQUATE, or expression containing the menu item name. The AddItem, AddItemMimic, or AddMenu methods set the item name.

*showflag* A Boolean constant, variable, EQUATE, or expression containing the status to which to set the item. A *showflag* value of one (1) indicates that the item will appear on the toolbox; zero (0) indicates that the item will not appear on the toolbox.

The **SetToolbox** method sets whether or not the menu item will appear on a popup toolbox. The Toolbox method displays the popup menu in a toolbox format.

Implementation: The SetToolbox method is used with the BrowseClass SetAlerts method when the BrowseClass ToolControl property is set to TRUE.

Example:

```
IF SELF.ToolControl
  SELF.Popup.AddItem( '-' )
  SELF.Popup.SetToolbox( SELF.Popup.AddItemMimic( DefaultToolName, |
    SELF.ToolControl, '!' & DefaultToolName ), 0 )
END
```

See Also: BrowseClass.SetAlerts

Toolbox

**SetTranslator (set run-time translator:PopupClass)****SetTranslator**( *translator* )

---

**SetTranslator** Sets the TranslatorClass object for the PopupClass object.*translator*        The label of the TranslatorClass object for this PopupClass object.

The **SetTranslator** method sets the TranslatorClass object for the PopupClass object. By specifying a TranslatorClass object for the PopupClass object, you can automatically translate the popup menu text--the TranslatorClass object does not otherwise translate popup menus because they are not part of the WINDOW structure.

Implementation:    The Ask method uses the TranslatorClass object to translate popup menu text before displaying it.

Example:

```

PopupMgr  PopupClass           !declare PopupMgr object
Translator TranslatorClass     !declare Translator object

!declare menu definition
MenuChoices  EQUATE('&Save Menu|&Restore Menu|&Close')
CODE
Translator.Init                !initialize Translator object
PopupMgr.Init(INIMgr)         !initialize PopupMgr object
PopupMgr.AddMenu(MenuChoices) !add Popup menu
PopupMgr.SetTranslator(Translator) !enable popup text translation
!program code
PopupMgr.Ask()                !display translated menu

```

See Also:        Ask

## Toolbox (start the popup toolbox menu)

**Toolbox**( *name* )

---

**Toolbox** Starts (displays) the popup menu toolbox.

*name* A string constant, variable, EQUATE, or expression containing the toolbox name. This name will appear in the title bar of the toolbox window.

The **Toolbox** method displays the toolbox popup window. All items enabled by the SetToolbox method are displayed as buttons, with the *name* parameter appearing in the toolbox window's title bar.

Implementation: The Toolbox method is called from the BrowseClass TakeEvent method when the ToolControl property is set to TRUE. It is used to support the BrowseToolbar control template.

Example:

```
BrowseClass.TakeEvent PROCEDURE  
  
CASE ACCEPTED()  
  OF SELF.ToolControl  
    SELF.Popup.Toolbox('Browse Actions')  
END
```

See Also: SetToolbox  
BrowseClass.TakeEvent

**ViewMenu (popup menu debugger)****ViewMenu**

The **ViewMenu** method displays information about the structure of the popup menu built up by the various 'Add' methods.

Implementation: The ViewMenu method only works when the program is compiled with debug information turned on.



---

# PrintPreviewClass

## PrintPreviewClass Overview

The PrintPreviewClass is a WindowManager that implements a full-featured print preview dialog.

## PrintPreviewClass Concepts

This print preview facility includes pinpoint zoom-in and zoom-out with configurable zoom magnification, random and sequential page navigation, plus thumbnail views of each report page. You can even specify how many rows and columns of thumbnails the print preview facility displays.

When you finish viewing the report, you can send it directly to the printer for immediate What You See Is What You Get (WYSIWYG) printing.

The PrintPreviewClass previews reports in the form of a Windows metafile (.WMF) per report page. The PREVIEW attribute generates reports in Windows metafile format, and the Clarion Report templates provide this capability as well. See PREVIEW in the *Language Reference* for more information, and see Procedure Templates--Report for more information on Report templates.

## PrintPreviewClass Relationship to Other Application Builder Classes

The PrintPreviewClass is derived from the WindowManager class (see Window Manager Class for more information).

The PrintPreviewClass relies on the PopupClass and, optionally, the TranslatorClass to accomplish some of its tasks. Therefore, if your program instantiates the PrintPreviewClass, it should also instantiate the PopupClass and may need the Translator class as well. Much of this is automatic when you INCLUDE the PrintPreviewClass header (ABREPORT.INC) in your program's data section. See the Conceptual Example.

The ASCIIPrintClass and the ReportManager use the PrintPreviewClass to provide a print preview facility.

## PrintPreviewClass ABC Template Implementation

The Report and Viewer Procedure templates and the Report Wizard Utility template automatically generate all the code and include all the classes necessary to provide the print preview facility for your application's reports.

These Report templates instantiate a PrintPreviewClass object called Previewer for *each* report procedure in the application. This object supports all the functionality specified in the **Preview Options** section of the Report template's **Report Properties** dialog. See *Procedure Templates--Report* for more information.

The template generated ReportManager object (ThisWindow) "drives" the Previewer object, so generally, the only references to the Previewer object within the template generated code are to initially configure the Previewer's properties.

## PrintPreviewClass Source Files

The PrintPreviewClass source code is installed by default to the Clarion \LIBSRC folder. The PrintPreviewClass source code and its respective components are contained in:

ABREPORT.INC	PrintPreviewClass declarations
ABREPORT.CLW	PrintPreviewClass method definitions
ABREPORT.TRN	PrintPreviewClass user interface text

## Zoom Configuration

The user interface text and the standard zoom choices the PrintPreviewClass displays at runtime are defined in the ABREPORT.TRN file. To modify or customize this text or the standard zoom choices, simply back up the ABREPORT.TRN file then edit it to suit your needs. See *ZoomIndex* for more information.

## PrintPreviewClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a PrintPreviewClass object and some related objects.

This example uses the PrintPreviewClass object to preview a very simple report before printing it. The program specifies an initial position and size for the print preview window and allows custom zoom factors.

```

PROGRAM

    INCLUDE('ABREPORT.INC')           !declare ReportManager & PrintPreviewClass
    MAP
    END

GlobalErrors ErrorClass
VCRRequest  LONG(0),THREAD

Customer    FILE,DRIVER('TOPSPEED'),PRE(CUS),THREAD
BYNUMBER    KEY(CUS:CUSTNO),NOCASE,OPT,PRIMARY
Record      RECORD,PRE()
CUSTNO      LONG
Name        STRING(30)
State       STRING(2)
            END
            END

Access:Customer CLASS(FileManager)    !declare Access:Customer object
Init           PROCEDURE
            END

Relate:Customer CLASS(RelationManager) !declare Relate:Customer object
Init           PROCEDURE
            END

CusView       VIEW(Customer)          !declare CusView VIEW
            END

PctDone       BYTE                    !track progress variable

```

```

report  REPORT,AT(1000,1542,6000,7458),PRE(RPT),FONT('Arial',10,,),THOUS
        HEADER,AT(1000,1000,6000,542),FONT(,,FONT:bold)
        STRING('Customers'),AT(2000,20),FONT(,14,,)
        STRING('Id'),AT(52,313),TRN
        STRING('Name'),AT(2052,313),TRN
        STRING('State'),AT(4052,313),TRN
        END
detail  DETAIL,AT(,,6000,281),USE(?detail)
        STRING(@n-14),AT(52,52),USE(CUS:CUSTNO)
        STRING(@s30),AT(2052,52),USE(CUS:NAME)
        STRING(@s2),AT(4052,52),USE(CUS:State)
        END
        FOOTER,AT(1000,9000,6000,219)
        STRING(@pPage <<<#p),AT(5250,31),PAGE NO,USE(?PageCount)
        END
        END

ProgressWindow WINDOW('Progress...'),AT(,,142,59),CENTER,TIMER(1),GRAY,DOUBLE
        PROGRESS,USE(PctDone),AT(15,15,111,12),RANGE(0,100)
        STRING(' '),AT(0,3,141,10),USE(?UserString),CENTER
        STRING(' '),AT(0,30,141,10),USE(?TxtDone),CENTER
        BUTTON('Cancel'),AT(45,42),USE(?Cancel)
        END

ThisProcedure CLASS(ReportManager)           !declare ThisProcedure object
Init          PROCEDURE(),BYTE,PROC,VIRTUAL
Kill          PROCEDURE(),BYTE,PROC,VIRTUAL
              END

CusReport     CLASS(ProcessClass)            !declare CusReport object
TakeRecord    PROCEDURE(),BYTE,PROC,VIRTUAL
              END

Previewer     PrintPreviewClass              !declare Previewer object
              ! for use with ThisProcedure

CODE
ThisProcedure.Run()                          !run the procedure

ThisProcedure.Init PROCEDURE()               !initialize ThisProcedure
ReturnValue    BYTE,AUTO
CODE
GlobalErrors.Init
Relate:Customer.Init
ReturnValue = PARENT.Init()
SELF.FirstField = ?PctDone
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:Customer.Open

```

```
OPEN(ProgressWindow)
SELF.Opened=True
CusReport.Init(CusView,Relate:Customer,?TxtDone,PctDone,RECORDS(Customer))
CusReport.AddSortOrder(CUS:BYNUMBER)
SELF.AddItem(?Cancel,RequestCancelled)
SELF.Init(CusReport,report,Previewer) !register Previewer with ThisProcedure
SELF.Zoom = PageWidth
Previewer.AllowUserZoom=True !allow custom zoom factors
Previewer.Maximize=True !initially maximize preview window
SELF.SetAlerts()
RETURN ReturnValue

ThisProcedure.Kill PROCEDURE()
ReturnValue BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Customer.Close
Relate:Customer.Kill
GlobalErrors.Kill
RETURN ReturnValue

CusReport.TakeRecord PROCEDURE()
ReturnValue BYTE,AUTO
SkipDetails BYTE
CODE
ReturnValue = PARENT.TakeRecord()
PRINT(RPT:detail)
RETURN ReturnValue

Access:Customer.Init PROCEDURE
CODE
PARENT.Init(Customer,GlobalErrors)
SELF.FileNameValue = 'Customer'
SELF.Buffer &= CUS:Record
SELF.Create = 0
SELF.LazyOpen = False
SELF.AddKey(CUS:BYNUMBER,'CUS:BYNUMBER',0)

Relate:Customer.Init PROCEDURE
CODE
Access:Customer.Init
PARENT.Init(Access:Customer,1)
```

## PrintPreviewClass Properties

The PrintPreviewClass contains properties that primarily allow configuration of the print preview window and its features. The PrintPreviewClass properties are described below.

### **AllowUserZoom (allow any zoom factor)**

**AllowUserZoom**      **BYTE**

The **AllowUserZoom** property indicates whether the PrintPreviewClass object provides user zoom capability for the end user. The user zoom lets the end user apply any zoom factor. Without user zoom, the end user may only apply the standard zoom choices.

The ZoomIndex property indicates whether a user zoom factor or a standard zoom factor is applied.

Implementation:      A value of one (1) enables user zoom capability; a value of zero (0) disables user zoom. The UserPercentile property contains the user zoom factor.

See Also:              UserPercentile, ZoomIndex

### **ConfirmPages (force 'pages to print' confirmation)**

**ConfirmPages**      **BYTE**

The **ConfirmPages** property indicates whether or not the AskPrintPages method should be called before printing.

Implementation:      Zero (0) is the default; a value of one (1) forces the enduser to choose the pages to print before the print job is sent to the printer.

See Also: AskPrintPages

### **CurrentPage (the selected report page)**

**CurrentPage**    **LONG**

The **CurrentPage** property contains the number of the selected report page. The PrintPreviewClass object uses this property to highlight the selected report page when more than one page is displayed, to navigate pages, and to display the current page number for the end user.

### **Maximize (number of pages displayed horizontally)**

**Maximize**    **BYTE**

The **Maximize** property indicates whether to open the preview window maximized. A value of one (1 or True) maximizes the window; a value of zero (0 or False) opens the window according to the WindowSizeSet property.

See Also:        WindowSizeSet

### **PagesAcross (number of pages displayed horizontally)**

**PagesAcross**    **USHORT**

The **PagesAcross** property contains the number of thumbnail pages the PrintPreviewClass object displays *horizontally* within the preview window. The PrintPreviewClass object uses this property to calculate appropriate positions and sizes when displaying several pages at a time.

The PrintPreviewClass object displays the PagesAcross value at runtime and lets the end user set the value as well.

## **PagesDown (number of vertical thumbnails)**

**PagesDown**    **USHORT**

The **PagesDown** property contains the number of thumbnail pages the PrintPreviewClass object displays *vertically* within the preview window. The PrintPreviewClass object uses this property to calculate appropriate positions and sizes when displaying several pages at a time.

The PrintPreviewClass object displays the PagesDown value at runtime and lets the end user set the value as well.

## **PagesToPrint (the pages to print)**

**PagesToPrint**    **CSTRING(256), PROTECTED**

The **PagesToPrint** property contains the page range to print.

The default value is 1-*n*, where *n* is equal to the total number of pages in the report. Individual pages can be printed by separating page numbers by commas. A range of pages to print can be specified by separating the first page number to print and the last page number to print by a dash (-). Combinations of individual pages and ranges of pages are allowed.

## **UserPercentile (custom zoom factor)**

**UserPercentile**    **USHORT**

The **UserPercentile** property contains the user specified zoom factor. The PrintPreviewClass object solicits this factor from the end user and applies it to the selected report page when the AllowUserZoom property is True. The SetZoomPercentile method sets the UserPercentile property.

See Also:            AllowUserZoom, SetZoomPercentile

## **WindowPosSet (use a non-default initial preview window position)**

### **WindowPosSet BYTE**

The **WindowPosSet** property contains a value indicating whether a non-default initial position is specified for the print preview window. The PrintPreviewClass object uses this property to determine the initial position of the print preview window.

Implementation: The SetPosition method sets the value of this property. A value of one (1) indicates a non-default initial position is specified and is applied; a zero (0) indicates no position is specified and the default position is applied.

See Also: SetPosition

## **WindowSizeSet (use a non-default initial preview window size)**

### **WindowSizeSet BYTE**

The **WindowSizeSet** property contains a value indicating whether a non-default initial size is specified for the print preview window. The PrintPreviewClass object uses this property to determine the initial size of the print preview window.

Implementation: The SetPosition method sets the value of this property. A value of one (1) indicates a non-default initial size is specified and is applied; a zero (0) indicates no size is specified and the default size is applied.

See Also: SetPosition

## **ZoomIndex (index to applied zoom factor)**

**ZoomIndex**     **BYTE**

The **ZoomIndex** property contains a value indicating which zoom factor is applied. The PrintPreviewClass object uses this property to identify and apply the selected zoom factor. The SetZoomPercentile method sets the ZoomIndex property.

Implementation:     The ZoomIndex value "points" to one of the 7 standard zoom settings or to a user zoom setting. The PrintPreviewClass object sets the ZoomIndex value when the end user selects a zoom setting from one of the zoom menus or from the zoom combo box. The standard zoom choices are defined in ABREPORT.TRN as follows:

No Zoom	Displays the specified number of pages (PagesAcross and PagesDown properties) in a tiled arrangement in the preview window.
Page Width	Displays a single page whose width is the same as the width of the preview window.
50%	Displays a single page at 50% of actual print size.
75%	Displays a single page at 75% of actual print size.
100%	Displays a single page at 100% of actual print size.
200%	Displays a single page at 200% of actual print size.
300%	Displays a single page at 300% of actual print size.

A ZoomIndex value of zero (0) indicates a nonstandard zoom factor is specified. Nonstandard zoom factors may be specified when the AllowUserZoom property is True. The UserPercentile property contains the nonstandard zoom factor.

See Also:     AllowUserZoom, PagesAcross, PagesDown, UserPercentile, SetZoomPercentile

## PrintPreviewClass Methods

The PrintPreviewClass contains the methods listed below.

### PrintPreviewClass Functional Organization--Expected Use

As an aid to understanding the PrintPreviewClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the PrintPreviewClass methods.

#### Non-Virtual Methods

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into two categories:

##### **Housekeeping (one-time) Use:**

Initv	initialize the PrintPreviewClass object
SetPosition	set initial preview window coordinates
Displayv	preview the report
Killv	shut down the PrintPreviewClass object

##### **Occasional Use:**

SetNIManager	save and restore window coordinates
SetPosition	set print preview position and size
SetZoomPercentile	set user or standard zoom factor

v These methods are also Virtual.

#### Virtual Methods

Typically you will not call these methods directly--the Display method calls them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Initv	initialize the PrintPreviewClass object
AskPage	prompt for new report page
AskThumbnails	prompt for new thumbnail configuration
Display	preview the report
Open	prepare preview window for display
TakeAccepted	process EVENT:Accepted events
TakeEvent	process all events
TakeFieldEvent	a virtual to process field events
TakeWindowEvent	process non-field events
Killv	shut down the PrintPreviewClass object

**AskPage (prompt for new report page)****AskPage, PROC, VIRTUAL, PROTECTED**

The **AskPage** method prompts the end user for a specific report page to display and returns a value indicating whether a new page is selected. A return value of one (1) indicates a new page is selected and a screen redraw is required; a return value of zero (0) indicates a new page is not selected and a screen redraw is not required.

Implementation: The PrintPreviewClass.Display method calls the AskPage method. The AskPage method displays a dialog that prompts for a specific report page.

Return Data Type: BYTE

Example:

```
!Virtual implementation of AskPage: a simplified version with no translator...
PrintPreviewClass.AskPage FUNCTION
JumpPage LONG,AUTO
RVal      BOOL(False)

JumpWin WINDOW('Jump to Page'),AT(,,181,26),CENTER,GRAY,DOUBLE
        PROMPT('&Page: '),AT(5,8),USE(?JumpPrompt)
        SPIN(@n5),AT(30,7),USE(JumpPage),RANGE(1,10),STEP(1)
        BUTTON('OK'),AT(89,7),USE(?OKButton),DEFAULT
        BUTTON('Cancel'),AT(134,7),USE(?CancelButton)
        END

CODE
JumpPage=SELF.CurrentPage
OPEN(JumpWin)
ACCEPT
CASE EVENT()
OF EVENT:OpenWindow
    ?JumpPage{PROP:RangeHigh}=RECORDS(SELF.ImageQueue)
OF EVENT:Accepted
CASE ACCEPTED()
OF ?OKButton
    IF JumpPage NOT=SELF.CurrentPage
        RVal=True          !SELF.CurrentPage changed
        SELF.CurrentPage=JumpPage
    END
    POST(EVENT:CloseWindow)
OF ?CancelButton
    POST(EVENT:CloseWindow)
END
END
END
CLOSE(JumpWin); RETURN RVal
```

**AskPrintPages (prompt for pages to print)**

**AskPrintPages, VIRTUAL, PROTECTED, PROC**

The **AskPrintPages** method prompts the end user for the number(s) of the pages to print from the previewed report.

Implementation: The PrintPreviewClass.TakeAccepted method calls the AskPrintPages method and returns TRUE (1) when completed or FALSE (0) if the user presses the cancel button. The AskPrintPages method displays a dialog that prompts for the page numbers to print.

Return Data Type: BYTE

Example:

```
!Virtual implementation of AskThumbnails
PrintPreviewClass.AskPrintPages PROCEDURE
Preserve LIKE(PrintPreviewClass.PagesToPrint),AUTO
Window WINDOW('Pages to Print'),AT(,,260,37),CENTER,SYSTEM,GRAY
  PROMPT('&Pages to Print:'),AT(4,8),USE(?Prompt)
  ENTRY(@s255),AT(56,4,200,11),USE(SELF.PagesToPrint, , ?PagesToPrint)
  BUTTON('&Reset'),AT(116,20,45,14),USE(?Reset)
  BUTTON('&Ok'),AT(164,20,45,14),USE(?Ok),DEFAULT
  BUTTON('&Cancel'),AT(212,20,45,14),USE(?Cancel),STD(STD:Close)
END
RVal BYTE(False)
CODE
Preserve = SELF.PagesToPrint
OPEN(Window)
ACCEPT
CASE EVENT()
OF EVENT:Accepted
CASE ACCEPTED()
OF ?Cancel
  SELF.PagesToPrint = Preserve
  POST(EVENT:CloseWindow)
OF ?Ok
  RVal = True
  POST(EVENT:CloseWindow)
OF ?Reset
  SELF.SetDefaultPages
  SELECT(?PagesToPrint)
END
```

```
OF EVENT:OpenWindow
  ! INIMgr code for FETCHing window settings
OF EVENT:CloseWindow
  ! INIMgr code for UPDATEing window settings
END
END
CLOSE(Window)
RETURN Rval
```

**AskThumbnails (prompt for new thumbnail configuration)****AskThumbnails, VIRTUAL, PROTECTED**

The **AskThumbnails** method prompts the end user for the number of pages to tile across and down the preview window.

Implementation: The PrintPreviewClass.Display method calls the AskThumbnails method. The AskThumbnails method displays a dialog that prompts for the number of thumbnails to display horizontally, and the number of thumbnails to display vertically.

Example:

```
!Virtual implementation of AskThumbnails
! a slightly simplified version with no translator...
PrintPreviewClass.AskThumbnails PROCEDURE

SelectWindow WINDOW('Pages Displayed'),AT(,,141,64),GRAY,DOUBLE
    GROUP('Across'),AT(7,10,62,32),BOXED
        SPIN(@N2),AT(13,22,15),USE(SELF.PagesAcross,,?PagesAcross),RANGE(1,10)
    END
    GROUP('Down'),AT(72,10,62,32),BOXED
        SPIN(@N2),AT(79,22,15),USE(SELF.PagesDown,,?PagesDown),RANGE(1,10)
    END
    BUTTON('OK'),AT(98,47,40,14),KEY(EnterKey),USE(?OK)
    END

CODE
OPEN(SelectWindow)
ACCEPT
    CASE EVENT()
    OF EVENT:Accepted
        CASE FIELD()
        OF ?OK
            IF SELF.PagesAcross*SELF.PagesDown>RECORDS(SELF.ImageQueue)
                SELECT(?PagesAcross)
            ELSE
                POST(EVENT:CloseWindow)
            END
        END
    END
END
CLOSE(SelectWindow)
```

## DeletelImageQueue (remove non-selected pages)

**DeletelImageQueue**(*page* ), VIRTUAL, PROC

---

**DeletelImageQueue** Removes a page number from the ImageQueue.

*page* An integer constant, variable, EQUATE, or expression containing the page number to delete.

The **DeletelImageQueue** method removes records from the ImageQueue, and the associated image file, which have not been selected for printing.

Implementation: The SyncImageQueue method calls the DeletelImageQueue method. The value contained in the PagesToPrint property determines which records and images are deleted.

Return Data Type: BYTE

Example:

```
PrintPreviewClass.SyncImageQueue PROCEDURE
i LONG,AUTO

CODE
LOOP i = RECORDS(SELF.ImageQueue) TO 1 BY -1
  IF ~SELF.InPageList(i)
    SELF.DeleteImageQueue(i)
  END
END
```

See Also: PagesToPrint,DeletelImageQueue

**Display (preview the report)**

**Display**( [*zoom*] [, *page*] [, *across*] [, *down*] ), **VIRTUAL, PROC**

---

<b>Display</b>	Displays the report image metafiles.
<i>zoom</i>	An integer constant, variable, EQUATE, or expression containing the initial zoom factor for the print preview display. If omitted, the Display method uses the default zoom factor in the ABREPORT.TRN file.
<i>page</i>	An integer constant, variable, EQUATE, or expression containing the initial page number to display. If omitted, <i>page</i> defaults to one (1).
<i>across</i>	An integer constant, variable, EQUATE, or expression containing the number of horizontal thumbnails for the initial print preview display. If omitted, <i>across</i> defaults to one (1).
<i>down</i>	An integer constant, variable, EQUATE, or expression containing the number of vertical thumbnails for the initial print preview display. If omitted, <i>down</i> defaults to one (1).

The **Display** method displays the report image metafiles and returns a value indicating whether or not to print them. A return value of one (1 or True) indicates the end user asked to print the report; a return value of zero (0 or False) indicates the end user did not ask to print the report.

The Display method is the print preview engine. It manages the print preview, providing navigation, zoom, thumbnail configuration, plus the option to immediately print the report.

Implementation: The Display method declares the preview WINDOW, then calls the WindowManager.Ask method to display the preview WINDOW and process its events. EQUATEs for the *zoom* parameter are declared in ABREPORT.INC:

```
NoZoom           EQUATE(-2)
PageWidth        EQUATE(-1)
```

In addition to the EQUATE values, you may specify any integer zoom factor, such as 50 (50% zoom) or 200 (200% zoom).

Return Data Type: **BYTE**

Example:

```
IF ReportCompleted           !if report was not cancelled
  ENDPAGE(report)           !force final page overflow
  IF PrtPrev.Display()       !preview the report on-line
    report{PROP:FlushPreview} = True !and print it if user asked to
  END
END
```

See Also: **WindowManager.Ask**

**Init (initialize the PrintPreviewClass object)**

**Init** (*image queue* ), VIRTUAL

---

<b>Init</b>	Initializes the PrintPreviewClass object.
<i>image queue</i>	The label of the QUEUE containing the filenames of the report image metafiles. See <i>PREVIEW</i> in the <i>Language Reference</i> for more information on report image metafiles.

The **Init** method initializes the PrintPreviewClass object.

Implementation: The PrintPreviewClass.Init method instantiates a PopupClass object for the PrintPreviewClass object, using the menu text defined in ABREPORT.TRN.

The image queue parameter names a QUEUE with the same structure as the PreviewQueue declared in \ABREPORT.INC as follows:

```
PreviewQueue    QUEUE,TYPE
Filename        STRING(128)
                END
```

Example:

```
PrintPreviewQueue PreviewQueue    !declare report image queue
PrtPrev          PrintPreviewClass !declare PrtPrev object
CODE
PrtPrev.Init(PrintPreviewQueue)    !initialize PrtPrev object
!program code
PrtPrev.Kill                        !shut down PrtPrev object
```

## InPageList (check page number)

**InPageList( *page* )**

---

**InPageList** Evaluates page against value(s) in PagesToPrint.

*page* An integer constant, variable, EQUATE, or expression containing the page number to check.

The **InPageList** method evaluates a page number against the value(s) contained in the PagesToPrint property, and returns TRUE (1) if the page is in PagesToPrint or FALSE (0) if it is not.

Implementation: The PageManagerClass.Draw (which is PRIVATE) and SyncImageQueue methods call the InPageList method to verify report pages for inclusion in the preview window and the printed report respectively.

Return Data Type: BYTE

Example:

```
PrintPreviewClass.SyncImageQueue PROCEDURE
i LONG,AUTO
CODE
LOOP i = RECORDS(SELF.ImageQueue) TO 1 BY -1
  IF ~SELF.InPageList(i)
    SELF.DeleteImageQueue(i)
  END
END
```

See Also: PagesToPrint

**Kill (shut down the PrintPreviewClass object)****Kill, VIRTUAL, PROC**

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code. Kill returns a value to indicate the status of the shut down.

Implementation: The Kill method calls the WindowManager.Kill method and returns Level:Benign to indicate a normal shut down. Return value EQUATEs are declared in ABERROR.INC.

Return Data Type: BYTE

Example:

```
PrintPreviewQueue PreviewQueue      !declare report image queue
PrtPrev          PrintPreviewClass  !declare PrtPrev object
CODE
PrtPrev.Init(PrintPreviewQueue)     !initialize PrtPrev object
!program code
PrtPrev.Kill                        !shut down PrtPrev object
```

See Also: WindowManager.Kill

## Open (prepare preview window for display)

### Open, VIRTUAL

The **Open** method prepares the PrintPreviewClass window for initial display. It is designed to execute on window opening events such as EVENT:OpenWindow and EVENT:GainFocus.

Implementation: The Open method sets the window's initial size and position, enables and disables controls as needed, and sets up the specified zoom configuration.

The WindowManager.TakeWindowEvent method calls the Open method.

Example:

```
ThisWindow.TakeWindowEvent PROCEDURE
CODE
CASE EVENT()
OF EVENT:OpenWindow
  IF ~BAND(SELF.Inited,1)
    SELF.Open
  END
OF EVENT:GainFocus
  IF BAND(SELF.Inited,1)
    SELF.Reset
  ELSE
    SELF.Open
  END
END
RETURN Level:Benign
```

See Also: WindowManager.TakeWindowEvent

## SetINIManager (save and restore window coordinates)

**SetINIManager**( *INI manager* )

---

**SetINIManager** Enables save and restore of preview window position and size between computing sessions.

*INI manager* The label of the INIClass object that saves and restores window coordinates. See *INI Class* for more information.

The **SetINIManager** method names an INIClass object to save and restore window coordinates between computing sessions.

Implementation: The Open method uses the *INI manager* to restore the window's initial size and position. The TakeEvent method uses the *INI manager* to save the window's size and position.

Example:

```
ThisWindow.Init PROCEDURE()  
  CODE  
  !procedure code  
  ThisWindow.Init(Process,report,Previewer)  
  Previewer.SetINIManager(INIMgr)
```

See Also: Open, TakeEvent

## SetPosition (set initial preview window coordinates)

**SetPosition**( [*x*] [*y*] [,*width*] [,*height*] )

---

<b>SetPosition</b>	Sets the initial position and size of the print preview window.
<i>x</i>	An integer constant, variable, EQUATE, or expression containing the initial horizontal position of the print preview window. If omitted, the print preview window opens to the default Windows position.
<i>y</i>	An integer constant, variable, EQUATE, or expression containing the initial vertical position of the print preview window. If omitted, the print preview window opens to the default Windows position.
<i>width</i>	An integer constant, variable, EQUATE, or expression containing the initial width of the print preview window. If omitted, the print preview window opens to its default width.
<i>height</i>	An integer constant, variable, EQUATE, or expression containing the initial height of the print preview window. If omitted, the print preview window opens to its default height.

The **SetPosition** method sets the initial position and size of the print preview window.

Implementation: The SetPosition method sets the WindowPosSet and WindowSizeSet properties.

The Display method definition determines the default width and height of the print preview window.

Example:

```
PrtPrev.SetPosition(1,1,300,250)      !set initial position and size
PrtPrev.SetPosition(1,1)             !set initial position only
PrtPrev.SetPosition(, ,300,250)      !set initial size only
```

See Also: WindowPosSet, WindowSizeSet

## **SetZoomPercentile (set user or standard zoom factor)**

**SetZoomPercentile**( *zoom factor* )

---

**SetZoomPercentile** Sets the ZoomIndex and UserPercentile properties.

*zoom factor* An integer constant, variable, EQUATE, or expression indicating the zoom factor to apply.

The **SetZoomPercentile** method sets the ZoomIndex property and the UserPercentile property.

Implementation: The SetZoomPercentile method assumes the AllowUserZoom property is True. If the *zoom factor* equals a defined ZoomIndex choice, SetZoomPercentile sets the ZoomIndex property to that choice and sets the UserPercentile property to zero. If the *zoom factor* does not equal a defined ZoomIndex choice, SetZoomPercentile sets the UserPercentile property to the *zoom factor* and sets the ZoomIndex property to zero.

Example:

```
ThisWindow.Init PROCEDURE()  
  CODE  
  !procedure code  
  ThisWindow.Init(Process,report,Previewer)  
  Previewer.SetZoomPercentile(120)
```

See Also: AllowUserZoom, UserPercentile, ZoomIndex

## SetDefaultPages (set the default pages to print)

### SetDefaultPages, VIRTUAL

The **SetDefaultPages** method sets the initial value of the PagesToPrint property. The initial value is 1-*n*, where *n* is equal to the total number of pages in the report.

Implementation: The Display and AskPrintPreview methods call the SetDefaultPages method.

Example:

```
!Virtual implementation of SetDefaultPages method
PrintPreviewClass.SetDefaultPages PROCEDURE
CODE
  SELF.PagesToPrint = '1-' & RECORDS(SELF.ImageQueue)
```

See Also: PagesToPrint

## SyncImageQueue (sync image queue with PagesToPrint)

### SyncImageQueue, VIRTUAL

The **SyncImageQueue** method synchronizes the image queue with the contents of PagesToPrint to ensure that only the specified pages are sent to the printer.

Implementation: The Display method calls the SyncImageQueue method. The value contained in the PagesToPrint property determines which pages are printed.

Example:

```
PrintPreviewClass.Display PROCEDURE
! Window declaration
! executable Display code
  IF SELF.PrintOk
    SELF.SyncImageQueue
  END
  RETURN SELF.PrintOK
```

See Also: PagesToPrint

## TakeAccepted (process EVENT:Accepted events:PrintPreviewClass)

### TakeAccepted, VIRTUAL, PROC

The **TakeAccepted** method processes EVENT:Accepted events for all the controls on the preview window, then returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeAccepted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeEvent method calls the TakeAccepted method. The TakeAccepted method calls the WindowManager.TakeAccepted method, then processes EVENT:Accepted events for all the controls on the preview window, including zoom controls, print button, navigation controls, thumbnail configuration controls, etc.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;    RVal = SELF.TakeRejected()
    OF EVENT:Selected;    RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;   RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also: TakeEvent, WindowManager.TakeEvent

## TakeEvent (process all events:PrintPreviewClass)

**TakeEvent, VIRTUAL, PROC**

The **TakeEvent** method processes all preview window events and returns a value indicating whether ACCEPT loop processing is complete and should stop. TakeEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The Ask method calls the TakeEvent method. The TakeEvent method calls the WindowManager.TakeEvent method, then processes EVENT:CloseWindow, EVENT:Sized and EVENT:AlertKey events for the preview window.

Return Data Type: BYTE

Example:

```
WindowManager.Ask PROCEDURE
CODE
IF SELF.Dead THEN RETURN .
CLEAR(SELF.LastInsertedPosition)
ACCEPT
CASE SELF.TakeEvent()
OF Level:Fatal
BREAK
OF Level:Notify
CYCLE ! Used for 'short-stopping' certain events
END
END
```

See Also: WindowManager.Ask

## TakeFieldEvent (a virtual to process field events:PrintPreviewClass)

### TakeFieldEvent, VIRTUAL, PROC

The **TakeFieldEvent** method is a virtual placeholder to process all field-specific/control-specific events for the window. It returns a value indicating whether window process is complete and should stop. TakeFieldEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeEvent method calls the TakeFieldEvent method. The TakeFieldEvent method processes EVENT:NewSelection events for the preview window SPIN controls.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;    RVal = SELF.TakeRejected()
    OF EVENT:Selected;    RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;   RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also: Ask

**TakeWindowEvent (process non-field events:PrintPreviewClass)****TakeWindowEvent, VIRTUAL, PROC**

The **TakeWindowEvent** method processes all non-field events for the preview window and returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeWindowEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeEvent method calls the TakeWindowEvent method. The TakeWindowEvent method calls the WindowManager.TakeWindowEvent method for all events except EVENT:GainFocus.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;    RVal = SELF.TakeRejected()
    OF EVENT:Selected;    RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;   RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also: TakeEvent



# ProcessClass

## ProcessClass Overview

The ProcessClass is a ViewManager with a progress window.

## ProcessClass Concepts

The ProcessClass lets you "batch" process a VIEW, applying sort orders, range limits, and filters as needed to process only the specific result set in the specific sequence you require; plus the ProcessClass supplies appropriate (configurable) visual feedback to the end user on the progress of the batch process.

## ProcessClass Relationship to Other Application Builder Classes

The ProcessClass is derived from the ViewManager, plus it relies on many of the other Application Builder Classes to accomplish its tasks. Therefore, if your program instantiates the ProcessClass, it must also instantiate these other classes. Much of this is automatic when you INCLUDE the ProcessClass header (ABREPORT.INC) in your program's data section. See the Conceptual Example.

The ReportManager uses the ProcessClass to process report data and provide appropriate visual feedback to the end user on the progress of the report.

## ProcessClass ABC Template Implementation

The ABC Templates automatically include all the classes necessary to support the batch processes (Process procedures and Report procedures) specified in your application.

The templates *derive* a class from the ProcessClass for *each* batch process (Process Procedures and Report Procedures) in the application. The derived classes are called ThisProcess and ThisReport. These derived ProcessClass objects support all the functionality specified in the Process or Report procedure template.

The derived ProcessClass is local to the procedure, is specific to a single process and relies on the global file-specific RelationManager and FileManager objects for the processed files.

## ProcessClass Source Files

The ProcessClass source code is installed by default to the Clarion \LIBSRC. The ProcessClass source code and their respective components are contained in:

ABREPORT.INC  
ABREPORT.CLW

ProcessClass declarations  
ProcessClass method definitions

## ProcessClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a ProcessClass object and related objects. This example processes selected records in a file, updates them, and displays a window with a progress bar to show the progress of the process.

```

PROGRAM
  INCLUDE('ABWINDOW.INC')           !declare WindowManager Class
  INCLUDE('ABREPORT.INC')          !declare Process Class
MAP
END
Customer FILE,DRIVER('TOPSPEED'),PRE(CUS),THREAD !declare Customer file
BYNUMBER KEY(CUS:CUSTNO),NOCASE,OPT,PRIMARY
Record RECORD,PRE()
CUSTNO LONG
Name STRING(30)
State STRING(2)
  END
  END
CusView VIEW(Customer)             !declare VIEW for process
  END
Access:Customer CLASS(FileManager) !declare Access:Customer object
Init PROCEDURE
  END
Relate:Customer CLASS(RelationManager)!declare Relate:Customer object
Init PROCEDURE
  END
ThisWindow CLASS(ReportManager) !declare ThisWindow object
Init PROCEDURE(),BYTE,PROC,VIRTUAL
Kill PROCEDURE(),BYTE,PROC,VIRTUAL
  END
ThisProcess CLASS(ProcessClass) !declare ThisProcess object
TakeRecord PROCEDURE(),BYTE,PROC,VIRTUAL
  END
ProgressMgr StepLongClass          !declare ProgressMgr object
GlobalErrors ErrorClass           !declare GlobalErrors object
VCRRequest LONG(0),THREAD
Thermometer BYTE                  !declare PROGRESS variable
ProgressWindow WINDOW('Progress...'),AT(,142,59),CENTER,TIMER(1),GRAY,DOUBLE
  PROGRESS,USE(Thermometer),AT(15,15,111,12),RANGE(0,100)
  STRING(' '),AT(0,3,141,10),USE(?UserString),CENTER
  STRING(' '),AT(0,30,141,10),USE(?PctText),CENTER
  BUTTON('Cancel'),AT(45,42),USE(?Cancel)
  END
CODE
ThisWindow.Run()                  !run the Process procedure

```

```

ThisWindow.Init PROCEDURE()          !initialize things
ReturnValue      BYTE,AUTO
CODE
GlobalErrors.Init                    !initialize GlobalErrors object
Relate:Customer.Init                 !initialize Relate:Customer object
ReturnValue = PARENT.Init()          !call base class init
SELF.FirstField = ?Thermometer       !set FirstField for ThisWindow
SELF.VCRRequest &= VCRRequest       !VCRRequest not used
SELF.Errors &= GlobalErrors          !set errorhandler for ThisWindow
Relate:Customer.Open                 !Open Customer and related files
OPEN(ProgressWindow)                !open the window
SELF.Opened=True                     !set Opened flag for ThisWindow
ProgressMgr.Init(ScrollSort:AllowNumeric) !initialize ProgressMgr object
!init ThisProcess by naming its VIEW, RelationManager,ProgressMgr & progress variables
ThisProcess.Init(CusView,Relate:Customer,?PctText,Thermometer,ProgressMgr,CUS:CUSTNO)
ThisProcess.AddSortOrder(CUS:BYNUMBER) !set the process sort order
SELF.Init(ThisProcess)               !process specific initialization
SELF.AddItem(?Cancel,RequestCancelled) !register Cancel with ThisWindow
SELF.SetAlerts()                     !alert keys for ThisWindow
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()          !shut down things
ReturnValue      BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()          !call base class shut down
Relate:Customer.Close               !close Customer and related files
Relate:Customer.Kill                !shut down Relate:Customer object
GlobalErrors.Kill                   !shut down GlobalErrors object
RETURN ReturnValue

ThisProcess.TakeRecord PROCEDURE()   !action for each record processed
ReturnValue      BYTE,AUTO
CODE
IF NOT CUS:State                     !if State is blank
  CUS:State = 'FL'                   ! set it to 'FL'
END
ReturnValue = PARENT.TakeRecord()     !call base class for each record
PUT(CusView)                         !write the updated record
IF ERRORCODE()                       !if write failed
  ThisWindow.Response = RequestCompleted ! shut down process
  ReturnValue = Level:Fatal           !Use IF Relate:Customer.Update()
END                                   !to apply RI constraints to
RETURN ReturnValue                     ! Customer and related files.

```

```
Access:Customer.Init PROCEDURE
CODE
PARENT.Init(Customer,GlobalErrors)
SELF.FileNameValue = 'Customer'
SELF.Buffer &= CUS:Record
SELF.LazyOpen = False
SELF.AddKey(CUS:BYNUMBER,'CUS:BYNUMBER',0)
```

```
Relate:Customer.Init PROCEDURE
CODE
Access:Customer.Init
PARENT.Init(Access:Customer,1)
```

## ProcessClass Properties

The ProcessClass inherits all the properties of the ViewManager class from which it is derived. See *ViewManager Properties* for more information.

In addition to the inherited properties, the ProcessClass contains the following properties:

### **CaseSensitiveValue (case sensitive flag)**

**CaseSensitiveValue**      **BYTE**

The **CaseSensitiveValue** property is set to zero (0 or False) when the key for the processed FILE is a case insensitive key, i.e. the NOCASE attribute is on the key definition.

### **Percentile (portion of process completed)**

**Percentile**              **&BYTE, PROTECTED**

The **Percentile** property is a reference to a variable whose contents indicates how much of the process is completed. The ProcessClass periodically updates the Percentile property so it can be the USE variable for a PROGRESS control.

The Init method initializes the Percentile property. See the *Conceptual Example*.

See Also:              Init

### **PText (progress control number)**

**PText**    **SIGNED**

The **PText** property contains the control number of a text based Window control such as a STRING or PROMPT. The ProcessClass uses this control to provide visual feedback to the end user.

The Init method initializes the PText property. See the *Conceptual Example*.

This property is PROTECTED, therefore, it can only be referenced by a ProcessClass method, or a method in a class derived from ProcessClass.

See Also:            Init

### **RecordsProcessed (number of elements processed)**

**RecordsProcessed**    **LONG**

The **RecordsProcessed** property contains the number of elements processed so far. The ProcessClass uses this property to calculate how much of the process is completed.

### **RecordsToProcess (number of elements to process)**

**RecordsToProcess**    **LONG**

The **RecordsToProcess** property contains the total number of elements to process. The ProcessClass uses this property to calculate how much of the process is completed.

## ProcessClass Methods

The ProcessClass inherits all the methods of the ViewManager class from which it is derived. See *ViewManager Properties* for more information.

### **ProcessClass Functional Organization--Expected Use**

As an aid to understanding the ProcessClass, it is useful to organize its methods into two categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the ProcessClass methods.

#### **Non-Virtual Methods**

##### **Housekeeping (one-time) Use:**

Init	initialize the ProcessClass object
AddRange <sub>i</sub>	add a range limit to the active sort order
AddSortOrder <sub>i</sub>	add a sort order
AppendOrder <sub>i</sub>	refine the active sort order
SetProgressLimits	calibrate the StepClass progress monitor
Kill <sub>v</sub>	shut down the ProcessClass object

##### **Mainstream Use:**

Open <sub>i</sub>	open the view
Next <sub>v</sub>	get the next result set element
Previous <sub>i</sub>	get the previous result set element
PrimeRecord <sub>i</sub>	prepare a record for adding
ValidateRecord <sub>i</sub>	validate the current result set element
SetFilter <sub>i</sub>	specify a filter for the active sort order
SetSort <sub>v</sub>	set the active sort order
ApplyFilter <sub>i</sub>	range limit and filter the result set
ApplyOrder <sub>i</sub>	sort the result set
ApplyRange <sub>i</sub>	conditionally range limit and filter the result set
Close <sub>i</sub>	close the view

##### **Occasional Use:**

GetFreeElementName <sub>i</sub>	return the free element field name
Reset <sub>v</sub>	reposition to the first result set element
SetOrder <sub>i</sub>	replace the active sort order

<sub>i</sub> These methods are inherited from the ViewManager class.

<sub>v</sub> These methods are also Virtual.

---

**Virtual Methods**

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Next	get the next result set element
Previous	get the previous result set element
Reset	reposition to the first result set element
SetSort	set the active sort order
ValidateRecord	validate the current result set element
Kill	shut down the ProcessClass object

| These methods are inherited from the ViewManager class.

**Init (initialize the ProcessClass object)**

```
Init( view, relationmanager [, progress txt] [, progress pct] | [, total records] | )
                                     |, stepclass, free element |
```

---

<b>Init</b>	Initializes the ProcessClass object.
<i>view</i>	The label of the VIEW to process.
<i>relationmanager</i>	The label of the <i>view</i> 's primary file RelationManager object.
<i>progress txt</i>	A numeric constant, variable, EQUATE, or expression that contains the control number of a text-based Window control. The ProcessClass uses this control to provide textual feedback to the end user. If omitted, <i>progress txt</i> defaults to zero (0) and the ProcessClass provides no textual feedback.
<i>progress pct</i>	The label of a BYTE variable whose contents indicates what percent of the process is completed. The ProcessClass periodically updates <i>progress pct</i> so it can be the USE variable for a PROGRESS control. If omitted, the ProcessClass provides no numeric feedback.
<i>total records</i>	A numeric constant, variable, EQUATE, or expression that contains the estimated number of records to process. The ProcessClass uses this value to calculate how much of the process is completed. You should use this parameter when you can easily estimate the number of records to be processed, that is, when the process is not dynamically filtered. If omitted, <i>totalrecords</i> defaults to zero.
<i>stepclass</i>	The label of a StepClass object to monitor the progress of the process. The ProcessClass uses this object to determine how much of the process is completed. You should use this parameter when you cannot easily estimate the number of records to be processed, that is, when the process is dynamically filtered.
<i>free element</i>	The label of the <i>view</i> 's free element field. The <i>stepclass</i> uses this field to determine how much of the process is completed. See <i>StepClass Methods--GetPercentile</i> for more information.

The **Init** method initializes the ProcessClass object. If you supply *total records* to process, the ProcessClass object calculates the progress of the process as a function of *total records* and the number of records processed so far. Otherwise, the ProcessClass object relies on the *stepclass* to calculate the progress of the process. See *StepClass Methods--GetPercentile* for more information.

Implementation: The Init method assigns *progress txt* to the PText property, reference assigns *progress pct* to the Percentile property, and assigns *total records* to the RecordsToProcess property. The Init method calls the ViewManager Init method.

Example:

```
!initialize the ProcessClass object
Process.Init( Process:View, |      !set the VIEW
               Relate:Client, |    !set the primary file RelationManager
               ?PctText,      |    !set the Window control for text messages
               PctDone,       |    !set the PROGRESS USE variable
               ProgressMgr,   |    !set StepClass object to monitor progress
               CLI:Name)      |    !set StepClass free element to monitor
```

See Also:           Percentile, PText, RecordsToProcess, ViewManager.Init

## Kill (shut down the ProcessClass object)

### Kill, VIRTUAL

The **Kill** method shuts down the ProcessClass object by freeing any memory allocated during the life of the object and executing any other required termination code.

Implementation: The Kill method calls the ViewManager.Kill method.

Example:

```
!initialize the ProcessClass object
Process.Init( Process:View, | !set the VIEW
    Relate:Client, | !set the primary file RelationManager
    ?PctText, | !set the Window control for text messages
    PctDone, | !set the PROGRESS USE variable
    ProgressMgr, | !set StepClass object to monitor progress
    CLI:Name) | !set StepClass free element to monitor
!procedure code
Process.Kill |!shut down the ProcessClass object
```

See Also: ViewManager.Kill

**Next (get next element)****Next**( [*process records* ] ), **VIRTUAL****Next** Gets the next element in the result set.

*process records* A boolean constant, variable, EQUATE, or expression that tells the ProcessClass object whether to update its progress indicators. A zero (0) value does not update the progress indicators; any other value does update the indicators. If omitted, *process records* defaults to one (1).

The **Next** method gets the next element in the result set and returns a value indicating its success or failure.

Implementation: The Next method calls the ViewManager.Next method. The ProcessClass.Next method updates both the RecordsProcessed property and the Percentile property.

Return Data Type: BYTE

Example:

```

ACCEPT
CASE EVENT()
OF Event:OpenWindow
  Process.Reset                !position to first record
  IF Process.Next()           !get first record
    POST(Event:CloseWindow)   !if no records, shut down
  CYCLE
END
OF Event:Timer                !process records with timer
  StartOfCycle=Process.RecordsProcessed
  LOOP WHILE Process.RecordsProcessed-StartOfCycle<RecordsPerCycle
    CASE Process.Next()       !get next record
    OF Level:Notify           !if end of file
      MESSAGE('Process Completed') ! tell end user
      POST(EVENT:CloseWindow)    ! and shut down
      BREAK
    OF Level:Fatal            !if fatal error
      POST(EVENT:CloseWindow)    ! shut down
      BREAK
    END
  END
END
END
END
END

```

See Also: Percentile, RecordsProcessed, ViewManager.Next

**Reset (position to the first element)****Reset, VIRTUAL**

The **Reset** method positions the process to the first element in the result set and resets the progress indicators.

Implementation: The Reset method resets the RecordsProcessed property to zero (0), conditionally calls the SetProgressLimits method, then calls the ViewManager.Reset method.

Example:

```

CASE EVENT( )
OF Event:OpenWindow
  Process.Reset           !position to first record
  IF Process.Next()      !get first record
    POST(Event:CloseWindow) !if no records, shut down
  CYCLE
END

```

See Also: SetProgressLimits, ViewManager.Reset

**SetProgressLimits (calibrate the progress monitor)****SetProgressLimits**

The **SetProgressLimits** method supplies the upper and lower boundaries of the result set--considering the active sort order, range limits, and filters--to the StepClass object that monitors the progress of the process.

The Init method specifies the StepClass object.

Implementation: The SetProgressLimits method assumes a StepClass object is specified. The Reset method conditionally calls the SetProgressLimits method. The SetProgressLimits method calls the StepClass.SetLimits method.

Example:

```

MyProcessClass.Reset PROCEDURE           !prepare to process the records
CODE
  SELF.RecordsProcessed = 0               !set RecordsProcessed to 0
  SELF.SetProgressLimits                 !set StepClass boundaries based
                                         ! on actual data processed
  PARENT.Reset                          !call ViewManager.Reset to
                                         !position to the first record

```

See Also: Init, Reset, StepClass.SetLimit

## TakeLocate (a virtual to process each filter)

### TakeLocate, VIRTUAL

The **TakeLocate** method does this.

Implementation: The ReportManager.TakeAccepted method calls the TakeLocate method for each report record.

Example:

```
ProcessClass.TakeLocate PROCEDURE
CODE
  IF ~SELF.Query&=NULL AND SELF.Query.Ask()
    SELF.SetFilter(SELF.Query.GetFilter())
  END
```

## TakeRecord (a virtual to process each report record)

### TakeRecord, VIRTUAL, PROC

The **TakeRecord** method is a virtual placeholder to process each record in the result set. It returns a value indicating whether processing should continue or should stop. TakeRecord returns Level:Benign to indicate processing should continue normally; it returns Level:Notify to indicate processing is completed and should stop.

Implementation: The ReportManager.TakeWindowEvent method calls the TakeRecord method for each report record. For a report, the TakeRecord method typically implements any DETAIL specific filters and PRINTs the unfiltered DETAILS for the ReportManager. For a process, the TakeRecord method typically implements any needed record action for the Process.

Return Data Type: BYTE

Example:

```
ThisWindow.TakeRecord PROCEDURE()
CODE
  IF ORD:Date = TODAY()
    PRINT(RPT:detail)
  END
  RETURN Level:Benign
```

See Also: ReportManager.TakeWindowEvent

# QueryClass

## QueryClass Overview

The QueryClass provides support for ad hoc queries against Clarion VIEWS. The query support includes a flexible user input dialog, a broad variety of search capabilities, and seamless integration with the BrowseClass. The QueryClass provides the following features:

- flexible user input dialog
- runtime setup of queryable fields
- queries against calculated fields (e.g., Qty\*Price>100)
- case sensitive or insensitive searches
- "begins with" searches
- "contains anywhere" searches
- exclusive searches (not equal, greater than, less than)
- inclusive searches (equal, greater than or equal, less than or equal)
- ranged searches (greater than low value AND less than high value)
- persistent queries for stepwise refinement of queries

## QueryClass Concepts

Use the AddItem method to define a standard user input dialog at runtime. Or create a custom dialog to plug into your QueryClass object. Use the Ask method to solicit end user query input or use the SetLimit method to programmatically set query search values. Finally, use the GetFilter method to build the filter expression to apply to your VIEW. You can apply the resulting filter with the ViewManager.SetFilter method, or directly with the PROP:Filter property.

## QueryClass Relationship to Other Application Builder Classes

The classes derived from the QueryClass are optionally used by the BrowseClass. Therefore, if your BrowseClass object uses a QueryClass object, it must instantiate the QueryClass object.

The BrowseClass automatically provides a default query dialog that solicits end user search values for each field displayed in the browse list. See the Conceptual Example.

## QueryClass ABC Template Implementation

The ABC Templates do not instantiate the QueryClass object independently. The templates instantiate the derived QueryFormClass instead.

**Tip:** Use the **BrowseQBEBUTTON** control template to add a QueryFormClass object to your template generated BrowseBoxes.

## QueryClass Source Files

The QueryClass source code is installed by default to the Clarion \LIBSRC folder. The specific QueryClass files and their respective components are:

ABQUERY.INC	QueryClass declarations
ABQUERY.CLW	QueryClass method definitions

## QueryClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a QueryClass object and related objects. The example plugs a QueryClass into a BrowseClass object. The QueryClass object simply filters on the current record.

Note that the WindowManager and BrowseClass objects internally handle the normal events surrounding the query.

```

PROGRAM

_AB CDllMode_ EQUATE(0)
_AB CLinkMode_ EQUATE(1)
INCLUDE( 'ABWINDOW.INC' )
INCLUDE( 'ABBROWSE.INC' )
INCLUDE( 'ABQUERY.INC' )

MAP
END

GlobalErrors  ErrorClass
Access:Customer CLASS(FileManager)
Init          PROCEDURE
              END

Relate:Customer CLASS(RelationManager)
Init          PROCEDURE
Kill          PROCEDURE,VIRTUAL
              END

GlobalRequest  BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD

Customer      FILE,DRIVER( 'TOPSPEED' ),PRE(CUS),CREATE,THREAD
CustomerIDKey KEY(CUS:ID),NOCASE,OPT,PRIMARY
NameKey       KEY(CUS:LastName),NOCASE,OPT
Record        RECORD,PRE()
ID            LONG
LastName      STRING(20)
FirstName     STRING(15)
City          STRING(20)
State         STRING(2)
ZIP           STRING(10)
              END
              END

```



```
BRW1.Q &= CustQ
BRW1.AddSortOrder(,CUS:NameKey)
BRW1.AddField(CUS:LastName,BRW1.Q.CUS:LastName)
BRW1.AddField(CUS:FirstName,BRW1.Q.CUS:FirstName)
BRW1.AddField(CUS:ZIP,BRW1.Q.CUS:ZIP)
BRW1.QueryControl = ?Query           !register Query button w/ BRW1
BRW1.UpdateQuery(Query)              !make each BRW1 field queryable
Query.AddItem('CUS:State','')       !make State field queryable too
SELF.SetAlerts()
RETURN ReturnValue
```

```
ThisWindow.Kill PROCEDURE()
ReturnValue BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
Relate:Customer.Close
RETURN ReturnValue
```

```
Access:Customer.Init PROCEDURE
CODE
PARENT.Init(Customer,GlobalErrors)
SELF.FileNameValue = 'Customer'
SELF.Buffer &= CUS:Record
SELF.Create = 1
SELF.AddKey(CUS:CustomerIDKey,'CUS:CustomerIDKey',1)
SELF.AddKey(CUS:NameKey,'CUS:NameKey',0)
```

```
Relate:Customer.Init PROCEDURE
CODE
Access:Customer.Init
PARENT.Init(Access:Customer,1)
```

```
Relate:Customer.Kill PROCEDURE
CODE
Access:Customer.Kill
PARENT.Kill
```

## QueryClass Properties

The QueryClass contains the following properties:

### QKCurrentQuery ( popup menu choice )

**QKCurrentQuery**      **CSTRING(100)**

The **QKCurrentQuery** property holds the value of the popup menu item if QuickQBE support is enabled.

### QKIcon ( icon for popup submenu )

**QKIcon**      **CSTRING(255)**

The **QKIcon** property holds the full pathname of the icon file to be used in the QuickQBE submenu items.

### QKMenuIcon ( icon for popup menu )

**QKMenuIcon**      **CSTRING(255)**

The **QKMenuIcon** property holds the full pathname of the icon file to be in the popup menu if QuickQBE has been enabled.

**QKSupport ( quickqbe flag)****QKSupport**     **BYTE**

The **QKSupport** property indicates that QuickQBE support is enabled.

**Window ( browse window:QueryClass )****Window**     **&Window**

The **Window** property is a reference to the QBE dialog window.

## QueryClass Methods

The QueryClass contains the following methods:

### QueryClass Functional Organization--Expected Use

As an aid to understanding the QueryClass, it is useful to organize its various methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the QueryClass methods.

#### Non-Virtual Methods

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the QueryClass object
AddItem	add a field to query
Killv	shut down the QueryClass object

##### **Mainstream Use:**

Askv	a virtual to accept query criteria
GetFilter	return filter expression

##### **Occasional Use:**

Reset	reset the QueryClass object
GetLimit	get searchvalues
SetLimit	set search values

These methods are also Virtual.

#### Virtual Methods

Typically you will not call these methods directly--other ABC Library methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Ask	a virtual to accept query criteria
Kill	shut down the QueryClass object

## AddItem (add field to query)

**AddItem**( *name*, *title* [ ,*picture* ] )

<b>AddItem</b>	Adds specific functionality to the QueryClass.
<i>name</i>	A string constant, variable, EQUATE, or expression containing the queryable item, typically the fully qualified name of a field in the view being queried.

**Tip:** This may also be an expression such as *UPPER(field1)* or *field1 \* field2*.

<i>title</i>	A string constant, variable, EQUATE, or expression containing the text to associate with the queryable item. This text appears as the prompt or header for the item in the query dialog presented to the end user.
<i>picture</i>	A string constant, variable, EQUATE, or expression containing the display picture for the queryable item. If omitted, <i>picture</i> defaults to S255 (unformatted string). See <i>Picture Tokens</i> in the <i>Language Reference</i> for more information.

The **AddItem** method adds a queryable item to the QueryClass object. The QueryClass object can then accept input for the item from the end user and build a filter expression to apply to the view being queried.

Other QueryClass methods, such as GetLimit and SetLimit, refer to the queryable item by its *name*.

**Tip:** You may use the **BrowseClass.UpdateQuery** method in combination with the **AddItem** method to define a query interface that contains the **BrowseClass** fields plus other queryable items.

Example:

```
QueryForm QueryFormClass
QueryVis QueryFormVisual
BRW1 CLASS(BrowseClass)
Q &CusQ
END
```

```
CusWindow.Init PROCEDURE()
CODE
!open files, views, window, etc.
BRW1.UpdateQuery(QueryForm) !add browse fields to query
QueryForm.AddItem('UPPER(CUS:NAME)', 'Name') !add caseless name to query
QueryForm.AddItem('CUS:ZIP_CODE', 'Name') !add zip code to query
QueryForm.AddItem('ITM:Qty+ITM:Price', 'Total')!add dynamic total to query
END
RETURN Level:Benign
```

See Also: [BrowseClass.UpdateQuery](#)

**Ask (a virtual to accept query criteria)****Ask( [ *uselast* ] ), VIRTUAL, PROC**

<b>Ask</b>	A virtual to accept query criteria (search values) from the end user.
<i>uselast</i>	An integer constant, variable, EQUATE, or expression that determines whether the QueryClass object carries forward previous query criteria. A value of one (1 or True) carries forward input from the previous query; a value of zero (0 or False) discards previous input.

The **Ask** method is a virtual to display a query dialog, process its events, and return a value indicating whether to apply the query or abandon it. A return value of Level:Notify indicates the QueryClass object should apply the query criteria; a return value of Level:Benign indicates the end user cancelled the query input dialog and the QueryClass object should not apply the query criteria.

The GetFilter method generates filter expressions using the search values set by the Ask method.

Implementation: For each item that can be queried (added by the AddItem method), the Ask method collects the query values from the selected item's file buffers rather than from a query input dialog. This default behavior automatically gives you query criteria (search values) for the current item without soliciting input from the end user. This allows you to, for example, use a regular update form as a special kind of query (QBE) form.

Return Data Type: **BYTE**

Example:

```
MyQueryForm.Ask PROCEDURE(BYTE UseLast)           !derived class Ask method
W WINDOW('Example values'),CENTER,SYSTEM,GRAY    !declare user input dialog
  BUTTON('&OK'),USE(?Ok,1000),DEFAULT
  BUTTON('Cancel'),USE(?Cancel,1001)
END
CODE
OPEN(W)
IF ~UseLast THEN SELF.Reset().                   !preserve or discard prior query
IF SELF.Win.Run()=RequestCancelled              !show dialog and handle events
  RETURN Level:Benign                            !return Cancel indicator
ELSE
  RETURN Level:Notify                            !return OK indicator
END
```

See Also: AddItem, GetFilter, QueryFormClass.Ask, QueryFormClass

**ClearQuery ( remove loaded query )****ClearQuery, PROTECTED**

The **ClearQuery** method clears the listbox on the QueryVisual dialog that contains the currently loaded query.

Implementation: The ClearQuery method is called by the Take, Restore, and QueryVisual.TakeAccepted methods. This method is used by the QuickQBE functionality.

**Note:** The Clear Query method does not remove the ad hoc filter from a Browse procedure. It only affects the query dialog used for managing a Browse's queries.

Example:

```
QueryClass.Take PROCEDURE(PopupClass P)
CODE
ASSERT(~P &= NULL)
IF SELF.QkSupport
  SELF.QkCurrentQuery = P.GetLastSelection()
  SELF.PopupList.PopupID = SELF.QkCurrentQuery
  GET(SELF.PopupList,SELF.PopupList.PopupID)
  IF Errorcode()
    SELF.ClearQuery()
  ELSE
    SELF.Restore(SELF.PopupList.QueryName)
  END
  SELF.Save('tsMRU') ! Save Most recently used for Browse\Report query sharing.
  RETURN 1
END
RETURN 0
```

See Also: Save

**Delete ( remove saved query )****Delete ( *queryname* ), PROTECTED**

<b>Delete</b>	Remove a saved query.
<i>queryname</i>	A string constant, variable, EQUATE or expression containing the name of a saved query.

Implementation: The Delete method is the mechanism by which the QuickQBE queries are deleted. This method is called when the user presses the Delete button on the Query dialog.

**Note:** The Delete method is primarily designed for use by the QuickQBE functionality.

See Also: Save

**GetFilter (return filter expression)****GetFilter**

The **GetFilter** method returns a filter expression. The GetFilter method builds the expression from values supplied by the AddItem, Ask, and SetLimit methods.

Implementation: The returned filter expression is up to 5000 characters long.

The GetFilter method generates filter expressions using the search values set by the Ask method, the SetLimit method, or both.

**Tip:** By default, the Ask method only sets the *equal* to value; it does not set lower and upper values.

The generated filter expression searches for values greater than *lower*, less than *upper*, and equal to *equal*. For string fields, the GetFilter method applies the following special meanings to these special search characters:

Symbol	Position	Filter Effect
^	prefix	caseless (case insensitive) search
*	prefix	contains search
*	suffix	begins with search
=	prefix	inclusive search
>	prefix	exclusive search--greater than
<	prefix	exclusive search--less than

For example:

<i>lower</i>	<i>upper</i>	<i>equal</i>	query searches for
fred			values > fred
	fred		values < fred
		fred	values = fred
=fred			values >= fred
	=fred		values <= fred
		>fred	values >= fred
fred	fred		values >= fred
fred	george	george	values <= george AND values > fred
		d*	values beginning with d (e.g., dog, david)
		*d	values containing d (e.g., dog, cod)
		^d	values d and D
		^d*	values beginning with d or D (e.g., dog, David)
		^*d	values containing d or D (e.g., dog, cod, coD)

Return Data Type: **STRING**

Example:

```
MyBrowseClass.TakeLocate PROCEDURE
CurSort  USHORT,AUTO
I         USHORT,AUTO
CODE
IF ~SELF.Query&=NULL AND SELF.Query.Ask()    !get query input from end user
  CurSort = POINTER(SELF.Sort)                !save current sort order
  LOOP I = 1 TO RECORDS(SELF.Sort)
    PARENT.SetSort(I)                        !step thru each sort order
    SELF.SetFilter(SELF.Query.GetFilter(),'9-QBE') !get filter expression from
Query
  END                                         ! and give it to Browse object
  PARENT.SetSort(CurSort)                    !restore current sort order
  SELF.ResetSort(1)                          !apply the filter expression
END
```

See Also:           AddItem, Ask, SetLimit

**GetLimit (get searchvalues)****GetLimit**( *name* [ ,*lower*] [ ,*upper*] [ ,*equal*] ), **PROTECTED**

<b>GetLimit</b>	Gets the QueryClass object's search values.
<i>name</i>	A string constant, variable, EQUATE, or expression containing the queryable item to set. Queryable items are established by the AddItem method.
<i>lower</i>	A CSTRING variable to receive the filter's lower boundary.
<i>upper</i>	A CSTRING variable to receive the filter's upper boundary.
<i>equal</i>	A CSTRING variable to receive the filter's exact match.

The **GetLimit** method gets the QueryClass object's search values. The Ask or SetLimit methods set the QueryClass object's search values.

Implementation: The GetFilter method generates filter expressions using the search values. The generated filter expression searches for values greater than *lower*, less than *upper*, and equal to *equal*.

Example:

```

QueryClass.Ask      PROCEDURE(BYTE UseLast=1)
I USHORT,AUTO
EV CSTRING(1000),AUTO
CODE
  SELF.Reset
  LOOP I = 1 TO RECORDS(SELF.Fields)
    GET(SELF.Fields,I)
    EV = CLIP(EVALUATE(SELF.Fields.Name))
    IF EV
      SELF.SetLimit(SELF.Fields.Name,,,EV)
    END
  END
RETURN Level:Notify

```

See Also: AddItem, Ask, SetLimit

**Init (initialize the QueryClass object)**

**Init**([*queryvisual*] [,*inimanager*,*family*,*errormanage*])

<b>Init</b>	The <b>Init</b> method initializes the QueryClass object.
<i>queryvisual</i>	The label of the query's QueryVisual object.
<i>inimanager</i>	The label of the query's INIManager object.
<i>family</i>	A string constant, variable, EQUATE, or expression that specifies the name to use for storing queries. By default this is the name of the procedure.
<i>errormanager</i>	The label of the query's ErrorManager object

Implementation: The Init method allocates a new queryable items queue.

Example:

```

ThisWindow.Init PROCEDURE()
ReturnValue BYTE,AUTO
CODE
!other initialization code
Query.Init(QueryWindow)
Query.AddItem('UPPER(CLI:LastName)','Name','s20')
Query.AddItem('CLI:ZIP+1','ZIP+1','')
RETURN ReturnValue
ThisWindow.Kill PROCEDURE()
ReturnValue BYTE,AUTO
CODE
!other termination code
Query.Kill
RETURN ReturnValue

```

See Also: Kill

## Kill (shut down the QueryClass object)

### Kill, VIRTUAL

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code.

Implementation:       The Kill method deallocates the queryable items queue.

Example:

```
ThisWindow.Init PROCEDURE()  
ReturnValue        BYTE,AUTO  
  CODE  
  !other initialization code  
  Query.Init(QueryWindow)  
  Query.AddItem('UPPER(CLI:LastName)', 'Name', 's20')  
  Query.AddItem('CLI:ZIP+1', 'ZIP+1', '')  
  RETURN ReturnValue
```

```
ThisWindow.Kill PROCEDURE()  
ReturnValue        BYTE,AUTO  
  CODE  
  !other termination code  
  Query.Kill  
  RETURN ReturnValue
```

See Also:            Init

**Reset (reset the QueryClass object)****Reset**( [ *name* ] )

<b>Reset</b>	Resets the QueryClass object.
<i>name</i>	A string constant, variable, EQUATE, or expression containing the queryable item to reset. Queryable items are established by the AddItem method. If omitted, the Reset method resets all the queryable items.

The **Reset** method resets the QueryClass object by clearing prior query values.

Implementation: The Reset method calls the SetLimit method to clear the search values for each queryable item.

Example:

```

MyQueryForm.Ask PROCEDURE(BYTE UseLast)           !derived class Ask method
W WINDOW('Example values'),CENTER,SYSTEM,GRAY    !declare user input dialog
  BUTTON('&OK'),USE(?Ok,1000),DEFAULT
  BUTTON('Cancel'),USE(?Cancel,1001)
END
CODE
OPEN(W)
IF ~UseLast THEN SELF.Reset().                   !preserve or discard prior query
IF SELF.Win.Run()=RequestCancelled              !show dialog and handle events
  RETURN Level:Benign                           !return Cancel indicator
ELSE
  RETURN Level:Notify                           !return OK indicator

```

See Also: AddItem, SetLimit

**Restore ( retrieve saved query )****Restore ( *queryname* )**

**Restore**            The **Restore** method retrieves a saved query from the INI file.

*queryname*        A string constant, variable, EQUATE or expression containing the name of a saved query.

Implementation:    The Restore method is called by the Take, QueryVisual.TakeAccepted, and QueryVisual.TakeFieldEvent methods.

**Note:**    The Restore method is primarily designed for use by the QuickQBE functionality.

See Also:            Save

**Save ( save a query )****Save ( *queryname* )****Save**

The **Save** method saves a query to the INI file.

*queryname*

A string constant, variable, EQUATE or expression containing the name of the query to save.

Implementation:

The Kill method deallocates the queryable items queue.

**Note:** The save method is primarily designed for use by the QuickQBE functionality.

See Also:

Restore

**SetLimit (set search values)**

**SetLimit**( *name* [ ,*lower* ] [ ,*upper* ] [ ,*equal* ] )

<b>SetLimit</b>	Sets the QueryClass object's search values.
<i>name</i>	A string constant, variable, EQUATE, or expression containing the queryable item to set. Queryable items are established by the AddItem method.
<i>lower</i>	A string constant, variable, EQUATE, or expression that specifies the filter's lower boundary--the query searches for values greater than <i>lower</i> . If you prefix the lower value with the equal sign (=), the query searches for values greater than or equal to <i>lower</i> . If omitted, SetLimit leaves the lower boundary intact.
<i>upper</i>	A string constant, variable, EQUATE, or expression that specifies the filter's upper boundary--the query searches for values less than <i>upper</i> . If you prefix the <i>upper</i> value with the equal sign (=), the query searches for values less than or equal to <i>upper</i> . If omitted, SetLimit leaves the upper boundary intact.
<i>equal</i>	A string constant, variable, EQUATE, or expression that specifies the filter's exact match--the query searches for values equal to <i>equal</i> . If you prefix the <i>equal</i> value with the greater sign (>), the query searches for values greater than or equal to <i>equal</i> ; if you prefix the <i>equal</i> value with the less sign (<), the query searches for values less than or equal to <i>equal</i> . If omitted, SetLimit leaves the exact match intact.

The **SetLimit** method sets the QueryClass object's search values. The GetLimit method gets the QueryClass object's search values.

Implementation: The GetFilter method generates filter expressions using the search values set by the Ask method, the SetLimit method, or both.

**Tip:** By default, the Ask method only sets the *equal* to value; it does not set lower and upper values.

The generated filter expression searches for values greater than *lower*, less than *upper*, and equal to *equal*. For string fields, the GetFilter method applies the following special meanings to these special search characters:

Symbol	Position	Filter Effect
^	prefix	caseless (case insensitive) search
*	prefix	contains search
*	suffix	begins with search
=	prefix	inclusive search
>	prefix	exclusive search--greater than
<	prefix	exclusive search--less than

For example:

<i>lower</i>	<i>upper</i>	<i>equal</i>	<b>query searches for</b>
fred			values > fred
	fred		values < fred
		fred	values = fred
=fred			values >= fred
	=fred		values <= fred
		>fred	values >= fred
fred	fred		values >= fred
fred	george	george	values <= george AND values > fred
		d*	values beginning with d (e.g., dog, david)
		*d	values containing d (e.g., dog, cod)
		^d	values d and D
		^d*	values beginning with d or D (e.g., dog, David)
		^*d	values containing d or D (e.g., dog, cod, coD)

Example:

```

QueryClass.Ask      PROCEDURE(BYTE UseLast=1)
I USHORT,AUTO
EV CSTRING(1000),AUTO
CODE
  SELF.Reset
  LOOP I = 1 TO RECORDS(SELF.Fields)
    GET(SELF.Fields,I)
    EV = CLIP(EVALUATE(SELF.Fields.Name))
    IF EV
      SELF.SetLimit(SELF.Fields.Name,,,EV)
    END
  END
END
RETURN Level:Notify

```

See Also:           AddItem, Ask, GetFilter, GetLimit

## **SetQuickPopup ( add QuickQBE to browse popup )**

SetQuickPopup ( *popup*, *query* )

**SetQuickPopup** Add QuickQBE items and submenu to Browse popup.

*popup* A string constant, variable, EQUATE, or expression containing the label of the browse PopupManager object.

*query* A string constant, variable, EQUATE, or expression containing the label of the QueryClass object

Implementation: The SetQuickPopup method adds a submenu to the BroweClass popup object, and an item to clear the current query and an item for every saved query for the current procedure.

**Note:** TheSetQuickPopup method is primarily designed for use by the QuickQBE functionality.

See Also: QkIcon, QkMenuIcon, BrowseClass.Popup, Save

**Take ( process QuickQBE popup menu choice )**

**Take** ( *popup* )

<b>Take</b>	Add QuickQBE items and submenu to Browse popup.
<i>popup</i>	A string constant, variable, EQUATE, or expression containing the label of the browse PopupManager object.

Implementation: The Take method is called by the BrowseClass.TakeEvent method. It returns one (1 or True) if QuickQBE support is enabled, and zero (0 or False) if QuickQBE is not enabled.

**Note:** The take method is primarily designed for use by the QuickQBE functionality.

Return Data Type: BYTE

See Also: QkSupport



# QueryFormClass

## QueryFormClass Overview

The QueryFormClass is a QueryClass with a "form" user interface. The QueryFormClass provides support for ad hoc queries against Clarion VIEWS. The form interface includes an entry field, a prompt, and an equivalence operator (equal, not equal, greater than, etc.) button for each queryable item.

## QueryFormClass Concepts

Use the AddItem method to define a user input dialog at runtime. Or create a custom dialog to plug into your QueryClass object. Use the Ask method to solicit end user query criteria (search values) or use the SetLimit method to programmatically set query search values. Finally, use the GetFilter method to build the filter expression to apply to your VIEW. Use the ViewManager.SetFilter method or the PROP:Filter property to apply the resulting filter.

## QueryFormClass Relationship to Other Application Builder Classes

The QueryFormClass is derived from the QueryClass, plus it relies on the QueryFormVisual class to display its input dialog and handle the dialog events.

The BrowseClass optionally uses the QueryFormClass to filter its result set. Therefore, if your BrowseClass object uses a QueryFormClass object, it must instantiate the QueryFormClass object and the QueryFormVisual object.

The BrowseClass automatically provides a default query dialog that solicits end user search values for each field displayed in the browse list. See the Conceptual Example.

## QueryFormClass ABC Template Implementation

The ABC Templates declare a local QueryFormClass class *and* object for each instance of the BrowseQBEBUTTON template. The ABC Templates automatically include all the code necessary to support the functionality specified in the BrowseQBEBUTTON template.

The templates optionally *derive* a class from the QueryFormClass for *each* BrowseQBEBUTTON control in the application. The derived class is called QBE# where # is the instance number of the BrowseQBEBUTTON template. The templates provide the derived class so you can use the BrowseQBEBUTTON template **Classes** tab to easily modify the query's behavior on an instance-by-instance basis.

**Tip:** Use the BrowseQBEBUTTON control template to add a QueryFormClass object to your template generated BrowseBoxes.

## QueryFormClass Source Files

The QueryFormClass source code is installed by default to the Clarion \LIBSRC folder. The specific QueryFormClass files and their respective components are:

ABQUERY.INC	QueryFormClass declarations
ABQUERY.CLW	QueryFormClass method definitions

## QueryFormClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a QueryFormClass object and related objects. The example plugs a QueryFormClass into a BrowseClass object. The QueryFormClass object solicits query criteria (search values) with a "form" dialog, and then generates a filter expression based on the end user input.

Note that the WindowManager and BrowseClass objects internally handle the normal events surrounding the query.

```

PROGRAM

  _ABCDllMode_  EQUATE(0)
  _ABCLinkMode_ EQUATE(1)
  INCLUDE( 'ABWINDOW.INC' )
  INCLUDE( 'ABBROWSE.INC' )
  INCLUDE( 'ABQUERY.INC' )

  MAP
  END

GlobalErrors  ErrorClass
Access:Customer CLASS(FileManager)
Init          PROCEDURE
              END

Relate:Customer CLASS(RelationManager)
Init          PROCEDURE
Kill          PROCEDURE,VIRTUAL
              END

GlobalRequest  BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD

Customer      FILE,DRIVER( 'TOPSPEED' ),PRE(CUS),CREATE,THREAD
CustomerIDKey KEY(CUS:ID),NOCASE,OPT,PRIMARY
NameKey       KEY(CUS:LastName),NOCASE,OPT
Record        RECORD,PRE( )
ID            LONG
LastName      STRING(20)
FirstName     STRING(15)
City          STRING(20)
State         STRING(2)
ZIP           STRING(10)
              END
              END

```

```

CustView    VIEW(Customer)
            END

CustQ       QUEUE
CUS:LastName LIKE(CUS:LastName)
CUS:FirstName LIKE(CUS:FirstName)
CUS:ZIP     LIKE(CUS:ZIP)
ViewPosition STRING(1024)
            END

CusWindow  WINDOW('Browse Customers'),AT(, ,210,105),IMM,SYSTEM,GRAY
            LIST,AT(5,5,200,80),USE(?CusList),IMM,HVSCROLL,FROM(CustQ),|
            FORMAT('80L(2)|M~Last~@s20@64L(2)|M~First~@s15@44L(2)|M~ZIP~@s10@')
            BUTTON('&Query'),AT(50,88),USE(?Query)
            BUTTON('Close'),AT(90,88),USE(?Close)
            END

ThisWindow  CLASS(WindowManager)                !declare ThisWindow object
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL
            END

Query       QueryFormClass                      !declare Query object
QBWindow    QueryFormVisual                    !declare QBWindow object
BRW1        CLASS(BrowseClass)                 !declare BRW1 object
Q           &CustQ
            END

CODE
GlobalErrors.Init
Relate:Customer.Init
GlobalResponse = ThisWindow.Run()              !ThisWindow handles all events
Relate:Customer.Kill
GlobalErrors.Kill

ThisWindow.Init PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?CusList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(?Close,RequestCancelled)
Relate:Customer.Open
BRW1.Init(?CusList,CustQ.ViewPosition,CustView,CustQ,Relate:Customer,ThisWindow)
OPEN(CusWindow)

```

```
SELF.Opened=True
Query.Init(QBEWindow)           !initialize Query object
BRW1.Q &= CustQ
BRW1.AddSortOrder(,CUS:NameKey)
BRW1.AddField(CUS:LastName,BRW1.Q.CUS:LastName)
BRW1.AddField(CUS:FirstName,BRW1.Q.CUS:FirstName)
BRW1.AddField(CUS:ZIP,BRW1.Q.CUS:ZIP)
BRW1.QueryControl = ?Query      !register Query button w/ BRW1
BRW1.UpdateQuery(Query)        !make each browse item Queryable
Query.AddItem('Cus:State','State') !make State field Queryable too
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
Relate:Customer.Close
RETURN ReturnValue

Access:Customer.Init PROCEDURE
CODE
PARENT.Init(Customer,GlobalErrors)
SELF.FileNameValue = 'Customer'
SELF.Buffer &= CUS:Record
SELF.Create = 1
SELF.AddKey(CUS:CustomerIDKey,'CUS:CustomerIDKey',1)
SELF.AddKey(CUS:NameKey,'CUS:NameKey',0)

Relate:Customer.Init PROCEDURE
CODE
Access:Customer.Init
PARENT.Init(Access:Customer,1)

Relate:Customer.Kill PROCEDURE
CODE
Access:Customer.Kill
PARENT.Kill
```

## QueryFormClass Properties

The QueryFormClass inherits all the properties of the *QueryClass* from which it is derived.

## QueryFormClass Methods

The QueryFormClass inherits all the methods of the QueryClass from which it is derived. See *QueryClass Methods* for more information.

### **QueryFormClass Functional Organization--Expected Use**

As an aid to understanding the QueryFormClass, it is useful to organize its various methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the QueryFormClass methods.

#### **Non-Virtual Methods**

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the QueryFormClass object
AddItem <sub>i</sub>	add a field to query
Kill <sub>v</sub>	shut down the QueryFormClass object

##### **Mainstream Use:**

Ask <sub>v</sub>	accept query criteria
GetFilter <sub>i</sub>	return filter expression

##### **Occasional Use:**

Reset <sub>i</sub>	reset the QueryFormClass object
GetLimit <sub>i</sub>	get search values
SetLimit <sub>i</sub>	set search values

<sub>v</sub> These methods are also Virtual.

<sub>i</sub> These methods are inherited from the QueryClass.

#### **Virtual Methods**

Typically you will not call these methods directly--other ABC Library methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Ask	accept query criteria
Kill	shut down the QueryFormClass object

**Ask (solicit query criteria)****Ask( [ *uselast* ] ), VIRTUAL, PROC**

<b>Ask</b>	Accepts query criteria (search values) from the end user.
<i>uselast</i>	An integer constant, variable, EQUATE, or expression that determines whether the QueryFormClass object carries forward previous query criteria. A value of one (1 or True) carries forward input from the previous query; a value of zero (0 or False) discards previous input.

The **Ask** method displays a query dialog, processes its events, and returns a value indicating whether to apply the query or abandon it. A return value of Level:Notify indicates the QueryFormClass object should apply the query criteria; a return value of Level:Benign indicates the end user cancelled the query input dialog and the QueryFormClass object should not apply the query criteria.

Implementation: The Ask method declares a generic (empty) dialog to accept query criteria. The Ask method calls the QueryFormClass object's WindowManager to define the dialog and process its events.

The GetFilter method generates filter expressions using the search values set by the Ask method.

The Init method sets the value of the QueryFormClass object's WindowManager.

Return Data Type: **BYTE**

Example:

```
MyBrowseClass.TakeLocate PROCEDURE
CurSort USHORT,AUTO
I USHORT,AUTO
CODE
IF ~SELF.Query&=NULL AND SELF.Query.Ask()
  CurSort = POINTER(SELF.Sort)
  LOOP I = 1 TO RECORDS(SELF.Sort)
    PARENT.SetSort(I)
    SELF.SetFilter(SELF.Query.GetFilter(),'9 - QBE')
  END
PARENT.SetSort(CurSort)
SELF.ResetSort(1)
END
```

See Also: GetFilter, Init, QueryFormVisual

**Init (initialize the QueryFormClass object)**

**Init**( *query window manager, inmanager, family, errormanager* )

<b>Init</b>	Initializes the QueryFormClass object.
<i>query window manager</i>	The label of the QueryFormVisual object that displays the query input dialog and processes it's events.
<i>inmanager</i>	The label of the INIManager object.
<i>family</i>	A string constant, variable, EQUATE, or expression containing the procedure name of the calling procedure.
<i>errormanager</i>	The label of the Global ErrorManager object.

The **Init** method initializes the QueryFormClass object.

Implementation: The Init method sets the QFC property for the *query window manager*.

Example:

```

ThisWindow.Init PROCEDURE()
ReturnValue          BYTE,AUTO
CODE
!other initialization code
Query.Init(QueryWindow)
Query.AddItem('UPPER(CLI:LastName)', 'Name', 's20')
Query.AddItem('CLI:ZIP+1', 'ZIP+1', '')
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue          BYTE,AUTO
CODE
!other termination code
Query.Kill
RETURN ReturnValue

```

See Also: Kill, QueryFormVisual, QueryFormVisual.QFC

## Kill (shut down the QueryFormClass object)

### Kill, VIRTUAL

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code.

Example:

```
ThisWindow.Init PROCEDURE()  
ReturnValuE          BYTE,AUTO  
CODE  
!other initialization code  
Query.Init(QueryWindow)  
Query.AddItem('UPPER(CLI:LastName)','Name','s20')  
Query.AddItem('CLI:ZIP+1','ZIP+1','')  
RETURN ReturnValuE
```

```
ThisWindow.Kill PROCEDURE()  
ReturnValuE          BYTE,AUTO  
CODE  
!other termination code  
Query.Kill  
RETURN ReturnValuE
```

See Also:            Init

# QueryFormVisual

## QueryFormVisual Overview

The QueryFormVisual class is a QueryVisualClass that displays a query input dialog and handles the dialog events. The query dialog includes an entry field, a prompt, and an equivalence operator (equal, not equal, greater than, etc.) button for each queryable item.

## QueryFormVisual Concepts

The QueryFormVisual provides the query window for a QueryFormClass object. The Init method defines and "programs" the query input dialog at runtime. The query input dialog contains a prompt, an entry field, and a query operator button for each queryable item. On each button press, the operator button cycles through the available operators: equal (=), greater than or equal (>=), less than or equal (<=), not equal (<>), and no filter ( ).

The QueryFormClass recognizes these operators and uses them to create valid filter expressions.

## QueryFormVisual Relationship to Other Application Builder Classes

The QueryFormVisual class is derived from the QueryVisualClass.

The BrowseClass uses the QueryFormVisual to provide the user interface to its query facility. Therefore, if your BrowseClass object provides a query, it must instantiate the QueryFormVisual object (and the QueryFormClass object). See the Conceptual Example.

## QueryFormVisual ABC Template Implementation

The ABC Templates declare a local QueryFormVisual class *and* object for each instance of the BrowseQBEBUTTON template. The ABC Templates automatically include all the code necessary to support the functionality specified in the BrowseQBEBUTTON template.

The templates optionally *derive* a class from the QueryFormVisual for *each* BrowseQBEBUTTON control in the application. The derived class is called QBV# where # is the instance number of the BrowseQBEBUTTON template. The templates provide the derived class so you can use the BrowseQBEBUTTON template **Classes** tab to easily modify the query's behavior on an instance-by-instance basis.

**Tip:** Use the BrowseQBEBUTTON control template to add a QueryFormClass object to your template generated BrowseBoxes.

## QueryFormVisual Source Files

The QueryFormVisual source code is installed by default to the Clarion \LIBSRC folder. The specific QueryFormVisual files and their respective components are:

ABQUERY.INC	QueryFormVisual declarations
ABQUERY.CLW	QueryFormVisual method definitions

## QueryFormVisual Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a QueryFormVisual object and related objects. The example plugs a QueryFormClass into a BrowseClass object. The QueryFormClass object uses the QueryFormVisual to solicit query criteria (search values) from the end user.

Note that the QueryVisualClass and BrowseClass objects internally handle the normal events surrounding the query.

```

PROGRAM
  _ABCDllMode_ EQUATE(0)
  _ABCLinkMode_ EQUATE(1)

  INCLUDE('ABWINDOW.INC')
  INCLUDE('ABBROWSE.INC')
  INCLUDE('ABQUERY.INC')

  MAP
  END

GlobalErrors ErrorClass
Access:Customer CLASS(FileManager)
Init           PROCEDURE
              END

Relate:Customer CLASS(RelationManager)
Init          PROCEDURE
Kill          PROCEDURE,VIRTUAL
              END

GlobalRequest  BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD

Customer       FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,THREAD
CustomerIDKey  KEY(CUS:ID),NOCASE,OPT,PRIMARY
NameKey        KEY(CUS:LastName),NOCASE,OPT
Record         RECORD,PRE()
ID             LONG
LastName       STRING(20)
FirstName      STRING(15)
City           STRING(20)
State          STRING(2)
ZIP            STRING(10)
              END
              END

```

```

CustView    VIEW(Customer)
            END

CustQ       QUEUE
CUS:LastName LIKE(CUS:LastName)
CUS:FirstName LIKE(CUS:FirstName)
CUS:ZIP     LIKE(CUS:ZIP)
ViewPosition STRING(1024)
            END

CusWindow   WINDOW('Browse Customers'),AT(,,210,105),IMM,SYSTEM,GRAY
            LIST,AT(5,5,200,80),USE(?CusList),IMM,HVSCROLL,FROM(CustQ),|
            FORMAT('80L(2)|M~Last~@s20@64L(2)|M~First~@s15@44L(2)|M~ZIP~@s10@')
            BUTTON('&Query'),AT(50,88),USE(?Query)
            BUTTON('Close'),AT(90,88),USE(?Close)
            END

ThisWindow  CLASS(WindowManager)           !declare ThisWindow object
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL
            END

Query       QueryFormClass                 !declare Query object
QBWindow   QueryFormVisual                 !declare QBWindow object
BRW1       CLASS(BrowseClass)              !declare BRW1 object
Q          &CustQ
            END

CODE
GlobalErrors.Init
Relate:Customer.Init
GlobalResponse = ThisWindow.Run()         !ThisWindow handles all events
Relate:Customer.Kill
GlobalErrors.Kill

ThisWindow.Init PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?CusList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(?Close,RequestCancelled)
Relate:Customer.Open
BRW1.Init(?CusList,CustQ.ViewPosition,CustView,CustQ,Relate:Customer,ThisWindow)
OPEN(CusWindow)

```

```
SELF.Opened=True
Query.Init(QBEWindow)           !initialize Query object
BRW1.Q &= CustQ
BRW1.AddSortOrder(,CUS:NameKey)
BRW1.AddField(CUS:LastName,BRW1.Q.CUS:LastName)
BRW1.AddField(CUS:FirstName,BRW1.Q.CUS:FirstName)
BRW1.AddField(CUS:ZIP,BRW1.Q.CUS:ZIP)
BRW1.QueryControl = ?Query      !register Query button w/ BRW1
BRW1.UpdateQuery(Query)         !make each browse item Queryable
Query.AddItem('Cus:State','State') !make State field Queryable too
SELF.SetAlerts()
RETURN ReturnValue
```

```
ThisWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
Relate:Customer.Close
RETURN ReturnValue
```

```
Access:Customer.Init PROCEDURE
CODE
PARENT.Init(Customer,GlobalErrors)
SELF.FileNameValue = 'Customer'
SELF.Buffer &= CUS:Record
SELF.Create = 1
SELF.AddKey(CUS:CustomerIDKey,'CUS:CustomerIDKey',1)
SELF.AddKey(CUS:NameKey,'CUS:NameKey',0)
```

```
Relate:Customer.Init PROCEDURE
CODE
Access:Customer.Init
PARENT.Init(Access:Customer,1)
```

```
Relate:Customer.Kill PROCEDURE
CODE
Access:Customer.Kill
PARENT.Kill
```

## QueryFormVisual Properties

The QueryFormVisual inherits all the properties of the QueryVisualClass from which it is derived. See QueryVisualClass properties for more information.

In addition to the inherited properties, the QueryFormVisual contains the following property:

### **QFC (reference to the QueryFormClass)**

#### **QFC   &QueryFormClass**

The **QFC** property is a reference to the QueryFormClass that uses this QueryFormVisual object to solicit query criteria (search values) from the end user.

Implementation:       The QueryFormClass.Init method sets the QFC property.

See Also:               QueryFormClass.Init

## QueryFormVisual Methods

The QueryFormVisual inherits all the methods of the QueryVisualClass from which it is derived. See QueryVisualClass methods for more information.

### QueryFormVisual Functional Organization--Expected Use

As an aid to understanding the QueryFormVisual class, it is useful to organize its various methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the QueryFormVisual methods.

#### Non-Virtual Methods

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Initv     program the QueryFormVisual object

##### **MainStream Use:**

none

##### **Occasional Use:**

none

v These methods are also Virtual.

#### Virtual Methods

Typically you will not call these methods directly--other ABC Library methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init	program the QueryFormVisual object
TakeCompleted	wrap up the query dialog
TakeAccepted	handle EVENT:Accepted events

**GetButtonFeq(returns a field equate label)****GetButtonFeq(*index*)**

<b>GetButtonFeq</b>	Returns the field equate label of the starting set of QBE controls.
<i>index</i>	An integer constant, variable, EQUATE, or expression that contains an indexed value to the starting set of QBE controls

The **GetButtonFeq** method returns a field equate label which corresponds to the starting set of QBE controls.

Implementation: The GetButtonFeq method is used in conjunction with the Web Builder template set. This method is called only if the WebServer.IsEnabled method returns a TRUE value.

Return Data Type: SIGNED

## Init (initialize the QueryFormVisual object)

### Init, VIRTUAL, PROC

The **Init** method initializes the QueryFormVisual object. Init returns Level:Benign to indicate normal initialization.

The Init method "programs" the QueryFormVisual object.

Implementation: The QueryFormClass.Ask method (indirectly) calls the Init method to configure the QueryFormClass WINDOW.

For each queryable item (defined by the QFC property), the Init method creates a series of window controls to accept search values. By default, each queryable item gets a prompt, an entry control, and an query operator button (equal, not equal, greater than, etc.).

The Init method sets the coordinates for the QueryFormClass WINDOW and for the individual controls.

Return Data Type: BYTE

Example:

```
MyQuery.Ask PROCEDURE(BYTE UseLast)
W WINDOW('Query values'),GRAY !declare an "empty" window
  BUTTON('&OK'),USE(?Ok,1000),DEFAULT
  BUTTON('Cancel'),USE(?Cancel,1001)
  END
CODE
OPEN(W)
IF SELF.Win.Run()=RequestCancelled !configure, display & process query dialog
  ! Win &= QueryFormVisual
  ! Win.Run calls Init, Ask & Kill
  ! Win.Init configures the dialog
  ! Win.Ask displays dialog & handles events
  ! Win.Kill shuts down the dialog

RETURN Level:Notify
ELSE
RETURN Level:Benign
END
```

See Also: QFC

## ResetFromQuery ( reset the QueryFormVisual object )

### ResetFromQuery, DERIVED

The **ResetFromQuery** method resets the QueryFormVisual object after a query.

Implementation:       The ResetFromQuery method calls the SetText method for each field available for query.

Example:

```
QueryFormVisual.ResetFromQuery PROCEDURE
I USHORT
CODE
LOOP I = 1 TO RECORDS(SELF.QFC.Fields)
  GET (SELF.QFC.Fields,I)
  SELF.SetText((Feq:StartControl+(I*3-1)),SELF.QFC.Fields.Middle)
END
Update()
RETURN
```

See Also:            SetText

**SetText ( set prompt text:QueryFormVisual )****SetText ( *control*, *entrytext* )**

<b>SetText</b>	The <b>SetText</b> method sets the prompt text for the QueryFormVisual object.
<i>control</i>	An integer constant, variable, EQUATE, or expression containing the control number of the control to act on.
<i>entrytext</i>	A string constant, variable, EQUATE, or expression containing the text to assign to the prompt.

Implementation: The ResetFromQuery method calls the SetText method for each field available for query.

Example:

```
QueryFormVisual.ResetFromQuery PROCEDURE
I USHORT
CODE
LOOP I = 1 TO RECORDS(SELF.QFC.Fields)
  GET(SELF.QFC.Fields,I)
  SELF.SetText((Feq:StartControl+(I*3-1)),SELF.QFC.Fields.Middle)
END
Update()
RETURN
```

See Also: ResetFromQuery

## TakeAccepted (handle query dialog Accepted events: QueryFormVisual)

### TakeAccepted, VIRTUAL, PROC

The **TakeAccepted** method processes EVENT:Accepted events for the query dialog's controls, and returns a value indicating whether ACCEPT loop processing is complete and should stop. TakeAccepted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: For each item that can be queried(defined by the QFC property), the TakeAccepted method implements cycling of operators for the query operator buttons. On each button press, the button cycles through the available filter operators: equal(=), greater than or equal(>=), less than or equal(<=), not equal(<>), and no filter( ).

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;    RVal = SELF.TakeRejected()
    OF EVENT:Selected;    RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;   RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also: QFC

## TakeCompleted (complete the query dialog: QueryFormVisual)

### TakeCompleted, VIRTUAL, PROC

The **TakeCompleted** method processes the EVENT:Completed event for the query dialog and returns a value indicating whether window ACCEPT loop processing is complete and should stop.

TakeCompleted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: Based on the current state of the querydialog, the TakeCompleted method sets the search values in the QFC property. The QFC property may use these search values to create a filter expression.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;    RVal = SELF.TakeRejected()
    OF EVENT:Selected;    RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;   RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also: QFC

**TakeFieldEvent (a virtual to process field events:QueryFormVisual)****TakeFieldEvent, DERIVED, PROC**

The **TakeFieldEvent** method is a virtual placeholder to process all field-specific/control-specific events for the window. It returns a value indicating whether window process is complete and should stop. TakeFieldEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: Return values are declared in ABERROR.INC. The TakeEvent method calls the TakeFieldEvent method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I USHORT,AUTO
CODE
IF ~FIELD()
  RVal = SELF.TakeWindowEvent()
  IF RVal THEN RETURN RVal.
END
CASE EVENT()
  OF EVENT:Accepted; RVal = SELF.TakeAccepted()
  OF EVENT:Rejected; RVal = SELF.TakeRejected()
  OF EVENT:Selected; RVal = SELF.TakeSelected()
  OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
  OF EVENT:Completed; RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
  RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
IF FIELD()
  RVal = SELF.TakeFieldEvent()
END
RETURN RVal
```

See Also: WindowManager.Ask

## UpdateFields ( process query values )

UpdateFields, DERIVED

The **UpdateFields** method processes the values entered into the query dialog for assignment to a filter statement.

Implementation:           The TakeCompleted method calls the UpdateFields method.

Example:

```
QueryFormVisual.TakeCompleted   PROCEDURE
CODE
SELF.SetResponse(RequestCompleted)
SELF.UpdateFields
RETURN Level:Benign
```

See Also:                TakeCompleted



# QueryListClass

## QueryListClass--Overview

The QueryListClass is a QueryClass with a "list" user interface. The QueryListClass provides support for ad hoc queries against Clarion VIEWS. The list interface includes is an edit-in-place, 3-column listbox with a field column, an equivalence operator (contains, begins, equal, not equal, greater than, less than) column, and a value (to query for) column.

## QueryListClass Concepts

Use the AddItem method to define a user input dialog at runtime. Or create a custom dialog to plug into your QueryClass object. Use the Ask method to solicit end user query criteria (search values) or use the SetLimit method to programmatically set query search values. Finally, use the GetFilter method to build the filter expression to apply to your VIEW. Use the ViewManager.SetFilter method or the PROP:Filter property to apply the resulting filter.

## QueryListClass--Relationship to Other Application Builder Classes

The QueryListClass is derived from the QueryClass, plus it relies on the QueryListVisual class to display its input dialog and handle the dialog events.

The BrowseClass optionally uses the QueryListClass to filter its result set. If your BrowseClass object uses a QueryListClass object, it must instantiate a QueryListClass object and a QueryListVisual object.

The BrowseClass automatically provides a default query dialog that solicits end user search values for each field displayed in the browse list. See the *Conceptual Example*.

## QueryListClass--ABC Template Implementation

The ABC Templates declare a local QueryClass class *and* object for each instance of the BrowseQBEBUTTON template. The ABC Templates automatically include all the code necessary to support the functionality specified in the BrowseQBEBUTTON template.

The templates optionally derive a QueryListClass object for *each* BrowseQBEBUTTON control in the application. The derived class is called QBE# where # is the instance number of the BrowseQBEBUTTON template. The templates provide the derived class so you can use the BrowseQBEBUTTON template **Classes** tab to easily modify the query's behavior on an instance-by-instance basis.

**Tip:** Use the BrowseQBEBUTTON control template to add a QueryListClass object to your template generated BrowseBoxes.

## QueryListClass Source Files

The QueryListClass source code is installed by default to the Clarion \LIBSRC folder. The specific QueryListClass files and their respective components are:

ABQUERY.INC	QueryListClass declarations
ABQUERY.CLW	QueryListClass method definitions

## QueryListClass--Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a QueryListClass object and related objects. The example plugs a QueryListClass into a BrowseClass object. The QueryListClass object solicits query criteria (search values) with a "list" dialog, then generates a filter expression based on the end user input.

```

PROGRAM

_ABCDllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)

INCLUDE('ABWINDOW.INC')
INCLUDE('ABBROWSE.INC')
INCLUDE('ABQUERY.INC')

MAP
END

GlobalErrors ErrorClass
Access:Customer CLASS(FileManager)
Init          PROCEDURE
              END

Relate:Customer CLASS(RelationManager)
Init          PROCEDURE
Kill          PROCEDURE,VIRTUAL
              END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD

Customer      FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,THREAD
CustomerIDKey KEY(CUS:ID),NOCASE,OPT,PRIMARY
NameKey       KEY(CUS:LastName),NOCASE,OPT
Record        RECORD,PRE()
ID            LONG
LastName      STRING(20)
FirstName     STRING(15)
City          STRING(20)
State         STRING(2)
ZIP           STRING(10)
              END
              END

```

```

CustView      VIEW(Customer)
              END
CustQ         QUEUE
CUS:LastName  LIKE(CUS:LastName)
CUS:FirstName LIKE(CUS:FirstName)
CUS:ZIP       LIKE(CUS:ZIP)
ViewPosition  STRING(1024)
              END

CusWindow WINDOW('Browse Customers'),AT(,,210,105),IMM,SYSTEM,GRAY
          LIST,AT(5,5,200,80),USE(?CusList),IMM,HVSCROLL,FROM(CustQ),|
          FORMAT('80L(2)|M~Last~@s20@64L(2)|M~First~@s15@44L(2)|M~ZIP~@s10@')
          BUTTON('&Query'),AT(50,88),USE(?Query)
          BUTTON('Close'),AT(90,88),USE(?Close)
          END

ThisWindow CLASS(WindowManager)           !declare ThisWindow object
Init       PROCEDURE(),BYTE,PROC,VIRTUAL
Kill       PROCEDURE(),BYTE,PROC,VIRTUAL
          END

Query      QueryListmClass                !declare Query object
QBWindow  QueryListVisual                 !declare QBWindow object
BRW1      CLASS(BrowseClass)              !declare BRW1 object
Q         &CustQ
          END

CODE
GlobalErrors.Init
Relate:Customer.Init
GlobalResponse = ThisWindow.Run()        !ThisWindow handles all events
Relate:Customer.Kill
GlobalErrors.Kill

ThisWindow.Init PROCEDURE()
ReturnValue     BYTE,AUTO
CODE
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?CusList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(?Close,RequestCancelled)
Relate:Customer.Open
BRW1.Init(?CusList,CustQ.ViewPosition,CustView,CustQ,Relate:Customer,ThisWindow)
OPEN(CusWindow)
SELF.Opened=True
Query.Init(QBWindow)                    !initialize Query object

```

```
BRW1.Q &= CustQ
BRW1.AddSortOrder(,CUS:NameKey)
BRW1.AddField(CUS:LastName,BRW1.Q.CUS:LastName)
BRW1.AddField(CUS:FirstName,BRW1.Q.CUS:FirstName)
BRW1.AddField(CUS:ZIP,BRW1.Q.CUS:ZIP)
BRW1.QueryControl = ?Query           !register Query button w/ BRW1
BRW1.UpdateQuery(Query)              !make each browse item Queryable
Query.AddItem('Cus:State','State')  !make State field Queryable too
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
RETURNValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
Relate:Customer.Close
RETURN ReturnValue

Access:Customer.Init PROCEDURE
CODE
PARENT.Init(Customer,GlobalErrors)
SELF.FileNameValue = 'Customer'
SELF.Buffer &= CUS:Record
SELF.Create = 1
SELF.AddKey(CUS:CustomerIDKey,'CUS:CustomerIDKey',1)
SELF.AddKey(CUS:NameKey,'CUS:NameKey',0)

Relate:Customer.Init PROCEDURE
CODE
Access:Customer.Init
PARENT.Init(Access:Customer,1)

Relate:Customer.Kill PROCEDURE
CODE
Access:Customer.Kill
PARENT.Kill
```

## QueryListClass Properties

The QueryListClass inherits all the properties of the QueryClass from which it is derived. See *QueryClass Properties* for more information.

## QueryListClass Methods

The QueryListClass inherits all the methods of the QueryClass from which it is derived. See *QueryClass Methods* for more information.

### **QueryListClass--Functional Organization--Expected Use**

As an aid to understanding the QueryListClass, it is useful to organize its various methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the QueryListClass methods.

#### **Non-Virtual Methods**

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into two categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the QueryListClass object
AddItem <sub>i</sub>	add a field to query
Kill <sub>v</sub>	shut down the QueryListListClass object

##### **Mainstream Use:**

Ask <sub>v</sub>	accept query criteria
GetFilter <sub>i</sub>	return filter expression

##### **Occasional Use:**

Reset <sub>i</sub>	reset the QueryListClass object
GetLimit <sub>i</sub>	get search values
SetLimit <sub>i</sub>	set search values

<sub>v</sub> These methods are also Virtual.

<sub>i</sub> These methods are inherited from the QueryClass.

#### **Virtual Methods**

Typically you will not call these methods directly--other ABC Library methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Ask	accept query criteria
Kill	shut down the QueryListClass object

**Ask (solicit query criteria:QueryListClass)****Ask( [ *uselast* ] ), DERIVED, PROC**


---

<b>Ask</b>	Accepts query criteria (search values) from the end user.
<i>uselast</i>	An integer constant, variable, EQUATE, or expression that determines whether the QueryListClass object carries forward previous query criteria. A value of one (1) carries forward input from the previous query; a value of zero (0) discards previous input.

The **Ask** method displays a query dialog, processes its events, and returns a value indicating whether to apply the query or abandon it. A return value of Level:Notify indicates the QueryListClass object should apply the query criteria; a return value of Level:Benign indicates the end user cancelled the query input dialog and the QueryListClass object should not apply the query criteria.

Implementation: The Ask method declares a generic (empty) dialog to accept query criteria. The Ask method calls the QueryListClass object's WindowManager to define the dialog and process it's events.

The GetFilter method generates filter expressions using the search values set by the Ask method.

The Init method sets the value of the QueryListClass object's WindowManager.

Return Data Type: BYTE

Example:

```
MyBrowseClass.TakeLocate PROCEDURE
CurSort USHORT,AUTO
I USHORT,AUTO
CODE
IF ~SELF.Query&=NULL AND SELF.Query.Ask()
  CurSort = POINTER(SELF.Sort)
  LOOP I = 1 TO RECORDS(SELF.Sort)
    PARENT.SetSort(I)
    SELF.SetFilter(SELF.Query.GetFilter(),'9 - QBE')
  END
  PARENT.SetSort(CurSort)
  SELF.ResetSort(1)
END
```

See Also: GetFilter, Init, QueryListVisual

**Init (initialize the QueryListClass object)**

**Init**( *querywindowmanager*, *inimanager*, *family*, *errormanager* )

---

<b>Init</b>	Initializes the QueryListClass object.
<i>querywindowmanager</i>	The label of the QueryListVisual object that displays the query input dialog list and processes it's events.
<i>inimanager</i>	The label of the INIManager object.
<i>family</i>	A string constant, variable, EQUATE, or expression containing the procedure name of the calling procedure.
<i>errormanager</i>	The label of the Global ErrorManager object.

The **Init** method initializes the QueryListClass object.

Implementation: The Init method sets the QFC property for the *querywindowmanager*.

Example:

```

ThisWindow.Init PROCEDURE()
ReturnValue          BYTE,AUTO
CODE
!other initialization code
Query.Init(QueryWindow)
Query.AddItem('UPPER(CLI:LastName)', 'Name', 's20')
Query.AddItem('CLI:ZIP+1', 'ZIP+1', '')
RETURN ReturnValue

```

```

ThisWindow.Kill PROCEDURE()
ReturnValue          BYTE,AUTO
CODE
!other termination code
Query.Kill
RETURN ReturnValue

```

See Also: Kill, QueryListVisual, QueryListVisual.QFC

## Kill (shut down the QueryListClass object)

### Kill, DERIVED

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code.

Example:

```
ThisWindow.Init PROCEDURE()  
ReturnValue          BYTE,AUTO  
CODE  
!other initialization code  
Query.Init(QueryWindow)  
Query.AddItem('UPPER(CLI:LastName)','Name','s20')  
Query.AddItem('CLI:ZIP+1','ZIP+1','')  
RETURN ReturnValue
```

```
ThisWindow.Kill PROCEDURE()  
ReturnValue          BYTE,AUTO  
CODE  
!other termination code  
Query.Kill  
RETURN ReturnValue
```

See Also:     Init

# QueryListVisual

## QueryListVisual--Overview

The QueryListVisual class is a QueryVisualClass that displays a query input dialog and handles the dialog events. The query dialog includes an edit-in-place, 3-column listbox, which allows the end user to choose the fields to query, the equivalence operator, and the value to query for.

## QueryListVisual Concepts

The QueryListVisual provides the query window for a QueryListClass object. The Init method defines and "programs" the query input dialog at runtime. The query interface includes an edit-in-place, 3-column listbox with a field column, an equivalence operator (contains, begins, equal, not equal, greater than, less than) column, and a value (to query for) column.

## QueryListVisual--Relationship to Other Application Builder Classes

The QueryListVisual class is derived from the QueryVisualClass.

The BrowseClass optionally uses the QueryListVisual class to provide the user an edit-in-place list interface to its query facility.

The QueryListClass requires the QueryListVisual class as a window manager.

## QueryListVisual--ABC Template Implementation

The ABC Templates declare a local QueryListVisual class *and* object for each instance of the BrowseQBEBButton template. The ABC Templates automatically include all the code necessary to support the functionality specified in the BrowseQBEBButton template.

The templates optionally *derive* a class from the QueryListVisual for *each* BrowseQBEBButton control in the application. The derived class is called QBV# where # is the instance number of the BrowseQBEBButton template. The templates provide the derived class so you can use the BrowseQBEBButton template **Classes** tab to easily modify the query's behavior on an instance-by-instance basis.

**Tip:** Use the BrowseQBEBButton control template to add a QueryListClass object to your template generated BrowseBoxes.

## QueryListVisual Source Files

The QueryListVisual source code is installed by default to the Clarion \LIBSRC folder. The specific QueryListVisual files and their respective components are:

ABQUERY.INC  
ABQUERY.CLW

QueryListVisual declarations  
QueryListVisual method definitions

## QueryListVisual--Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a QueryListVisual object and related objects. The example plugs a QueryListClass into a BrowseClass object. The QueryListClass object uses the QueryListVisual to solicit query criteria (search values) from the end user.

Note that the QueryVisualClass and BrowseClass objects internally handle the normal events surrounding the query.

```

PROGRAM

_ABCDllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)

INCLUDE('ABWINDOW.INC')
INCLUDE('ABBROWSE.INC')
INCLUDE('ABQUERY.INC')

MAP
END

GlobalErrors ErrorClass
Access:Customer CLASS(FileManager)
Init PROCEDURE
      END

Relate:Customer CLASS(RelationManager)
Init PROCEDURE
Kill PROCEDURE,VIRTUAL
      END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest LONG(0),THREAD

Customer FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,THREAD
CustomerIDKey KEY(CUS:ID),NOCASE,OPT,PRIMARY
NameKey KEY(CUS:LastName),NOCASE,OPT
Record RECORD,PRE()
ID LONG
LastName STRING(20)
FirstName STRING(15)
City STRING(20)
State STRING(2)
ZIP STRING(10)
      END
      END

```

```

CustView      VIEW(Customer)
              END

CustQ         QUEUE
CUS:LastName  LIKE(CUS:LastName)
CUS:FirstName LIKE(CUS:FirstName)
CUS:ZIP       LIKE(CUS:ZIP)
ViewPosition  STRING(1024)
              END

CusWindow WINDOW('Browse Customers'),AT(,,210,105),IMM,SYSTEM,GRAY
          LIST,AT(5,5,200,80),USE(?CusList),IMM,HVSCROLL,FROM(CustQ),|
          FORMAT('80L(2)|M~Last~@s20@64L(2)|M~First~@s15@44L(2)|M~ZIP~@s10@')
          BUTTON('&Query'),AT(50,88),USE(?Query)
          BUTTON('Close'),AT(90,88),USE(?Close)
          END

ThisWindow CLASS(WindowManager)           !declare ThisWindow object
Init       PROCEDURE(),BYTE,PROC,VIRTUAL
Kill       PROCEDURE(),BYTE,PROC,VIRTUAL
          END

Query      QueryListClass                 !declare Query object
QBWindow  QueryListVisual                 !declare QBWindow object
BRW1      CLASS(BrowseClass)              !declare BRW1 object
Q         &CustQ
          END

CODE
GlobalErrors.Init
Relate:Customer.Init
GlobalResponse = ThisWindow.Run()         !ThisWindow handles all events
Relate:Customer.Kill
GlobalErrors.Kill

ThisWindow.Init PROCEDURE()
ReturnValue     BYTE,AUTO
CODE
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?CusList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(?Close,RequestCancelled)
Relate:Customer.Open
BRW1.Init(?CusList,CustQ.ViewPosition,CustView,CustQ,Relate:Customer,ThisWindow)
OPEN(CusWindow)

```

```
SELF.Opened=True
Query.Init(QBEWindow)           !initialize Query object
BRW1.Q &= CustQ
BRW1.AddSortOrder(,CUS:NameKey)
BRW1.AddField(CUS:LastName,BRW1.Q.CUS:LastName)
BRW1.AddField(CUS:FirstName,BRW1.Q.CUS:FirstName)
BRW1.AddField(CUS:ZIP,BRW1.Q.CUS:ZIP)
BRW1.QueryControl = ?Query      !register Query button w/ BRW1
BRW1.UpdateQuery(Query)         !make each browse item Queryable
Query.AddItem('Cus:State','State') !make State field Queryable too
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
Relate:Customer.Close
RETURN ReturnValue

Access:Customer.Init PROCEDURE
CODE
PARENT.Init(Customer,GlobalErrors)
SELF.FileNameValue = 'Customer'
SELF.Buffer &= CUS:Record
SELF.Create = 1
SELF.AddKey(CUS:CustomerIDKey,'CUS:CustomerIDKey',1)
SELF.AddKey(CUS:NameKey,'CUS:NameKey',0)

Relate:Customer.Init PROCEDURE
CODE
Access:Customer.Init
PARENT.Init(Access:Customer,1)

Relate:Customer.Kill PROCEDURE
CODE
Access:Customer.Kill
PARENT.Kill
```

## QueryListVisual Properties

The QueryListVisual inherits all the properties of the QueryVisualClass from which it is derived. See QueryVisualClass properties for more information.

In addition to the inherited properties, the QueryListVisual contains the following properties:

### **QFC (reference to the QueryListClass)**

#### **QFC    &QueryListClass**

The **QFC** property is a reference to the QueryListClass that uses this QueryListVisual object to solicit query criteria (search values) from the end user.

Implementation:    The QueryListClass.Init method sets the QFC property.

See Also:            QueryListClass.Init

### **OpsEIP (reference to the EditDropListClass)**

#### **OpsEIP    &EditDropListClass,PROTECTED**

The **OpsEIP** property is a reference to the EditDropListClass that displays the available operators in the QueryList dialog.

### **FldsEIP (reference to the EditDropListClass)**

#### **FldsEIP    &EditDropListClass,PROTECTED**

The **FldsEIP** property is a reference to the EditDropListClass that displays the available fields to query in the QueryList dialog.

**ValueEIP(reference to QEditEntryClass)**

**ValueEIP**      **&EditEntryClass,PROTECTED**

The **ValueEIP** property is a reference to the QEditEntryClass that enables edit-in-place entry fields in the QBE window.

Implementation:      The ValueEIP is initialized in the QueryListVisual.Init method and updated in the QueryListVisual.UpdateControl method.

See Also:              QueryListVisual.Init, QueryListVisual.UpdateControl, QueryListVisual.Kill

## QueryListVisual Methods

The QueryListVisual inherits all the methods of the QueryVisualClass from which it is derived. See QueryVisualClass methods for more information.

### **QueryListVisual--Functional Organization--Expected Use**

As an aid to understanding the QueryListVisual class, it is useful to organize its various methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the QueryListVisual methods.

#### **Non-Virtual Methods**

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init<sub>o</sub>                      program the QueryListVisual object

##### **MainStream Use:**

none

##### **Occasional Use:**

none

o These methods are Derived.

#### **Derived Methods**

---

Typically you will not call these methods directly--other ABC Library methods call them. However, we anticipate you will often want to override these methods, and because they are derived, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init	program the QueryListVisual object
SetAlerts	prepare the query dialog for EIP
TakeEvent	Handle events for the query dialog
TakeCompleted	wrap up the query dialog
TakeAccepted	handle EVENT:Accepted events

**Init (initialize the QueryListVisual object)****Init, DERIVED PROC**

The **Init** method initializes the QueryListVisual object. Init returns Level:Benign to indicate normal initialization.

The Init method "programs" the QueryListVisual object.

Implementation: The QueryListClass.Ask method (indirectly) calls the Init method to configure the QueryListClass WINDOW.

The Init method reads each queryable item (defined by the QFC property) from a queue, then creates an edit-in-place, 3-column listbox with a field column, an equivalence operator (equal, not equal, greater than, etc.) column, and a value (to query for) column.

The Init method sets the coordinates for the QueryListClass WINDOW and for the individual controls.

Return Data Type: **BYTE**

Example:

```

QueryListClass.Ask      PROCEDURE(BYTE UseLast)
W WINDOW('Query'),AT(,,300,200),FONT('MS SansSerif',8,,FONT:regular),SYSTEM,GRAY,DOUBLE
  LIST,AT(5,5,290,174),USE(?List,FEQ:ListBox),|
  FORMAT('91L|M~Field~@s20@44C|M~Operator~L@s10@120C|M~Value~L@s30@')
  BUTTON('Insert'),AT(5,183,45,14),USE(?Insert,FEQ:Insert)
  BUTTON('Change'),AT(52,183,45,14),USE(?Change,FEQ:Change)
  BUTTON('Delete'),AT(99,183,45,14),USE(?Delete,FEQ:Delete)
  BUTTON('&OK'),AT(203,183,45,14),USE(?Ok,FEQ:OK),DEFAULT
  BUTTON('Cancel'),AT(250,183,45,14),USE(?Cancel,FEQ:Cancel)
END
CODE
OPEN(W)
IF ~UseLast THEN SELF.Reset().
RETURN CHOOSE(SELF.Win.Run()=RequestCancelled,Level:Benign,Level:Notify)

```

See Also: **QFC**

**Kill (shutdown the QueryListVisual object)****Kill, PROC, DERIVED**

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code.

Return Data Type:       BYTE

## ResetFromQuery ( reset the QueryList Visual object )

### ResetFromQuery, DERIVED

The **ResetFromQuery** method resets the QueryListVisual object after a query.

Implementation: The ResetFromQuery method calls the GetLimit method for each field available for query.

Example:

```
QueryListVisual.ResetFromQuery PROCEDURE
I USHORT
CaseLess BYTE,AUTO
High BYTE
CODE
FREE(SELF.Vals)
LOOP I = 1 TO RECORDS(SELF.QFC.Fields)
  GET(SELF.QFC.Fields,I)
  LOOP
    High = SELF.QFC.GetLimit(SELF.Vals.Value,SELF.Vals.Ops,CaseLess,High)
    IF SELF.Vals.Value
      IF CaseLess AND SELF.Vals.Value[1] ~= '^'
        SELF.Vals.Value = '^' & SELF.Vals.Value
      END
      SELF.Vals.Field = SELF.QFC.Fields.Title
      ADD(SELF.Vals)
    END
  WHILE High
END
RETURN
```

See Also: QueryClass.GetLimit

## SetAlerts (alert keystrokes for the edit control:QueryListVisual)

### SetAlerts, DERIVED

The **SetAlerts** method method alerts appropriate keystrokes for the edit-in-place control.

Implementation: The Init method calls the CreateControl method to create the input control and set the FEQ property. The Init method then calls the SetAlerts method to alert specific keystrokes for the query dialog. Alerted keys are:

MouseLeft2	!edit selected record
InsertKey	!add a query field
CtrlEnter	!edit selected record
DeleteKey	!delete query field

Example:

```

EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
CODE
SELF.ListBoxFeq = ListBox
SELF.CreateControl()
ASSERT(SELF.Feq)
SELF.UseVar &= UseVar
SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
SELF.Feq{PROP:Use} = UseVar
SELF.SetAlerts

```

See Also:        Init

## TakeAccepted (handle query dialog EVENT:Accepted events)

### TakeAccepted, DERIVED, PROC

The **TakeAccepted** method processes EVENT:Accepted events for the query dialog's controls, and returns a value indicating whether ACCEPT loop processing is complete and should stop. TakeAccepted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeAccepted method handles the processing of the update buttons (Insert, Change, Delete) on the Query list dialog.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;    RVal = SELF.TakeRejected()
    OF EVENT:Selected;    RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;   RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also: QFC

**TakeCompleted (complete the query dialog)****TakeCompleted, DERIVED, PROC**

The **TakeCompleted** method processes the EVENT:Completed event for the query dialog and returns a value indicating whether window ACCEPT loop processing is complete and should stop.

TakeCompleted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: Based on the current state of the querydialog, the TakeCompleted method sets the search values in the QFC property. The QFC property may use these search values to create a filter expression.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I   USHORT,AUTO
  CODE
  IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
  END
  CASE EVENT()
  OF EVENT:Accepted;   RVal = SELF.TakeAccepted()
  OF EVENT:Rejected;   RVal = SELF.TakeRejected()
  OF EVENT:Selected;   RVal = SELF.TakeSelected()
  OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
  OF EVENT:Completed;  RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVal = SELF.TakeCloseEvent()
  END
  IF RVal THEN RETURN RVal.
  IF FIELD()
    RVal = SELF.TakeFieldEvent()
  END
  RETURN RVal
```

See Also: QFC

**TakeEvent (process edit-in-place events:QueryListVisual)****TakeEvent( *event* ), VIRTUAL**


---

**TakeEvent** Processes an event for the QueryListVisualClass object.

*event* An integer constant, variable, EQUATE, or expression that contains the event number (see EVENT in the *Language Reference*).

The **TakeEvent** method processes an event for the QueryListVisualClass object and returns a value indicating the user requested action. Valid actions are none, insert (InsertKey), change (MouseLeft2 or CtrlEnter), or delete (DeleteKey).

Implementation: The EIPManager.TakeFieldEvent method calls the TakeEvent method. The TakeEvent method process an EVENT:AlertKey for the edit-in-place control and returns a value indicating the user requested action.

Return Data Type: BYTE

Example:

```

EIPManager.TakeFieldEvent      PROCEDURE
I UNSIGNED(1)
CODE
IF FIELD() = SELF.ListControl THEN RETURN Level:Benign .
LOOP I = 1 TO RECORDS(SELF.EQ)+1
! Optimised to pick up subsequent events from same field
IF ~SELF.EQ.Control &= NULL AND SELF.EQ.Control.Feq = FIELD()
SELF.TakeAction(SELF.EQ.Control.TakeEvent(EVENT()))
RETURN Level:Benign
END
GET(SELF.EQ,I)
END
! Not a known field
IF ?{PROP:Type} <> CREATE:Button OR EVENT() <> EVENT:Selected
!Wait to post accepted for button
SELF.Repost = EVENT()
SELF.RepostField = FIELD()
SELF.TakeFocusLoss
END
RETURN Level:Benign

```

See Also: EIPManager.TakeFieldEvent, SetAlerts

## TakeFieldEvent (a virtual to process field events:QueryListVisual)

### TakeFieldEvent, DERIVED, PROC

The **TakeFieldEvent** method is a virtual placeholder to process all field-specific/control-specific events for the window. It returns a value indicating whether window process is complete and should stop. TakeFieldEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: Return values are declared in ABERROR.INC. The TakeEvent method calls the TakeFieldEvent method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I USHORT,AUTO
CODE
IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
END
CASE EVENT()
    OF EVENT:Accepted; RVal = SELF.TakeAccepted()
    OF EVENT:Rejected; RVal = SELF.TakeRejected()
    OF EVENT:Selected; RVal = SELF.TakeSelected()
    OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
    OF EVENT:Completed; RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
IF FIELD()
    RVal = SELF.TakeFieldEvent()
END
RETURN RVal
```

See Also: WindowManager.Ask

## UpdateControl(updates the edit-in-place entry control)

### UpdateControl(*fieldname*)

---

**UpdateControl** Updates the value for the edit-in-place entry control.

*fieldname* A string constant, variable, EQUATE, or expression containing the name of the edit-in-place field.

The **UpdateControl** method updates the value for the edit-in-place entry control.

Implementation: The UpdateControl method is called from the QEIPManager.TakeEvent, when the MouseRight button is clicked.

## UpdateFields ( process query values )

UpdateFields, DERIVED

The **UpdateFields** method processes the values entered into the query dialog for assignment to a filter statement.

Implementation: The TakeCompleted method calls the UpdateFields method.

Example:

```
QueryListVisual.TakeCompleted PROCEDURE
CODE
SELF.SetResponse(RequestCompleted)
SELF.UpdateFields
RETURN Level:Benign
```

See Also: TakeCompleted



# QueryVisualClass

## QueryVisualClass: Overview

The QueryVisualClass is a WindowManager that displays a query input dialog and handles the dialog events. The QueryVisualClass is an abstract class that handles all of the basic Window functionality for the query dialog.

## QueryVisualClass Concepts

The QueryVisualClass is the parent class for the Query dialogs. It is designed to encapsulate the standard query requirements for the window manager.

## QueryVisualClass:Relationship to Other Application Builder Classes

The QueryVisualClass is derived from the WindowManager. The classes derived from the QueryVisualClass are optionally used by the QueryClass object.

The QueryFormVisual and the QueryListVisual classes are derived QueryVisualClasses.

## QueryVisualClass:ABC Template Implementation

The ABC Templates do not instantiate the QueryClass object independently. The templates instantiate the derived QueryFormClass or QueryListClass instead.

## QueryVisualClass Source Files

The QueryVisualClass source code is installed by default to the Clarion \LIBSRC folder. The specific QueryVisualClass files and their respective components are:

ABQUERY.INC	QueryVisual declarations
ABQUERY.CLW	QueryVisual method definitions

## QueryVisualClass Properties

The QueryVisualClass inherits all the properties of the WindowManager from which it is derived. See WindowManager Properties for more information.

In addition to the inherited properties, the QueryVisualClass contains the following properties:

### QC (reference to the QueryClass)

**QC**                      **&QueryClass**

The **QC** property is a reference to the QueryClass that uses this QueryVisualClass object to solicit query criteria (search values) from the end user.

Implementation:    The QueryFormVisual.Init and QueryListVisual.Init methods set the QC property.

See Also:            QueryFormVisual.Init, QueryListVisual.Init

### Resizer (reference to the WindowResizeClass:QueryVisualClass)

**Resizer**                      **&WindowResizeClass**

The **Resizer** property is a reference to the WindowResizeClass that is used by this QueryVisualClass object to handle resizing of the Window controls at runtime.

Implementation:    The Init method sets the Resizer property.

See Also:            Init, Kill

## QueryVisualClass Methods

The QueryVisualClass inherits all the methods of the WindowManager from which it is derived. See WindowManager Methods for more information.

### Init (initialize the QueryVisual object )

#### Init, DERIVED, PROC

The **Init** method initializes the QueryVisual object. Init returns Level:Benign to indicate normal initialization. The Init method "programs" the QueryVisual object.

Implementation: The Init method is called from the Init methods of both the QueryFormVisual and the QueryListVisual as PARENT calls. Typically, the Init method is paired with the Kill method, performing the converse of the Kill method tasks.

Return Data Type: BYTE

Example:

```
QueryFormVisual.Init          PROCEDURE
  CODE
  QFC &= SELF.QFC
  CLEAR(SELF)
  SELF.QFC &= QFC
  SELF.QC &= QFC
  RVal = PARENT.Init()        ! The call to the Init
  IF RVal THEN RETURN RVal.
  ! Saved query code
  RETURN RVal
```

See Also: Kill

## Kill (shut down the QueryVisual object)

### Kill, DERIVED, PROC

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code. Kill returns a value to indicate the status of the shut down.

Implementation: Kill sets the Dead property to True and returns Level:Benign to indicate a normal shut down. If the Dead property is already set to True, Kill returns Level:Notify to indicate it is taking no additional action.

Typically, the Kill method is paired with the Init method, performing the converse of the Init method tasks.

Return value EQUATES are declared in ABERROR.INC.

Return Data Type: BYTE

Example:

```
ThisWindow.Kill PROCEDURE()  
CODE  
IF PARENT.Kill() THEN RETURN Level:Notify.  
IF FilesOpened  
    Relate:Defaults.Close  
END  
IF SELF.Opened  
    INIMgr.Update('Main',AppFrame)  
END  
GlobalResponse = CHOOSE(LocalResponse=0,RequestCancelled,LocalResponse)
```

See Also: Init

**Reset ( reset the dialog for display:QueryVisualClass )****Reset ( *forcerest* ), DERIVED**


---

<b>Reset</b>	Resets the QueryVisual object.
<i>forcerest</i>	A numeric constant, variable, EQUATE, or expression that indicates whether to conditionally or uncondition-ally reset the window. A value of one (1 or True) uncon-ditionally resets the window; a value of zero (0 or False) only resets the window if circumstances require, such as a new sort on browse object or a changed reset field on a browse object. If omitted, <i>forcerest</i> defaults to zero (0).
Implementation:	The Reset method calls the WindowMangaer.Reset and handles the logic for enabling and disabling the Load and Save buttons. The Reset is called by the TakeFieldEvent and TakeAccepted methods.

Example:

```

QueryVisual.TakeFieldEvent PROCEDURE
CODE
CASE FIELD()
OF FEQ:QueryNameField
CASE EVENT()
OF EVENT:NewSelection
SELF.Reset
END
OF FEQ:SaveListBox
CASE Event()
OF EVENT:AlertKey
IF Keycode() = MouseLeft2
GET( SELF.Queries,CHOICE(FEQ:SaveListBox))
SELF.QC.Restore(SELF.Queries.Item)
SELF.ResetFromQuery
POST(EVENT:Accepted,FEQ:Ok)
END
OF EVENT:NewSelection
GET(SELF.Queries,CHOICE(FEQ:SaveListBox))
FEQ:QueryNameField{Prop:ScreenText} = SELF.Queries.Item
Update(FEQ:QueryNameField)
SELF.Reset
END
END
RETURN PARENT.TakeFieldEvent()

```

See Also:           TakeFieldEvent, TakeAccepted

**TakeAccepted (handle query dialog EVENT:Accepted events)****TakeAccepted, DERIVED, PROC**

The **TakeAccepted** method processes EVENT:Accepted events for the query dialog's controls, and returns a value indicating whether ACCEPT loop processing is complete and should stop. TakeAccepted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: For each queryable item (defined by the QC property), the TakeAccepted method implements cycling of operators for the query operator buttons. On each button press, the button cycles through the available filter operators: equal(=), greater than or equal(>=), less than or equal(<=), not equal(<>), and no filter( ).

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I USHORT,AUTO
CODE
IF ~FIELD()
  RVal = SELF.TakeWindowEvent()
  IF RVal THEN RETURN RVal.
END
CASE EVENT()
  OF EVENT:Accepted; RVal = SELF.TakeAccepted()
  OF EVENT:Rejected; RVal = SELF.TakeRejected()
  OF EVENT:Selected; RVal = SELF.TakeSelected()
  OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
  OF EVENT:Completed; RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
  RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
IF FIELD()
  RVal = SELF.TakeFieldEvent()
END
RETURN RVal
```

See Also: QC

## TakeFieldEvent (a virtual to process field events:QueryVisualClass)

### TakeFieldEvent, DERIVED, PROC

The **TakeFieldEvent** method is a virtual placeholder to process all field-specific/control-specific events for the window. It returns a value indicating whether window process is complete and should stop. TakeFieldEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: Return values are declared in ABERROR.INC. The TakeEvent method calls the TakeFieldEvent method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I USHORT,AUTO
CODE
IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
END
CASE EVENT()
    OF EVENT:Accepted; RVal = SELF.TakeAccepted()
    OF EVENT:Rejected; RVal = SELF.TakeRejected()
    OF EVENT:Selected; RVal = SELF.TakeSelected()
    OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
    OF EVENT:Completed; RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
IF FIELD()
    RVal = SELF.TakeFieldEvent()
END
RETURN RVal
```

See Also: WindowManager.Ask

**TakeWindowEvent (a virtual to process non-field events:QueryVisualClass)****TakeWindowEvent, DERIVED, PROC**

The **TakeWindowEvent** method processes all non-field events for the window and returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeWindowEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: TakeWindowEvent implements standard handling of EVENT:OpenWindow (Open method), EVENT:LoseFocus, EVENT:GainFocus (Reset method), and EVENT:Sized (WindowResizeClass.Resize method). Return values are declared in ABERROR.INC.

The TakeEvent method calls the TakeWindowEvent method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I USHORT,AUTO
CODE
IF ~FIELD()
  RVal = SELF.TakeWindowEvent()
  IF RVal THEN RETURN RVal.
END
CASE EVENT()
  OF EVENT:Accepted; RVal = SELF.TakeAccepted()
  OF EVENT:Rejected; RVal = SELF.TakeRejected()
  OF EVENT:Selected; RVal = SELF.TakeSelected()
  OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
  OF EVENT:Completed; RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
  RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
IF FIELD()
  RVal = SELF.TakeFieldEvent()
END
RETURN RVal
```

See Also: WindowManager.TakeEvent

# RelationManager

## RelationManager Overview

The RelationManager class declares a relation manager object that does the following:

- Consistently and flexibly defines relationships between files--the relationships need not be defined in a data dictionary; they may be defined directly (dynamically) to the relation manager object.
- Reliably enforces discrete specified levels of referential integrity (RI) constraints between the related files--the RI constraints need not be defined in a data dictionary; they may be defined directly (dynamically) to the relation manager object.
- Conveniently forwards appropriate file commands to related files--for example, when a relation manager object opens its primary file, it also opens any related files.

The RelationManager class provides "setup" methods that let you describe the file relationships, their linking fields, and their associated RI constraints; plus other methods to perform the cascadable or constrainable database operations such as open, change, delete, and close.

## Relation Manager Concepts and Conventions

### Cascading Commands and Referential Constraints

---

You can fully describe a set of file relationships with a series of RelationManager objects--one RelationManager object for each file. Each RelationManager object defines the relationships between its primary file and any files *directly* related to the primary file. However, each RelationManager object also knows about its related files' RelationManager objects, so indirectly, it knows about those secondary relationships too.

For example, consider three related files: Customer <->> Order <->> Item, where <->> indicates a one:many relationship. The RelationManager object for the Customer file knows about the relationship between Customer and Order, but it also knows about the Order file's RelationManager object, so indirectly, it knows about the relationship between Order and Item too.

The benefit of this chain of RelationManager awareness, is that you can issue a file command such as open or close to any one of the RelationManager objects and it forwards the command up *and* down the chain of related files; and for deletes or changes, it enforces any relational integrity constraints up and down the chain of related files.

### Me and Him

---

Some of the RelationManager methods refer to its primary file as "MyFile" or "Me" and its related files as "HisFile" or "Him." See Relation Manager Properties for more information.

---

## Left and Right (and Buffer)

---

Some of the RelationManager methods refer to its primary file record buffer as "Left," the associated queue buffer as "Right" and the associated save area for the record as "Buffer." See BufferedPairsClass and FieldPairsClass for more information.

## RelationManager ABC Template Implementation

The ABC Templates *derive* a class from the RelationManager class for *each* file the application processes. The derived classes are called Hide:Relate:*filename*, but may be referenced as Relate:*filename*. These derived classes and their methods are declared and implemented in the generated *appnaBC0.CLW* through *appnaBC9.CLW* files (depending on how many files your application uses). The derived class methods are specific to the file being managed, and they enforce the file relationships and referential integrity constraints specified in the data dictionary.

The ABC Templates generate housekeeping procedures to initialize and shut down the RelationManager objects. The procedures are DctInit and DctKill. They are generated into the *appnaBC.CLW* file.

The derived RelationManager classes are configurable with the **Global Properties** dialog. See *Template Overview--File Control Options* and *Classes Options* for more information.

## RelationManager Relationship to Other Application Builder Classes

---

### FileManager and BufferedPairsClass

---

The RelationManager relies on both the FileManager and the BufferedPairsClass to do much of its work. Therefore, if your program instantiates the RelationManager it must also instantiate the FileManager and the BufferedPairsClass. Much of this is automatic when you INCLUDE the RelationManager header (ABFILE.INC) in your program's data section. See the Conceptual Example and see File Manager Class and Field Pairs Classes for more information.

---

### ViewManager

---

Perhaps more significantly, the RelationManager serves as the foundation or "errand boy" of the ViewManager. If your program instantiates the ViewManager it must also instantiate the RelationManager. See View Manager Class for more information.

## RelationManager Source Files

The RelationManager source code is installed by default to the Clarion \LIBSRC folder. The RelationManager source code and its respective components are contained in:

ABFILE.INC	RelationManager declarations
ABFILE.CLW	RelationManager method definitions

## RelationManager Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate some RelationManager objects.

This example uses the RelationManager class to cascade new key values from parent file records to the corresponding child file records.

```

PROGRAM
  INCLUDE( 'ABFILE.INC' )
  INCLUDE( 'ABREPORT.INC' )
MAP
END

CUSTOMER  FILE, DRIVER( 'TOPSPEED' ), NAME( 'CUSTOMER' ), PRE( CUS ), BINDABLE, CREATE, THREAD
BYNUMBER  KEY( CUS: CUSTNO ), NOCASE, OPT, PRIMARY
Record    RECORD, PRE( )
CUSTNO    LONG
NAME      STRING( 30 )
ZIP       DECIMAL( 5 )
          END
          END

PHONES    FILE, DRIVER( 'TOPSPEED' ), NAME( 'PHONES' ), PRE( PHO ), BINDABLE, CREATE, THREAD
BYCUSTOMER KEY( PHO: CUSTNO, PHO: PHONE ), DUP, NOCASE, OPT
Record    RECORD, PRE( )
CUSTNO    LONG
PHONE     STRING( 20 )
TYPE      STRING( 8 )
          END
          END

GlobalErrors  ErrorClass

Access: CUSTOMER  CLASS( FileManager )
Init           PROCEDURE
              END

Relate: CUSTOMER  CLASS( RelationManager )
Init           PROCEDURE
              END

Access: PHONES    CLASS( FileManager )
Init           PROCEDURE
              END

Relate: PHONES    CLASS( RelationManager )
Init           PROCEDURE
              END

```

```

RecordsPerCycle  LONG(25)
StartOfCycle     LONG,AUTO
PercentProgress  BYTE
ProgressMgr      StepLongClass
CustView         VIEW(CUSTOMER)
                END
Process          ProcessClass
Progress:Bar     BYTE

ProgressWindow  WINDOW('Processing...'),AT(, ,142,59),CENTER,TIMER(1),GRAY,DOUBLE
                PROGRESS,USE(Progress:Bar),AT(15,15,111,12),RANGE(0,100)
                STRING(' '),AT(0,3,141,10),USE(?Progress:UserString),CENTER
                STRING(' '),AT(0,30,141,10),USE(?Progress:Text),CENTER
                BUTTON('Cancel'),AT(45,42,50,15),USE(?Progress:Cancel)
                END

CODE
GlobalErrors.Init
Relate:CUSTOMER.Init
Relate:PHONES.Init
ProgressMgr.Init(ScrollSort:AllowNumeric)
Process.Init(CustView,Relate:CUSTOMER,|
            ?Progress:Text,Progress:Bar,|
            ProgressMgr,CUS:CUSTNO)
Process.AddSortOrder( CUS:BYNUMBER )
Relate:CUSTOMER.Open
OPEN(ProgressWindow)
?Progress:Text{Prop:Text} = '0% Completed'
ACCEPT
CASE EVENT()
OF Event:OpenWindow
  Process.Reset
  IF Process.Next()
    POST(Event:CloseWindow)
  CYCLE
END
OF Event:Timer
  StartOfCycle=Process.RecordsProcessed
  LOOP WHILE Process.RecordsProcessed-StartOfCycle<RecordsPerCycle
    CUS:CUSTNO+=100                !change parent key value
    IF Relate:CUSTOMER.Update()    !cascade change to children
      BREAK
    END
  CASE Process.Next()
  OF Level:Notify
    ?Progress:Text{Prop:Text} = 'Process Completed'
    DISPLAY(?Progress:Text)
    POST(EVENT:CloseWindow)
    BREAK

```

```
        OF Level:Fatal
        POST(EVENT:CloseWindow)
        BREAK
    END
END
END
CASE FIELD()
OF ?Progress:Cancel
CASE Event()
OF Event:Accepted
    POST(Event:CloseWindow)
END
END
END
ProgressMgr.Kill
Relate:CUSTOMER.Close
Relate:CUSTOMER.Kill
Relate:PHONES.Kill
GlobalErrors.Kill

Access:CUSTOMER.Init PROCEDURE
CODE
PARENT.Init(Customer, GlobalErrors)
SELF.FileNameValue = 'CUSTOMER'
SELF.Buffer &= CUS:Record
SELF.AddKey(CUS:BYNUMBER, 'CUS:BYNUMBER',1)

Relate:CUSTOMER.Init PROCEDURE
CODE
Access:CUSTOMER.Init
PARENT.Init(Access:CUSTOMER,1)
SELF.AddRelation(Relate:PHONES,RI:Cascade,RI:Restrict,PHO:BYCUSTOMER)
SELF.AddRelationLink(CUS:CUSTNO,PHO:CUSTNO)

Access:PHONES.Init PROCEDURE
CODE
PARENT.Init(Phones, GlobalErrors)
SELF.FileNameValue = 'PHONES'
SELF.Buffer &= PHO:Record
SELF.AddKey(PHO:BYCUSTOMER, 'PHO:BYCUSTOMER')

Relate:PHONES.Init PROCEDURE
CODE
Access:PHONES.Init
PARENT.Init(Access:PHONES,1)
SELF.AddRelation( Relate:CUSTOMER )
```

## RelationManager Properties

### RelationManager Properties

The Relation Manager contains the following properties.

#### Me (the primary file's FileManager object)

##### **Me      &FileManager**

The **Me** property is a reference to the FileManager object for the RelationManager's primary file. By definition, the file referenced by this FileManager object is the RelationManager's primary file. The Me property identifies the primary file's FileManager object for the various RelationManager methods.

Implementation:      The Init method sets the value of the Me property.

See Also:              Init

#### UseLogout (transaction framing flag)

##### **UseLogout      BYTE**

The **UseLogout** property determines whether cascaded updates or deletes are done within a transaction frame (LOGOUT/COMMIT). A value of zero (0) indicates no transaction framing; a value of one (1) indicates transaction framing.

Implementation:      The Init method sets the value of the UseLogout property.

The ABC Templates set the UseLogout property based on the **Enclose RI code in transaction frame** check box in the **Global Properties** dialog.

See Also:              Init

## RelationManager Methods

### RelationManager Functional Organization--Expected Use

As an aid to understanding the RelationManager, it is useful to organize its methods into two categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the RelationManager methods.

#### Non-Virtual Methods

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the RelationManager object
AddRelation	set a file relationship
AddRelationLink	set linking fields for a relationship
SetAlias	add/set a file alias
Kill	shut down the RelationManager object

##### **Mainstream Use:**

Open <sub>v</sub>	open a file and any related files
Save <sub>v</sub>	copy current and designated related records
Update <sub>v</sub>	update current record subject to RI constraints
Delete <sub>v</sub>	delete current record subject to RI constraints
Close <sub>v</sub>	close a file and any related files

<sub>v</sub> These methods are also Virtual.

##### **Occasional Use:**

ListLinkingFields	map pairs of linked fields
SetQuickScan	enable QuickScan across related files

#### Virtual Methods

We anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Open	open a file and any related files
CancelAutoInc	undo PrimeAutoInc actions
Save	copy current and designated related records
Update	update current record subject to RI constraints
Delete	delete current record subject to RI constraints
Close	close a file and any related files

**AddRelation (set a file relationship)****AddRelation( *relationmanager* [,*updatemode* ,*deletemode* ,*relatedkey*] ), PROTECTED****AddRelation** Describes a relationship between this object's primary file (see *Me*) and another file.*relationmanager* The label of the related file's RelationManager object.*updatemode* A numeric constant, variable, EQUATE, or expression that indicates the referential integrity constraint to apply upon updates to the primary file's linking field. Valid constraints are none, clear, restrict, and cascade. If omitted, then *deletemode* and *relatedkey* must also be omitted, and the relationship is unconstrained.*deletemode* A numeric constant, variable, EQUATE, or expression that indicates the referential integrity constraint to apply upon deletes of the primary file's linking field. Valid constraints are none, clear, restrict, and cascade. If omitted, then *updatemode* and *relatedkey* must also be omitted, and the relationship is unconstrained.*relatedkey* The label of the related file's linking KEY. If included, the call to AddRelation must be followed by a call to AddRelationLink for each linking component field of the key. If omitted, then *updatemode* and *deletemode* must also be omitted, and the relationship is unconstrained.

The **AddRelation** method, in conjunction with the AddRelationLink method, describes a relationship between this object's primary file (see *Me*) and another file so that other RelationManager methods can cascade or constrain file operations across the related files when appropriate.

Implementation: You should typically call AddRelation after the Init method is called (or within your derived Init method).

The EQUATEs for *updatemode* and *deletemode* are declared in FILE.INC as follows:

```

ITEMIZE(0),PRE(RI)
None      EQUATE      !no action on related files
Clear     EQUATE      !clear the linking fields in related files
Restrict  EQUATE      !disallow the operation if linked records exist
Cascade   EQUATE      !update the linking fields in related files, or
END                               !delete the linked records in related files

```

Example:

```

Orders      FILE,DIVER('TOPSPEED'),PRE(ORD),CREATE
ByCustomer  KEY(ORD:CustNo,ORD:OrderNo),DUP,NOCASE,OPT

```

```
Record      RECORD,PRE( )
CustNo      LONG
OrderNo     LONG
OrderDate   LONG
Reference   STRING(24)
ShipTo      STRING(32)
Shipped     BYTE
Carrier     STRING(1)
            END
            END
```

```
Items      FILE,DRIVER('TOPSPEED'),PRE(ITEM),CREATE
AsEntered  KEY(ITEM:CustNo,ITEM:OrderNo,ITEM:LineNo),NOCASE,OPT,PRIMARY
Record     RECORD,PRE( )
CustNo     LONG
OrderNo    LONG
LineNo     SHORT
ProdCode   SHORT
Quantity   SHORT
            END
            END
```

```
CODE
!program code
```

```
Relate:Orders.Init PROCEDURE
```

```
CODE
  SELF.AddRelation( Relate:Items,0,0, ITEM:AsEntered )
  SELF.AddRelationLink( ORD:CustNo, ITEM:CustNo )
  SELF.AddRelationLink( ORD:OrderNo, ITEM:OrderNo )
  SELF.AddRelation( Relate:Customer )
```

See Also:      AddRelationLink, Init

**AddRelationLink (set linking fields for a relationship)****AddRelationLink( *parentkey*, *childkey* ), PROTECTED****AddRelationLink**

Identifies the linking fields for a relationship between this object's primary file (see *Me*) and another file.

*parentkey*      The label of the primary file's linking field.

*childkey*        The label of the related file's linking field.

The **AddRelationLink** method, in conjunction with the AddRelation method, describes a relationship between this object's primary file (see *Me*) and another file so that other RelationManager methods can cascade or constrain file operations across the related files when appropriate.

You must call AddRelationLink for each pair of linking fields, and the calls must be in sequence of high order linking fields to low order linking fields.

Implementation:    You should typically call AddRelationLink after the Init method is called (or within your derived Init method).

Example:

```

Orders            FILE, DRIVER( ' TOPSPEED' ), PRE( ORD ), CREATE
ByCustomer      KEY( ORD: CustNo, ORD: OrderNo ), DUP, NOCASE, OPT
Record           RECORD, PRE( )
CustNo           LONG
OrderNo          LONG
OrderDate        LONG
Reference        STRING( 24 )
ShipTo           STRING( 32 )
Shipped          BYTE
Carrier          STRING( 1 )
                  END
                  END

Items            FILE, DRIVER( ' TOPSPEED' ), PRE( ITEM ), CREATE
AsEntered        KEY( ITEM: CustNo, ITEM: OrderNo, ITEM: LineNo ), NOCASE, OPT, PRIMARY
Record           RECORD, PRE( )
CustNo           LONG
OrderNo          LONG
LineNo           SHORT
ProdCode         SHORT
Quantity         SHORT
                  END
                  END

```

```
CODE
!program code
Relate:Orders.Init PROCEDURE
CODE
  SELF.AddRelation( Relate:Items,0,0, ITEM:AsEntered )
  SELF.AddRelationLink( ORD:CustNo, ITEM:CustNo )
  SELF.AddRelationLink( ORD:OrderNo, ITEM:OrderNo )
  SELF.AddRelation( Relate:Customer )
```

See Also:      AddRelation, Init

**CancelAutoInc (undo autoincrement)****CancelAutoInc, VIRTUAL, PROC**

The **CancelAutoInc** method restores the managed file to its pre-PrimeAutoInc state, typically when an insert operation is cancelled. CancelAutoInc returns a value indicating its success or failure. A return value of zero (0 or Level:Benign) indicates success; any other return value indicates a problem.

Implementation: The CancelAutoInc method calls the FileManager.CancelAutoInc method for its primary file, passing SELF as the *relation manager* parameter.

Return value EQUATEs are declared in ABERROR.INC as follows:

```
! Severity of error
Level:Benign    EQUATE(0)
Level:User      EQUATE(1)
Level:Program   EQUATE(2)
Level:Fatal     EQUATE(3)
Level:Cancel    EQUATE(4)
Level:Notify    EQUATE(5)
```

Return Data Type: BYTE

Example:

```
WindowManager.TakeCloseEvent PROCEDURE
CODE
  IF SELF.Response <> RequestCompleted
  !procedure code
  IF SELF.OriginalRequest=InsertRecord AND SELF.Response=RequestCancelled
  IF SELF.Primary.CancelAutoInc() !undo PrimeAutoInc - cascade
  SELECT(SELF.FirstField)
  RETURN Level:Notify
  END
END
!procedure code
END
RETURN Level:Benign
```

See Also: FileManager.CancelAutoInc, FileManager.PrimeAutoInc

**Close (close a file and any related files)****Close( *cascading* ), VIRTUAL, PROC**

---

<b>Close</b>	Closes this object's primary file (see <i>Me</i> ) and any related files.
<i>cascading</i>	A numeric constant, variable, EQUATE, or expression that indicates whether this method was called by itself (recursive). A value of zero (0) indicates a non-recursive call; a value of one (1) indicates a recursive call. This allows the method to stop when it has processed all the related files in a circular relationship. If omitted, <i>cascading</i> defaults to zero (0). You should <i>always</i> omit this parameter when calling the Close method from your program.

The **Close** method closes this object's primary file (see *Me*) if no other procedure needs it, and any related files, and returns a value indicating its success or failure.

Implementation: The Close method uses the FileManager.Close method to close each file. The Close method returns the FileManager.Close method's return value. See *File Manager Class* for more information.

Return Data Type: BYTE

Example:

```
Relate:Customer.Open          !open Customer and related files
!program code                 !process the files
Relate:Customer.Close        !close Customer and related files
```

See Also: FileManager.Close

**Delete (delete record subject to referential constraints)****Delete( [ *confirm* ] ), VIRTUAL**


---

**Delete** Deletes the record from the primary file subject to any specified referential integrity constraints.

*confirm* An integer constant, variable, EQUATE, or expression that indicates whether to confirm the delete with the end user. A value of one (1 or True) deletes only on confirmation from the end user; a value of zero (0 or false) deletes without confirmation. If omitted, *confirm* defaults to one (1).

The **Delete** method deletes the current record from the primary file (see *Me*) applying any specified referential integrity constraints, then returns a value indicating its success or failure. The deletes are done within a transaction frame if the *Init* method's *uselayout* parameter is set to one (1).

Implementation: Delete constraints are specified by the *AddRelation* method. If the constraint is RI:Restrict, the method deletes the current record only if there are no related child records. If the constraint is RI:Cascade, the method also deletes any related child records. If the constraint is RI:None, the method unconditionally deletes only the primary file record. If the constraint is RI:Clear, the method unconditionally deletes the primary file record, and clears the linking field values in any related child records.

The *Delete* method calls the primary file *FileManager.Throw* method to confirm the delete with the end user.

Return Data Type: BYTE

Example:

DeleteCustomer PROCEDURE

```

CODE
Relate:Customer.Open           !Open Customer & related files
IF NOT GlobalErrors.Throw(Msg:ConfirmDelete) !have user confirm delete
  LOOP                          !allow retry if delete fails
    IF Relate:Customer.Delete()  !delete subject to constraints
      IF NOT GlobalErrors.Throw(Msg:RetryDelete)!if del fails, offer to try again
        CYCLE                    !if user accepts, try again
      END                        ! otherwise, fall thru
    END                          !if del succeeds or user declines
  UNTIL 1                       ! fall out of loop
END

```

See Also: AddRelation, Init

---

**GetNbFiles(returns number of children)**

**GetNbFiles**(*relationmanager*)

---

**GetNbFiles** Returns the number of child files

*relationmanager* The label of the related file's RelationManager object.

The **GetNbFiles** method returns the number of child files related to this objects primary file.

Return Data Type: LONG

---

**GetNbRelations(returns number of relations)**

**GetNbRelations**

The **GetNbRelations** method returns the number of relations defined for this objects primary file.

Return Data Type: LONG

**GetRelation(returns reference to relation manager)**

```

GetRelation(|relationposition   |)
              |relatedfile     |

```

---

**GetRelation** Returns a reference to the objects relation manager.

*relationposition* An integer constant, variable, EQUATE or expression that contains the relation number for the objects primary file.

*relatedfile* The label of the file to query relations.

The **GetRelation** method returns a reference to the objects relation manager based on the specified relation position or related file.

**GetRelation**(*relationposition*)

Returns the relation manager for the specified *relationposition* in the internal list of relations.

**GetRelation**(*relatedfile*)

Returns the relation manager for the specified file.

Return Data Type: \*RelationManager

**GetRelationType(returns relation type)**

```

GetRelationType(whichrelation)

```

---

**GetRelationType** Returns the type of relation for the objects primary file.

*whichrelation* An integer constant, variable, EQUATE or expression that contains the relation number.

The **GetRelationType** method returns the type of the relation for the specified relation number.

Implementation: The **GetRelationType** method returns a -1 when *whichrelation* is not a valid relation number. It returns a one (1) if the relationship is determined to be a one-to-many relationship. It returns a zero (0) for all other relation types.

Return Data Type: LONG

**Init (initialize the RelationManager object)**

**Init**( *filemanager* [, *uselayout*] )

---

<b>Init</b>	Initializes the RelationManager object.
<i>filemanager</i>	The label of the FileManager object for the RelationManager's primary file. By definition, the file referenced by this FileManager object is the RelationManager's primary file.
<i>uselayout</i>	A numeric constant, variable, EQUATE, or expression that determines whether cascaded updates or deletes are done within a transaction frame (LOGOUT/COMMIT). A value of zero (0) indicates no transaction framing; a value of one (1) indicates transaction framing. If omitted, <i>logout</i> defaults to zero (0).

The **Init** method initializes the RelationManager object. To implement the RelationManager's transaction framing, all the files within a transaction must use the same file driver and that file driver must support LOGOUT.

Implementation: The Init method sets the value of the Me and UseLogout properties. The ABC Templates set the *uselayout* parameter based on the **Enclose RI code in transaction frame** check box in the **Global Properties** dialog.

Example:

```

PROGRAM
  INCLUDE('FILE.INC')                !declare RelationManager class
Access:Client CLASS(FileManager)    !declare Access:Client class
Init      PROCEDURE
      END

Client    FILE,DRIVER('TOPSPEED'),PRE(CLI),THREAD !declare Client file
IDKey     KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record    RECORD,PRE()
ID        LONG
Name      STRING(20)
StateCode STRING(2)
      END
      END

CODE
Access:Client.Init                !initialize Access:Client obj
Relate:Client.Init(Access:Client,1) !init Relate:Client--use logout
Relate:Client.AddRelation( Relate:States )!relate Client to States file
!program code
Relate:Client.Kill                !shut down Relate:Client object
Access:Client.Kill                !shut down Access:Client object

```

See Also: Me



**ListLinkingFields (map pairs of linked fields)**

**ListLinkingFields**( *relationmanager*, *fieldpairs* [, *recursed*] )

---

<b>ListLinkingFields</b>	Maps pairs of linking fields between the primary file and a related file.
<i>relationmanager</i>	The label of the related file's RelationManager object.
<i>fieldpairs</i>	The label of the FieldPairsClass object to receive the linking field references.
<i>recursed</i>	A numeric constant, variable, EQUATE, or expression that indicates whether this method was called by itself (recursive). A value of zero (0) indicates a non-recursive call; a value of one (1) indicates a recursive call. This allows the method to get the list of linking fields from the <i>relationmanager</i> if necessary--since only one side of the relationship maintains the list of linking fields. If omitted, <i>recursed</i> defaults to zero (0). You should <i>always</i> omit this parameter when calling the ListLinkingFields method from your program.

The **ListLinkingFields** method maps pairs of linking fields between the primary file and a related file.

Implementation: The RelationManager object does not use the resulting mapped fields, but provides this mapping service for the ViewManager class, etc.

Example:

```
ViewManager.AddRange PROCEDURE(*? Field,RelationManager MyFile,RelationManager HisFile)
CODE
SELF.Order.LimitType = Limit:File !add range limit to view
SELF.Order.LimitType = Limit:File !set limit type: relationship
MyFile.ListLinkingFields(HisFile,SELF.Order.RangeList)!get linking fields
ASSERT(RECORDS(SELF.Order.RangeList.List)) !confirm Range limits exist
SELF.SetFreeElement !set free key element
```

**Open (open a file and any related files)****Open( *cascading* ), VIRTUAL, PROC**


---

<b>Open</b>	Opens this object's primary file (see <i>Me</i> ) and any related files.
<i>cascading</i>	A numeric constant, variable, EQUATE, or expression that indicates whether this method was called by itself (recursive). A value of zero (0) indicates a non-recursive call; value of one (1) indicates a recursive call. This allows the method to stop when it has processed all the related files in a circular relationship. If omitted, <i>cascading</i> defaults to zero (0). You should <i>always</i> omit this parameter when calling the Open method from your program.

The **Open** method Opens this object's primary file (see *Me*) and any related files, and returns a value indicating its success or failure.

Implementation: The Open method uses the FileManager.Open method to Open each file. The Open method returns the FileManager.Open method's return value. See *File Manager Class* for more information.

Return Data Type: BYTE

Example:

```
Relate:Customer.Open      !open Customer and related files
!program code            !process the files
Relate:Customer.Close    !Close Customer and related files
```

See Also: FileManager.Open

**Save (copy the current record and any related records)****Save, VIRTUAL**

The **Save** method copies the current record in the primary file and any related files. The copies may be used to detect subsequent changes to the current record or restore the current record to its previous state.

Implementation: The Save method uses the BufferedPairsClass.AssignLeftToBuffer method to Save each record. See Field Pairs Classes for more information.

**SetAlias (set a file alias)****SetAlias**( *relationmanager* )

---

**SetAlias**        Identifies an alias of this object's primary file.

*relationmanager*        The label of the alias file's RelationManager object.

The **SetAlias** method identifies an alias of this RelationManager object's primary file so that, when appropriate, the RelationManager only processes the file one time. For example, if both the primary file and its alias are part of a framed transaction (LOGOUT/COMMIT), the RelationManager recognizes the alias and appropriately applies the LOGOUT only to the primary file.

Example:

```
Customer FILE,DRIVER('TOPSPEED'),PRE(CLI),NAME('Customer') !declare Customer file
IDKey    KEY(CLI:ID),NOCASE,OPT,PRIMARY
Record   RECORD,PRE()
ID       LONG
Name     STRING(20)
        END
        END
```

```
Client FILE,DRIVER('TOPSPEED'),PRE(CUS),NAME('Customer') !declare Client 'alias'
IDKey    KEY(CUS:ID),NOCASE,OPT,PRIMARY
Record   RECORD,PRE()
ID       LONG
Name     STRING(20)
        END
        END
```

```
Relate:Customer.SetAlias( Relate:Client ) !Client = alias of Customer
```

**SetQuickScan (enable QuickScan on a file and any related files)****SetQuickScan( on [,propagate] ), VIRTUAL**


---

**SetQuickScan** Enables or disables quick scanning on this object's primary file and on the *propagated* related files.

*on* A numeric constant, variable, EQUATE, or expression that enables or disables quick scanning. A value of zero (0) disables quick scanning; a value of one (1) enables quick scanning.

*propagate* A numeric constant, variable, EQUATE, or expression that indicates which related files to include. Valid propagation options are none, one:many, many:one, and all. If omitted, *propagate* defaults to none.

The **SetQuickScan** method enables or disables quick scanning on this object's primary file and on the *propagated* related files.

Implementation: The SetQuickScan method SENDs the QUICKSCAN driver string to the file driver for each specified file. The QUICKSCAN driver string is supported by the ASCII, BASIC, and DOS drivers. See *Database Drivers* for more information.

Corresponding EQUATEs for the valid propagate options are declared in FILE.INC as follows:

```

ITEMIZE(0),PRE(Propagate)
None      EQUATE      !do primary file only, no related files
OneMany   EQUATE      !do 1-Many relations only
ManyOne   EQUATE      !do Many-1 relations only
All       EQUATE      !do all related files
END

```

Example:

```

Relate:Customer.SetQuickScan(1,Propagate:OneMany) !enable quickscan for 1:Many
Relate:Orders.SetQuickScan(1)                   !enable quickscan for primary
Relate:Orders.SetQuickScan(0)                   !disable quickscan for primary

```

**Update (update record subject to referential constraints)****Update( *fromform* ), VIRTUAL**


---

<b>Update</b>	Updates this object's primary file (see <i>Me</i> ) subject to the specified referential integrity constraints.
<i>fromform</i>	A numeric constant, variable, EQUATE, or expression that indicates whether this method was called from a (form) procedure with field history (restore) capability. A value of zero (0) indicates no restore capability; a value of one (1) indicates restore capability. This allows the method to issue an appropriate message when the update fails.

The **Update** method updates the current record in the primary file (see *Me*) applying any specified referential integrity constraints, then returns a value indicating its success or failure.

Implementation: Update constraints are specified by the AddRelation method and they apply to the values in the linking fields. If the constraint is RI:Restrict, the method does not update the current record if the change would result in orphaned child records. If the constraint is RI:Cascade, the method updates the primary file record as well as the linking field values in any related child records. If the constraint is RI:None, the method unconditionally updates only the primary file record. If the constraint is RI:Clear, the method unconditionally updates the primary file record, and clears the linking field values in any related child records.

Return Data Type: **BYTE**

Example:

**ChangeOrder ROUTINE**

```

IF Relate:Orders.Update(0)           !update subject to constraints
  MESSAGE('Update Failed')          ! if fails, acknowledge
ELSE                                  ! otherwise
  POST(Event:CloseWindow)           ! shut down
END

```

See Also: **AddRelation**



---

# ReportManager Class

## ReportManager Overview

The ReportManager is a WindowManager that uses a ProcessClass object to process report records in the background, and optionally uses a PrintPreviewClass object to provide a full-featured print preview facility.

## ReportManager Concepts

The ReportManager supports a batch report procedure, complete with progress window, print preview, DETAIL specific record filtering, and optimized sharing of machine resources.

## ReportManager Relationship to Other Application Builder Classes

The ReportManager is derived from the WindowManager because it supports a progress window to provide appropriate visual feedback to the end user (see *WindowManager* for more information).

The ReportManager uses the ProcessClass to manage the batch processing of the REPORT's underlying VIEW. The ReportManager optionally uses the PrintPreviewClass to provide a full-featured print preview for the report.

If your program instantiates the ReportManager, it should also instantiate the ProcessClass and may need the PrintPreviewClass as well. Much of this is automatic when you INCLUDE the ReportManager header (ABREPORT.INC) in your program's data section. See the Conceptual Example.

## ReportManager ABC Template Implementation

The Report Procedure template and the Report Wizard Utility template automatically generate all the code and include all the classes necessary to support your application's template generated reports.

These Report templates generate code to instantiate a ReportManager object called ThisWindow for each report procedure. The Report templates also instantiate a ProcessClass object and optionally a PrintPreviewClass object for the ThisWindow object to use.

The ThisWindow object supports all the functionality specified in the Report template's **Report Properties** dialog. See *Procedure Templates--Report* for more information.

## ReportManager Source Files

The ReportManager source code is installed by default to the Clarion \LIBSRC folder. The ReportManager source code and their respective components are contained in:

ABREPORT.INC	ReportManager declarations
ABREPORT.CLW	ReportManager method definitions

## ReportManager Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use and terminate a ReportManager and related objects.

This example uses the ReportManager object to preview a very simple report before printing it. The program specifies a maximized print preview window.

```

PROGRAM
  INCLUDE( 'ABREPORT.INC' )           !declare ReportManager &
                                      ! and PrintPreviewClass

MAP
END

GlobalErrors ErrorClass
VCRRequest LONG(0),THREAD

Customer FILE,DRIVER( 'TOPSPEED' ),PRE(CUS),THREAD
BYNUMBER KEY(CUS:CUSTNO),NOCASE,OPT,PRIMARY
Record RECORD,PRE()
CUSTNO LONG
Name STRING(30)
State STRING(2)
      END
      END

Access:Customer CLASS(FileManager)    !declare Access:Customer object
Init PROCEDURE
      END
Relate:Customer CLASS(RelationManager) !declare Relate:Customer object
Init PROCEDURE
      END
CusView VIEW(Customer)               !declare CusView VIEW
      END
PctDone BYTE                          !track progress variable

report REPORT,AT(1000,1542,6000,7458),PRE(RPT),FONT('Arial',10,,),THOUS
  HEADER,AT(1000,1000,6000,542),FONT(,,FONT:bold)
  STRING('Customers'),AT(2000,20),FONT(,14,,)
  STRING('Id'),AT(52,313),TRN
  STRING('Name'),AT(2052,313),TRN
  STRING('State'),AT(4052,313),TRN
  END
detail DETAIL,AT(,,6000,281),USE(?detail)
  STRING(@n-14),AT(52,52),USE(CUS:CUSTNO)
  STRING(@s30),AT(2052,52),USE(CUS:NAME)

```

```

        STRING(@s2),AT(4052,52),USE(CUS:State)
    END
    FOOTER,AT(1000,9000,6000,219)
        STRING(@pPage <<<#p),AT(5250,31),PAGE NO,USE(?PageCount)
    END
END

ProgressWindow WINDOW('Progress...'),AT(, ,142,59),CENTER,TIMER(1),GRAY,DOUBLE
    PROGRESS,USE(PctDone),AT(15,15,111,12),RANGE(0,100)
    STRING(' '),AT(0,3,141,10),USE(?UserString),CENTER
    STRING(' '),AT(0,30,141,10),USE(?TxtDone),CENTER
    BUTTON('Cancel'),AT(45,42),USE(?Cancel)
END

ThisProcedure CLASS(ReportManager)           !declare ThisProcedure object
Init          PROCEDURE(),BYTE,PROC,VIRTUAL
Kill          PROCEDURE(),BYTE,PROC,VIRTUAL
            END

CusReport     CLASS(ProcessClass)            !declare CusReport object
TakeRecord   PROCEDURE(),BYTE,PROC,VIRTUAL
            END

Previewer     PrintPreviewClass              !declare Previewer object
            ! for use with ThisProcedure

            CODE
            ThisProcedure.Run()              !run the report procedure

ThisProcedure.Init PROCEDURE()                !initialize ThisProcedure
ReturnValue   BYTE,AUTO
            CODE
            GlobalErrors.Init
            Relate:Customer.Init
            ReturnValue = PARENT.Init()
            SELF.FirstField = ?PctDone
            SELF.VCRRequest &= VCRRequest
            SELF.Errors &= GlobalErrors      !set error handler for ThisProcedure
            Relate:Customer.Open            !open Customer & related files
            OPEN(ProgressWindow)
            SELF.Opened=True
            !do report specific initialization
            CusReport.Init(CusView,Relate:Customer,?TxtDone,PctDone,RECORDS(Customer))
            CusReport.AddSortOrder(CUS:BYNUMBER) !set report sort order
            SELF.AddItem(?Cancel,RequestCancelled) !set action on cancel
            SELF.Init(CusReport,report,Previewer) !register Previewer & CusReport with
            !ThisProcedure

            SELF.Zoom = PageWidth
            Previewer.AllowUserZoom=True      !allow custom zoom factors

```

```
Previewer.Maximize=True           !initially maximize preview window
SELF.SetAlerts()                  !alert keys for ThisProcedure
RETURN ReturnValue

ThisProcedure.Kill PROCEDURE()     !shut down ThisProcedure
ReturnValue      BYTE,AUTO
CODE
  ReturnValue = PARENT.Kill()       !call base class shut down
  Relate:Customer.Close           !close Customer & related files
  Relate:Customer.Kill            !shut down Relate:Customer object
  GlobalErrors.Kill               !shut down GlobalErrors object
  RETURN ReturnValue

CusReport.TakeRecord PROCEDURE()   !do any per record process
ReturnValue      BYTE,AUTO
SkipDetails     BYTE
CODE
  ReturnValue = PARENT.TakeRecord() !standard process for each record
  PRINT(RPT:detail)                !print detail for each record
  RETURN ReturnValue

Access:Customer.Init PROCEDURE
CODE
  PARENT.Init(Customer,GlobalErrors)
  SELF.FileNameValue = 'Customer'
  SELF.Buffer &= CUS:Record
  SELF.Create = 0
  SELF.LazyOpen = False
  SELF.AddKey(CUS:BYNUMBER, 'CUS:BYNUMBER',0)

Relate:Customer.Init PROCEDURE
CODE
  Access:Customer.Init
  PARENT.Init(Access:Customer,1)
```

## ReportManager Properties

The ReportManager inherits all the properties of the WindowManager class from which it is derived. See *WindowManager Properties* for more information.

In addition to the inherited properties, the ReportManager contains the following properties:

### **Attribute (ReportAttributeManager object)**

**Attribute**                      **&ReportAttributeManager, PROTECTED**

The **Attribute** property is a reference to the ReportAttributeManager object that the ReportManager uses to manage the attribute processing of the target report's controls. The Attribute property applies any information set by the report control's EXTEND parameter.

Implementation:    The Init method sets the Attribute property.

See Also:    Init

### **BreakMan (BreakManagerClass object)**

**BreakMan**                      **&BreakManagerClass, PROTECTED**

The **BreakMan** property is a reference to the BreakManagerClass object that the ReportManager uses to manage the customized break processing of the target report. The BreakMan property applies any break information set by the report.

Implementation:    The AddItem method sets the BreakMan property.

See Also:    AddItem

## **DeferOpenReport (defer open)**

**DeferOpenReport**      **BYTE, PROTECTED**

The **DeferOpenReport** property controls whether the ReportManager opens the report with the Open method or delays opening the report until the first timer cycle. A value of one (1 or True) delays the open until the first timer cycle; a value of zero (0 or False) opens the report immediately.

The DeferOpenReport property gives you an opportunity to query the end user about items such as filters and sort orders before the report starts printing.

Implementation:      The Open and TakeWindowEvent methods implement the behavior specified by the DeferOpenReport property.

See Also:              Open, TakeWindowEvent

## **DeferWindow (defer progress window)**

**DeferWindow**              **USHORT, PROTECTED**

The **DeferWindow** property controls whether the ReportManager opens the progress window with the Open method or delays opening the progress window until the first timer cycle. A value of one (1 or True) delays the open until the first timer cycle; a value of zero (0 or False) opens the window immediately.

The DeferWindow property gives you an opportunity to leave the progress window closed until there are actual records to process.

Implementation:      The TakeWindowEvent method implements the behavior specified by the DeferWindow property.

See Also:              TakeWindowEvent

**KeepVisible (keep progress window visible)****KeepVisible          BYTE, PROTECTED**

The **KeepVisible** property controls whether the ReportManager closes the progress window before or after the Print Preview window is displayed. A value of one (1 or True) delays the close until after the Print Preview window is closed; a value of zero (0 or False) closes the window prior to the Print Preview window open.

The KeepVisible property gives you an opportunity to leave the progress window open and viewable with the Print Preview. Useful if you have results that you would like to display on the progress window.

Implementation:    The TakeCloseEvent method implements the behavior specified by the KeepVisible property.

See Also:          TakeCloseEvent

**OutputFileQueue (advanced report generation filenames)****OutputFileQueue          &OutputFileQueue, PROTECTED**

The **OutputFileQueue** property is a reference to a structure containing the full pathnames of the report's file advanced generation output for each report page. The ReportManager object uses this property to provide to output the report after previewing.

Implementation:    The ReportManager only uses the OutputFileQueue property if the ReportTarget property is active.

The OutputFileQueue structure is actually an EQUATE to the PrintPreviewFileQueue, and declared in EQUATES.CLW as follows:

```
PrintPreviewFileQueue    QUEUE, TYPE
Filename                STRING(FILE:MaxFileName)
PrintPreviewImage       STRING(FILE:MaxFileName), OVER(Filename)
                          END
```

## Preview (PrintPreviewClass object)

### **Preview                      &PrintPreviewClass, PROTECTED**

The **Preview** property is a reference to the PrintPreviewClass object the ReportManager uses to provide an online preview of the report.

Implementation:    The Init method sets the Preview property.

See Also:            Init

## PreviewQueue (report metafile pathnames)

### **PreviewQueue &PreviewQueue, PROTECTED**

The **PreviewQueue** property is a reference to a structure containing the full pathnames of the report's Windows metafiles (\*.WMF)--one metafile for each report page. The ReportManager object uses this property to provide an online preview of the report, and to print the report after previewing. See *PREVIEW* in the *Language Reference* for more information on report metafiles.

Implementation:    The ReportManager only uses the PreviewQueue property if the Preview property is set.

                      The PreviewQueue structure is declared in ABREPORT.INC as follows:

```
PreviewQueue    QUEUE,TYPE
Filename        STRING(128)
                  END
```

See Also:            Preview

## **Process (ProcessClass object)**

### **Process                      &ProcessClass, PROTECTED**

The **Process** property is a reference to the ProcessClass object the ReportManager uses to manage the "batch" processing of the report's data. The Process property applies sort orders, range limits, and filters as needed, and supplies appropriate visual feedback to the end user on the progress of the batch process.

Implementation:    The Init method sets the Process property.

See Also:            Init

## **QueryControl (query button)**

### **QueryControl SIGNED**

The **QueryControl** property contains the number of the reports query control. This is typically the value of the Query BUTTON's field equate. The ReportManager methods use this value to process the report based on a user defined query.

Implementation:    The Init method does not initialize the QueryControl property. You should initialize the QueryControl property after the Init method is called. See the *Conceptual Example*. On EVENT:Accepted for the QueryControl, the TakeEvent method calls the TakeLocate method to collect (from the end user) and apply the ad hoc query.

**The ABC ProcessReportQBEBUTTON template generates code to declare and support a QBE button.**

## Report (the managed REPORT)

### **Report      &WINDOW**

The **Report** property is a reference to the managed REPORT structure. The ReportManager uses this property to open, print, and close the REPORT.

Implementation:      The Init method sets the Report property.

See Also:              Init

## ReportTarget (IReportGenerator interface)

### **ReportTarget      &IReportGenerator, PROTECTED**

The **ReportTarget** property is a reference to the IReportGenerator interface that the ReportManager uses to manage the type of advanced report generation output (Text, PDF, HTML or XML). The ReportTarget property is set by the active TargetSelector property, which is a reference to the ReportTargetSelector Class.

Implementation:      The TakeAccepted method of the PrintPreview Class and the SetReportTarget method of the ReportManager sets the ReportTarget property.

See Also:              SetReportTarget

## SkipPreview (print rather than preview)

### **SkipPreview      BYTE**

The **SkipPreview** property controls whether the ReportManager provides an on-line preview when requested, or prints the report instead. A value of one (1 or True) prints rather than previews the report; a value of zero (0 or False) previews the report. The SkipPreview property is only effective if the Preview property is set.

The SkipPreview property lets you suppress the on-line print preview anytime before the AskPreview method executes.

Implementation:      The AskPreview method implements the behavior specified by the SkipPreview property.

See Also:              AskPreview, Preview

**TargetSelector (ReportTargetSelectorClass object)****TargetSelector            &ReportTargetSelectorClass, PROTECTED**

The **TargetSelector** property is a reference to the ReportTargetSelectorClass object that the ReportManager uses to determine whether to process the report as a standard printed report, or redirect the report output to another output type (Text, PDF, HTML or XML).

Implementation:    The Init method of the PrintPreview Class sets the TargetSelector property.

See Also:            PrintPreviewClass.Init , Init

**TargetSelectorCreated (report target active)****TargetSelectorCreated            BYTE, PROTECTED**

The **TargetSelectorCreated** property is used to signal the ReportManager that a report redirection to an alternative output (Text, PDF, HTML or XML) is active.

Implementation:    The Init method of the ReportManager sets the TargetSelectorCreated property.

See Also:            Init

## **TimeSlice (report resource usage)**

**TimeSlice**      **USHORT**

The **TimeSlice** property contains the amount of time in hundredths of a second the ReportManager tries to "fill up" for each processing "cycle." A cycle begins with an EVENT:Timer (see *TIMER* in the *Language Reference*), and ends about TimeSlice later. For example, for a TimeSlice of 100, the ReportManager processes as many records as it can within about 100/100 (one) second before yielding control back to the operating system. To provide efficient sharing of machine resources, we recommend setting the TIMER to something less than or equal to TimeSlice.

Implementation:      The Init method sets TimeSlice to one (100). The TakeWindowEvent method continuously adjusts the number of records processed per cycle to fill the specified TimeSlice--that is, to process as many records as possible within the TimeSlice. This provides both efficient report processing and reasonable sharing of machine resources, provided the TIMER value is less than or equal to the TimeSlice value. This leaves the user in control in a multi-tasking environment, especially when processing a large data set.

See Also:              Init, TakeWindowEvent

## **WaitCursor (activate Wait cursor during report processing)**

**WaitCursor**      **BYTE, PROTECTED**

The **WaitCursor** property is used to signal the ReportManager that a wait cursor will be active as the report is generating. A value of one (1 or True) activates the wait cursor; a value of zero (0 or False) uses the default.

Implementation:      The Ask method of the ReportManager implements the WaitCursor property.

See Also:              Ask

## **WMFParser (WMFDocumentParser object)**

**WMFParser**      **&WMFDocumentParser, PROTECTED**

The **WMFParser** property is a reference to the WMFDocumentparser object that the ReportManager uses to process embedded attributes and data from the generated Windows metafiles (\*.WMF).

Implementation:      The Init method of the Reportmanger sets the WMFparser property.

See Also:              Init

## **Zoom (initial report preview magnification)**

### **Zoom SHORT**

The **Zoom** property controls the initial zoom or magnification factor for the on-line report preview. A value of zero (0) uses the PrintPreviewClass object's default zoom setting. Any other value specifies the initial preview zoom factor.

The Zoom property lets you override the PrintPreviewClass object's default zoom setting. The PrintPreviewClass object determines the actual zoom factor applied.

The Zoom property is only effective if the Preview property is set.

Implementation: The AskPreview method implements the behavior specified by the Zoom property by passing the Zoom value to the PrintPreviewClass.Display method.

If the PrintPreviewClass object allows custom zoom factors, then the initial magnification equals the Zoom value (81 gives 81%, 104 gives 104%, etc.). If the PrintPreviewClass object only supports a limited set of discrete magnifications, the initial magnification is the one closest to the Zoom value (81 gives 75%, 104 gives 100%, etc.).

See Also: AskPreview, Preview, PrintPreviewClass.ZoomIndex

## ReportManager Methods

The ReportManager inherits all the methods of the WindowManager class from which it is derived. See *WindowManager Methods* for more information.

### ReportManager Functional Organization--Expected Use

As an aid to understanding the ReportManager, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the ReportManager methods.

#### **Non-Virtual Methods**

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the ReportManager object
Askv	display window and process its events
Killv	shut down the ReportManager object

v These methods are also Virtual.

#### **Virtual Methods**

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Ask	display window and process its events
AskPreview	preview or print the report
CancelPrintReport	cancel printing of report
EndReport	close the report
Kill	shut down the ReportManager object
Next	get next report record
Open	prepare progress window
OpenReport	prepare report for execution
PrintReport	prints the report
ProcessResultFiles	process redirected output
SetDynamicControlsAttributes	set control attributes prior to band printing
SetStaticControlsAttributes	set control attributes after report is opened
TakeNoRecords	handle empty report
TakeCloseEvent	process EVENT:CloseWindow events
TakeWindowEvent	process non-field events

**AddItem (program the ReportManager object)***AddItem(RecordProcessor)***AddItem** Adds specific functionality to the ReportManager.*RecordProcessor* The label of a RecordProcessor interface.

The AddItem method registers an ABC Library interface with the ReportManager object and adds the interface's specific functionality to the ReportManager.

**Ask (display window and process its events:ReportManager)****Ask, VIRTUAL**

The **Ask** method initiates the event processing (ACCEPT loop) for the report procedure. This virtual method handles any special processing immediately before or after the report procedure's event processing.

Implementation: The Run method calls the Ask method. The Ask method calls the PARENT.Ask method to manage the ACCEPT loop for the report procedure.

Example:

**MyReporter.Ask PROCEDURE**

**CODE**

```

SETCURSOR(CURSOR:Wait)      !special pre event handling code
PARENT.Ask                  !process events (ACCEPT)
SETCURSOR()                 !special post event handling code

```

See Also: WindowManager.Ask, WindowManager.Run

## AskPreview (preview or print the report)

### AskPreview, VIRTUAL

The **AskPreview** method previews or prints the report, only if the Preview property references an operative PrintPreviewClass object.

If the SkipPreview property is true, AskPreview does not preview the report, but prints it instead.

Implementation: The TakeCloseEvent method calls the AskPreview method to print or preview the report. The AskPreview method calls the PrintPreviewClass.Display method to preview the report.

Typically, the Init method sets the Preview reference.

Example:

```
MyReporter.TakeCloseEvent PROCEDURE
CODE
IF EVENT() = EVENT:CloseWindow
  SELF.AskPreview()
  IF ~SELF.Report&=NULL
    CLOSE(SELF.Report)
  END
END
RETURN Level:Benign
```

See Also: Ask, PrintPreviewClass.Display, Init, Preview, SkipPreview

## CancelPrintReport (cancel report printing)

### CancelPrintReport, VIRTUAL

The **CancelPrintReport** method is a virtual method that allows you to write any clean up code to execute when canceling in a Print Preview window.

Implementation: The AskPreview method CancelPrintReport method. The method is empty, but generates an embed point to allow custom code to be implemented.

Example:

ReportManager.AskPreview PROCEDURE

```
CODE
IF NOT SELF.Report &= NULL AND SELF.Response = RequestCompleted
  IF SELF.EndReport()=Level:Benign
    IF NOT SELF.Preview &= NULL
      IF CHOOSE (NOT SELF.SkipPreview, SELF.Preview.Display(SELF.Zoom), TRUE)
        SELF.PrintReport()
      ELSE
        SELF.CancelPrintReport()
      END
      FREE(SELF.Preview.ImageQueue)
    ELSE
      SELF.PrintReport()
      FREE(SELF.PreviewQueue)
    END
  ELSIF NOT SELF.Preview &= NULL
    FREE(SELF.Preview.ImageQueue)
  ELSE
    FREE(SELF.PreviewQueue)
  END
END
```

## EndReport (close the report)

### EndReport, VIRTUAL

The **EndReport** method prepares the report to close and returns a value indicating success or failure. This is a good place to add a final print summary or last page. Valid return value is

Level:Benign            report closed successfully

Implementation:    The EndReport method is called by the ReportManager's AskPreview method.

It checks to see if any final break logic needs to be processed by the BreakManager, then issues an ENDPAGE statement. You can use the method's embed point to perform any post processing needed prior to closing the report and accessing the Preview window.

Return Data Type:    BYTE

Example:

**ReportManager.AskPreview PROCEDURE**

```
CODE
IF NOT SELF.Report &= NULL AND SELF.Response = RequestCompleted
  IF SELF.EndReport()= Level:Benign
    IF NOT SELF.Preview &= NULL
      IF CHOOSE (NOT SELF.SkipPreview, SELF.Preview.Display(SELF.Zoom), TRUE)
        SELF.PrintReport()
      ELSE
        SELF.CancelPrintReport()
      END
      FREE(SELF.Preview.ImageQueue)
    ELSE
      SELF.PrintReport()
      FREE(SELF.PreviewQueue)
    END
  ELSIF NOT SELF.Preview &= NULL
    FREE(SELF.Preview.ImageQueue)
  ELSE
    FREE(SELF.PreviewQueue)
  END
END
```

**Init (initialize the ReportManager object)**

**Init**( *process object* [, *report*] [, *preview object*] )

---

<b>Init</b>	Initializes the ReportManager object.
<i>process object</i>	The label of the ProcessClass object the ReportManager uses to batch process the <i>report</i> VIEW and provide appropriate visual feedback to the end user on the progress of the <i>report</i> .
<i>report</i>	The label of the managed REPORT structure. If omitted, the ReportManager becomes a batch VIEW processor with automatic resource management.
<i>preview object</i>	The label of the PrintPreviewClass object the ReportManager uses to preview or print the <i>report</i> . If omitted, the ReportManager prints the report without generating preview image files.

The **Init** method does the report-specific initialization of the ReportManager object. This Init method is in addition to the Init method inherited from the WindowManager class which does general window procedure initialization.

Implementation: Typically, the Init method calls the Init(*process*, *report*, *preview*) method to do report-specific initialization. The Init method sets the Preview, Process, Report, and TimeSlice properties.

Example:

```
PrintPhones    PROCEDURE
report REPORT,AT(1000,1540,6000,7460),PRE(RPT)
detail  DETAIL,AT(, ,6000,280)
        STRING(@s20),AT(50,50,5900,170),USE(PHO:Number)
        END
        END
Previewer PrintPreviewClass      !declare Previewer object
Process   ProcessClass           !declare Process object
ThisWindow CLASS(ReportManager)  !declare derived ThisWindow object
Init      PROCEDURE(),BYTE,PROC,VIRTUAL
Kill      PROCEDURE(),BYTE,PROC,VIRTUAL
        END
!procedure data
CODE
  ThisWindow.Run                  !run the procedure (init,ask,kill)
ThisWindow.Init PROCEDURE()
CODE
!procedure code
  ThisWindow.Init(Process,report,Previewer)  !call the report-specific Init
!procedure code
```

See Also:      WindowManager.Init

## Kill (shut down the ReportManager object)

### Kill, VIRTUAL, PROC

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code. Kill returns a value to indicate the status of the shut down. Valid return values are:

Level:Benign	normal shut down
Level:Notify	no action taken

Implementation: The Run method calls the Kill method. If the Dead property is True, Kill returns Level:Notify and takes no other action. Otherwise, the Kill method, among other things, calls the WindowManager.Kill method.

Return value EQUATEs are declared in ABERROR.INC.

Return Data Type: BYTE

Example:

```
ThisWindow.Kill PROCEDURE()  
CODE  
IF PARENT.Kill() THEN RETURN Level:Notify.  
IF FilesOpened  
  Relate:Defaults.Close  
END  
IF SELF.Opened  
  INIMgr.Update('Main',AppFrame)  
END  
GlobalResponse = CHOOSE(LocalResponse=0,RequestCancelled,LocalResponse)
```

See Also: WindowManager.Dead, WindowManager.Run

**Next (get next report record)****Next, VIRTUAL, PROC**

The **Next** method gets the next report record and returns a value indicating whether the report is completed, cancelled, or in progress. Valid return values are:

Level:Benign	proceeding normally
Level:Notify	completed normally
Level:Fatal	cancelled or ended abnormally

Implementation: The **Next** method calls the **ProcessClass.Next** method to get the next report record. When the report is completed or canceled, the **Next** method sets the **Response** property and POSTs an **EVENT:CloseWindow** to end the progress window procedure.

Return Data Type: BYTE

Example:

```
ReportManager.Open PROCEDURE
CODE
PARENT.Open
SELF.Process.Reset
IF ~SELF.Next()
IF ~SELF.Report&=NULL
OPEN(SELF.Report)
IF ~SELF.Preview &= NULL
SELF.Report{PROP:Preview} = SELF.PreviewQueue.Filename
END
END
END
```

See Also: ProcessClass.Next, WindowManager.Response

**Open (a virtual to execute on EVENT:OpenWindow--ReportManager)****Open, VIRTUAL**

The **Open** method prepares the progress window for display. It is designed to execute on window opening events such as EVENT:OpenWindow.

Implementation: The TakeWindowEvent method calls the Open method. The Open method calls the WindowManager.Open method, then conditionally (based on the DeferOpenReport property) calls the OpenReport method to reset the ProcessClass object and get the first report record.

Example:

```
WindowManager.TakeWindowEvent PROCEDURE
RVal BYTE(Level:Benign)
CODE
CASE EVENT()
  OF EVENT:OpenWindow
    IF ~BAND(SELF.Inited,1)
      SELF.Open                !handle EVENT:OpenWindow
    END
    IF SELF.FirstField
      SELECT(SELF.FirstField)
    END
  OF EVENT:LoseFocus
    IF SELF.ResetOnGainFocus
      SELF.ForcedReset = 1
    END
  OF EVENT:GainFocus
    IF BAND(SELF.Inited,1)
      SELF.Reset
    ELSE
      SELF.Open                !handle EVENT:GainFocus
    END
  OF EVENT:Completed
    RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVal = SELF.TakeCloseEvent()
END
RETURN RVal
```

See Also: DeferOpenReport, OpenReport, WindowManager.Open, WindowManager.TakeWindowEvent

## OpenReport (prepare report for execution)

### OpenReport, PROC, PROTECTED, VIRTUAL

The **OpenReport** method prepares the report to execute and returns a value indicating success or failure. This is a good place to add any filters or keys specified at runtime. Valid return values are:

Level:Benign	report opened successfully
Level:Notify	no records found
Level:Fatal	failed, cause unknown

Implementation: The TakeWindowEvent method or the Open method calls the OpenReport method depending on the value of the DeferOpenReport property. The OpenReport method calls the Process.Reset method to reset the ProcessClass object, calls the Next method to get the first report record, then opens the REPORT structure.

The OpenReport method resets the DeferOpenReport property to zero so that if deferred, the OpenReport only happens with the first timer event.

Return Data Type: BYTE

Example:

```
ReportManager.Open PROCEDURE
  CODE
  PARENT.Open
  IF ~SELF.DeferOpenReport
    SELF.OpenReport      !call OpenReport if not deferred
  END

MyReportManager.TakeWindowEvent PROCEDURE
!procedure data
  CODE
  IF EVENT() = EVENT:Timer
    IF SELF.DeferOpenReport
      SELF.OpenReport      !if deferred, call OpenReport on timer
    ELSE
!procedure code
```

```

MyReportManager.OpenReport PROCEDURE
CODE
SELF.Process.SetFilter(UserFilter) !set dynamic filter
SELF.DeferOpenReport = 0
SELF.Process.Reset
IF ~SELF.Next()
IF ~SELF.Report&=NULL
OPEN(SELF.Report)
IF ~SELF.Preview &= NULL
SELF.Report{PROP:Preview} = SELF.PreviewQueue.FileName
END
END
END

```

See Also: DeferOpenReport, Next, Open, TakeWindowEvent, Process.Reset

## PrintReport (print the report)

### PrintReport, VIRTUAL

The **PrintReport** method is used to determine where a report output will be redirected. Based on the ReportTarget value, the report will be processed via the WMFparser, or as a standard report is handled by the ProcessResultFiles method.

Implementation: The PrintReport method is called by the ReportManager's AskPreview method.

Example:

```

ReportManager.AskPreview PROCEDURE
CODE
IF NOT SELF.Report &= NULL AND SELF.Response = RequestCompleted
IF SELF.EndReport()=Level:Benign
IF NOT SELF.Preview &= NULL
IF CHOOSE (NOT SELF.SkipPreview, SELF.Preview.Display(SELF.Zoom), TRUE)
SELF.PrintReport() !Print after previewing
ELSE
SELF.CancelPrintReport()
END
FREE(SELF.Preview.ImageQueue)
ELSE
SELF.PrintReport() !Print without previewing
FREE(SELF.PreviewQueue)
END
ELSIF NOT SELF.Preview &= NULL
FREE(SELF.Preview.ImageQueue)
ELSE
FREE(SELF.PreviewQueue)
END
END

```

**ProcessResultFiles (process generated output files)****ProcessResultFiles (OutputFileQueue), VIRTUAL**

The **ProcessResultFiles** virtual method is provided as an entry point to process the contents of the *OutputFileQueue* before the queue is freed and the report is either saved to disk in one of several formats, or sent to the printer.

Implementation: The ProcessResultFiles method is called by the ReportManager's PrintReport method. By default, the method returns Level:Benign. If any other value is returned, the report's printing will be aborted.

Return Data Type: BYTE

Example:

```
ReportManager.PrintReport          PROCEDURE ( )
Rt          BYTE
lIndex      SHORT
CODE
! Used select the target at the beginning if this is not a standard report
IF NOT SELF.ReportTarget &= NULL THEN
  IF RECORDS(SELF.PreviewQueue) THEN
    IF SELF.ReportTarget.SupportResultQueue()=True THEN
      SELF.ReportTarget.SetResultQueue(SELF.OutputFileQueue)
    END
    IF SELF.ReportTarget.AskProperties(False)=Level:Benign THEN
      SELF.WMFParser.Init(SELF.PreviewQueue, SELF.ReportTarget, SELF.Errors)
      IF SELF.WMFParser.GenerateReport()=Level:Benign THEN
        IF SELF.ReportTarget.SupportResultQueue()=True THEN
          Rt = SELF.ProcessResultFiles(SELF.OutputFileQueue) !after generating
        END
      END
    END
  END
ELSE
  FREE(SELF.OutputFileQueue)
  LOOP lIndex=1 TO RECORDS(SELF.PreviewQueue)
    GET(SELF.PreviewQueue,lIndex)
    IF NOT ERRORCODE() THEN
      SELF.OutputFileQueue.FileName = SELF.PreviewQueue.FileName
      ADD(SELF.OutputFileQueue)
    END
  END
  !prior to flushing to printer
  IF SELF.ProcessResultFiles(SELF.OutputFileQueue)=Level:Benign THEN
    SELF.Report{PROP:FlushPreview} = True
  ELSE
    SELF.Report{PROP:FlushPreview} = FALSE
  END
END
```

## SetReportTarget (set ReportGenerator target)

**SetReportTarget** ( *IReportGenerator pReportTarget* )

The **SetReportTarget** method sets the report's target output as passed by the *IReportGenerator* interface. This target can be text, HTML, PDF or XML formats.

## SetStaticControlsAttributes (set report's static controls)

**SetStaticControlsAttributes**, VIRTUAL

The **SetStaticControlsAttributes** method is a virtual method used to set any static attribute property on a report's controls prior to opening the report. The control must have the **EXTEND** attribute applied. This method is used to set attributes that will be recognized by the *WMFParser* to generate a particular output format.

Implementation: The **SetStaticControlsAttributes** method is a virtual method called by the *ReportManager*'s **Open** method.

Example:

```
ReportManager.Open PROCEDURE
CODE
  PARENT.Open
  IF ~SELF.DeferOpenReport
  SELF.OpenReport
  IF NOT SELF.OpenFailed THEN
    IF SELF.Report{PROPPRINT:Extend}=1 THEN
      SELF.SetStaticControlsAttributes()
    END
  END
END
```

## SetDynamicControlsAttributes (set report's static controls)

### SetDynamicControlsAttributes, VIRTUAL

The **SetDynamicControlsAttributes** method is a virtual method used to set any dynamic attribute property on a report's controls as each record is processed. The control must have the EXTEND attribute applied. This method is used to set attributes that will be recognized by the WMFParser to generate a particular output format.

Implementation: The **SetDynamicControlsAttributes** method is a virtual method called by the ReportManager's TakeRecord method.

Example:

```
ReportManager.TakeRecord PROCEDURE
I          LONG,AUTO
RVal      BYTE(Level:Benign)
CODE
  IF ~SELF.BreakMan &= NULL THEN
    SELF.BreakMan.AskBreak()
  END
  IF SELF.Report{PROPPRINT:Extend}=1 THEN
    SELF.SetDynamicControlsAttributes()
  END
  RVal = SELF.Process.TakeRecord()
  DO CheckState
  LOOP I = 1 TO RECORDS(SELF.Processors)
    GET(SELF.Processors,I)
    RVal = SELF.Processors.P.TakeRecord()
    DO CheckState
  END
  IF SELF.Next() THEN
    TARGET{PROP:Timer} = 0
    RETURN Level:Notify
  END
RETURN Rval
```

## TakeAccepted (process Accepted event)

### TakeAccepted, PROC, PROTECTED, DERIVED

The **TakeAccepted** method processes the accepted event for the ReportManager. This occurs after each record is read. This method processes the record according to the end user query (filter) set by the ProcessReportQBEButton.

Return Data Type: BYTE

See Also: ProcessClass.TakeLocate

## TakeCloseEvent (a virtual to process EVENT:CloseWindow)

### TakeCloseEvent, VIRTUAL, PROC

The **TakeCloseEvent** method handles EVENT:CloseWindow for the ReportManager and returns a value indicating whether window ACCEPT loop processing is complete and should stop.

TakeCloseEvent returns returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeEvent method calls the TakeCloseEvent method. The TakeCloseEvent method calls the AskPreview method to preview or print the report, then closes the report.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I   USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;   RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;  RVal = SELF.TakeRejected()
    OF EVENT:Selected;  RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed; RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also: AskPreview, WindowManager.TakeEvent

## TakeNoRecords (process empty report)

### TakeNoRecords, VIRTUAL

The **TakeNoRecords** method implements any special processing required for a report with no records.

Implementation: The **OpenReport** method calls the **TakeNoRecords** method. The **TakeNoRecords** method issues a message indicating there are no records, and therefore no report.

You can use the **TakeNoRecords** method to print a page indicating an empty report. The default action is to issue the message and print nothing.

Example:

```
MyReporttr.TakeNoRecords PROCEDURE
CODE
PARENT.TakeNoRecords
CLI:CustomerName = 'No Customers'
PRINT(CustomerDetail)
```

See Also: OpenReport

## TakeRecord(process each record)

### TakeRecord, VIRTUAL, PROC

The **TakeRecord** method processes each item in the result set. It returns a value indicating whether processing should continue or should stop. **TakeRecord** returns **Level:Benign** to indicate processing should continue normally; it returns **Level:Notify** to indicate processing is completed and should stop.

Return Data Type: BYTE

See Also: ProcessClass.TakeRecord

**TakeWindowEvent (a virtual to process non-field events:ReportManager)****TakeWindowEvent, VIRTUAL, PROC**

The **TakeWindowEvent** method processes all non-field events for the progress window and returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeWindowEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeEvent method calls the TakeWindowEvent method.

The TakeWindowEvent method processes EVENT:Timer events for the report. The TakeWindowEvent method either calls OpenReport (if DeferOpenReport is True) or begins processing a "cycle" of report records. Each timer event begins a "cycle" of report record processing which ends about TimeSlice later.

TakeWindowEvent calls the TakeRecord method and the Next method for each record within a processing cycle.

TakeWindowEvent adjusts the number of records processed per cycle to fill the TimeSlice and optimize sharing of machine resources.

Finally, TakeWindowEvent calls the WindowManager.TakeWindowEvent method to handle any other non-field events.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;    RVal = SELF.TakeRejected()
    OF EVENT:Selected;    RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;  RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also:      DeferOpenReport, Next, TimeSlice, TakeRecord, WindowManager.TakeEvent,  
WindowManager.TakeWindowEvent

# RuleManager

## Overview

The Rule Manager classes work together to provide a central repository for business rules logic and a methodology for checking and responding to business rules in Clarion procedures. The Rule Manager classes also provide the option of visual indicators when a rule has been broken and conditional hiding or disabling of controls in the presence of broken rules.

### Why Use RuleManager?

A primary requirement of a database system is the validation of data, or, more generally, enforcement of business rules. A common methodology is to put all validation code in the update procedure. However, the same tables may be handled by multiple procedures, *e.g.*, manual input and batch processes, and individual columns may also be updated by procedures primarily concerned with other tables.

The result is that code for validating a particular column is repeated in multiple places throughout the application, providing opportunities for inconsistency and, as the system grows, creating a maintenance nightmare.

The Rule Manager Classes and associated templates provide a mechanism for centralizing validation code so that validation, wherever it is needed, is always provided in exactly the same way by code which can be maintained in a single, central location.

In addition, the Rule Manager Classes provide facilities for identifying broken rules in a convenient manner and enforcing rules through selective disabling or hiding of controls.

## RuleManager Concepts

Rule Manager is implemented by a set of three classes:

- The **Rule** CLASS stores the definition of, and provides a method for testing, a single rule.
- The **RulesCollection** CLASS manages a collection of related rules, and provides methods for defining those rules, testing them singly or jointly, counting and/or displaying broken rules, and hiding or disabling controls when one or more rules is broken.
- The **RulesManager** CLASS manages multiple instances of RulesCollection.

These objects work together to implement Rule Manager features.

## A Collection Class of Rules

A RulesCollection object is a collection class that is composed of an indefinite number of Rule objects. Each Rule object contains a single expression which when evaluated indicates whether the rule is broken or not. Rules can be explicitly checked by name, numeric id or all at once.

## Visual Indicators

Each Rule object within the RulesCollection object has the option of displaying a small button with an image next to a specified control when the rule has been checked and found to be broken.

## Evaluation of rules

**When a rule is checked, the EVALUATE() function is called internally by the RuleUsBroken() method. Therefore all variables used by a BrokenRulesManager object need to be explicitly bound. If the result of the EVALUATE() returns FALSE, it means that the rule has been broken.**

## Responding to Broken Rules

RulesCollection provides the ability to keep a tally or how many rules are broken at one time. It does not explicitly re-check all the rules, but rather counts all the rules that have already been flagged as broken. This will provide you with a single, clean reference for determining whether a procedure is in a valid state or not.

## Rule Manager Relationship to Other Application Builder Classes

The Rule Manager Classes work in conjunction with the WindowManager Class to provide visual error indicators. Template-placed code allows Rule Manager to work within the ACCEPT loop of WindowManager to handle events related to error indicators.

## RuleManager ABC Template Implementation

Three templates support the use of the Rule Manager classes. They are:

- The **Global Business Rules Manager**, which establishes logical connections between business rules and particular table columns, other fields, and controls.
- The **Local Business Rules Manager**, which implements global rules wherever relevant items are populated and allows for the addition of local rules having effect in only one procedure. The Local Business Rules Manager is automatically added to every procedure in any application which contains the Global Business Rules Manager.
- A code template, the **Error Handler for Business Rules**, which provides enhanced functionality where needed.

### RuleManager Source Files

The Rule Manager Classes source code is installed by default to the Clarion \LIBSRC folder. The Rule Manager source code and its respective components are contained in:

ABRULE.INC	Rule Manager Class declarations
ABRULE.CLW	Rule Manager Class method definitions

### RuleManager Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use and terminate a RulesCollection and related Rule objects in a standard ABC Window procedure. This example defines several rules and checks them at critical points. Each rule is bound to a specific control (this is optional) and when the rule is checked and found to be broken, a graphic error indicator is displayed next to the associated control. The OK button is disabled when there are broken rules.

```
MEMBER('app.clw')           ! This is a MEMBER module

INCLUDE('ABRULES.INC'),ONCE

INCLUDE('ABTOOLBA.INC'),ONCE
INCLUDE('ABWINDOW.INC'),ONCE

MAP
  INCLUDE('APP001.INC'),ONCE  !Local module procedure declarations
```

```

END

Main PROCEDURE                                !Generated from procedure template - Window

CusName          STRING(20)
CusAddress       STRING(20)
CusPhone         STRING(10)
CustomerRules &RuleManager !Rule manager for Rules for the customer
Window WINDOW('Example of using RulesManager'),AT(,,169,124), FONT(,,,CHARSET:ANSI),|
GRAY,DOUBLE
SHEET,AT(3,4,159,116),USE(?Sheet1)
TAB('Customer Info'),USE(?Tab1)
SHEET,AT(8,26,149,62),USE(?Sheet2)
TAB('Name'),USE(?Tab3)
ENTRY(@s20),AT(51,55,60,10),USE(CusName),IMM
PROMPT('Cus Name: '),AT(13,55),USE(?CusName:Prompt)
END
TAB('Address'),USE(?Tab4)
PROMPT('Cus Address: '),AT(12,53),USE(?CusAddress:Prompt)
ENTRY(@s20),AT(69,53,60,10),USE(CusAddress),IMM
END
TAB('Phone'),USE(?Tab5)
PROMPT('Cus Phone: '),AT(12,52),USE(?CusPhone:Prompt)
ENTRY(@s10),AT(69,50,60,10),USE(CusPhone),IMM
END
END
BUTTON('OK'),AT(118,96,32,14),USE(?Button:OK),STD(STD:Close)
BUTTON('View Customer Broken Rules'),AT(8,95,101,14),USE(?Button:ListAll)
END
END
END

ThisWindow          CLASS(WindowManager)
Init                PROCEDURE(),BYTE,PROC,DERIVED
Kill                PROCEDURE(),BYTE,PROC,DERIVED
TakeAccepted        PROCEDURE(),BYTE,PROC,DERIVED
TakeFieldEvent      PROCEDURE(),BYTE,PROC,DERIVED
TakeNewSelection    PROCEDURE(),BYTE,PROC,DERIVED
END

Toolbar            ToolbarClass

CODE
GlobalResponse = ThisWindow.Run()

ThisWindow.Init PROCEDURE

ReturnValue          BYTE,AUTO

CODE
GlobalErrors.SetProcedureName('Main')

```

```

SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()

!Bind the variables used by RulesManager
BIND('CusName',CusName)                !RulesManager Hotfield
BIND('CusAddress',CusAddress)          !RulesManager Hotfield
BIND('CusPhone',CusPhone)             !RulesManager Hotfield

!Define RulesManager
CustomerRules &= New(RuleManager)
CustomerRules.SetErrorImage('~SMCROSS.ICO')
CustomerRules.SetDescription('Rules for the customer')

!Defining rules in RulesManager
CustomerRules.AddRule|
('CusNameReq','Customer name is required','len(clip(CusName))>0',?CusName,3)
  CustomerRules.AddRule|
('Addreq','Customer address is required','len(clip(CusAddress))>0',?CusAddress,3)
CustomerRules.AddRule('PhoneReq','Phone is required','Len(clip(CusPhone))>0',?CusPhone,3)

IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?CusName
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(Toolbar)
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
OPEN(Window)
SELF.Opened=True
!Check all Rules in RulesManager and show error indicators
CustomerRules.CheckAllRules(1)

SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE

ReturnValue          BYTE,AUTO

CODE
ReturnValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
GlobalErrors.SetProcedureName

!UnBind the variables used by RulesManager
UNBIND('CusName')                !RulesManager Hotfield
UNBIND('CusAddress')            !RulesManager Hotfield
UNBIND('CusPhone')             !RulesManager Hotfield
Dispose(CustomerRules)

RETURN ReturnValue

```

```

ThisWindow.TakeAccepted PROCEDURE

ReturnValue          BYTE,AUTO

Looped BYTE
CODE
!Pass the Accepted control to RulesManager for processing in case
!the control clicked was an error indicator. If it was an error indicator,
!a MessageBox containing the description of the broken rule will be displayed
CustomerRules.TakeAccepted(Accepted())
LOOP
  IF Looped
    RETURN Level:Notify
  ELSE
    Looped = 1
  END
ReturnValue = PARENT.TakeAccepted()
CASE ACCEPTED()

  OF ?Button:ListAll
    ThisWindow.Update

!This will Cause a window to popup that contains a list of all broken rules,
!and if the user double clicked one of the rules in the list, the relevant
!control will be selected.

Select(CustomerRules.EnumerateBrokenRules|
(CustomerRules.GetDescription(),CustomerRules.GetErrorImage()))

  END
  RETURN ReturnValue
END
ReturnValue = Level:Fatal
RETURN ReturnValue

ThisWindow.TakeFieldEvent PROCEDURE

ReturnValue          BYTE,AUTO

Looped BYTE
CODE
LOOP
  IF Looped
    RETURN Level:Notify
  ELSE
    Looped = 1
  END

!Disable the save button if there are any broken rules
?Button:OK{Prop:Disable} = CustomerRules.BrokenRuleCount()

!Hide the ListAll button if there are no broken rules

```

```
?Button:ListAll{Prop:Hide} = Choose(CustomerRules.BrokenRuleCount() = 0)

ReturnValue = PARENT.TakeFieldEvent()
RETURN ReturnValue
END
ReturnValue = Level:Fatal
RETURN ReturnValue
```

ThisWindow.TakeNewSelection PROCEDURE

```
ReturnValue          BYTE,AUTO
```

```
Looped BYTE
```

```
CODE
```

```
LOOP
```

```
IF Looped
```

```
RETURN Level:Notify
```

```
ELSE
```

```
Looped = 1
```

```
END
```

```
ReturnValue = PARENT.TakeNewSelection()
```

```
CASE FIELD()
```

```
OF ?CusName
```

```
UPDATE
```

```
CustomerRules.CheckRule('CusNameReq',1)!Check for broken rule in RulesManager
```

```
OF ?CusAddress
```

```
UPDATE
```

```
CustomerRules.CheckRule('AddReq',1) !Checking for broken rule in RulesManager
```

```
OF ?CusPhone
```

```
UPDATE
```

```
CustomerRules.CheckRule('PhoneReq',1)!Checking for broken rule in RulesManager
```

```
END
```

```
RETURN ReturnValue
```

```
END
```

```
ReturnValue = Level:Fatal
```

```
RETURN ReturnValue
```

## Implementation Steps using hand code

Although there is a powerful template that is included with Clarion to help you implement the RuleManager, there are times when you may need to hand code its properties and methods into your source. The following are the recommend steps to implementing the RuleManager in your hand coded projects.

1. Identify specific rules and assign each rule a name. For example, if *Cus:Name* is required, call it *CusNameReq*.

2. For each rule, write a line of code that returns the value of TRUE when the rule is unbroken. For example, if *Cus:Name* is required, the corresponding code will be:

```
Len(clip(Cus:Name)) > 0
```

If *Cus:Address* is required if *Cus:Name* <> "Unknown", the corresponding code will be

```
Choose(Upper(clip(Cus:Name))<<>'UNKNOWN' and len(clip(Cus:Address))=0,0,1)
```

3. Bind all of the variables used in each expression.

```
BIND('Cus:Name',Cus:Name)
BIND('Cus:Address',Cus:Address)
```

4. Define a RuleManager Object.

```
CustomerRules &RuleManager
```

5. Instantiate the RuleManager Object

```
CustomerRules &= New(RuleManager)
CustomerRules.SetErrorImage('~SMCROSS.ICO')
CustomerRules.SetDescription('Rules for the customer')
```

6. Define the rules for the RulesManager

```
CustomerRules.AddRule('CusNameReq','Customer name is required' ,|
  'len(clip(CusName))>0',?CusName,3)
!A small button with the icon SMCROSS.ICO' will be displayed 3 pixels to
the left of !?CusName when the expression evaluates to false

CustomerRules.AddRule|
('Addreq','Customer address is required if customer name is not
"Unknown"',|
  'Choose(Upper(clip(CusName))<<>'UNKNOWN' and |
  len(clip(CusAddress))=0,0,1)',?CusAddress,3)
!A small button with the icon SMCROSS.ICO' will be displayed 3 pixels to
the left of !?CusAddress when the expression evaluates to false
```

7. Check the rules.

```
CustomerRules.CheckAllRules(1)           !Checks all the rules
CustomerRules.CheckRule('CusNameReq',1) !Checks specific rule
```

8. Trap a mouse click on the error indicator button.

```
CustomerRules.TakeAccepted(Accepted())

!If a description is provided with the corresponding error, a
!message with the corresponding error will appear
```

9. Count the Broken Rules.

```
?OK{Prop:Disable} = CustomerRules.BrokenRuleCount()
!The ?OK button is disabled when there are broken rules
```

10. View All Broken Rules.

```
Select(CustomerRules.EnumerateBrokenRules|
(CustomerRules.GetDescription(),CustomerRules.GetErrorImage()))

!Call a popup listbox of broken rules, and use the default RulesManager
icon as an icon. !If the user double clicks one of the broken rules, the
corresponding control will be !selected.
```

## Rule Class Properties

The Rule Class contains no public properties.

## Rule Class Methods

### Access Methods

Although all of the properties of the Rule class are PRIVATE, public access to these properties is provided by a set of methods:

**GetName( )** returns a STRING containing the Name property.

**GetExpression( )** returns a STRING containing the Expression property.

**GetControlNum( )** returns a LONG containing the ControlNum Property

**GetDescription( )** returns a STRING containing the Description property

**GetErrorIndicator( )** returns a LONG containing the ErrorIndicator property

**GetErrorImage( )** returns a STRING containing the ErrorImage property

**GetIsBroken( )** returns a LONG containing the IsBroken property

**GetOffsetRight( )** returns a LONG containing the OffsetRight property

**SetName( string )** sets the Name property

**SetExpression( string )** sets the Expression property

**SetControlNum( value )** sets the ControlNum property

**SetDescription( string )** sets the Description property

**SetErrorIndicator( value )** sets the ErrorIndicator property

**SetErrorImage( string )** sets the ErrorImage property

**SetIsBroken( value )** sets the IsBroken property

**SetOffsetRight( value )** sets the OffestRight property

---

**Setxxx/Getxxx** Sets or retrieves a property

*string* A string constant, variable, EQUATE, or expression that contains a value appropriate to the property being set.

*value* A LONG constant, variable, EQUATE, or expression that contains a value appropriate to the property being set.

Example:

```
RulesCollection.AddRule PROCEDURE (STRING RuleName, STRING RuleDescription, |
                          STRING RuleExpression, <LONG ControlNum>, |
                          LONG OSR=3)
Counter      LONG
Found        BYTE
CODE
SELF.BrokenRuleQueue.BrokenRuleInstance &= NEW(Rule)
SELF.BrokenRuleQueue.BrokenRuleInstance.SetName(RuleName)
SELF.BrokenRuleQueue.BrokenRuleInstance.SetExpression(RuleExpression)
SELF.BrokenRuleQueue.BrokenRuleInstance.SetDescription(RuleDescription)
SELF.BrokenRuleQueue.BrokenRuleInstance.SetErrorImage(SELF.ErrorImage)
IF ~OMITTED(5)
  SELF.BrokenRuleQueue.BrokenRuleInstance.SetControlNum(ControlNum)
  SELF.BrokenRuleQueue.BrokenRuleInstance.SetErrorIndicator(ControlNum+1000)
  SELF.BrokenRuleQueue.BrokenRuleInstance.SetOffsetRight(OSR)
END
ADD(SELF.BrokenRuleQueue)
```

## SetGlobalRule (post address to GlobalRule)

### SetGlobalRule

---

**SetGlobalRule** Posts address of current Rule object to SELF.GlobalRule.

The **SetGlobalRule** method tests the GlobalRule reference property for NULL. If the reference is not NULL, the address of the current instance of Rule (ADDRESS(SELF)) is posted to the referenced variable.

Example:

```
Rule.RuleIsBroken PROCEDURE(Byte DisplayIndicator)
RetVal              LONG
EvaluateResults byte
CODE
SELF.SetGlobalRule
EvaluateResults = EVALUATE(SELF.Expression)
IF INLIST(ERRORCODE(),800,810,1011)
    RetVal = ERRORCODE()
    STOP('Evaluate Syntax error')
ELSE
    RetVal = CHOOSE(EvaluateResults = 0,1,0)
    SELF.SetIsBroken(RetVal)
    IF DisplayIndicator
        SELF.SetIndicator(RetVal)
    END
END
SELF.ResetGlobalRule
RETURN(RetVal)
```

See Also: ResetGlobalRule, RuleIsBroken

## ResetGlobalRule (clear address in GlobalRule)

### ResetGlobalRule

---

**ResetGlobalRule** Sets SELF.GlobalRule to zero.

The **ResetGlobalRule** method tests the GlobalRule reference property for NULL. If the reference is not NULL, a zero is posted to the referenced variable.

Example:

```

Rule.RuleIsBroken PROCEDURE(Byte DisplayIndicator)
RetVal              LONG
EvaluateResults byte
CODE
SELF.SetGlobalRule
EvaluateResults = EVALUATE(SELF.Expression)
IF INLIST(ERRORCODE(),800,810,1011)
    RetVal = ERRORCODE()
    STOP('Evaluate Syntax error')
ELSE
    RetVal = CHOOSE(EvaluateResults = 0,1,0)
    SELF.SetIsBroken(RetVal)
    IF DisplayIndicator
        SELF.SetIndicator(RetVal)
    END
END
SELF.ResetGlobalRule
RETURN(RetVal)

```

See Also: RuleIsBroken

## RuleIsBroken (test rule and return result)

### **RuleIsBroken**( *display* )

---

**RuleIsBroken** Tests the rule and returns the result

*display* An integer constant, variable, EQUATE, or expression that indicates whether to display an error indicator if the rule is broken.

The **RuleIsBroken** method uses EVALUATE to test SELF.Expression. If the result is zero, the rule is considered broken. Otherwise, the rule is considered sound. The method returns True if the rule is broken, False if not. If the value of *display* is non-zero, an error indicator is set using the SetIndicator method.

Implementation: The RuleIsBroken method relies on the EVALUATE function to test a string containing an expression. Accordingly, all names used in the expression must be bound. If a procedure name is used, it must return a STRING, REAL or LONG value. If the procedures accept parameters, those parameters must be STRINGS passed by value and may not be omissible.

The RuleIsBroken method tests explicitly for errors related to the EVALUATE process and ignores errors which might be posted by called procedures. These procedures must therefore handle any errors which may be posted during their execution.

The RuleIsBroken method calls SELF.SetGlobalRule before evaluating the expression and SELF.ResetGlobalRule immediately after evaluating the expression. During the EVALUATE process, then, a called procedure can use the address posted to call back to the current Rule instance.

Return Data Type: BYTE

Example:

```
RulesCollection.CheckAllRules Procedure(<BYTE DisplayIndicator>)
RetVal          LONG
LBR             &Rule
Counter        LONG
Recs           LONG
lIsBroken      BYTE
CODE
  Recs      = SELF.RuleCount()
  Retval    = 0
  LOOP Counter = 1 TO Recs
    LBR     &= SELF.Item(Counter)
    lIsBroken= LBR.RuleIsBroken(DisplayIndicator)
    RetVal +=lIsBroken
  END
  IF SELF.ChangeControlsStatus
    SELF.SetControlsStatus()
  END
RETURN(RetVal)
```

See Also:      SetGlobalRule, ResetGlobalRule, SetIndicator

## SetIndicator (set error indicator)

### **SetIndicator**( *broken* )

The **SetIndicator** method creates or destroys an error indicator as appropriate.

*broken*            An integer constant, variable, EQUATE, or expression that indicates whether the current rule is broken.

Implementation:    The SetIndicator method destroys any previously created error indicator for this rule and conditionally creates a new one.

Example:

```
RetVal = CHOOSE(EvaluateResults = 0,1,0)
SELF.SetIsBroken(RetVal)
IF DisplayIndicator
  SELF.SetIndicator(RetVal)
END
```

## RulesCollection Class Properties

The RulesCollection Class contains no public properties.

## RulesCollection Class Methods

### Access Methods

Although all of the properties of the RulesCollection class are PRIVATE, public access to these properties is provided by a set of methods:

**GetDescription()** returns a STRING containing the Description property

**GetErrorImage()** returns a STRING containing the ErrorImage property

**GetChangeControls()** returns a BYTE containing the ChangeControlsStatus property

**SetDescription( string )** sets the Description property

**SetErrorImage( string )** sets the ErrorImage property

**SetChangeControls( status )** sets the ChangeControlsStatus property

---

**Setxxx/Getxxx** Sets or retrieves a property

*string* A string constant, variable, EQUATE, or expression that contains a value appropriate to the property being set.

*status* A Boolean constant, variable, EQUATE, or expression that contains either True or False.

Example:

```
Customer.SetErrorImage('~BRuleNo.ico')
Customer.SetDescription('Customer File Rules')
Customer.AddRule('ShortNameRequired','Short name is required', |
                 'cus:ShortName <<> '''' OR CheckShortName()' |
                 ,?cus:ShortName,3)
```

## Construct (initialize RulesCollection object)

### **Construct**

**Construct** Initializes the RulesCollection object. The Construct method is executed automatically when the object is instantiated.

## Destruct (shut down RulesCollection object)

### Destruct

**Destruct** Performs necessary cleanup prior to the disposal of the RulesCollection object. The Destruct method is executed automatically when the object is disposed of.

## RuleCount (count rules in the collection)

### RuleCount()

**RuleCount** Returns the number of rules being monitored by this RulesCollection object.

Return Data Type: LONG

Example:

```
NumberOfRules = SELF.RuleCount()
```

## BrokenRuleCount (count rules in the collection which are broken)

### BrokenRuleCount()

**BrokenRuleCount** Returns the number of rules being monitored by this RulesCollection object that are broken.

Return Data Type: LONG

Example:

```
CustomerRules RulesCollection  
CODE  
... code omitted ...  
BrokenRules = CustomerRules.BrokenRuleCount()
```

## AddRule (add a rule to this collection)

**AddRule**( *name,description,expression,control,offset* )

---

<b>AddRule</b>	Adds and initializes a rule to the collection being managed by this RuleManager object
<i>name</i>	A string constant, variable, EQUATE, or expression that contains a name for this rule. If this name is not unique, results may be unpredictable.
<i>description</i>	A string constant, variable, EQUATE, or expression that contains a description of this rule. The primary use of this description is as a message identifying a broken rule. The description should be worded with this use in mind.
<i>expression</i>	A string constant, variable, EQUATE, or expression that contains the logical expression which defines this rule. If the expression evaluates to False, the rule is considered broken.
<i>control</i>	A numeric constant, variable, EQUATE, or expression that specifies the Field Equate of the control linked to this rule. If the rule is broken, an error indicator will be placed to the right of this control.
<i>offset</i>	A numeric constant, variable, EQUATE, or expression that specifies the distance, in dialog units, between the right side of the linked control and the error indicator.

The **AddRule** method creates a Rule object and adds it to the broken rule queue.

Example:

Customer RulesCollection

```

CODE
Customer.SetErrorImage('~BRuleNo.ico')
Customer.SetDescription('Customer File Rules')
Customer.AddRule('ShortNameRequired','Short name is required', |
    'cus:ShortName <<> '''' OR CheckShortName()', |
    ?cus:ShortName,3)
Customer.AddRule('Company Boolean','Must select individual (0) or company (1)', |
    'INRANGE(cus:Company,0,1)',?cus:Company,3)
Customer.AddRule('CompanyNameRequired','Company name is required', |
    'cus:CompanyName <<> '''' OR NOT cus:Company',?cus:CompanyName,3)
Customer.AddRule('LastNameRequired','Last name is required', |
    'cus:LastName <<> '''' OR cus:Company',?cus:LastName,3)
Customer.AddRule('CityRequired','City is required','cus:City <<> ''''',?cus:City,3)
Customer.AddRule('StateRequired','State is required','cus:State <<> ''''',?cus:State,
Customer.AddRule('ZipOK','Postal code must have valid format.',
    'CheckCustomerZip()',?cus:ZipCode,3)
Customer.AddRule('CreditLimit','Credit limit must not exceed $1000.00',
    'cus:CreditLimit =<< 1000',?cus:CreditLimit,3)

```

## AddControl (add managed control )

### **AddControl( *feq,action* )**

---

**AddControl** Adds a control to the controls queue so that a specified action is taken when any rule in the managed collection is broken.

*feq* A numeric constant, variable, EQUATE, or expression that indicates the Field Equate number of a control whose hidden or disabled status will be determined by the existence of one or more broken rules.

*action* A numeric constant, variable, EQUATE, or expression that indicates the action to be taken with respect to this control when a rule is broken, defined as follows:

RuleAction:None	EQUATE(0)
RuleAction:Hide	EQUATE(1)
RuleAction:UnHide	EQUATE(2)
RuleAction:Disable	EQUATE(3)
RuleAction:Enable	EQUATE(4)

Example:

```
Customer.SetErrorImage('~BRuleNo.ico')
Customer.SetDescription('Customer File Rules')
Customer.AddRule('StateRequired','State is required','cus:State <<> ''',?cus:State,
Customer.AddRule('ZipOK','Postal code must have valid format.',
    'CheckCustomerZip()',?cus:ZipCode,3)
Customer.AddRule('CreditLimit','Credit limit must not exceed $1000.00',
    'cus:CreditLimit =<< 1000',?cus:CreditLimit,3)
Customer.AddControl(?Ok,RuleAction:Disable) ! Disable OK button if any rule broken
```

## AddControlToRule (add managed control )

### AddControlToRule( *rule,feq,action* )

---

**AddControlToRule** Adds a control to the controls queue so that a specified action is taken when a particular rule in the managed collection is broken.

*rule* A string constant, variable, EQUATE, or expression that contains the name of this rule. If rule names are not unique within the collection managed by this RuleManager object, results may be unpredictable.

*feq* A numeric constant, variable, EQUATE, or expression that indicates the Field Equate number of a control whose hidden or disabled status will be determined by the status of this rule.

*action* A numeric constant, variable, EQUATE, or expression that indicates the action to be taken with respect to this control when a rule is broken, defined as follows:

RuleAction:None	EQUATE(0)
RuleAction:Hide	EQUATE(1)
RuleAction:UnHide	EQUATE(2)
RuleAction:Disable	EQUATE(3)
RuleAction:Enable	EQUATE(4)

Example:

```
Customer.SetErrorImage('~BRuleNo.ico')
Customer.SetDescription('Customer File Rules')
Customer.AddRule('StateRequired','State is required','cus:State <<> ''',?cus:State,
Customer.AddRule('ZipOK','Postal code must have valid format.',
    'CheckCustomerZip()',?cus:ZipCode,3)
Customer.AddControlToRule('StateRequired, ?cus:ZipCode,RuleAction:Hide)
    ! Hide Zip Code field until state is filled in.
```

## CheckRule (check a particular rule)

### **CheckRule**( *rule,display*)

---

**CheckRule** Checks a particular rule and optionally sets the error indicator for its associated control.

*rule* A string constant, variable, EQUATE, or expression that contains the name of this rule. If rule names are not unique within the collection managed by this RuleManager object, results may be unpredictable.

*display* A Boolean constant, variable, EQUATE, or expression that, if True, will cause the error indicator to be set for the control associated with this rule.

Return Data Type: BYTE

Example:

```
Customer.CheckRule(ZipOK,True) ! Validate zip code and set indicator if invalid
```

## CheckAllRules (check all rules in this collection)

### **CheckAllRules**( *display*)

---

**CheckAllRules** Checks all rules in the collection managed by this FileManager object and optionally sets the error indicators controls associated with those rules.

*display* A Boolean constant, variable, EQUATE, or expression that, if True, will cause the error indicator to be set for the control associated with this rule.

Return Data Type: LONG

Example:

```
Customer.CheckAllRules(False) ! Validate all fields but do not set indicators.
```

## Item (locate a particular rule)

```
Item( | rulename | )
      | position  |
```

---

<b>Item</b>	Locates a specified rule and retrieves its entry from the broken rule queue. t.
<i>rulename</i>	A string constant, variable, EQUATE, or expression that contains the name of the rule to be located. If rule names are not unique within the collection managed by this RuleManager object, results may be unpredictable.
<i>position</i>	A string constant, variable, EQUATE, or expression that contains a numeric value corresponding to the position in the broken rule queue of the rule to be located.

The **Item** method locates the specified rule and returns the address of its Rule object. If the specified rule name does not exist in the broken rule queue or if the specified position is outside the range of entries in the broken rule queue, the Item method returns zero.

Return Data Type: LONG

Example:

```
RulesCollection.BrokenRuleCount PROCEDURE
```

```
LBR          &Rule
```

```
NumberOfRules LONG
```

```
Counter      LONG
```

```
RetVal       LONG
```

```
CODE
```

```
NumberOfRules = SELF.RuleCount()
```

```
LOOP Counter = 1 TO NumberOfRules
```

```
  LBR &= SELF.Item(Counter)
```

```
  IF LBR.GetIsBroken()
```

```
    Retval += 1
```

```
  END
```

```
END
```

```
RETURN(Retval)
```

## TakeAccepted (handle acceptance of error indicators)

### TakeAccepted( *control* )

---

**TakeAccepted** Determines whether or not the specified control is an error indicator for one of the Rule objects managed by this RulesCollection object. If so, the TakeAccepted method displays the description of the rule and provides the user with an option to view the status of all broken rules.

*control* A numeric constant, variable, EQUATE, or expression that contains a field equate value. The control indicated by this value is assumed to have been Accepted.

Return Data Type: BYTE

Implementation: When the control that has been accepted is an error indicator, the description of the associated rule is displayed in a message box. This message box always provides a Close button. If the total number of broken rules found exceeds one, a second button offers to display all of them. If the Supervisor property is not NULL, this RulesCollection object uses that property to access the EnumerateBrokenRules method of its supervising RulesManagerObject rather than its own. As a result, the user gets a full list of broken rules even when the controls on a screen involve rules from more than one collection.

Example:

```

ThisWindow.TakeAccepted PROCEDURE
Return Value          BYTE,AUTO
Looped BYTE
CODE
Customer.TakeAccepted(Accepted())! RulesCollection traps to determine if
                                ! error-indicator was clicked

LOOP
  IF Looped
    RETURN Level:Notify
  ELSE
    Looped = 1
  END
CASE ACCEPTED()
END
EMD

```

See Also: CheckAllRules, EnumerateBrokenRules

## SetEnumerateIcons (set icons for broken rules display)

**SetEnumerateIcons**( *windowicon*, *validicon*, *brokenicon*)

---

**SetEnumerateIcons** Sets the icons for the display produced by the EnumerateBrokenRules method.

*windowicon* A string constant, variable, EQUATE, or expression that contains the file name of the icon which will appear on the title bar of the broken rules listing.

*validicon* A string constant, variable, EQUATE, or expression that contains the file name of the icon which will appear on lines containing valid rules.

*brokenicon* A string constant, variable, EQUATE, or expression that contains the file name of the icon which will appear on lines containing broken rules.

Return Data Type: LONG

Example:

```
Customer. SetEnumerateIcons('~BRules.ico', '~BRuleOk.ico', |
                               '~BRuleNo.ico')
```

See Also: EnumerateBrokenRules

## EnumerateBrokenRules (display a list of rules with status of each)

**EnumerateBrokenRules**( *header,brokenonly* )

---

**EnumerateBrokenRules**      Displays a list of the rules managed by this RulesCollection object.

*header*                    A string constant, variable, EQUATE, or expression that contains a header to be displayed in the title bar of the enumerated rules display.

*broken*                    A Boolean constant, variable, EQUATE, or expression that, if True, causes the EnumerateBrokenRules method to display only rules which are broken.

The **EnumerateBrokenRules** method provides a convenient way to display the set of rules (or broken rules) managed by this RulesCollection object. If the user selects a rule from this display, the field `equate` of the control associated with that rule is returned, allowing the appropriate field to be SELECTed,

Example:

```
IF MESSAGE(Desc,'Error Information',ICON:ASTERISK,'Close |' & MoreString) = 2
  SelectControl = SELF.EnumerateBrokenRules(SELF.GetDescription())
  IF SelectControl
    SELECT(SelectControl)
  END
END
```

See Also:      TakeAccepted

## SetControlsStatus (set status of managed controls)

**SetControlsStatus**( [*control* [, *action* ] ] )

---

**SetControlsStatus** Sets the hidden or disabled status of a control or controls managed by this RulesCollection object.

*action* A numeric constant, variable, EQUATE, or expression that indicates the action to be taken with respect to this control when a rule is broken, defined as follows:

RuleAction:None	EQUATE(0)
RuleAction:Hide	EQUATE(1)
RuleAction:UnHide	EQUATE(2)
RuleAction:Disable	EQUATE(3)
RuleAction:Enable	EQUATE(4)

*control* A numeric constant, variable, EQUATE, or expression that contains a field equate value.

The **SetControlsStatus** method sets the status of controls managed by this RulesCollection object depending on the parameters supplied. If *control* is supplied, only status changes involving that control are processed. If *action* is supplied, only status changes involving that action are processed. If neither is supplied, then all changes for all controls are processed.

Example:

```
RulesCollection.SetControlsStatus PROCEDURE
  lIndex LONG
  lIndex2 LONG
  lChange BYTE
  CODE
  LOOP lIndex=1 TO RECORDS(SELF.Controls)
    GET(SELF.Controls,lIndex)
    IF ERRORCODE() THEN BREAK.
    SELF.SetControlsStatus(SELF.Controls.Control,pAction)
  END
```

## NeedChangeControlStatus (check if control status needs to change)

**NeedChangeControlStatus**( *control*, *action*, *found* )

---

**NeedChangeControlStatus** Examines the conditions under which the hidden or disabled status of control needs to change and returns information about whether the change is needed now.

*control* A numeric constant, variable, EQUATE, or expression that contains a field equate value.

*action* A numeric constant, variable, EQUATE, or expression that indicates the action to be taken with respect to this control when a rule is broken, defined as follows:

RuleAction:None	EQUATE(0)
RuleAction:Hide	EQUATE(1)
RuleAction:UnHide	EQUATE(2)
RuleAction:Disable	EQUATE(3)
RuleAction:Enable	EQUATE(4)

*found* A numeric variable which will be updated with an action code.

The **NeedChangeControlStatus** method returns True if the status of *control* needs to change, False if not. On return, the *found* variable is set to the action that needs to be taken.

Example:

```
lActionExist = RuleAction:None
lChange = SELF.NeedChangeControlStatus(pControlFeq,pAction,lActionExist)
IF lActionExist<>pAction THEN RETURN.
```

## RulesManager Properties

The RulesManager Class contains no public properties.

## RulesManager Methods

### Access Methods

Although most of the properties of the RulesCollection class are PRIVATE, public access to these properties is provided by a set of methods:

**GetChangeControls()** returns a BYTE containing the ChangeControlsStatus property

**SetChangeControls( status )** sets the ChangeControlsStatus property

---

**Setxxx/Getxxx** Sets or retrieves a property

*status* A Boolean constant, variable, EQUATE, or expression that contains either True or False.

Example:

```
RulesManager.AddRulesCollection PROCEDURE(RulesCollection pRM)
    CODE
    SELF.Rules.RM &= pRM
    ADD( SELF.Rules )
    SELF.Rules.RM.ChangeControlsStatus = False
```

## Construct (initialize RulesManager object)

### **Construct**

**Construct** Initializes the RulesManager object. The Construct method is executed automatically when the object is instantiated.

## Destruct (shut down RulesManager object)

### **Destruct**

**Destruct** Performs necessary cleanup prior to the disposal of the RulesManager object. The Destruct method is executed automatically when the object is disposed of.

## RulesManagerCount (count rules in the collection)

### **RulesManagerCount()**

**RuleManagerCount** Returns the number of RulesCollection objects being monitored by this RulesCollection object.

Return Data Type: LONG

Example:

```
NumberOfRulesCollections = SELF.RulesManagerCount()
```

## BrokenRulesCount (count rules in the collection which are broken)

### **BrokenRulesCount()**

**BrokenRulesCount** Returns the number of rules being monitored, by all of the RulesCollection objects managed by this RulesManager object, which are broken.

Return Data Type: LONG

Example:

```
BrokenRules = SELF.BrokenRuleCount()
```

## AddRulesCollection (add a rule to this collection)

### AddRulesCollection( *RulesCollection* )

---

**AddRulesCollection** Adds a RulesCollection instance to the collection being managed by this RulesManager object

*RulesCollection* The label of a RulesCollection object.

The **AddRulesCollection** method adds the specified RulesCollection object to the RuleManager and adds it to the Rules queue.

Example:

```
BusinessRulesManager.AddRulesCollection(Customer)
BusinessRulesManager.SetEnumerateIcons('~BRules.ico', '~BRuleOk.ico', |
                                     '~BRuleNo.ico')
BusinessRulesManager.AddControl(?Ok,RuleAction:Disable)
BusinessRulesManager.SetGlobalRuleReferences(GlobalRule)
```

## CheckAllRules (check all rules in all collections)

### CheckAllRules( *display* )

---

**CheckAllRules** Checks all rules in the collections (RulesCollection objects) managed by this FilesManager object, optionally sets the error indicators controls associated with those rules, and returns a count of broken rules.

*display* A Boolean constant, variable, EQUATE, or expression that, if True, will cause the error indicator to be set for the control associated with this rule.

Return Data Type: LONG

Example:

```
BusinessRulesManager.CheckAllRules(True) ! Validate all fields and set indicators.
```

## TakeAccepted (handle acceptance of error indicators)

### TakeAccepted( *control* )

---

**TakeAccepted** Determines whether or not the specified control is an error indicator for one of the Rule objects monitored by RulesCollection objects which are in turn managed by this RulesManager object. If so, the TakeAccepted method displays the description of the rule and provides the user with an option to view the status of all broken rules.

*control* A numeric constant, variable, EQUATE, or expression that contains a field equate value. The control indicated by this value is assumed to have been Accepted.

Return Data Type: BYTE

Example:

```
ThisWindow.TakeAccepted PROCEDURE
ReturnValue          BYTE,AUTO
Looped BYTE
CODE
  BusinessRulesManager.TakeAccepted(Accepted())! RulesManager trap to determine
                                           ! if error-indicator was clicked

LOOP
  IF Looped
    RETURN Level:Notify
  ELSE
    Looped = 1
  END
  CASE ACCEPTED()
  END
EMD
```

## SetEnumerateIcons (set icons for broken rules display)

**SetEnumerateIcons**( *windowicon*, *validicon*, *brokenicon* )

---

**SetEnumerateIcons** Sets the icons for the display produced by the EnumerateBrokenRules method.

*windowicon* A string constant, variable, EQUATE, or expression that contains the file name of the icon which will appear on the title bar of the broken rules listing.

*validicon* A string constant, variable, EQUATE, or expression that contains the file name of the icon which will appear on lines containing valid rules.

*brokenicon* A string constant, variable, EQUATE, or expression that contains the file name of the icon which will appear on lines containing broken rules.

Return Data Type: LONG

Example:

```
BusinessRulesManager.SetEnumerateIcons('~BRules.ico', '~BRuleOk.ico', |  
                                         '~BRuleNo.ico')
```

See Also: EnumerateBrokenRules

## EnumerateBrokenRules (display a list of rules with status of each)

**EnumerateBrokenRules**( *header,brokenonly* )

---

### EnumerateBrokenRules

Displays a list of the rules monitored by RulesCollection objects which in turn are managed by this RulesManager object.

*header* A string constant, variable, EQUATE, or expression that contains a header to be displayed in the title bar of the enumerated rules display.

*broken* A Boolean constant, variable, EQUATE, or expression that, if True, causes the EnumerateBrokenRules method to display only rules which are broken.

The **EnumerateBrokenRules** method provides a convenient way to display the set of rules (or broken rules) monitored by RulesCollection objects managed by this RulesManager object. If the user selects a rule from this display, the field equate of the control associated with that rule is returned, allowing the appropriate field to be SELECTed,

Example:

```
IF MESSAGE(Desc,'Error Information',ICON:ASTERISK,'Close |' & MoreString) = 2
  SelectControl = SELF.EnumerateBrokenRules(SELF.GetDescription(), |
                                           SELF.GetErrorImage())
  IF SelectControl
    SELECT(SelectControl)
  END
END
```

See Also: TakeAccepted

## SetControlsStatus (set status of managed controls)

### **SetControlsStatus( )**

---

#### **SetControlsStatus**

Sets the hidden or disabled status of a control or controls managed by this RulesManager object and by RulesCollection objects assigned to this RulesManager object.

The **SetControlsStatus** method sets the status of controls managed by this RulesManager object and subsidiary RulesCollection objects.



# SelectFileClass

## SelectFileClass Concepts

The SelectFileClass object manages the Windows File Dialog--both 16-bit (short filenames) and 32-bit versions (long filenames)--to select a single file or multiple files.

## SelectFileClass Relationship to Other Application Builder Classes

The ASCIIViewerClass uses the the SelectFileClass to let the end user choose the file to view. Otherwise, the SelectFileClass is completely independent of other Application Builder Classes.

## SelectFileClass ABC Template Implementation

The ABC DOSFileLookup control template generates code to declare a local SelectFileClass class *and* object for each instance of the SelectFile Control Template.

The class is named SelectFile# where # is the instance number of the DOSFileLookup control template. The template provides the derived class so you can use the **Classes** tab to easily modify the select file behavior on an instance-by-instance basis.

## SelectFileClass Source Files

The SelectFileClass source code is installed by default to the Clarion \LIBSRC folder. The SelectFileClass source code and its respective components are contained in:

ABUTIL.INC	SelectFileClass declarations
ABUTIL.CLW	SelectFileClass method definitions
ABUTIL.TRN	SelectFileClass default text, mask, flags

## SelectFileClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a SelectFileClass object. This example displays a dialog that alternatively allows single file or multi-file selection.

```

PROGRAM
  INCLUDE('ABUTIL.INC')          !declare SelectFileClass
  MAP
  END
SelectFile  SelectFileClass      !declare SelectFile object
FileQ       SelectFileQueue      !declare FileName QUEUE
FileQCount  USHORT,AUTO          !declare Q counter
FileNames   CSTRING(255)         !variable to hold file names
FileMask    CSTRING('Text *.txt|*.txt|All *.*|*.*') !File dialog file masks
MultiFiles  BYTE                  !single/multiple file switch
GetFile     WINDOW('Select File'),AT(,,173,40),SYSTEM,GRAY,RESIZE
            ENTRY(@s254),AT(6,6,144,12),USE(FileNames)
            BUTTON('...'),AT(156,6,12,12),USE(?SelectFiles)
            OPTION,AT(6,20,),USE(MultiFiles)
            RADIO('One File'),AT(5,25),USE(?1File),VALUE('0')
            RADIO('Multiple Files'),AT(45,25),USE(?MultiFile),VALUE('1')
            END
            BUTTON('Close'),AT(119,24),USE(?Close)
            END
CODE
OPEN(GetFile)
ACCEPT
  IF EVENT() = EVENT:OpenWindow    !on open window
  SelectFile.Init                   !initialize SelectFile object
  SelectFile.AddMask('Clarion source|*.clw;*.inc') !set default file mask
  SelectFile.AddMask(FileMask)      !set additional file masks
  END

```

```
CASE FIELD()
OF ?SelectFiles                                !on get file button
  IF EVENT() = EVENT:Accepted                 !if user clicked it
    IF MultiFiles                             !if multiple files requested
      SelectFile.WindowTitle='Select multiple files' !set file dialog titlebar
      SelectFile.Ask(FileQ,0)                 !display file dialog
      LOOP FileQCount=1 TO RECORDS(FileQ)     !for each selected file
        GET(FileQ,FileQCount)                 !get the file information
        MESSAGE(FileQ.Name)                   !process the file
      END
    ELSE                                       !if single file requested
      SelectFile.WindowTitle = 'Select one file' !set file dialog titlebar
      FileNames = SelectFile.Ask(1)           !display file dialog
      DISPLAY(?FileNames)                     !redraw Filenames field
    END
  END
OF ?Close                                       !on close button
  IF EVENT() = EVENT:Accepted                 !if user clicked it
    POST(Event:CloseWindow)                   !shut down
  END
END
END
```

## SelectFileClass Properties

The SelectFileClass contains the following properties.

### DefaultDirectory (initial path)

**DefaultDirectory**                      **CSTRING(File:MaxFilePath)**

The **DefaultDirectory** property contains the directory the Windows file dialog initially opens to. If DefaultDirectory is null, the file dialog opens to the current directory.

### DefaultFile (initial filename/filemask)

**DefaultFile**                      **CSTRING(File:MaxFilePath)**

The **DefaultFile** property contains the filename that initially appears in the Windows file dialog filename field. The filename may contain wildcard characters such as \* to filter the file dialog's file list.

## **Flags (file dialog behavior)**

### **Flags BYTE**

The **Flags** property is a bitmap that indicates the type of file action the Windows file dialog performs (select, multi-select, save directory, lock directory, suppress errors). The Flags property operates identically to the FILEDIALOG *flag* parameter. See *FILEDIALOG* in the *Language Reference* for more information.

Implementation: The Init method sets the Flags property to its default value declared in ABUTIL.TRN--select a file from any directory.

See Also: Init

## **WindowTitle (file dialog title text)**

### **WindowTitle CSTRING(80)**

The **WindowTitle** property contains a string that sets the title bar text in the Windows file dialog.

Implementation: The Init method sets the WindowTitle property to its default value declared in ABUTIL.TRN. The SelectFileClass uses the WindowTitle property as the *title* parameter to the FILEDIALOG function. See *FILEDIALOG* in the *Language Reference* for more information.

See Also: Init

## SelectFileClass Methods

### AddMask (add file dialog file masks)

**AddMask**( | *description*, *masks* |)  
                   | *mask string*           |

---

<b>AddMask</b>	Adds file masks to the file dialog's <b>List Files of Type</b> drop-down list.
<i>description</i>	A string constant, variable, EQUATE, or expression that contains a file mask description such as 'all files-*. *' or 'source files-*.inc;*.clw'. The mask value may be included in the description for information only.
<i>masks</i>	A string constant, variable, EQUATE, or expression that defines the file mask or masks corresponding to the <i>description</i> , such as '*. *' or '*.inc;*.clw'. Multiple masks are separated by a semi-colon (;).
<i>mask string</i>	A string constant, variable, EQUATE, or expression that defines both the file masks and their descriptions.

The **AddMask** method adds file masks and their descriptions to the file dialog's **List Files of Type** drop-down list. The first mask is the default selection in the file dialog.

The **SetMask** method replaces file masks and their descriptions.

The *mask string* parameter must contain one or more descriptions followed by their corresponding file masks in the form *description|masks|description|masks*. All elements in the string must be delimited by the vertical bar (|). For example, 'all files \*.\*|\*.\*|Clarion source \*.clw;\*.inc|\*.clw;\*.inc' defines two selections for the File Dialog's **List Files of Type** drop-down list. See the *extensions* parameter to the FILEDIALOG function in the *Language Reference* for more information.

Example:

```
FileMask CSTRING('Text *.txt|*.txt|All *.*|*.*')      !File dialog file masks
CODE
!program code
  IF EVENT() = EVENT:OpenWindow                      !on open window
    SelectFile.Init                                  !initialize SelectFile object
    SelectFile.SetMask('Clarion source', '*.clw;*.inc') !set default file mask
    SelectFile.AddMask(FileMask)                      !set additional file masks
  END
```

See Also:       SetMask

**Ask (display Windows file dialog)**

**Ask**( [ *file queue*] [, *restore path*] )

---

<b>Ask</b>	Displays the Windows file dialog.
<i>file queue</i>	The label of a QUEUE structure that receives information for the selected files. The structure must be the same as the SelectFileQueue structure declared in ABUTIL.INC. If omitted, the end user may select only one file, for which the Ask method returns the full pathname.
<i>restore path</i>	An integer constant, variable, EQUATE, or expression that indicates whether to restore the current path to its pre-file dialog state. A <i>restore path</i> value of one (1) restores the current path; a value of zero (0) does not restore the current path. If omitted, <i>restore path</i> defaults to zero (0).

The **Ask** method displays the Windows file dialog and returns information, primarily the full pathname, for the selected file or files.

Implementation: The *file queue* parameter must name a QUEUE that begins the same as the SelectFileQueue structure declared in ABUTIL.INC:

```
SelectFileQueue QUEUE,TYPE
Name          STRING(File:MaxFilePath)
ShortName     STRING(File:MaxFilePath)
              END
```

Return Data Type: STRING

Example:

```
FileQ      SelectFileQueue      !declare FileName QUEUE
FileQCount BYTE
CODE
!program code
SelectFile.Ask(FileQ,0)        !multi file dialog, don't restore directory
LOOP FileQCount=1 TO RECORDS(FileQ) !for each selected file
  GET(FileQ,FileQCount)       !get the file information
  MESSAGE(FileQ.Name)         !process the file
END

FileNames = SelectFile.Ask(1)  !single file dialog, restore directory
```

## Init (initialize the SelectFileClass object)

### Init

The **Init** method initializes the SelectFileClass object.

Implementation: The Init method WindowTitle and Flags properties to their default values declared in ABUTIL.TRN.

Example:

```
IF EVENT() = EVENT:OpenWindow           !on open window
  SelectFile.Init                       !initialize SelectFile object
  SelectFile.AddMask('Clarion source|*.clw;*.inc') !set default file mask
  SelectFile.AddMask(FileMask)          !set additional file masks
END
```

See Also: Flags, WindowTitle

**SetMask (set file dialog file masks)**

**SetMask**( | *description*, *masks* |)

| *mask string* |

<b>SetMask</b>	Sets the file masks available in the file dialog's <b>List Files of Type</b> drop-down list.
<i>description</i>	A string constant, variable, EQUATE, or expression that contains a file mask description such as 'all files-*.*)' or 'source files-.inc;*.clw'. The mask value may be included in the description for information only.
<i>masks</i>	A string constant, variable, EQUATE, or expression that defines the file mask or masks corresponding to the <i>description</i> , such as '*.*)' or '*.inc;*.clw'. Multiple masks are separated by a semi-colon (;).
<i>mask string</i>	A string constant, variable, EQUATE, or expression that defines both the file masks and their descriptions.

The **SetMask** method sets the file masks and their descriptions available in the file dialog's **List Files of Type** drop-down list. The first mask is the default selection in the file dialog.

The AddMask method appends file masks and their descriptions.

The *mask string* parameter must contain one or more descriptions followed by their corresponding file masks in the form description|masks|description|masks. All elements in the string must be delimited by the vertical bar (|). For example, 'all files \*.\*|\*.\*|Clarion source \*.clw;\*.inc|\*.clw;\*.inc' defines two selections for the File Dialog's **List Files of Type** drop-down list. See the *extensions* parameter to the FILEDIALOG function in the *Language Reference* for more information.

Example:

```
FileMask CSTRING('Text *.txt|*.txt|All *.*|*.*')      !File dialog file masks
CODE
!program code
IF EVENT() = EVENT:OpenWindow                      !on open window
  SelectFile.Init                                  !initialize SelectFile object
  SelectFile.SetMask('Clarion source','*.clw;*.inc')!set default file mask
  SelectFile.AddMask(FileMask)                     !set additional file masks
END
```

See Also:      AddMask



---

# StandardBehavior Class

## StandardBehavior Overview

The StandardBehavior class provides a central point for specification of standard basic browse behavior.

## StandardBehavior Class Concepts

The StandardBehavior class provides a set of standard methods that can be used for all classes that will emulate a browse box.

## Relationship to Other Application Builder Classes

The StandardBehavior class implements the IListControl interface and the BrowseQueue interface.

## StandardBehavior Source Files

The StandardBehavior source code is installed by default to the Clarion \LIBSRC folder. The specific StandardBehavior source code and their respective components are contained in:

ABBROWSE.INC	StandardBehavior declarations
ABBROWSE.CLW	StandardBehavior method definitions

## StandardBehavior Properties

The StandardBehavior class contains no public properties.

## StandardBehavior Methods

### StandardBehavior Methods

The StandardBehavior class inherits all of the methods from the BrowseQueue and IListControl interfaces which it implements. See BrowseQueue and IListControl interfaces for more information.

### Init(initialize the StandardBehavior object)

Init(*listqueue*, *viewposition*, *listcontrol*)

Init	<b>Initializes the StandardBehavior object.</b>
<i>listqueue</i>	The label of the list control's data source QUEUE.
<i>viewposition</i>	The label of a string field within the <i>listqueue</i> containing the POSITION of the VIEW.
<i>listcontrol</i>	A numeric constant, variable, EQUATE, or expression containing the control number of the browse's LIST control.

The **Init** method initializes the StandardBehavior object.

Implementation: The Init method is called by the BrowseClass.Init method. The BrowseClass.Init method creates an instance of the StandardBehavior object before the Init method is called.

See Also: BrowseClass.Init



---

# StandardErrorLogClass

## StandardErrorLogClass Overview

The StandardErrorLogClass manages the opening and closing of an error log file. This class implements the ErrorLogInterface.

## StandardErrorLogClass Source Files

The StandardErrorLogClass source code is installed by default to the Clarion \LIBSRC. The specific StandardErrorLogClass source code and their respective components are contained in:

ABERROR.INC    StandardErrorLogClass declarations  
ABERROR.CLW    StandardErrorLogClass method definitions

## ABC Template Implementation

The StandardErrorLogClass is instantiated in the ErrorClass.Init method.

## StandardErrorLogClass Properties

The StandardErrorLogClass contains no public properties.

## StandardErrorLogClass Methods

### Close (close standarderrorlog file)

**Close**(*force*), PROC, PROTECTED

<b>Close</b>	Close the ErrorlogFile.
<i>force</i>	An numeric constant, variable, EQUATE, or expression that indicates whether the log file must be closed or whether it should be conditionally closed. A value of one (1 or True) unconditionally closes the errorlog file; a value of zero (0 or False) only closes the errorlog file as circumstances require.

The **Close** method closes the ErrorLog file. Level:Benign is returned from this method. A Level:Fatal is returned if an error occurs.

### Construct (initialize StandardErrorLogClass object)

**Construct**

The **Construct** method initializes the StandardErrorLogClass. It is automatically called when the object is created.

### Destruct (remove the StandardErrorLogClass object)

**Destruct**

The **Destruct** method destroys the StandardErrorLogClass object. This method is automatically called when the object is destroyed.

**Open (open standarderrorlog file)****Open**(*force*), PROC, PROTECTED

<b>Open</b>	Create and open the ErrorLog file.
<i>force</i>	An numeric constant, variable, EQUATE, or expression that indicates whether the log file must be opened or whether it should be conditionally opened. A value of one (1 or True) unconditionally opens the errorlog file; a value of zero (0 or False) only opens the errorlog file as circumstances require.

The **Open** method creates and opens the ErrorLog file. Level:Benign is returned from this method. A Level:Fatal is returned if an error occurs.

# StepClass

## StepClass Overview

The StepClass estimates the relative position of a given record within a keyed dataset. The StepClass is an abstract class--it is not useful by itself. However, other useful classes are derived from it and other structures (such as the BrowseClass and ProcessClass) use it to reference any of its derived classes.

## StepClass Concepts

The classes derived from the StepClass let you define an upper and a lower boundary as well as a series of steps between the boundaries. Then the classes help you traverse or navigate the defined steps with a scrollbar thumb, a progress bar, or any control that shows a relative linear position within a finite range.

The classes derived from the StepClass implement some of the common variations in boundaries (alphanumeric or numeric) and steps (alphabetic distribution, surname distribution, normal distribution) that occur in the context of a browse or batch process.

The StepClass requires that the data be traversed with a key. If you are traversing data without a key, you can track your progress simply by counting records, and no StepClass is needed.

## StepClass Relationship to Other Application Builder Classes

The BrowseClass and ProcessClass optionally use the classes derived from the StepClass. Therefore, if your BrowseClass or ProcessClass objects use a StepClass, then your program must instantiate a StepClass for each use.

The StepCustomClass, StepStringClass, StepLongClass, and StepRealClass are all derived from the StepClass. Each of these derived classes provides slightly different behaviors and characteristics.

### StepCustomClass

Use the StepCustomClass when the data you are processing has an alphanumeric key with a skewed distribution.

### StepStringClass

Use the StepStringClass when the data you are processing has an alphanumeric key with a normal distribution.

### StepLongClass

Use the StepLongClass when the data you are processing has an integer key with a normal distribution.

**StepRealClass**

Use the StepRealClass when the data you are processing has a non-integer numeric key with a normal distribution.

**StepClass ABC Template Implementation**

Because the StepClass is abstract, the ABC Template generated code does not directly reference the StepClass--rather, it references classes derived from the StepClass.

**StepClass Source Files**

The StepClass source code is installed by default to the Clarion \LIBSRC folder. The StepClass source code and its respective components are contained in:

ABBROWSE.INC  
ABBROWSE.CLW

StepClass declarations  
StepClass method definitions

## StepClass Properties

The StepClass has a single property--Controls. This property is inherited by classes derived from StepClass. The Controls property is described below.

### **Controls (the StepClass sort sequence)**

#### **Controls      BYTE**

The **Controls** property contains a value that identifies for the StepClass object:

- the characters included in the sort sequence
- the direction of the sort (ascending or descending)

The Init method sets the value of the Controls property.

A StepClass object may be associated with a BrowseClass object sort order. The BrowseClass.AddSortOrder method sets the sort orders for a BrowseClass object.

Implementation:      The Controls property is a single byte bitmap that contains several important pieces of information for the StepClass object. Set the value of the Controls property with the Init method.

See Also:              Init, BrowseClass.AddSortOrder

## StepClass Methods

### GetPercentile (return a value's percentile:StepClass)

**GetPercentile**( *value* ), VIRTUAL

---

**GetPercentile** Returns the specified *value*'s percentile relative to the StepClass object's boundaries.

*value* A constant, variable, EQUATE, or expression that specifies the value for which to calculate the percentile.

The **GetPercentile** method returns the specified *value*'s percentile relative to the StepClass object's upper and lower boundaries.

The GetPercentile method is a placeholder method for classes derived from StepClass--StepLongClass, StepRealClass, StepStringClass, StepCustomClass, etc.

Return Data Type: BYTE

See Also: StepLongClass.GetPercentile, StepRealClass.GetPercentile, StepStringClass.GetPercentile, StepCustomClass.GetPercentile

**GetValue (return a percentile's value:StepClass)****GetValue( *percentile* ), VIRTUAL**

---

**GetValue** Returns the specified *percentile's* value relative to the StepClass object's boundaries.

*percentile* An integer constant, variable, EQUATE, or expression that specifies the percentile for which to retrieve the value.

The **GetValue** method returns the specified *percentile's* value relative to the StepClass object's upper and lower boundaries.

The GetValue method is a placeholder method for classes derived from StepClass--StepLongClass, StepRealClass, StepStringClass, StepCustomClass, etc.

Return Data Type: **STRING**

See Also: StepLongClass.GetValue, StepRealClass.GetValue, StepStringClass.GetValue, StepCustomClass.GetValue

**Init (initialize the StepClass object)****Init**( *controls* )

---

<b>Init</b>	Initializes the StepClass object.
<i>controls</i>	An integer constant, variable, EQUATE, or expression that contains several important pieces of information for the StepClass object.

The **Init** method initializes the StepClass object.

The *controls* parameter identifies for the StepClass object:

- the characters included in the sort sequence
- whether the key is case sensitive
- the direction of the sort (ascending or descending)

Implementation: The Init method sets the value of the Controls property. Set the value of the Controls property by adding together the applicable EQUATEs declared in ABBROWSE.INC as follows:

```

ITEMIZE,PRE(ScrollSort)
AllowAlpha    EQUATE(1)    !include characters ABCDEFGHIJKLMNOPQRSTUVWXYZ
AllowAlt      EQUATE(2)    !include characters `!"#$%^&*()' '-=_+][#;~@:/.,? \ |
AllowNumeric  EQUATE(4)    !include characters 0123456789
CaseSensitive EQUATE(8)    !include characters abcdefghijklmnopqrstuvwxyz
Descending   EQUATE(16)   !the sort is descending
END

```

Example:

```
MyStepClass.Init(ScrollSort:AllowAlpha+ScrollSort:AllowNumeric)
```

See Also:      Controls

**Kill (shut down the StepClass object)****Kill, VIRTUAL**

The **Kill** method is a virtual method to shut down the StepClass object.

The Kill method is a placeholder method for classes derived from StepClass-- StepStringClass, StepCustomClass, etc.

See Also: StepStringClass.Kill, StepCustomClass.Kill

**SetLimit (set smooth data distribution:StepClass)****SetLimit( *lower, upper* ), VIRTUAL**

---

<b>SetLimit</b>	Sets the StepClass object's upper and lower boundaries.
<i>lower</i>	A constant, variable, EQUATE, or expression that specifies the StepClass object's lower boundary. The value may be numeric or alphanumeric.
<i>upper</i>	A constant, variable, EQUATE, or expression that specifies the StepClass object's upper boundary. The value may be numeric or alphanumeric.

The **SetLimit** method sets the StepClass object's upper and lower boundaries.

The SetLimit method is a placeholder method for classes derived from StepClass--StepLongClass, StepRealClass, StepStringClass etc.

See Also: StepLongClass.SetLimit, StepRealClass.SetLimit, StepStringClass.SetLimit

**SetLimitNeeded (return static/dynamic boundary flag:StepClass)****SetLimitNeeded, VIRTUAL**

The **SetLimitNeeded** method returns a value indicating whether the StepClass object's boundaries are static (set at compile time) or dynamic (set at runtime). A return value of one (1) indicates dynamic boundaries that may need to be reset when the monitored result set changes (records are added, deleted, or filtered). A return value of zero (0) indicates the boundaries are fixed at compile time (name or alpha distribution) and are not adjusted when the monitored result set changes.

The SetLimitNeeded method is a placeholder method for classes derived from StepClass, such as StepStringClass.

Return Data Type: BYTE

See Also: StepStringClass.SetLimitNeeded



# StepCustomClass

## StepCustomClass Overview

The StepCustomClass is a StepClass that handles a numeric or alphanumeric key with a skewed distribution (data is not evenly distributed between the lowest and highest key values). You can provide information about the data distribution so that the StepCustomClass object returns accurate feedback about the data being processed.

## StepCustomClass Concepts

You can specify a custom data distribution for a StepCustomClass object that fits a specific data set (the other StepClass objects apply one of several predefined data distributions). Use the AddItem method to set the steps or distribution points for the StepCustomClass object.

For example, your CustomerKey may contain values ranging from 1 to 10,000, but 90 percent of the values fall between 9,000 and 10,000. If your StepClass object assumes the values are *evenly* distributed between 1 and 10,000 (StepLongClass with Runtime distribution), then your progress bars and vertical scroll bar thumbs will give a misleading visual representation of the data. However, if your StepClass object knows the actual data distribution (StepCustomClass object with 90 percent of the steps between 9,000 and 10,000), then your progress bars and vertical scroll bar thumbs will give an accurate visual representation of the data.

**Tip:** Use the StepLongClass for integer keys with normal distribution. Use the StepStringClass for alphanumeric keys with smooth or skewed distribution. Use the StepRealClass for fractional keys with normal distribution.

Use the StepCustomClass when the data (key) is skewed (data is not evenly distributed between the lowest and highest key values), and the skew does not match any of the standard StepStringClass distribution options (see StepStringClass for more information).

## StepCustomClass Relationship to Other Application Builder Classes

The BrowseClass and the ProcessClass optionally use the StepCustomClass. Therefore, if your BrowseClass or ProcessClass uses the StepCustomClass, your program must instantiate the StepCustomClass for each use. See the Conceptual Example.

## StepCustomClass ABC Template Implementation

The ABC Templates (BrowseBox, Process, and Report) automatically include all the classes and generate all the code necessary to use the StepCustomClass with your BrowseBoxes, Reports, and Processes.

---

## Process and Report Procedure Templates

---

By default, the Process and Report templates declare a StepStringClass, StepLongClass, or StepRealClass called ProgressMgr. However, you can use the **Report Properties** Classes tab (the **Progress Class** button) to declare a StepCustomClass (or derive from the StepCustomClass) instead. Similarly, you can use the **Process Properties** General tab (the **Progress Manager** button) to declare a StepCustomClass (or derive from the StepCustomClass). The templates provide the derived class so you can modify the ProgressMgr behavior on an instance-by-instance basis.

If you specify a StepCustomClass object for a Process or Report procedure, you must embed calls to the AddItem method (ProgressMgr.AddItem) to set the custom "steps" or distribution points.

---

## Browse Procedure and BrowseBox Control Templates

---

By default, the BrowseBox template declares a StepStringClass, StepLongClass, or StepRealClass called BRWn::Sort#:StepClass, where *n* is the BrowseBox template instance number, and # is the sort order sequence (identifies the key). You can use the BrowseBox's **Scroll Bar Behavior** dialog to specify a StepCustomClass and to set the custom "steps" or distribution points. You can use the **Step Class** button to derive from the StepCustomClass so you can modify the StepCustomClass behavior on an instance-by-instance basis.

## StepCustomClass Source Files

The StepCustomClass source code is installed by default to the Clarion \LIBSRC folder. The StepCustomClass source code and its respective components are contained in:

ABBROWSE.INC	StepCustomClass declarations
ABBROWSE.CLW	StepCustomClass method definitions

## StepCustomClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a BrowseClass object and related objects. The example initializes and page-loads a LIST, then handles a number of associated events, including searching, scrolling, and updating. When they are initialized properly, the BrowseClass and WindowManager objects do most of the work (default event handling) internally.

```

INCLUDE( 'ABBROWSE.INC' )
INCLUDE( 'ABREPORT.INC' )

MAP
CustomerProcess PROCEDURE
END

CustomerProcess PROCEDURE

FilesOpened      BYTE
Thermometer      BYTE
Process:View     VIEW(Customer)
                END
ProgressWindow  WINDOW('Progress...'),AT(,,142,59),CENTER,TIMER(1),GRAY,DOUBLE
                PROGRESS,USE(Thermometer),AT(15,15,111,12),RANGE(0,100)
                STRING(' '),AT(0,3,141,10),USE(?UserString),CENTER
                STRING(' '),AT(0,30,141,10),USE(?PctText),CENTER
                BUTTON('Cancel'),AT(45,42,50,15),USE(?Cancel)
                END

ThisWindow CLASS(ReportManager)
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL
                END
ThisProcess  ProcessClass          !declare ThisProcess object
ProgressMgr  StepCustomClass       !declare ProgressMgr object
CODE
GlobalResponse = ThisWindow.Run()

ThisWindow.Init PROCEDURE()
ReturnValue BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?Thermometer
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
CLEAR(GlobalRequest)

```

```

CLEAR(GlobalResponse)
Relate:Customer.Open
FilesOpened = True
OPEN(ProgressWindow)
SELF.Opened=True

ProgressMgr.Init(ScrollSort:AllowNumeric) !initialize ProgressMgr object
! ignores inapplicable parameters
LOOP i# = 1 TO 9000 BY 1000 !build skewed distribution steps
  Step="i# !10% of customerids fall between 1 & 9000
  ProgressMgr.AddItem(Step")
END
LOOP i# = 9010 TO 10000 BY 11 !90% of customerids between 9000 & 10000
  Step="i#
  ProgressMgr.AddItem(Step")
END

ThisProcess.Init(Process:View,Relate:Customer,?PctText,Thermometer,ProgressMgr,CUS:ID)
ThisProcess.AddSortOrder(CUS:CustomerIDKey)
SELF.Init(ThisProcess)
SELF.AddItem(?Progress:Cancel,RequestCancelled)
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
IF FilesOpened
  Relate:Customer.Close
END
RETURN ReturnValue

```

## StepCustomClass Properties

The StepCustomClass inherits all the properties of the StepClass from which it is derived. See *StepClass Properties* and *StepClass Concepts* for more information.

In addition to its inherited properties, the StepCustomClass also contains the following properties:

### **Entries (expected data distribution)**

#### **Entries            &CStringList, PROTECTED**

The **Entries** property is a reference to a structure containing the markers or boundaries that define the expected data distribution for the StepCustomClass object. This property defines the expected data distribution points (or steps), as well as the upper and lower boundaries the StepCustomClass object implements. This, plus the actual data distribution, ultimately determines how "far" the indicator (thumb or progress bar) actually moves as records are processed.

The AddItem method sets the value of the Entries property.

Implementation:    The Entries property is a reference to a QUEUE declared in BROWSE.INC as follows:

```
CStringList  QUEUE,TYPE
Item         &CSTRING
            END
```

See Also:            AddItem

## StepCustomClass Methods

### StepCustomClass Methods

The StepCustomClass inherits all the methods of the StepClass from which it is derived. See *StepClass Concepts* and *StepClass Methods* for more information.

### AddItem (add a step marker)

**AddItem**( *stepmarker* )

---

**AddItem** Adds a step marker to the expected data distribution for the StepCustomClass object.

*stepmarker* A string constant, variable, EQUATE, or expression that specifies the next step boundary for each step of the StepCustomClass object's expected data distribution.

The **AddItem** method adds a step marker to the expected data distribution for the StepCustomClass object.

Implementation: The AddItem method sets the value of the Entries property.

Example:

```
GradeStepClass.AddItem('0')      !Failing:      0-65
GradeStepClass.AddItem('65')     !Below Average: 65-75
GradeStepClass.AddItem('75')     !Average:      75-85
GradeStepClass.AddItem('85')     !Better Than Average:85-95
GradeStepClass.AddItem('95')     !Outstanding:  95-
GradeStepClass.AddItem('1000')   !Catchall upper boundary
```

See Also: Entries

**GetPercentile (return a value's percentile:StepCustomClass)****GetPercentile( *value* ), VIRTUAL**

---

**GetPercentile** Returns the specified *value*'s percentile relative to the StepCustomClass object's boundaries.

*value* A string constant, variable, EQUATE, or expression that specifies the value for which to calculate the percentile.

The **GetPercentile** method returns the specified *value*'s percentile relative to the StepCustomClass object's "steps."

Implementation: The AddItem method sets the StepCustomClass object's steps.

Return Data Type: BYTE

Example:

```
IF FIELD() = ?Locator           !focus on locator field
  IF EVENT() = EVENT:Accepted   !if accepted
    MyBrowse.TakeAcceptedLocator !BrowseClass handles it
    ?MyList{PROP:VScrollPos}=MyStep.GetPercentile(Locator) !position thumb to match
  END
END
```

See Also: AddItem

**GetValue (return a percentile's value:StepCustomClass)****GetValue( *percentile* ), VIRTUAL**

---

**GetValue** Returns the specified *percentile's* value relative to the StepCustomClass object's boundaries.

*percentile* An integer constant, variable, EQUATE, or expression that specifies the percentile for which to retrieve the value.

The **GetValue** method returns the specified *percentile's* value relative to the StepCustomClass object's "steps."

Implementation: The AddItem method sets the StepCustomClass object's steps.

Return Data Type: STRING

Example:

```
IF FIELD() = ?MyList                                !focus on browse list
  IF EVENT() = EVENT:ScrollDrag                      !if thumb moved
    Locator=MyStep.GetValue(?MyList{PROP:VScrollPos})!update locator to match
  END
END
```

See Also: AddItem

## Init (initialize the StepCustomClass object)

### **Init**( *controls* )

---

<b>Init</b>	Initializes the StepCustomClass object.
<i>controls</i>	An integer constant, variable, EQUATE, or expression that contains several important pieces of information for the StepCustomClass object.

The **Init** method initializes the StepCustomClass object.

The *controls* identifies for the StepCustomClass object:

- the case sensitivity
- the direction of the sort (ascending or descending)

Implementation: The Init method sets the value of the Controls property. Set the value of the Controls property by adding together the applicable EQUATEs declared in BROWSE.INC as follows:

```
ITEMIZE,PRE(ScrollSort)
CaseSensitive EQUATE(8)      !include abcdefghijklmnopqrstuvwxyz
Descending    EQUATE(16)    !the sort is descending
END
```

Example:

```
MyStepCustomClass.Init(ScrollSort:CaseSensitive)
!program code
MyStepCustomClass.Kill
```

See Also: StepClass.Controls

**Kill (shut down the StepCustomClass object)****Kill, VIRTUAL**

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code.

Implementation: The Kill method frees memory allocated for the Custom property.

Example:

```
MyStepCustomClass.Init(ScrollSort:AllowAlpha+ScrollSort:AllowNumeric)
!program code
MyStepCustomClass.Kill
```

---

# StepLongClass

## StepLongClass Overview

The StepLongClass is a StepClass that handles integer keys with a normal distribution (data is evenly distributed between the lowest and highest key values).

## StepLongClass Concepts

The StepLongClass object applies a normal data distribution between its upper and lower boundaries. Use the SetLimit method to set the expected data distribution for the StepLongClass object.

Use the StepLongClass with integer keys that have a normal distribution (data is evenly distributed between the lowest and highest key values).

**Tip:** Use the StepCustomClass for integer keys with other skews. Use the StepRealClass for non-integer numeric keys. Use the StepStringClass for alphanumeric keys.

## StepLongClass Relationship to Other Application Builder Classes

The BrowseClass and the ProcessClass optionally use the StepLongClass. Therefore, if your BrowseClass or ProcessClass uses the StepLongClass, your program must instantiate the StepLongClass for each use. See the Conceptual Example.

## StepLongClass ABC Template Implementation

The ABC Templates (BrowseBox, Process, and Report) automatically include all the classes and generate all the code necessary to use the StepLongClass with your BrowseBoxes, Reports, and Processes.

## Process and Report Procedure Templates

---

By default, the Process and Report templates declare a StepLongClass for integer keys called ProgressMgr. You can use the **Report Properties** Classes tab (the **Progress Class** button) or the **Process Properties** General tab (the **Progress Manager** button) to derive from the StepLongClass instead. The templates provide the derived class so you can modify the ProgressMgr behavior on an instance-by-instance basis.

## Browse Procedure and BrowseBox Control Templates

---

By default, the BrowseBox template declares a StepLongClass for integer keys called BRWn::Sort#:StepClass, where *n* is the BrowseBox template instance number, and # is the sort order sequence (identifies the key). You can use the BrowseBox's **Scroll Bar Behavior** dialog--**Step Class** button to derive from the StepLongClass so you can modify the StepLongClass behavior on an instance-by-instance basis.

## StepLongClass Source Files

The StepLongClass source code is installed by default to the Clarion \LIBSRC folder. The StepLongClass source code and its respective components are contained in:

ABBROWSE.INC	StepLongClass declarations
ABBROWSE.CLW	StepLongClass method definitions

## StepLongClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a StepLongClass object and related objects. The example batch processes a Customer file on an integer key--CustomerID.

```

INCLUDE( 'ABBROWSE.INC' )
INCLUDE( 'ABREPORT.INC' )

MAP
CustomerProcess PROCEDURE
END

CustomerProcess PROCEDURE

FilesOpened      BYTE
Thermometer      BYTE
Process:View     VIEW(Customer)
                END
ProgressWindow WINDOW('Progress...'),AT(,,142,59),CENTER,TIMER(1),GRAY,DOUBLE
                PROGRESS,USE(Thermometer),AT(15,15,111,12),RANGE(0,100)
                STRING('',AT(0,3,141,10),USE(?UserString),CENTER
                STRING('',AT(0,30,141,10),USE(?PctText),CENTER
                BUTTON('Cancel'),AT(45,42,50,15),USE(?Cancel)
                END

ThisWindow      CLASS(ReportManager)
Init            PROCEDURE(),BYTE,PROC,VIRTUAL
Kill           PROCEDURE(),BYTE,PROC,VIRTUAL
                END

ThisProcess     ProcessClass      !declare ThisProcess object
ProgressMgr    StepLongClass      !declare ProgressMgr object

```

```
CODE
GlobalResponse = ThisWindow.Run()

ThisWindow.Init PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?Thermometer
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
Relate:Customer.Open
FilesOpened = True
OPEN(ProgressWindow)
SELF.Opened=True
ProgressMgr.Init(ScrollSort:AllowNumeric)    !initialize ProgressMgr object
                                              ! ignores inapplicable parameters
ThisProcess.Init(Process:View,Relate:Customer,?PctText,Thermometer,ProgressMgr,CUS:ID)
ThisProcess.AddSortOrder(CUS:CustomerIDKey)
SELF.Init(ThisProcess)
SELF.AddItem(?Progress:Cancel,RequestCancelled)
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
IF FilesOpened
    Relate:Customer.Close
END
RETURN ReturnValue
```

## StepLongClass Properties

The StepLongClass inherits all the properties of the StepClass from which it is derived. See *StepClass Properties* for more information.

In addition to its inherited properties, the StepLongClass also contains the following properties:

### **Low (lower boundary:StepLongClass)**

**Low**    **LONG**

The **Low** property contains the value of the StepLongClass object's lower boundary.

The SetLimit method sets the value of the Low property.

See Also:        SetLimit

### **High (upper boundary:StepLongClass)**

**High**    **LONG**

The **High** property contains the value of the StepLongClass object's upper boundary.

The SetLimit method sets the value of the High property.

See Also:        SetLimit

## StepLongClass Methods

The StepLongClass inherits all the methods of the StepClass from which it is derived. See *StepClass Methods* for more information.

In addition to (or instead of) the inherited methods, the StepLongClass contains the following methods:

### **GetPercentile (return a value's percentile:StepLongClass)**

#### **GetPercentile( *value* ), VIRTUAL**

---

**GetPercentile** Returns the specified *value*'s percentile relative to the StepLongClass object's boundaries.

*value* A constant, variable, EQUATE, or expression that specifies the value for which to calculate the percentile.

The **GetPercentile** method returns the specified *value*'s percentile relative to the StepLongClass object's upper and lower boundaries. For example, if the bounds are 0 and 1000 then GetPercentile(750) returns 75.

Implementation: The SetLimit method sets the StepLongClass object's upper and lower boundaries.

Return Data Type: BYTE

Example:

```
IF FIELD() = ?Locator           !focus on locator field
  IF EVENT() = EVENT:Accepted   !if accepted
    MyBrowse.TakeAcceptedLocator !BrowseClass handles it
    ?MyList{PROP:VScrollPos}=MyStep.GetPercentile(Locator) !position thumb to match
  END
END
```

See Also: SetLimit

## GetValue (return a percentile's value:StepLongClass)

### GetValue( *percentile* ), VIRTUAL

---

**GetValue** Returns the specified *percentile's* value relative to the StepLongClass object's boundaries.

*percentile* An integer constant, variable, EQUATE, or expression that specifies the percentile for which to retrieve the value.

The **GetValue** method returns the specified *percentile's* value relative to the StepLongClass object's upper and lower boundaries. For example, if the bounds are 0 and 1000 then GetValue(25) returns '250'.

Implementation: The SetLimit method sets the StepLongClass object's upper and lower boundaries.

Return Data Type: STRING

Example:

```
IF FIELD() = ?MyList                                !focus on browse list
  IF EVENT() = EVENT:ScrollDrag                      !if thumb moved
    Locator=MyStep.GetValue(?MyList{PROP:VScrollPos})!update locator to match
  END
END
```

See Also: SetLimit

## SetLimit (set smooth data distribution:StepLongClass)

**SetLimit**( *lower*, *upper* ), VIRTUAL

---

<b>SetLimit</b>	Sets the StepLongClass object's evenly distributed steps between <i>upper</i> and <i>lower</i> .
<i>lower</i>	An integer constant, variable, EQUATE, or expression that specifies the StepLongClass object's lower boundary.
<i>upper</i>	An integer constant, variable, EQUATE, or expression that specifies the StepLongClass object's upper boundary.

The **SetLimit** method sets the StepLongClass object's evenly distributed steps between *upper* and *lower*. The StepLongClass object (GetPercentile and GetValue methods) uses these steps to estimate key values and percentiles for the processed data.

Implementation: The BrowseClass.ResetThumbLimits (a PRIVATE method) and the ProcessClass.SetProgressLimits methods call the SetLimit method to calculate the expected data distribution for the data. The SetLimit method sets 100 evenly distributed "steps" or markers between *lower* and *upper*.

Example:

```
MyStep.SetLimit(1,9700)           !establish scrollbar steps and boundaries
```

See Also:     GetPercentile, GetValue, ProcessClass.SetProgressLimits



# StepLocatorClass

## StepLocatorClass Overview

The StepLocatorClass is a LocatorClass that accepts a *single character* search value, and does a *continuous (wrap around) search* starting from the current item so you can, for example, find the next item that begins with the search value (say, 'T'), then continue to the next item that begins with the same search value, etc. If there are no matching values, the step locator proceeds to the next highest item.

Use a Step Locator when the search field is a STRING, CSTRING, or PSTRING, a single character search is sufficient (a step locator is not appropriate when there are many key values that begin with the same character), and you want the search to take place immediately upon the end user's keystroke. Step Locators are not appropriate for numeric keys.

## StepLocatorClass Concepts

A Step Locator is a single-character locator with no locator control required.

The StepLocatorClass lets you specify a locator control and a sort field on which to search (the free key element) for a BrowseClass object. The BrowseClass object uses the StepLocatorClass to locate and scroll to the nearest matching item.

When the BrowseClass LIST has focus and the user types a character, the BrowseClass object advances the list to the next matching item (or the subsequent item if there is no match).

## StepLocatorClass Relationship to Other Application Builder Classes

The BrowseClass uses the StepLocatorClass to locate and scroll to the nearest matching item. Therefore, if your program's BrowseClass objects use a Step Locator, your program must instantiate the StepLocatorClass for each use. Once you register the StepLocatorClass object with the BrowseClass object (see BrowseClass.AddLocator), the BrowseClass object uses the StepLocatorClass object as needed, with no other code required. See the Conceptual Example.

## StepLocatorClass ABC Template Implementation

The ABC BrowseBox template generates code to instantiate the StepLocatorClass for your BrowseBoxes. The StepLocatorClass objects are called BRW*n*::Sort*#*:Locator, where *n* is the template instance number and *#* is the sort sequence (id) number. As this implies, you can have a different locator for each BrowseClass object sort order.

You can use the BrowseBox's **Locator Behavior** dialog (the **Locator Class** button) to derive from the EntryLocatorClass. The templates provide the derived class so you can modify the locator's behavior on an instance-by-instance basis.

## StepLocatorClass Source Files

The StepLocatorClass source code is installed by default to the Clarion \LIBSRC folder. The StepLocatorClass source code and its respective components are contained in:

ABBROWSE.INC	StepLocatorClass declarations
ABBROWSE.CLW	StepLocatorClass method definitions

## StepLocatorClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a BrowseClass object and related objects, including a StepLocatorClass object. The example initializes and page-loads a LIST, then handles a number of associated events, including scrolling, updating, and locating records.

Note that the WindowManager and BrowseClass objects internally handle the normal events surrounding the locator.

```

PROGRAM
  INCLUDE('ABWINDOW.INC')           !declare WindowManager class
  INCLUDE('ABBROWSE.INC')           !declare BrowseClass and Locator
  MAP
  END
State          FILE,DRIVER('TOPSPEED'),PRE(ST),THREAD
StateCodeKey   KEY(ST:STATECODE),NOCASE,OPT
Record         RECORD,PRE()
STATECODE     STRING(2)
STATENAME     STRING(20)
              END
              END
StView        VIEW(State)           !declare VIEW to process
              END
StateQ        QUEUE                 !declare Q for LIST
ST:STATECODE  LIKE(ST:STATECODE)
ST:STATENAME  LIKE(ST:STATENAME)
ViewPosition  STRING(512)
              END
Access:State  CLASS(FileManager)    !declare Access:State object
Init         PROCEDURE
              END
Relate:State  CLASS(RelationManager) !declare Relate:State object
Init         PROCEDURE
              END
VCRRequest   LONG(0),THREAD
StWindow     WINDOW('Browse States'),AT(, ,123,152),IMM,SYSTEM,GRAY
              LIST,AT(8,5,108,124),USE(?StList),IMM,HVSCROLL,FROM(StateQ),|
              FORMAT(' 27L(2)|M~CODE~@s2@80L(2)|M~STATENAME~@s20@')
              END

```

```

ThisWindow CLASS(WindowManager)      !declare ThisWindow object
Init      PROCEDURE(),BYTE,PROC,VIRTUAL
Kill      PROCEDURE(),BYTE,PROC,VIRTUAL
      END
BrowseSt  CLASS(BrowseClass)         !declare BrowseSt object
Q         &StateQ
      END
StLocator StepLocatorClass           !declare StLocator object
StStep    StepStringClass            !declare StStep object

CODE
ThisWindow.Run()                     !run the window procedure

ThisWindow.Init PROCEDURE()          !initialize things
ReturnValue  BYTE,AUTO
CODE
ReturnValue = PARENT.Init()          !call base class init
IF ReturnValue THEN RETURN ReturnValue.
Relate:State.Init                    !initialize Relate:State object
SELF.FirstField = ?StList            !set FirstField for ThisWindow
SELF.VCRRequest &= VCRRequest       !VCRRequest not used
Relate:State.Open                    !open State and related files
!Init BrowseSt object by naming its LIST,VIEW,Q,RelationManager & WindowManager
BrowseSt.Init(?StList,StateQ.ViewPosition,StView,StateQ,Relate:State,SELF)
OPEN(StWindow)
SELF.Opened=True
BrowseSt.Q &= StateQ                 !reference the browse QUEUE
StStep.Init(+ScrollSort:AllowAlpha,ScrollBy:Runtime)!initialize the StStep
object
BrowseSt.AddSortOrder(StStep,ST:StateCodeKey)!set the browse sort order
BrowseSt.AddLocator(StLocator)       !plug in the browse locator
StLocator.Init(,ST:STATECODE,1,BrowseSt) !initialize the locator object
BrowseSt.AddField(ST:STATECODE,BrowseSt.Q.ST:STATECODE) !set a column to browse
BrowseSt.AddField(ST:STATENAME,BrowseSt.Q.ST:STATENAME) !set a column to browse
SELF.SetAlerts()                    !alert any keys for ThisWindow
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()          !shut down things
ReturnValue  BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()          !call base class shut down
IF ReturnValue THEN RETURN ReturnValue.
Relate:State.Close                  !close State and related files
Relate:State.Kill                   !shut down Relate:State object
GlobalErrors.Kill                   !shut down GlobalErrors object
RETURN ReturnValue

```

## StepLocatorClass Properties

The StepLocatorClass inherits all the properties of the LocatorClass from which it is derived. See *LocatorClass Properties* for more information.

## StepLocatorClass Methods

### StepLocatorClass Methods

The StepLocatorClass inherits all the methods of the LocatorClass from which it is derived. See *LocatorClass Methods* for more information.

In addition to (or instead of) the inherited methods, the StepLocatorClass contains the following methods:

### Set (restart the locator:StepLocatorClass)

#### **Set, DERIVED**

The **Set** method prepares the locator for a new search.

Implementation: The Set method does nothing because each new step locator search reprimed the locator's FreeElement--since the step locator is a single character search.

Example:

```
BrowseClass.SetSort PROCEDURE(BYTE B,BYTE Force)
CODE
IF SELF.SetSort(B)
  IF ~SELF.Sort.Locator &= NULL
    SELF.Sort.Locator.Set
  END
END
```

## TakeKey (process an alerted keystroke:StepLocatorClass)

### TakeKey, DERIVED

The **TakeKey** method processes an alerted keystroke for the LIST control and returns a value indicating whether the browse list display must change.

**Tip:** By default, all alphanumeric keys are alerted for LIST controls.

Implementation: The TakeKey method primes the FreeElement property with the appropriate search value, then returns one (1) if a new search is required or returns zero (0) if no new search is required. A search is required only if the keystroke is a valid search character.

Return Data Type: BYTE

Example:

```
IF SELF.Sort.Locator.TakeKey()      ! process the search key
  SELF.Reset(1)                      ! if valid, reset the view
  SELF.ResetQueue( Reset:Done )      ! and the browse queue
END
```

See Also: FreeElement

---

# StepRealClass

## StepRealClass Overview

The StepRealClass is a StepClass that handles fractional (non-integer) keys with a normal distribution (data is evenly distributed between the lowest and highest key values).

## StepRealClass Concepts

The StepRealClass object applies a normal data distribution between its upper and lower boundaries. Use the SetLimit method to set the expected data distribution for the StepRealClass object. Use the StepRealClass with non-integer numeric keys that have a normal distribution (data is evenly distributed between the lowest and highest key values).

**Tip:** Use the StepLongClass for integer numeric keys. Use the StepStringClass for alphanumeric keys. Use the StepCustomClass for keys with skewed distributions.

## StepRealClass Relationship to Other Application Builder Classes

The BrowseClass and the ProcessClass optionally use the StepRealClass. Therefore, if your BrowseClass or ProcessClass uses the StepRealClass, your program must instantiate the StepRealClass for each use. See the Conceptual Example.

## StepRealClass ABC Template Implementation

The ABC Templates (BrowseBox, Process, and Report) automatically include all the classes and generate all the code necessary to use the StepRealClass with your BrowseBoxes, Reports, and Processes.

---

## Process and Report Procedure Templates

By default, the Process and Report templates declare a StepRealClass for fractional keys called ProgressMgr. You can use the **Report Properties** Classes tab (the **Progress Class** button) or the **Process Properties** General tab (the **Progress Manager** button) to derive from the StepRealClass instead. The templates provide the derived class so you can modify the ProgressMgr behavior on an instance-by-instance basis.

---

## Browse Procedure and BrowseBox Control Templates

By default, the BrowseBox template declares a StepRealClass for non-integer numeric keys called BRWn::Sort#:StepClass, where *n* is the BrowseBox template instance number, and # is the sort order sequence (identifies the key). You can use the BrowseBox's **Scroll Bar Behavior** dialog--**Step Class** button to derive from the StepRealClass so you can modify the StepRealClass behavior on an instance-by-instance basis.

## StepRealClass Source Files

The StepRealClass source code is installed by default to the Clarion \LIBSRC folder. The StepRealClass source code and its respective components are contained in:

ABBROWSE.INC	StepRealClass declarations
ABBROWSE.CLW	StepRealClass method definitions

## StepRealClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a StepRealClass object and related objects. The example batch-processes a Customer file on a fractional (non-integer) key--CustomerID.

```

INCLUDE('ABBROWSE.INC')
INCLUDE('ABREPORT.INC')

MAP
CustomerProcess PROCEDURE
END

CustomerProcess PROCEDURE

FilesOpened      BYTE
Thermometer      BYTE
Process:View     VIEW(Customer)
                END
ProgressWindow WINDOW('Progress...'),AT(,,142,59),CENTER,TIMER(1),GRAY,DOUBLE
                PROGRESS,USE(Thermometer),AT(15,15,111,12),RANGE(0,100)
                STRING(''),AT(0,3,141,10),USE(?UserString),CENTER
                STRING(''),AT(0,30,141,10),USE(?PctText),CENTER
                BUTTON('Cancel'),AT(45,42,50,15),USE(?Cancel)
                END

ThisWindow CLASS(ReportManager)
Init       PROCEDURE(),BYTE,PROC,VIRTUAL
Kill       PROCEDURE(),BYTE,PROC,VIRTUAL
                END

ThisProcess ProcessClass      !declare ThisProcess object
ProgressMgr StepRealClass     !declare ProgressMgr object

CODE
GlobalResponse = ThisWindow.Run()

ThisWindow.Init PROCEDURE()
ReturnValue     BYTE,AUTO
CODE
```

```
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?Thermometer
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
Relate:Customer.Open
FilesOpened = True
OPEN(ProgressWindow)
SELF.Opened=True
ProgressMgr.Init(ScrollSort:AllowNumeric)    !initialize ProgressMgr object
                                              ! ignores inapplicable parameters
ThisProcess.Init(Process:View,Relate:Customer,?PctText,Thermometer,ProgressMgr,CUS:ID)
ThisProcess.AddSortOrder(CUS:CustomerIDKey)
SELF.Init(ThisProcess)
SELF.AddItem(?Progress:Cancel,RequestCancelled)
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
  IF FilesOpened
    Relate:Customer.Close
  END
RETURN ReturnValue
```

## StepRealClass Properties

### **StepRealClass Properties**

The StepRealClass inherits all the properties of the StepClass from which it is derived. See *StepClass Properties* for more information.

In addition to its inherited properties, the StepRealClass also contains the following properties:

### **Low (lower boundary:StepRealClass)**

#### **Low REAL**

The **Low** property contains the value of the StepRealClass object's lower boundary.

The SetLimit method sets the value of the Low property.

See Also:      SetLimit

### **High (upper boundary:StepRealClass)**

#### **High REAL**

The **High** property contains the value of the StepRealClass object's upper boundary.

The SetLimit method sets the value of the High property.

See Also:      SetLimit



**GetValue (return a percentile's value:StepRealClass)****GetValue( *percentile* ), VIRTUAL**


---

**GetValue** Returns the specified *percentile's* value relative to the StepRealClass object's boundaries.

*percentile* An integer constant, variable, EQUATE, or expression that specifies the percentile for which to retrieve the value.

The **GetValue** method returns the specified *percentile's* value relative to the StepRealClass object's upper and lower boundaries. For example, if the bounds are 0 and 1000 then GetValue(25) returns '250'.

Implementation: The SetLimit method sets the StepRealClass object's upper and lower boundaries.

Return Data Type: STRING

Example:

```

IF FIELD() = ?MyList           !focus on browse list
  IF EVENT() = EVENT:ScrollDrag !if thumb moved
    Locator=MyStep.GetValue(?MyList{PROP:VScrollPos})!update locator to match
  END
END

```

See Also: SetLimit

## SetLimit (set smooth data distribution:StepRealClass)

**SetLimit**( *lower*, *upper* ), VIRTUAL

---

**SetLimit** Sets the StepRealClass object's evenly distributed steps between *upper* and *lower*.

*lower* An integer constant, variable, EQUATE, or expression that specifies the StepRealClass object's lower boundary.

*upper* An integer constant, variable, EQUATE, or expression that specifies the StepRealClass object's upper boundary.

The **SetLimit** method sets the StepRealClass object's evenly distributed steps between *upper* and *lower*. The StepRealClass object (GetPercentile and GetValue methods) uses these steps to estimate key values and percentiles for the processed data.

Implementation: The BrowseClass.ResetThumbLimits (a PRIVATE method) and the ProcessClass.SetProgressLimits methods call the SetLimit method to calculate the expected data distribution for the data. The SetLimit method sets 100 evenly distributed "steps" or markers between *lower* and *upper*.

Example:

```
MyStep.SetLimit(1,9700) !establish scrollbar steps and boundaries
```

See Also: GetPercentile, GetValue, ProcessClass.SetProgressLimits



# StepStringClass

## StepStringClass Overview

The StepStringClass is a StepClass that handles alphanumeric keys with a normal distribution (data is evenly distributed between the lowest and highest key values) or with English Alphabet or US Surname distribution. You can provide information about the data distribution so that the StepStringClass object returns accurate feedback about the data being processed.

## StepStringClass Concepts

You can set the expected data distribution for a StepStringClass object--the StepStringClass object applies one of several predefined data distributions. Use the Init and SetLimit methods to set the expected data distribution for the StepStringClass object.

For example, your NameKey may contain US Surname values ranging from 'Aabel' to 'Zuger.' If your StepClass assumes the values are evenly distributed between these values, then your progress bars and vertical scroll bar thumbs will give an inaccurate visual representation of the data. However, if your StepClass assumes a typical US Surname distribution, then your progress bars and vertical scroll bar thumbs will give an accurate visual representation of the data.

Use the StepStringClass with alphanumeric keys that have a normal distribution (data is evenly distributed between the lowest and highest key values) or with English Alphabet or US Surname distribution.

**Tip:** Use the StepLongClass for integer keys with normal distribution. Use the StepRealClass for fractional keys with normal distribution. Use the StepCustomClass for numeric or alphanumeric keys with skewed distribution.

## StepStringClass Relationship to Other Application Builder Classes

The BrowseClass and the ProcessClass optionally use the StepStringClass. Therefore, if your BrowseClass or ProcessClass uses the StepStringClass, your program must instantiate the StepStringClass for each use. See the Conceptual Example.

## StepStringClass ABC Template Implementation

The ABC Templates (BrowseBox, Process, and Report) automatically include all the classes and generate all the code necessary to use the StepStringClass with your BrowseBoxes, Reports, and Processes.

### Process and Report Procedure Templates

---

By default, the Process and Report templates declare a StepStringClass for alphanumeric keys called ProgressMgr. You can use the **Report Properties** Classes tab (the **Progress Class** button) or the **Process Properties** General tab (the **Progress Manager** button) to derive from the StepStringClass instead. The templates provide the derived class so you can modify the ProgressMgr behavior on an instance-by-instance basis.

### Browse Procedure and BrowseBox Control Templates

---

By default, the BrowseBox template declares a StepStringClass for alphanumeric keys called BRWn::Sort#:StepClass, where *n* is the BrowseBox template instance number, and # is the sort order sequence (identifies the key). You can use the BrowseBox's **Scroll Bar Behavior** dialog to specify the expected data distribution (normal distribution, English alphabet, or US surname). You can use the **Step Class** button to derive from the StepStringClass so you can modify the StepStringClass behavior on an instance-by-instance basis.

## StepStringClass Source Files

The StepStringClass source code is installed by default to the Clarion \LIBSRC folder. The StepStringClass source code and its respective components are contained in:

ABBROWSE.INC	StepStringClass declarations
ABBROWSE.CLW	StepStringClass method definitions

## StepStringClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a StepStringClass object and related objects. The example initializes and page-loads a LIST, then handles a number of associated events, including scrolling.

The StepStringClass object's steps are calculated based on the poles of the actual browsed data-a list of State abbreviations.

```

PROGRAM
INCLUDE('ABWINDOW.INC')           !declare WindowManager class
INCLUDE('ABBROWSE.INC')           !declare BrowseClass & StepClasses
MAP
END

State      FILE,DRIVER('TOPSPEED'),PRE(ST),THREAD
StateCodeKey KEY(ST:STATECODE),NOCASE,OPT
Record     RECORD,PRE()
STATECODE  STRING(2)
STATENAME  STRING(20)
          END
          END

StView     VIEW(State)             !declare VIEW to process
          END

StateQ     QUEUE                   !declare Q for LIST
ST:STATECODE LIKE(ST:STATECODE)
ST:STATENAME LIKE(ST:STATENAME)
ViewPosition STRING(512)
          END

GlobalErrors ErrorClass
Access:State CLASS(FileManager)
Init        PROCEDURE
          END

Relate:State CLASS(RelationManager)
Init        PROCEDURE
          END

VCRRequest LONG(0),THREAD

StWindow WINDOW('Browse States'),AT(, ,123,152),IMM,SYSTEM,GRAY
          LIST,AT(8,5,108,124),USE(?StList),IMM,HVSCROLL,FROM(StateQ),|
          FORMAT(' 27L(2)|M~CODE~@s2@80L(2)|M~STATENAME~@s20@')
          BUTTON('&Insert'),AT(8,133),USE(?Insert)
          BUTTON('&Change'),AT(43,133),USE(?Change),DEFAULT
          BUTTON('&Delete'),AT(83,133),USE(?Delete)

```

```

        END

ThisWindow CLASS(WindowManager)
Init      PROCEDURE(),BYTE,PROC,VIRTUAL
Kill     PROCEDURE(),BYTE,PROC,VIRTUAL
        END

BrowseSt  CLASS(BrowseClass)          !declare BrowseSt object
Q        &StateQ
        END

StStep   StepStringClass             !declare StStep object

CODE
ThisWindow.Run()                    !run the window procedure

ThisWindow.Init PROCEDURE()          !initialize things
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
GlobalErrors.Init
Relate:State.Init
SELF.FirstField = ?StList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
Relate:State.Open
BrowseSt.Init(?StList,StateQ.ViewPosition,StView,StateQ,Relate:State,SELF)
OPEN(StWindow)
SELF.Opened=True
BrowseSt.Q &= StateQ
StStep.Init(+ScrollSort:AllowAlpha,ScrollBy:Runtime)!initialize the StStep object
BrowseSt.AddSortOrder(StStep,ST:StateCodeKey)    ! & plug in to the BrowseSt
            ! BrowseSt calls SetLimit to
            ! calculate data distribution
            ! from the poles of the data
BrowseSt.AddField(ST:STATECODE,BrowseSt.Q.ST:STATECODE)
BrowseSt.AddField(ST:STATENAME,BrowseSt.Q.ST:STATENAME)
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()          !shut down things
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
IF ReturnValue THEN RETURN ReturnValue.
Relate:State.Close
Relate:State.Kill

```

```
GlobalErrors.Kill  
RETURN ReturnValue
```

```
Access:State.Init PROCEDURE  
CODE  
PARENT.Init(State,GlobalErrors)  
SELF.FileNameValue = 'State'  
SELF.Buffer &= ST:Record  
SELF.AddKey(ST:StateCodeKey,'ST:StateCodeKey',0)
```

```
Relate:State.Init PROCEDURE  
CODE  
Access:State.Init  
PARENT.Init(Access:State,1)
```

## StepStringClass Properties

The `StepStringClass` inherits all the properties of the `StepClass` from which it is derived. See *StepClass Properties* for more information. In addition to its inherited properties, the `StepStringClass` also contains the following properties:

### LookupMode (expected data distribution)

#### LookupMode BYTE

The **LookupMode** property sets the *expected* data distribution the `StepStringClass` object implements. This, plus the *actual* data distribution, ultimately determines how "far" the indicator (scrollbar thumb or progress bar) actually moves as records are processed. The `Init` method sets the value of the `LookupMode` property.

Implementation: Valid data distribution options are U.S. surnames, English alphabet, and runtime data distribution calculated from the poles of the actual data. Corresponding `LookupMode` EQUATEs are declared in `ABBROWSE.INC` as follows:

```
ITEMIZE,PRE(ScrollBy)
Name    EQUATE  !U.S. surnames distribution
Alpha   EQUATE  !English alphabet distribution
Runtime EQUATE  !calculate distribution from runtime poles
END
```

The U.S. surnames and English alphabet data distributions are defined in `ABBROWSE.CLW` as follows:

```
Scroll:Alpha STRING(' AFANATB BFBNBTC CFCNCT' |
    &'D DFDNDTE EFENETF FFFNFT' |
    &'G GFGNGTH HFNHTI IFINIT' |
    &'J JFJNJTK KFKNKT LFLNLT' |
    &'M MFMNMTN NFNNTTO OFONOT' |
    &'P PFPNPTQ QNR RFRNRTS SF' |
    &'SNSTT TFTNTTU UFUNUTV VF' |
    &'VNVTW WFWNWTX XFXNXTY YF' |
    &'YNYTZ ZN')
Scroll:Name  STRING(' ALBAMEARNBAKBATBENBIABOBBRA' |
    &'BROBUACACCARCENCHRCOECONCORCRU' |
    &'DASDELDIADONDURELDEVEFELFISFLO' |
    &'FREFUTGARGIBGOLGOSGREGUTHAMHEM' |
    &'HOBHOTINGJASJONKAGKEAKIRKORKYO' |
    &'LATLEOLIGLOUMACMAQMARMAUMCKMER' |
    &'MILMONMORNATNOLOKEPAGPAUPETPIN' |
    &'PORPULRAUREYROBROSUBSALSCASCH' |
    &'SCRSHASIGSKI SNASOUSTESTISUNTAY' |
    &'TIRTUCVANWACWASWEIWIWIMWOLYOR')
```

See Also:        `Init`

## Root (the static portion of the step)

### **Root   &CSTRING, PROTECTED**

The **Root** property is a reference to a structure containing the static or non-determinative characters of a step. For example, if the step bounds are 'abbey' and 'abracadabra' then Root contains 'ab'. The related property TestLen is equal to the length of Root, that is, 2.

Implementation:    The GetPercentile and GetValue methods use the Root and TestLen properties to efficiently traverse the defined steps.

See Also:            GetPercentile, GetValue, TestLen

## SortChars (valid sort characters)

### **SortChars   &CSTRING**

The **SortChars** property is a reference to a structure containing the valid sort characters for the StepStringClass object. The StepStringClass object uses the SortChars property to compute steps. For example if SortChars contains only 'ABYZ' then that is the information the StepStringClass uses to compute your steps.

The Init method sets the value of the SortChars property.

Implementation:    The SortChars property only affects StepStringClass objects with a LookupMode specifying runtime data distribution. The SetLimit method computes the runtime data distribution.

See Also:            Init, LookupMode, SetLimit

**TestLen (length of the static step portion)****TestLen**      **BYTE, PROTECTED**

The **TestLen** property contains the length of the Root property. For example, if the step bounds are 'abbey' and 'abracadabra' then Root contains 'ab'. The related property TestLen is equal to the length of Root, that is, 2.

The Init method sets the value of the TestLen property.

Implementation:      The GetPercentile and GetValue methods use the Root and TestLen properties to efficiently traverse the defined steps.

The value of the TestLen property depends on the value of the LookupMode property. LookupMode of U.S. surnames uses TestLen of 3, English alphabet uses TestLen of 2, and runtime data distribution uses TestLen of 4.

See Also:              Init, LookupMode, Root

## StepStringClass Methods

The StepStringClass inherits all the methods of the StepClass from which it is derived. See *StepClass Methods* for more information.

In addition to (or instead of) the inherited methods, the StepStringClass contains the following methods:

### GetPercentile (return a value's percentile)

**GetPercentile**( *value* ), VIRTUAL

---

**GetPercentile** Returns the specified *value*'s percentile relative to the StepStringClass object's boundaries.

*value* A string constant, variable, EQUATE, or expression that specifies the value for which to calculate the percentile.

The **GetPercentile** method returns the specified *value*'s percentile relative to the StepStringClass object's upper and lower boundaries. For example, if the bounds are 'A' and 'Z' then GetPercentile('M') returns 50.

Implementation: The SetLimit method sets the StepStringClass object's upper and lower boundaries.

Return Data Type: BYTE

Example:

```

IF FIELD() = ?Locator           !focus on locator field
  IF EVENT() = EVENT:Accepted   !if accepted
    MyBrowse.TakeAcceptedLocator !BrowseClass handles it
    ?MyList{PROP:VScrollPos}=MyStep.GetPercentile(Locator) !position thumb to match
  END
END

```

See Also: SetLimit

## GetValue (return a percentile's value)

### GetValue( *percentile* ), VIRTUAL

---

**GetValue** Returns the specified *percentile's* value relative to the StepStringClass object's boundaries.

*percentile* An integer constant, variable, EQUATE, or expression that specifies the percentile for which to retrieve the value.

The **GetValue** method returns the specified *percentile's* value relative to the StepStringClass object's upper and lower boundaries. For example, if the bounds are 'A' and 'Z' then GetValue(50) returns 'M'.

Implementation: The SetLimit method sets the StepStringClass object's upper and lower boundaries.

Return Data Type: STRING

Example:

```
IF FIELD() = ?MyList                !focus on browse list
  IF EVENT() = EVENT:ScrollDrag      !if thumb moved
    Locator=MyStep.GetValue(?MyList{PROP:VScrollPos})!update locator to match
  END
END
```

See Also: SetLimit

**Init (initialize the StepStringClass object)****Init**( *controls*, *mode* )

---

<b>Init</b>	Initializes the StepStringClass object.
<i>controls</i>	An integer constant, variable, EQUATE, or expression that contains several important pieces of information for the StepClass object.
<i>mode</i>	An integer constant, variable, EQUATE, or expression that determines the data distribution points (or steps) the StepStringClass object implements.

The **Init** method initializes the StepStringClass object.

The *controls* parameter identifies for the StepClass object:

- the characters included in the calculated runtime distribution
- whether the key is case sensitive
- the direction of the sort (ascending or descending)

A *mode* parameter value of ScrollBy:Name gives U.S. surname distribution, ScrollBy:Alpha gives English alphabet distribution, and ScrollBy:Runtime gives a smooth data distribution from the poles of the actual data, as calculated by the SetLimit method.

Implementation: The Init method sets the value of the Controls and LookupMode properties. Set the value of the Controls property by adding together the applicable EQUATES declared in ABBROWSE.INC as follows:

```
ITEMIZE,PRE(ScrollSort)
AllowAlpha EQUATE(1) !include ABCDEFGHIJKLMNOPQRSTUVWXYZ
AllowAlt EQUATE(2) !include `!"#$%^&*()' '-=_+][#;~@:/.,?|\
AllowNumeric EQUATE(4) !include 0123456789
CaseSensitive EQUATE(8)!include abcdefghijklmnopqrstuvwxyz
Descending EQUATE(16) !the sort is descending
```

EQUATES for the *mode* parameter are declared in ABBROWSE.INC as follows:

```
ITEMIZE,PRE(ScrollBy)
Name EQUATE !US Surname distribution
Alpha EQUATE !English alphabet distribution
Runtime EQUATE !calculate normal distribution from data poles
END
```

Example:

```
MyStepStringClass.Init(ScrollSort:AllowAlpha+ScrollSort:AllowNumeric)
!program code
MyStepStringClass.Kill
```

See Also: StepClass.Controls, LookupMode, SetLimit

**Kill (shut down the StepStringClass object)****Kill, VIRTUAL**

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code.

Implementation: The Kill method frees memory allocated for the Ref, Root, and SortChars properties.

Example:

```
MyStepStringClass.Init(ScrollSort:AllowAlpha+ScrollSort:AllowNumeric)
!program code
MyStepStringClass.Kill
```

**SetLimit (set smooth data distribution:StepStringClass)****SetLimit( *lower*, *upper* ), VIRTUAL**

---

<b>SetLimit</b>	Sets the StepStringClass object's evenly distributed steps between <i>upper</i> and <i>lower</i> .
<i>lower</i>	A string constant, variable, EQUATE, or expression that specifies the StepStringClass object's lower boundary. The value may be numeric or alphanumeric.
<i>upper</i>	A string constant, variable, EQUATE, or expression that specifies the StepStringClass object's upper boundary. The value may be numeric or alphanumeric.

The **SetLimit** method sets the StepStringClass object's evenly distributed steps between *upper* and *lower*. The StepStringClass object (GetPercentile and GetValue methods) uses these steps to estimate key values and percentiles for the processed data.

Implementation: The BrowseClass.ResetThumbLimits (a PRIVATE method) and the ProcessClass.SetProgressLimits methods call the SetLimit method to calculate the expected data distribution for the data. The SetLimit method sets 100 evenly distributed "steps" or markers between *lower* and *upper*. SetLimit considers the Controls property (as set by the Init method) when calculating the expected data distribution.

Example:

```
MyStep.SetLimit('A','Z')      !establish uppercase alphabetic scrollbar limits
```

See Also: GetPercentile, GetValue, Init, ProcessClass.SetProgressLimits, StepClass.Controls

**SetLimitNeeded (return static/dynamic boundary flag:StepStringClass)****SetLimitNeeded, VIRTUAL**

The **SetLimitNeeded** method returns a value indicating whether the StepClass object's steps and boundaries are static (set at compile time) or dynamic (set at runtime). A return value of one (1) indicates dynamic boundaries that may need to be reset when the monitored result set changes (records are added, deleted, or filtered). A return value of zero (0) indicates the boundaries are fixed at compile time (name or alpha distribution) and are not adjusted when the monitored result set changes.

Implementation: The SetLimitNeeded method returns one (1 or True) if the LookupMode property equals ScrollBy:RunTime; otherwise it returns zero (0 or False).

Return Data Type: BYTE

Example:

```

BrowseClass.ResetThumbLimits PROCEDURE
HighValue ANY
CODE
  IF SELF.Sort.Thumb &= NULL OR ~SELF.Sort.Thumb.SetLimitNeeded()
    RETURN
  END
  SELF.Reset
  IF SELF.Previous()
    RETURN
  END
  HighValue = SELF.Sort.FreeElement
  SELF.Reset
  IF SELF.Next()
    RETURN
  END
  SELF.Sort.Thumb.SetLimit(SELF.Sort.FreeElement,HighValue)

```

See Also: StepClass.SetLimitNeeded

# TagHTMLHelp Class

## TagHTMLHelpOverview

HTML Help is emerging as the new standard help file format. HTML Help is distributed in a single (.chm) file. This file is compressed and made from several .html files. All images, table of contents, index, and search files are compiled into this single .chm file. This makes for easier distribution of your help system.

HTML Help is available on any 32-bit windows platform, Windows 95/98, Windows 2000, and NT 4. On Windows 95 and NT 4, the HTML Help Display Engine, Internet Explorer 4.0 or later, or Microsoft Office 2000 must be installed on the user's system.

## TagHTMLHelp Class Concepts

Clarion's HTML Help implementation is a DLL that communicates with the Microsoft HTML System. The DLL is accessed by the TagHTMLHelp Class. Clarion's HTML Help intercepts the call to the Windows Help system when the F1 key is pressed. These calls are re-directed to the HTML Help System. There are several templates available which make the HTML Help system easily accessible from your Clarion program. Accessing Topics, Table of Contents, Indexing, Searching, Keywords, and Hyperlinks are a snap.

## Relationship to Other Application Builder Classes

The TagHTMLHelp class works independently of all other ABC classes.

## TagHTMLHelp ABC Template Implementation

Once the cwHH global template extension is added to the application, the templates instantiate a TagHTMLHelp object into the generated code for the main procedure of the application. This is also where the TagHTMLHelp object is initialized.

## TagHTMLHelp Source Files

The TagHTMLHelp class declarations are installed by default to the Clarion \LIBSRC folder. The TagHTMLHelp component is distributed as a LIB/DLL, therefore the source code for the methods is not available. However, the methods are defined in this chapter and may be implemented in applications provided the required LIB/DLL is available at runtime.

cwHH.INC	Help Class Definition
cwHHL.INC	Help Class Definition Local Compile
cwHH60.dll	Help DLL
cwHH60.lib	Help LIB
cwHHL.lib	Help Local LIB

## TagHTMLHelp Methods

The TagHTMLHelp class contains the following methods.

### **AlinkLookup (associative link lookup)**

**AlinkLookup**( *keywords*, | *messagetext*, *messagetitle* | )

<b>AlinkLookup</b>	Look up one or more Associative link (Alink) names within a compiled help (.chm) file.
<i>keywords</i>	A string constant, variable, EQUATE, or expression containing the Associative link ( <i>keyword</i> ) to search for. Multiple keywords can be used by concatenating them with semicolons (;).
<i>messagetext</i>	A string constant, variable, EQUATE, or expression containing the message to display in a message box if the Alink is not found.
<i>messagetitle</i>	A string constant, variable, EQUATE, or expression containing the title of the message box dialog.

The **AlinkLookup** method looks up one or more Associative link (Alink) names within a compiled help (.chm) file. Associative links are used to link related help topics to each other. When a link that contains an Alink is clicked, a popup window appears with the list of related topics.

Example:

```
oHH.AlinkLookup(sSearch,sMsg,sTitle )!Find sSearch Alink
```

**CloseHelp (close HTML help file)****CloseHelp**

The **CloseHelp** method closes any Help windows opened by the application.

Example:

```
oHH.CloseHelp()!Close all Help windows
```

**GetHelpFile (get help file name)****GetHelpFile**

The **GetHelpFile** method retrieves the current HTML Help (.chm) file name. The file name is returned.

Return Data Type:      **STRING**

Example:

```
sHelpFileName =oHH.GetHelpFile()!Get Help file name
```

**GetTopic (get current topic name)****GetTopic**

The **GetTopic** method retrieves the current HTML Help topic name. The topic name is returned.

Return Data Type:      **STRING**

Example:

```
sHelpFileName =oHH.GetTopic()!Get topic name
```

**Init (initialize HTML Help object)**

**Init**( *helpfile*, [*keycode*])

<b>AlinkLookup</b>	Look up one or more Associative link (Alink) names within a compiled help (.chm) file.
<i>helpfile</i>	A string constant, variable, EQUATE, or expression containing the help file name. This should be enclosed in single quotes.
<i>keycode</i>	A numeric keycode or keycode EQUATE. If omitted, the default value is 123, F12Key. Keycode equates can be found in KEYCODES.CLW.

The **Init** method initializes the TagHTMLHelp object and opens the specified help file. This method also aliases the F1 key to the F12 (keycode 123) key by default in order for the rest of the methods to be able to check for this keystroke. If F12 is used by some other function in the application this may be changed to whatever keystroke is unused in the application.

Example:

```
oHH.Init( 'cwhh.chm' )
```

Return Data Type:     **BYTE**

## KeyWordLookup (lookup keyword)

**KeyWordLookup**( *keywords*, | *messagetext*, *messagetitle* |)

**KeyWordLookup** Look up one or more keywords.

*keywords* A string constant, variable, EQUATE, or expression containing the Associative link (*keyword*) to search for. Multiple keywords can be used by concatenating them with semicolons (;).

*messagetext* A string constant, variable, EQUATE, or expression containing the message to display in a message box if the Alink is not found.

*messagetitle* A string constant, variable, EQUATE, or expression containing the title of the message box dialog.

The **KeyWordLookup** method looks up Keywords within a compiled help (.chm) file. Keywords are a collection of words and phrases that make up the help file's index. They are used to find specific help topics.

Example:

```
oHH.KeyWordLookUp( 'Demo' )
```

## Kill (shutdown the TagHTMLHelp object)

**Kill**

The **Kill** method shuts down the TagHTMLHelp object by freeing any memory used during the life of the object.

Example:

```
oHH.Kill()
```

Return Data Type:     BYTE

**SetHelpFile (set the current HTML Help file name)**

**SetHelpFile**( *helpfile* )

<b>SetHelpFile</b>	Set the HTML Help file name.
<i>helpfile</i>	A string constant, variable, EQUATE, or expression containing the help file name. This should be enclosed in single quotes.

The **SetHelpFile** method sets the name of the current compiled HTML help file (.chm) that the application will use

Example:

```
oHH.SetHelpFile('Demo.chm ' ) !Set help file name for application
```

## SetTopic (set the current HTML Help file topic)

**SetTopic**( *iControl* [, *topic*]

<b>SetTopic</b>	Set the HTML Help file topic.
<i>control</i>	An integer constant, variable, EQUATE, or expression containing the control number (FEQ).
<i>topic</i>	A string constant, variable, EQUATE, or expression containing a help file topic. This should be enclosed in single quotes.

The **SetTopic** method sets the current topic name. This should be set before displaying the topic. To tie a help topic to load when a specific control has focus, specify an optional *control* value. To specify a help topic use the 'SectionName/PageName.htm' naming convention. If the topic page is in the default section you simply use 'PageName.htm'. Remember that when using the Section/Page convention, the slash is a forward slash.

Example:

```
oHH.SetTopic( 'Class_Interface/Class_Interface.htm' )  
oHH.SetTopic( ?OKButton, 'myform/ok.htm' ) .
```

## ShowIndex (open the HTML Help index tab)

**ShowIndex**( *keyword*)

<b>ShowIndex</b>	Open HTML Help index tab.
<i>keyword</i>	A string constant, variable, EQUATE, or expression containing the keyword to search for. The keyword should be enclosed in single quotes.

The **ShowIndex** method opens the Index tab in the Navigation pane of the HTML Help Viewer and searches for the keyword, if specified.

Example:

```
oHH.ShowIndex()!Opens Index tab  
oHH.ShowIndex('Demo ' )!Opens Index tab and searches for keyword
```

## ShowSearch (open the HTML Help search tab)

### ShowSearch

The **ShowSearch** method opens the Search tab in the Navigation pane of the HTML Help Viewer.

Example:

```
oHH.ShowSearch( )
```

## ShowTOC (open the HTML Help contents tab)

### ShowTOC

The **ShowTOC** method opens the Contents tab in the Navigation pane of the HTML Help Viewer.

Example:

```
oHH.ShowTOC( )
```

## ShowTopic (display a help topic)

### ShowTopic( *topic* )

<b>ShowTopic</b>	Display the HTML Help file topic.
<i>topic</i>	A string constant, variable, EQUATE, or expression containing a help file topic. This should be enclosed in single quotes.

The **ShowTopic** method displays the help topic specified by the SetTopic method. If a *topic* is specified it will override the current topic name, however after the call to display the topic, the default topic name is still the default. This method does not change the default topic name.

Example:

```
oHH.ShowTopic( )
```

---

# TextWindowClass

## TextWindowClass Overview

The TextWindowClass is a Class that manages a Window that is used for the editing of EIP memo and large string fields.

## TextWindowClass Concepts

The TextWindowClass manages a window (TxtWindow) that is defined in ABEIP.CLW. This window contains a TEXT control along with OK and Cancel buttons.

## Relationship to Other Application Builder Classes

The TextWindowClass is derived from the WindowManager class. It manages the opening and closing of a special text window. This text window is where the memo or large string can be edited. The class also provides a method to handle the OK and Cancel buttons.

The TextWindowClass is instantiated within the EditTextClass.TakeEvent method. When the user presses the ellipsis button to edit the text, the special window is initiated.

## ABC Template Implementation

If the EditTextClass is used to allow special Edit-in-Place editing of memos or large string fields, the TextWindow class is automatically generated. This class manages the window that opens when the user presses the ellipsis button on the special EIP Combo control.

## TextWindowClass Source Files

The TextWindowClass source code is installed by default to the Clarion \LIBSRC folder. The specific TextWindowClass source code and their respective components are contained in:

ABEIP.INC	TextWindowClass declarations
ABEIP.CLW	TextWindowClass method definitions

## TextWindowClass Properties

The TextWindowClass inherits all the properties of the WindowManager from which it is derived.

In addition to the inherited properties, the TextWindowClass contains the following properties:

### **SeE (ending edit position)**

**SeE    UNSIGNED, AUTO**

The **SeE** property identifies the ending edit position (character) in a TEXT control.

Implementation:        The SeE property value is initialized in the TextWindowClass.Init method.

See Also:                TextWindowClass.Init

### **SeS (starting edit position)**

**SeS    UNSIGNED, AUTO**

The **SeS** property identifies the starting edit position (character) in a TEXT control.

Implementation:        The SeS property value is initialized in the TextWindowClass.Init method.

See Also:                TextWindowClass.Init

### **Txt (field equate number)**

**Txt    ANY, AUTO**

The **Txt** property is the field equate number (FEQ) of the TEXT control holding the data available for edit.

Implementation:        The Txt property is initialized in the TextWindowClass.Init method.

See Also:                TextWindowClass.Init

## TextWindowClass Methods

The TextWindowClass inherits all the methods of the WindowManager class from which it is derived.

In addition to (or instead of) the inherited methods, the TextWindowClass contains the following methods:

### Init (initialize TextWindow object)

**Init**(*entryFEQ*, *title*)  
**Init**, **DERIVED**, **PROC**

<b>Init</b>	Initializes and opens the special Text window.
<i>entryFEQ</i>	An integer constant, variable, EQUATE, or expression that represents the field number (FEQ) of the text control.
<i>title</i>	A string constant, variable, EQUATE, or expression that sets the title bar text in the dialog containing the text control.

The **Init** method initializes and opens the special Text window. A Level:Benign is returned from this method.

Implementation: The EditTextClass.TakeEvent method calls the TextWindowClass.Init method and initiates the ACCEPT loop for the text window.

Return Data Type: BYTE for Init, DERIVED, PROC prototype

See Also: EditTextClass.TakeEvent

### Kill (shutdown TextWindow object)

**Kill**, **DERIVED**, **PROC**

The **Kill** method closes the Text window and updates the control with the new value. It also disposes any memory allocated during the object's lifetime by calling the PARENT.KILL method. KILL returns a value to indicate the status of the shutdown.

Return Data Type: BYTE

**TakeAccepted (process window controls)****TakeAccepted, PROC, DERIVED**

The **TakeAccepted** method processes EVENT:Accepted for the OK and Cancel button controls. TakeAccepted returns Level:Benign to indicate processing of this event should continue normally; Level:Notify is returned to indicate processing is complete for this event and the ACCEPT loop should CYCLE; Level:Fatal is returned to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeAccepted method calls the WindowManager.SetResponse to register the methods' result (complete or cancelled) and triggers the normal procedure shutdown.

Return Data Type: BYTE

See Also: WindowManager.SetReponse

# ToolBarClass

## ToolBarClass Overview

ToolBarClass and ToolBarTarget objects work together to reliably "convert" an event associated with a toolbar button into an appropriate event associated with a specific control or window.

ToolBarClass objects communicate with zero or more ToolBarTarget objects. Each ToolBarTarget object is associated with a specific entity, such as a browse list, relation tree, or update form. The ToolBarClass object forwards events and method calls to the *active* ToolBarTarget object. Only one target is active at a time.

This lets you use a single toolbar to drive a variety of targets, such as update forms, browse lists, relation tree lists, etc. A single toolbar can even drive multiple targets (two or more BrowseBoxes) in a single procedure.

## ToolBarClass Concepts

Within an MDI application, the ToolBarClass and ToolBarTarget work together to reliably interpret and pass an event (EVENT:Accepted) associated with a toolbar button into an event associated with a specific control or window. For example, the end user CLICKS on a toolbar button (say the "Insert" button) on the MDI application frame. The frame procedure forwards the event to the active thread (POST(EVENT:Accepted,ACCEPTED(),SYSTEM{Prop:Active})). The active thread (procedure) manages a window that displays two LIST controls, and one of the LISTs has focus. This procedure has a ToolBarClass object plus a ToolBarTarget object for each LIST control. The ToolBarClass object takes the event (ToolBarClass.TakeEvent)<sub>1</sub> and forwards the event to the *active* ToolBarTarget object (the target that represents the LIST with focus). The ToolBarTarget object takes the event (ToolBarListBoxClass.TakeEvent) and handles it by posting an appropriate event to a specific control or to the window, for example:

```
POST(EVENT:ACCEPTED,SELF.InsertButton) !insert a record
POST(EVENT:PageDown,SELF.Control)      !scroll a LIST
POST(EVENT:Completed)                  !complete an update form
POST(EVENT:CloseWindow)                !select a record
etc.
```

If the procedure has a WindowManager object, the WindowManager object takes the event (WindowManager.TakeEvent) and forwards it to the ToolBarClass object (WindowManager.TakeAccepted).

## ToolBarClass Relationship to Other Application Builder Classes

### ToolBarTarget

---

The ToolBarClass object keeps a list of ToolBarTarget objects so it can forward events and method calls to a particular target. Each ToolBarTarget object is associated with a specific entity,

such as a browse list, relation tree, or update form. At present, the ABC Library has three classes derived from the `ToolbarTarget`:

<code>ToolbarListboxClass</code>	BrowseClass toolbar target
<code>ToolbarReltreeClass</code>	Reltree control toolbar target
<code>ToolbarUpdateClass</code>	Form procedure toolbar target

These `ToolbarTarget` objects implement the event handling specific to the associated entity. There may be zero or more `ToolbarTarget` objects within a procedure; however, *only one is active* at a time. The `SetTarget` method sets the active `ToolbarTarget` object.

### **BrowseClass and WindowManager**

---

The `WindowManager` optionally uses the `ToolbarClass`, as does the `BrowseClass`. Therefore, if your program uses a `WindowManager` or `BrowseClass` object, it may also need the `ToolbarClass`. Much of this is automatic when you `INCLUDE` the `WindowManager` or `BrowseClass` headers (`ABWINDOW.INC` and `ABBROWSE.INC`) in your program's data section. See the Conceptual Example.

## ToolBarClass ABC Template Implementation

The ABC procedure templates instantiate a ToolbarClass object called Toolbar within each procedure containing a template that asks for global toolbar control--that is, the BrowseBox template, the FormVCRControls template, and the RelationTree template.

The templates generate code to instantiate the ToolbarClass object and to register the ToolbarClass object with the WindowManager object. You may see code such as the following in your template-generated procedures.

```
ToolBar    ToolbarClass    !declare Toolbar object
CODE
!
ThisWindow.Init PROCEDURE
    SELF.AddItem(Toolbar)                !register Toolbar with WindowManager
    BRW1.AddToolbarTarget(Toolbar)       !register BrowseClass as target
    Toolbar.AddTarget(REL1::Toolbar,?RelTree) !register RelTree as target
    SELF.AddItem(ToolbarForm)           !register update form as target
```

The WindowManager and BrowseClass are both programmed to use ToolbarClass objects. Therefore most of the interaction between these objects is encapsulated within the Application Builder Class code, and is only minimally reflected in the ABC Template generated code.

## ToolBar Class Source Files

The ToolbarClass source code is installed by default to the Clarion \LIBSRC folder. The ToolbarClass source code and its respective components are contained in:

ABTOOLBA.INC	ToolBarClass declarations
ABTOOLBA.CLW	ToolBarClass method definitions

## ToolBarClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a ToolBarClass object and related ToolBarTarget objects.

This example uses the ToolBarClass to allow a global toolbar to drive two separate but related LISTS within a single MDI procedure. The primary LIST shows client information and the related LIST shows phone numbers for the selected client. The toolbar drives whichever list has focus.

The program POSTs toolbar events to the active MDI window using the SYSTEM{Prop:Active} property. Then the local ToolBarClass object calls on the active ToolBarTarget object to handle the event.

```

PROGRAM

INCLUDE('ABBROWSE.INC')           !declare BrowseClass
INCLUDE('ABTOOLBA.INC')          !declare Toolbar classes
INCLUDE('ABWINDOW.INC')          !declare WindowManager
CODE
!program code

Main PROCEDURE                   !contains global toolbar
AppFrame APPLICATION('Toolbars'),AT(,,275,175),SYSTEM,MAX,RESIZE,IMM
MENUBAR
  ITEM('Browse Customers'),USE(?BrowseCustomer)
END
TOOLBAR,AT(0,0,400,22)           !must use ABTOOLBA.INC EQUATES:
BUTTON,AT(4,2),USE(?Top,Toolbar:Top),DISABLE,ICON('VCRFIRST.ICO'),FLAT
BUTTON,AT(16,2),USE(?PageUp,Toolbar:PageUp),DISABLE,ICON('VCRPRIOR.ICO'),FLAT
BUTTON,AT(28,2),USE(?Up,Toolbar:Up),DISABLE,ICON('VCRUP.ICO'),FLAT
BUTTON,AT(40,2),USE(?Down,Toolbar:Down),DISABLE,ICON('VCRDOWN.ICO'),FLAT
BUTTON,AT(52,2),USE(?PageDown,Toolbar:PageDown),DISABLE,ICON('VCRNEXT.ICO'),FLAT
BUTTON,AT(64,2),USE(?Bottom,Toolbar:Bottom),DISABLE,ICON('VCRLAST.ICO'),FLAT
END
END

Frame          CLASS(WindowManager)
Init           PROCEDURE(),BYTE,PROC,VIRTUAL
TakeAccepted  PROCEDURE(),BYTE,PROC,VIRTUAL
              END

Toolbar  ToolbarClass          !declare Toolbar object
CODE
Frame.Run()

Frame.Init PROCEDURE()
ReturnValue BYTE,AUTO
CODE

```

```

ReturnValue = PARENT.Init()
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(ToolBar)           !register Toolbar with WindowManager
OPEN(AppFrame)
SELF.Opened=True
SELF.SetAlerts()
RETURN ReturnValue
Frame.TakeAccepted PROCEDURE()
ReturnValue      BYTE,AUTO
Looped          BYTE
CODE
LOOP
  IF Looped THEN RETURN Level:Notify ELSE Looped=1.
  CASE ACCEPTED()
  OF Toolbar:First TO Toolbar:Last           !for EVENT:Accepted on toolbar
    POST(EVENT:Accepted,ACCEPTED(),SYSTEM{Prop:Active}) !transfer it to active thread
    CYCLE                                       ! and stop
  END
  ReturnValue = PARENT.TakeAccepted()
  IF ACCEPTED() = ?BrowseCustomer
    START(BrowseCustomer,050000)
  END
  RETURN ReturnValue
END

BrowseCustomer PROCEDURE           !contains local Toolbar and targets
CusView  VIEW(Customer)
        END
CusQ     QUEUE
CUS:CUSTNO  LIKE(CUS:CUSTNO)
CUS:NAME    LIKE(CUS:NAME)
ViewPosition STRING(512)
        END
PhView    VIEW(Phones)
        END
PhQ       QUEUE
PH:NUMBER  LIKE(PH:NUMBER)
PH:ID      LIKE(PH:ID)
ViewPosition STRING(512)
        END

```

```

CusWindow WINDOW('Browse Customers'),AT(,,246,131),IMM,SYSTEM,GRAY,MDI
LIST,AT(8,7,160,100),USE(?CusList),IMM,HVSCROLL,FROM(CusQ),|
  FORMAT('51R(2)|M~CUSTNO~C(0)@n-14@80L(2)|M~NAME~@s30@')
BUTTON('&Insert'),AT(17,111,45,14),USE(?InsertCus),SKIP
BUTTON('&Change'),AT(66,111,45,14),USE(?ChangeCus),SKIP,DEFAULT
BUTTON('&Delete'),AT(115,111,45,14),USE(?DeleteCus),SKIP
LIST,AT(176,7,65,100),USE(?PhList),IMM,FROM(PhQ),FORMAT('80L~Phones~L(1)')
BUTTON('&Insert'),AT(187,41,42,12),USE(?InsertPh),HIDE
BUTTON('&Change'),AT(187,54,42,12),USE(?ChangePh),HIDE
BUTTON('&Delete'),AT(187,67,42,12),USE(?DeletePh),HIDE
END
ThisWindow CLASS(WindowManager)          !declare ThisWindow object
Init      PROCEDURE(),BYTE,PROC,VIRTUAL
Kill      PROCEDURE(),BYTE,PROC,VIRTUAL
TakeSelected PROCEDURE(),BYTE,PROC,VIRTUAL
      END
Toolbar   ToolbarClass                   !declare Toolbar object to receive
      ! and process toolbar events from Main
CusBrowse CLASS(BrowseClass)             !declare CusBrowse object
Q         &CusQ
      END
PhBrowse  CLASS(BrowseClass)             !declare PhBrowse object
Q         &PhQ
      END
CODE
ThisWindow.Run()

ThisWindow.Init PROCEDURE()
ReturnValue   BYTE,AUTO
CODE
ReturnValue = PARENT.Init()
SELF.FirstField = ?CusList           !CusList gets initial focus
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(Toolbar)                !register Toolbar with WindowManager
Relate:Customer.Open
CusBrowse.Init(?CusList,CusQ.ViewPosition,CusView,CusQ,Relate:Customer,SELF)
PhBrowse.Init(?PhList,PhQ.ViewPosition,PhView,PhQ,Relate:Phones,SELF)
OPEN(CusWindow)
SELF.Opened=True
CusBrowse.Q &= CusQ
CusBrowse.AddSortOrder(,CUS:BYNUMBER)
CusBrowse.AddField(CUS:CUSTNO,CusBrowse.Q.CUS:CUSTNO)
CusBrowse.AddField(CUS:NAME,CusBrowse.Q.CUS:NAME)
PhBrowse.Q &= PhQ
PhBrowse.AddSortOrder(,PH:IDKEY)
PhBrowse.AddRange(PH:ID,Relate:Phones,Relate:Customer)
PhBrowse.AddField(PH:NUMBER,PhBrowse.Q.PH:NUMBER)

```

```
PhBrowse.AddField(PH:ID,PhBrowse.Q.PH:ID)
CusBrowse.InsertControl=?InsertCus
CusBrowse.ChangeControl=?ChangeCus
CusBrowse.DeleteControl=?DeleteCus
CusBrowse.AddToolbarTarget(Toolbar)           !Make CusBrowse a toolbar target
PhBrowse.InsertControl=?InsertPh
PhBrowse.ChangeControl=?ChangePh
PhBrowse.DeleteControl=?DeletePh
PhBrowse.AddToolbarTarget(Toolbar)           !Make PhBrowse a toolbar target
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Customer.Close
RETURN ReturnValue

ThisWindow.TakeSelected PROCEDURE()
ReturnValue    BYTE,AUTO
Looped        BYTE
CODE
LOOP
IF Looped THEN RETURN Level:Notify ELSE Looped=1.
ReturnValue = PARENT.TakeSelected()
CASE FIELD()
  OF ?CusList           !if selected,
    Toolbar.SetTarget(?CusList) ! make ?CusList the active target
  OF ?PhList           !if selected
    IF RECORDS(PhBrowse.Q) > 1 !and contains more than one record,
      Toolbar.SetTarget(?PhList) ! make ?PhList the active target
  END
END
RETURN ReturnValue
END
```

## ToolBarClass Methods

The ToolBarClass contains the methods listed below.

### **ToolBarClass Functional Organization--Expected Use**

As an aid to understanding the ToolBarClass, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the ToolBarClass methods.

#### **Non-Virtual Methods**

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into two categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the ToolBarClass object
AddTarget	register toolbar driven entity
Kill <sub>v</sub>	shut down the ToolBarClass object

##### **Mainstream Use:**

SetTarget	set active target & appropriate toolbar state
TakeEvent <sub>v</sub>	process toolbar event for active target

##### **Occasional Use:**

DisplayButtons <sub>v</sub>	enable appropriate toolbar buttons
-----------------------------	------------------------------------

<sub>v</sub> These methods are also Virtual.

#### **Virtual Methods**

Typically you will not call these methods directly--other base class methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

DisplayButtons	enable appropriate toolbar buttons
TakeEvent	process toolbar event for active target
Kill	shut down the ToolBarClass object

## AddTarget (register toolbar driven entity)

**AddTarget**( *target*, *control* )

---

**AddTarget** Adds a toolbar target to the ToolBarClass object's list of potential toolbar targets.

*target* The label of a ToolbarTarget object.

*control* An integer constant, variable, EQUATE, or expression containing the *target*'s ID number. For targets associated with a control, this is the control number (usually represented by the control's Field Equate Label).

The **AddTarget** method adds a toolbar target (ToolbarTarget object) to the ToolBarClass object's list of potential toolbar targets.

The last added target is the active target until supplanted by a subsequent call to AddTarget or SetTarget.

Example:

```
CODE
ToolBar.Init                !initialize Toolbar object
ToolBar.AddTarget( ToolBarForm, -1 )    !register an Update Form target
ToolBar.AddTarget( REL1::ToolBar, ?RelTree )!register a RelTree target
BRW1.AddToolBarTarget( ToolBar )       !register a BrowseBox target...
                                      !BrowseClass method calls AddTarget
```

See Also:      SetTarget

## **DisplayButtons (enable appropriate toolbar buttons:ToolbarClass)**

### **DisplayButtons, VIRTUAL**

The **DisplayButtons** method enables and disables the appropriate toolbar buttons for the active toolbar target.

The **SetTarget** method sets the active toolbar target.

Implementation: The **DisplayButtons** method calls the **ToolbarTarget.DisplayButtons** method for the active toolbar target.

Example:

#### **CODE**

```
Toolbar.Init                !initialize Toolbar object
  Toolbar.AddTarget( ToolBarForm, -1 ) !register an Update Form target
  Toolbar.DisplayButtons    !and enable appropriate toolbar buttons
                             !for that target
```

See Also: [SetTarget](#)

## Init (initialize the ToolbarClass object)

### Init

The **Init** method initializes the ToolbarClass object.

Implementation: The Init method allocates a new list of potential toolbar targets.

Example:

```
CODE
ToolBar.Init                !initialize Toolbar object
!program code
ACCEPT
!program code
END
ToolBar.Kill                !shut down Toolbar object
```

## Kill (shut down the ToolbarClass object)

### Kill, VIRTUAL

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code.

Implementation: The Kill method disposes of the list of potential toolbar targets.

Example:

```
CODE
ToolBar.Init                !initialize Toolbar object
!program code
ACCEPT
!program code
END
ToolBar.Kill                !shut down Toolbar object
```

**SetTarget (sets the active target)****SetTarget**( [ *ID* ] )**SetTarget** Sets the ToolbarClass object's active toolbar target.*ID* An integer constant, variable, EQUATE, or expression containing the *target's* ID number. For targets associated with a control, this is the control number (usually represented by the control's Field Equate Label). If omitted or zero (0), SetTarget sets the most likely target.The **SetTarget** method sets the ToolbarClass object's active toolbar target (ToolbarTarget object), and adjusts the TOOLBAR state appropriate to that target.

Implementation: The SetTarget method calls the ToolbarTarget.TakeToolbar or ToolbarTarget.TryTakeToolbar method to set the toolbar buttons' TIP attributes and enabled/disabled status appropriate to the active toolbar target.

Example:

```

ACCEPT
CASE EVENT()
OF EVENT:OpenWindow           !on open window
DO RefreshWindow              !load the browse QUEUES
OF EVENT:Accepted             !for Accepted events (which may
CASE FOCUS()                  ! come from the global toolbar)
OF ?ClientList                ! make the list with FOCUS
  Toolbar.SetTarget(?ClientList) ! the active toolbar target
OF ?PhoneList                 ! and enable appropriate toolbar
  Toolbar.SetTarget(?PhoneList) ! buttons and TIP attributes
END
Toolbar.TakeEvent(VCRRequest, WM)!the Toolbar object calls the
END                             ! active target's event handler
END                             ! which in turn scrolls, inserts,
                               ! deletes, helps, etc. The event
                               ! handler often simply POSTs
                               ! another event to the correct
                               ! control, e.g.
                               ! Event:Accepted to ?Insert or
                               ! Event:PageUp to ?ClientList

```

See Also: ToolbarTarget.TakeToolbar, ToolbarTarget.TryTakeToolbar

**TakeEvent (process toolbar event:ToolBarClass)**

**TakeEvent**( [ *vcr* ], *window manager* ), **VIRTUAL**

---

**TakeEvent** Processes toolbar events for the active toolbar target.

*vcr* An integer variable to receive the control number of the accepted VCR navigation button. This lets the TakeEvent method specify an appropriate subsequent action. If omitted, the ToolbarTarget object does no "post processing" navigation.

*Windowmanager*

The label of the ToolbarTarget object's WindowManager object. See *Window Manager* for more information.

The **TakeEvent** method processes toolbar events for the active toolbar target (ToolbarTarget object).

The *vcr* parameter lets the TakeEvent method specify an appropriate subsequent or secondary action. For example, the ToolbarUpdateClass.TakeEvent method (for a FORM), may interpret a *vcr* scroll down as "save and then scroll." The method takes the necessary action to save the item and accomplishes the secondary scroll action by setting the *vcr* parameter.

The SetTarget method sets the active toolbar target.

Implementation: The WindowManager.TakeEvent method calls the TakeEvent method. The TakeEvent method calls the ToolbarTarget.TakeEvent method for the active toolbar target.

Example:

```
MyWindowManager.TakeAccepted PROCEDURE
  CODE
  IF ~SELF.Toolbar &= NULL
    SELF.Toolbar.TakeEvent ( SELF.VCRRequest , SELF )
  END
!procedure code
```

See Also: SetTarget, WindowManager.TakeEvent



# ToolBarListBoxClass

## ToolBarListBoxClass Overview

The ToolBarListBoxClass is a ToolbarTarget that handles events for a BrowseClass LIST. See BrowseClass and Control Templates--BrowseBox for more information.

## ToolBarListboxClass Concepts

ToolBarListBoxClass objects implement the event handling specific to a BrowseClass LIST. The LIST specific events are primarily scrolling events, but also include the event to select a single list item (EVENT:Accepted for a Select button). There may be zero or several ToolbarTarget objects within a procedure; however, *only one is active* at a time.

## ToolBarListBoxClass Relationship to Other Application Builder Classes

The ToolBarListboxClass is derived from the ToolbarTarget class.

The ToolbarClass keeps a list of ToolbarTarget objects (including ToolBarListboxClass objects) so it can forward events and method calls to a particular target.

## ToolBarListBoxClass ABC Template Implementation

The ToolBarListboxClass is completely encapsulated within the BrowseClass and is not referenced in the template-generated code.

## ToolBarListboxClass Source Files

The ToolBarListboxClass source code is installed by default to the Clarion \LIBSRC folder. The ToolBarListboxClass source code and its respective components are contained in:

ABTOOLBA.INC	ToolBarListboxClass declarations
ABTOOLBA.CLW	ToolBarListboxClass method definitions

## ToolBarListBoxClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a ToolbarClass object and related ToolBarListboxClass objects.

This example uses a global toolbar to drive two separate but related LISTS within a single MDI procedure. The primary LIST shows client information and the related LIST shows phone numbers for the selected client. The toolbar drives whichever list has focus. See also *ToolBarUpdateClass--Conceptual Example*.

The program POSTs toolbar events to the active MDI window using the SYSTEM{Prop:Active} property. Then the local ToolbarClass object calls on the active ToolbarTarget object to handle the event.

```

PROGRAM
  INCLUDE('ABBROWSE.INC')           !declare BrowseClass
  INCLUDE('ABTOOLBA.INC')          !declare Toolbar classes
  INCLUDE('ABWINDOW.INC')          !declare WindowManager
  CODE
  !program code

Main PROCEDURE                      !contains global toolbar
AppFrame APPLICATION('Toolbars'),AT(,,275,175),SYSTEM,MAX,RESIZE,IMM
  MENUBAR
    ITEM('Browse Customers'),USE(?BrowseCustomer)
  END
  TOOLBAR,AT(0,0,400,22)           !must use ABTOOLBA.INC EQUATES:
  BUTTON,AT(4,2),USE(?Top,Toolbar:Top),DISABLE,ICON('VCRFIRST.ICO'),FLAT
  BUTTON,AT(16,2),USE(?PageUp,Toolbar:PageUp),DISABLE,ICON('VCRPRIOR.ICO'),FLAT
  BUTTON,AT(28,2),USE(?Up,Toolbar:Up),DISABLE,ICON('VCRUP.ICO'),FLAT
  BUTTON,AT(40,2),USE(?Down,Toolbar:Down),DISABLE,ICON('VCRDOWN.ICO'),FLAT
  BUTTON,AT(52,2),USE(?PageDown,Toolbar:PageDown),DISABLE,ICON('VCRNEXT.ICO'),FLAT
  BUTTON,AT(64,2),USE(?Bottom,Toolbar:Bottom),DISABLE,ICON('VCRLAST.ICO'),FLAT
  END
  END
Frame      CLASS(WindowManager)
Init       PROCEDURE(),BYTE,PROC,VIRTUAL
TakeAccepted PROCEDURE(),BYTE,PROC,VIRTUAL
  END
Toolbar    ToolbarClass           !declare Toolbar object
  CODE
  Frame.Run()

Frame.Init PROCEDURE()
ReturnValue BYTE,AUTO
  CODE
  ReturnValue = PARENT.Init()
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
  SELF.AddItem(Toolbar)          !register Toolbar with WindowManager
  OPEN(AppFrame)
  SELF.Opened=True
  SELF.SetAlerts()
  RETURN ReturnValue

Frame.TakeAccepted PROCEDURE()
ReturnValue BYTE,AUTO
Looped      BYTE

```

```

CODE
CASE ACCEPTED()
OF Toolbar:First TO Toolbar:Last      !for EVENT:Accepted on toolbar
  POST(EVENT:Accepted,ACCEPTED(),SYSTEM{Prop:Active}) !transfer it to active thread
  RETURN Level:Notify
OF ?BrowseCustomer
  START(BrowseCustomer,050000)
END
RETURN PARENT.TakeAccepted()

BrowseCustomer PROCEDURE                !contains local Toolbar and targets
CusView  VIEW(Customer)
  END
CusQ     QUEUE
CUS:CUSTNO  LIKE(CUS:CUSTNO)
CUS:NAME    LIKE(CUS:NAME)
ViewPosition STRING(512)
  END
PhView  VIEW(Phones)
  END
PhQ     QUEUE
PH:NUMBER  LIKE(PH:NUMBER)
PH:ID      LIKE(PH:ID)
ViewPosition STRING(512)
  END
CusWindow WINDOW('Browse Customers'),AT(,,246,131),IMM,SYSTEM,GRAY,MDI
  LIST,AT(8,7,160,100),USE(?CusList),IMM,HVSCROLL,FROM(CusQ),|
  FORMAT(' 51R(2)|M~CUSTNO~C(0)@n-14@80L(2)|M~NAME~@s30@')
  BUTTON('&Insert'),AT(17,111,45,14),USE(?InsertCus),SKIP
  BUTTON('&Change'),AT(66,111,45,14),USE(?ChangeCus),SKIP,DEFAULT
  BUTTON('&Delete'),AT(115,111,45,14),USE(?DeleteCus),SKIP
  LIST,AT(176,7,65,100),USE(?PhList),IMM,FROM(PhQ),|
  FORMAT(' 80L~Phones~L(1)@s20@')
  BUTTON('&Insert'),AT(187,41,42,12),USE(?InsertPh),HIDE
  BUTTON('&Change'),AT(187,54,42,12),USE(?ChangePh),HIDE
  BUTTON('&Delete'),AT(187,67,42,12),USE(?DeletePh),HIDE
  END

ThisWindow CLASS(WindowManager)        !declare ThisWindow object
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL
TakeSelected PROCEDURE(),BYTE,PROC,VIRTUAL
  END

Toolbar  ToolbarClass                  ! declare Toolbar object to receive
                                              ! and process toolbar events from Main

CusBrowse CLASS(BrowseClass)          !declare CusBrowse object
Q        &CusQ

```

```

                END
PhBrowse    CLASS(BrowseClass)           !declare PhBrowse object
Q           &PhQ
                END

CODE
ThisWindow.Run()

ThisWindow.Init  PROCEDURE()
ReturnValue     BYTE,AUTO
CODE
ReturnValue = PARENT.Init()
SELF.FirstField = ?CusList               !CusList gets initial focus
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(Toolbar)                   !register Toolbar with WindowManager
Relate:Customer.Open
CusBrowse.Init(?CusList,CusQ.ViewPosition,CusView,CusQ,Relate:Customer,SELF)
PhBrowse.Init(?PhList,PhQ.ViewPosition,PhView,PhQ,Relate:Phones,SELF)
OPEN(CusWindow)
SELF.Opened=True
CusBrowse.Q &= CusQ
CusBrowse.AddSortOrder(,CUS:BYNUMBER)
CusBrowse.AddField(CUS:CUSTNO,CusBrowse.Q.CUS:CUSTNO)
CusBrowse.AddField(CUS:NAME,CusBrowse.Q.CUS:NAME)
PhBrowse.Q &= PhQ
PhBrowse.AddSortOrder(,PH:IDKEY)
PhBrowse.AddRange(PH:ID,Relate:Phones,Relate:Customer)
PhBrowse.AddField(PH:NUMBER,PhBrowse.Q.PH:NUMBER)
PhBrowse.AddField(PH:ID,PhBrowse.Q.PH:ID)
CusBrowse.InsertControl=?InsertCus
CusBrowse.ChangeControl=?ChangeCus
CusBrowse.DeleteControl=?DeleteCus
CusBrowse.AddToolbarTarget(Toolbar)      !Make CusBrowse a toolbar target
PhBrowse.InsertControl=?InsertPh
PhBrowse.ChangeControl=?ChangePh
PhBrowse.DeleteControl=?DeletePh
PhBrowse.AddToolbarTarget(Toolbar)      !Make PhBrowse a toolbar target
SELF.SetAlerts()
RETURN ReturnValue

ThisWindow.Kill  PROCEDURE()
ReturnValue     BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Customer.Close
RETURN ReturnValue

ThisWindow.TakeSelected  PROCEDURE()

```

```
ReturnValue          BYTE,AUTO
CODE
ReturnValue = PARENT.TakeSelected()
CASE FIELD()
OF ?CusList
    Toolbar.SetTarget(?CusList)           !if selected,
                                           ! make ?CusList the active target
OF ?PhList
    IF RECORDS(PhBrowse.Q) > 1           !if selected
        Toolbar.SetTarget(?PhList)       !and contains more than one record,
                                           ! make ?PhList the active target
    END
END
RETURN ReturnValue
```

## ToolbarListBoxClass Properties

The `ToolbarListBoxClass` inherits all the properties of the `ToolbarTarget` from which it is derived. See *ToolbarTarget Properties* for more information.

In addition to its inherited properties, the `ToolbarListBoxClass` contains the following properties.

### **Browse (BrowseClass object)**

#### **Browse &BrowseClass**

The **Browse** property is a reference to the `ToolbarListBoxClass` object's `BrowseClass` object. The `ToolbarListBoxClass` object uses this property to access the `BrowseClass` object's properties and methods.

Implementation: The `BrowseClass.AddToolbarTarget` method sets the value of the `Browse` property.

The `TryTakeToolbar` method uses the `Browse` property to determine whether the associated `LIST` control is visible.

See Also: `BrowseClass.AddToolbarTarget`

## ToolBarListBoxClass Methods

The ToolBarListBoxClass inherits all the methods of the ToolbarTarget from which it is derived. See *ToolbarTarget Methods* for more information.

In addition to (or instead of) the inherited methods, the ToolBarListBoxClass contains the following methods:

### **DisplayButtons (enable appropriate toolbar buttons:ToolBarListBoxClass)**

#### **DisplayButtons, VIRTUAL**

The **DisplayButtons** method enables and disables the appropriate toolbar buttons for the ToolBarListBoxClass object based on the values of the HelpButton, InsertButton, ChangeButton, DeleteButton, and SelectButton properties.

Implementation: The TakeToolbar method calls the DisplayButtons method. The DisplayButtons method calls the PARENT.DisplayButtons method (ToolBarTarget.DisplayButtons) to handle buttons common to all ToolbarTargets.

Example:

```
CODE
ToolBar.Init                               !initialize Toolbar object
  BRW1.AddToolbarTarget( Toolbar )         !register a BrowseBox target
  ToolBar.SetTarget( ?Browse:1)           !calls DisplayButtons via TakeToolbar
```

```
MyToolBarListBoxClass.DisplayButtons PROCEDURE !a derived class virtual
CODE
  DISABLE(ToolBar:History)                !disable toolbar ditto button
  ENABLE(ToolBar:Locate)                  !enable locator button
  PARENT.DisplayButtons                    !call base class DisplayButtons
  !your custom code here
```

See Also: HelpButton, InsertButton, ChangeButton, DeleteButton, SelectButton, TakeToolbar, ToolbarTarget.DisplayButtons

**TakeEvent (convert toolbar events:ToolbarListboxClass )**

**TakeEvent**( [ *vcr* ], *window manager* ), VIRTUAL

---

<b>TakeEvent</b>	Handles toolbar events for the ToolbarListboxClass object.
<i>vcr</i>	An integer variable to receive the control number of the accepted vcr button. This lets the TakeEvent method specify an appropriate subsequent action. If omitted, the ToolbarListboxClass object does no "post processing" navigation.
<i>windowmanager</i>	The label of the ToolbarListboxClass object's WindowManager object. See <i>Window Manager</i> for more information.

The **TakeEvent** method handles toolbar events for the ToolbarListboxClass object.

The *vcr* parameter lets the TakeEvent method specify an appropriate subsequent or secondary action. For example, the ToolbarListboxClass.TakeEvent method, may interpret a scroll down as "save and then scroll." The method takes the necessary action to save the item and accomplishes the secondary scroll action by setting the *vcr* parameter.

Implementation: The ToolbarClass.TakeEvent method calls the TakeEvent method for the active ToolbarTarget object. The ToolbarClass.SetTarget method sets the active ToolbarTarget object.

Example:

```
ToolbarClass.TakeEvent PROCEDURE(<*LONG VCR>,WindowManager WM)
  CODE
  ASSERT(~SELF.List &= NULL)
  IF RECORDS(SELF.List)
    SELF.List.Item.TakeEvent(VCR,WM)
  END
```

See Also: ToolbarClass.SetTarget, ToolbarClass.TakeEvent

## TakeToolbar (assume control of the toolbar)

### TakeToolbar, VIRTUAL

The **TakeToolbar** method sets the toolbar state appropriate to the ToolbarListBoxClass object.

Implementation: The TakeToolbar method sets appropriate TIP attributes for the toolbar buttons and enables and disables toolbar buttons appropriate for the ToolbarListBoxClass object. The ToolbarClass.SetTarget method and the TryTakeToolbar method call the TakeToolbar method.

Example:

```
MyToolBarClass.SetTarget PROCEDURE(SIGNED Id)
I USHORT,AUTO
Hit USHORT
CODE
  ASSERT(~ (SELF.List &= NULL))
  IF Id                                !set explicitly requested target
    SELF.List.Id = Id
    GET(SELF.List,SELF.List.Id)
    ASSERT (~ERRORCODE())
    SELF.List.Item.TakeToolbar
  ELSE                                  !set any (last) valid target
    LOOP I = 1 TO RECORDS(SELF.List)
      GET(SELF.List,I)
      IF SELF.List.Item.TryTakeToolbar() THEN Hit = I.
    END
    IF Hit THEN GET(SELF.List,Hit).
  END
```

See Also: TryTakeToolbar, ToolbarClass.SetTarget

**TryTakeToolbar (return toolbar control indicator:ToolbarListBoxClass)****TryTakeToolbar, VIRTUAL**

The **TryTakeToolbar** method returns a value indicating whether the ToolbarTarget object successfully assumed control of the toolbar. A return value of one (1 or True) indicates success; a value of zero (0 or False) indicates failure to take control of the toolbar.

Implementation: The ToolbarClass.SetTarget method calls the TryTakeToolbar method. The TryTakeToolbar method calls the TakeToolbar method if the ToolbarListboxClass object's LIST is visible.

Return Data Type: BYTE

Example:

```
ToolbarClass.SetTarget PROCEDURE(SIGNED Id)
I USHORT,AUTO
Hit USHORT
CODE
  ASSERT(~ (SELF.List &= NULL))
  IF Id                                !set explicitly requested target
    SELF.List.Id = Id
    GET(SELF.List,SELF.List.Id)
    ASSERT (~ERRORCODE())
    SELF.List.Item.TakeToolbar
  ELSE                                  !set a valid target
    LOOP I = 1 TO RECORDS(SELF.List)
      GET(SELF.List,I)
      IF SELF.List.Item.TryTakeToolbar() THEN Hit = I.
    END
    IF Hit THEN GET(SELF.List,Hit).
  END
```

See Also: TakeToolbar, ToolbarClass.SetTarget

# ToolBarReltreeClass

## ToolBarReltreeClass Overview

The ToolBarReltreeClass is a ToolbarTarget that handles events for a RelationTree control LIST. See Control Templates--RelationTree for more information.

## ToolBarReltreeClass Concepts

ToolBarReltreeClass objects implement the event handling specific to a RelationTree control LIST. The LIST specific events are primarily scrolling events, but may include other events. There may be zero or several ToolbarTarget objects within a procedure; however, *only one is active* at a time.

## ToolBarReltreeClass Relationship to Other Application Builder Classes

The ToolBarReltreeClass is derived from the ToolbarTarget class.

The ToolbarClass keeps a list of ToolbarTarget objects (including ToolBarReltreeClass objects) so it can forward events and method calls to a particular target.

## ToolBarReltreeClass ABC Template Implementation

The RelationTree control template derives a ToolBarReltreeClass object called REL#::ToolBar, where # is the RelationTree template's instance number. The template generates code to register the REL#::ToolBar object with the Toolbar object for the procedure that contains the RelationTree control template. Finally, the template generates the REL#::ToolBar.TakeEvent method to convert toolbar events into actions specific to the RelationTree LIST control.

## ToolBar ToolBarReltreeClass Source Files

The ToolBarReltreeClass source code is installed by default to the Clarion \LIBSRC folder. The source code and its respective components are in:

ABTOOLBA.INC  
ABTOOLBA.CLW

ToolBarReltreeClass declarations  
ToolBarReltreeClass method definitions

## ToolbarReltreeClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a ToolbarClass object and a related ToolbarReltreeClass (ToolbarTarget) object.

This example uses a global toolbar to drive a template generated RelTree control. The program POSTs toolbar events to the active MDI window using the SYSTEM{Prop:Active} property. Then the ToolbarClass object calls on the active ToolbarReltreeClass object to handle the (scrolling) events.

```

PROGRAM

_ABCDllMode_ EQUATE(0)
_ABCLinkMode_ EQUATE(1)
INCLUDE('ABERROR.INC')
INCLUDE('ABFILE.INC')
INCLUDE('ABWINDOW.INC')
INCLUDE('ABTOOLBA.INC')
INCLUDE('KEYCODES.CLW')

MAP
Main      PROCEDURE
OrderTree PROCEDURE
END
GlobalErrors ErrorClass
Access:Customer CLASS(FileManager)
Init            PROCEDURE
                END

Relate:Customer CLASS(RelationManager)
Init            PROCEDURE
Kill            PROCEDURE,VIRTUAL
                END

Access:Orders CLASS(FileManager)
Init            PROCEDURE
                END

Relate:Orders CLASS(RelationManager)
Init            PROCEDURE
Kill            PROCEDURE,VIRTUAL
                END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD

```

```

Customer      FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,BINDABLE,THREAD
KeyCustNumber KEY(CUS:CustNumber),NOCASE,OPT
KeyCompany    KEY(CUS:Company),DUP,NOCASE
Record        RECORD,PRE()
CustNumber    LONG
Company       STRING(20)
ZipCode       LONG
              END
              END

Orders        FILE,DRIVER('TOPSPEED'),PRE(ORD),CREATE,BINDABLE,THREAD
KeyOrderNumber KEY(ORD:OrderNumber),NOCASE,OPT,PRIMARY
KeyCustNumber  KEY(ORD:CustNumber),DUP,NOCASE,OPT
Record        RECORD,PRE()
CustNumber    LONG
OrderNumber   SHORT
InvoiceAmount DECIMAL(7,2)
              END
              END

CODE
GlobalErrors.Init
Relate:Customer.Init
Relate:Orders.Init
Main          !run Application Frame w/ toolbar
Relate:Customer.Kill
Relate:Orders.Kill
GlobalErrors.Kill

Main PROCEDURE          !Application Frame w/ toolbar
Frame APPLICATION('Application'),AT(,,310,210),SYSTEM,MAX,RESIZE,IMM
  MENUBAR
    ITEM('Orders'),USE(?OrderTree)
  END
  TOOLBAR,AT(0,0,,20) !must use toolbar EQUATES
  BUTTON,AT(4,4),USE(?Toolbar:Top,Toolbar:Top),DISABLE,ICON('VCRFIRST.ICO')
  BUTTON,AT(16,4),USE(?Toolbar:PageUp,Toolbar:PageUp),DISABLE,ICON('VCRPRIOR.ICO')
  BUTTON,AT(28,4),USE(?Toolbar:Up,Toolbar:Up),DISABLE,ICON('VCRUP.ICO')
  BUTTON,AT(40,4),USE(?Toolbar:Down,Toolbar:Down),DISABLE,ICON('VCRDOWN.ICO')
  BUTTON,AT(52,4),USE(?Toolbar:PageDown,Toolbar:PageDown),DISABLE,ICON('VCRNEXT.ICO')
  BUTTON,AT(64,4),USE(?Toolbar:Bottom,Toolbar:Bottom),DISABLE,ICON('VCRLAST.ICO')
  END
  END

ThisWindow CLASS(WindowManager)
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
TakeAccepted PROCEDURE(),BYTE,PROC,VIRTUAL
              END

```

```

CODE
  ThisWindow.Run()

ThisWindow.Init PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
  ReturnValue = PARENT.Init()
  SELF.FirstField = 1
  OPEN(Frame)
  SELF.Opened=True
  RETURN ReturnValue

ThisWindow.TakeAccepted PROCEDURE()
CODE
CASE ACCEPTED()
OF Toolbar:First TO Toolbar:Last      !post toolbar event to active thread
  POST(EVENT:Accepted,ACCEPTED(),SYSTEM{Prop:Active})
  RETURN Level:Notify
OF ?OrderTree
  START(OrderTree,25000)                !start OrderTree procedure/thread
END
RETURN PARENT.TakeAccepted()
OrderTree    PROCEDURE                  !template generated Window procedure
                                                ! with RelTree control template

DisplayString    STRING(255)
Toolbar          ToolbarClass          !declare Toolbar object
REL1::Toolbar    CLASS(ToolbarReltreeClass) !derive REL1::Toolbar object (target)
TakeEvent        PROCEDURE(<*LONG VCR>,WindowManager WM),VIRTUAL
                END
!template generated RelTree QUEUES and vaiables

window WINDOW('Browse Orders'),AT(,,115,110),SYSTEM,GRAY,DOUBLE,MDI
  LIST,AT(5,4,106,100),USE(?RelTree),FORMAT('800LT@s200@'),FROM(Queue:RelTree)
  END

ThisWindow    CLASS(WindowManager)      !derive ThisWindow object
Init          PROCEDURE(),BYTE,PROC,VIRTUAL
Kill          PROCEDURE(),BYTE,PROC,VIRTUAL
TakeFieldEvent PROCEDURE(),BYTE,PROC,VIRTUAL
                END

```

```
CODE
GlobalResponse = ThisWindow.Run()
!template generated RelTree ROUTINES

ThisWindow.Init PROCEDURE()
ReturnValue          BYTE,AUTO
CODE
ReturnValue = PARENT.Init()
SELF.FirstField = ?RelTree
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(Toolbar)          !register Toolbar with ThisWindow
Relate:Customer.Open
DO REL1::ContractAll
OPEN(window)
SELF.Opened=True
Toolbar.AddTarget(REL1::Toolbar,?RelTree) !make REL1::Toolbar a toolbar target
Toolbar.SetTarget(?RelTree)          !make REL1::Toolbar the active target
?RelTree{Prop:Selected} = 1
SELF.SetAlerts()
RETURN ReturnValue

REL1::Toolbar.TakeEvent PROCEDURE(<*LONG VCR>,WindowManager WM)
CODE          !convert toolbar events to
CASE ACCEPTED()          ! Reltree-specific actions
OF Toolbar:Bottom TO Toolbar:Up
  SELF.Control{PROPLIST:MouseDownRow} = CHOICE(SELF.Control)
  EXECUTE(ACCEPTED()-Toolbar:Bottom+1)
  DO REL1::NextParent          !on Toolbar:Bottom
  DO REL1::PreviousParent      !on Toolbar:Top
  DO REL1::NextLevel           !on Toolbar:PageDown
  DO REL1::PreviousLevel       !on Toolbar:PageUp
  DO REL1::NextRecord          !on Toolbar:Down
  DO REL1::PreviousRecord      !on Toolbar:Up
END
END
```

## ToolbarReltreeClass Properties

The `ToolbarReltreeClass` inherits all the properties of the `ToolbarTarget` from which it is derived. See *ToolbarTarget Properties* for more information.

## ToolBarReltreeClass Methods

The ToolBarReltreeClass inherits all the methods of the ToolbarTarget from which it is derived. See *ToolbarTarget Methods* for more information.

In addition to (or instead of) the inherited methods, the ToolBarReltreeClass contains the following methods:

### **DisplayButtons (enable appropriate toolbar buttons:ToolBarReltreeClass)**

#### **DisplayButtons, VIRTUAL**

The **DisplayButtons** method enables and disables the appropriate toolbar buttons for the ToolBarReltreeClass object based on the values of the HelpButton, InsertButton, ChangeButton, DeleteButton, and SelectButton properties.

Implementation: The TakeToolbar method calls the DisplayButtons method.

Example:

```

CODE
ToolBar.Init                               !initialize Toolbar object
  ToolBar.AddTarget( ToolBarForm, -1 )      !register an Update Form target
  ToolBar.AddTarget( REL1::ToolBar, ?RelTree ) !register a RelTree target
  ToolBar.SetTarget( ?RelTree )            !calls DisplayButtons via TakeToolbar
!program code

MyToolBarReltreeClass.DisplayButtons PROCEDURE !a derived class virtual
CODE
DISABLE(ToolBar:History)                   !disable toolbar ditto button
ENABLE(ToolBar:Locate)                     !enable locator button
PARENT.DisplayButtons                      !call base class DisplayButtons
!your custom code here

```

See Also: HelpButton, InsertButton, ChangeButton, DeleteButton, SelectButton, TakeToolbar

## TakeToolbar (assume control of the toolbar:ToolbarReltreeClass)

### TakeToolbar, VIRTUAL

The **TakeToolbar** method sets the toolbar state appropriate to the ToolbarReltreeClass object.

Implementation: The TakeToolbar method sets appropriate TIP attributes for the toolbar buttons and enables and disables toolbar buttons appropriate for the ToolbarReltreeClass object. The ToolbarClass.SetTarget method calls the TakeToolbar method.

Example:

```
CODE
Toolbar.Init                               !initialize Toolbar object
  ToolBar.AddTarget( ToolBarForm, -1 )      !register an Update Form target
  Toolbar.AddTarget( REL1::Toolbar, ?RelTree ) !register a RelTree target
  ToolBar.SetTarget( ?RelTree )            !calls TakeToolbar
!program code

MyToolbarReltreeClass.TakeToolbar PROCEDURE !a derived class virtual
CODE
!your custom code here
SELF.DisplayButtons                        !enable appropriate buttons
```

See Also:      ToolbarClass.SetTarget

# ToolBarTargetClass

## ToolBarTarget Overview

ToolBarClass and ToolBarTarget objects work together to reliably "convert" an event associated with a toolbar button into an appropriate event associated with a specific control or window. This lets you use a single toolbar to drive a variety of targets, such as update forms, browse lists, relation tree lists, etc. A single toolbar can even drive multiple targets (two or more BrowseBoxes) in a single procedure.

Although the ToolBarTarget is useful by itself, other more useful classes are derived from it (ToolBarListBoxClass, the ToolBarRelTreeClass, and the ToolBarUpdateClass), and other structures, such as the ToolBarClass, use it to reference any of these derived classes. The classes derived from ToolBarTarget let you set the state of the toolbar appropriate to the toolbar driven entity (set tooltips, enable/disable buttons, etc.), then process toolbar events for the entity by converting the generic toolbar events into appropriate entity-specific events.

## ToolBarTarget Concepts

Within an MDI application, the ToolBarClass and ToolBarTarget work together to reliably interpret and pass an event (EVENT:Accepted) associated with a toolbar button into an event associated with a specific control or window. For example, the end user CLICKS on a toolbar button (say the "Insert" button) on the MDI application frame. The frame procedure forwards the event to the active thread (POST(EVENT:Accepted,ACCEPTED( ),SYSTEM{Prop:Active})). The active thread (procedure) manages a window that displays two LIST controls, and one of the LISTS has focus. This procedure has a ToolBarClass object plus a ToolBarTarget object for each LIST control. The ToolBarClass object takes the event (ToolBarClass.TakeEvent)<sub>1</sub> and forwards the event to the *active* ToolBarTarget object (the target that represents the LIST with focus). The ToolBarTarget object takes the event (ToolBarListBoxClass.TakeEvent) and handles it by posting an appropriate event to a specific control or to the window, for example:

```
POST(EVENT:ACCEPTED,SELF.InsertButton) !insert a record
POST(EVENT:PageDown,SELF.Control)      !scroll a LIST
POST(EVENT:Completed)                   !complete an update form
POST(EVENT:CloseWindow)                 !select a record
etc.
```

1If the procedure has a WindowManager object, the WindowManager object takes the event (WindowManager.TakeEvent) and forwards it to the ToolBarClass object (WindowManager.TakeAccepted).

## ToolbarTarget Relationship to Other Application Builder Classes

At present, the ABC Library has three classes derived from the ToolbarTarget class:

ToolbarListboxClass	BrowseClass toolbar target
ToolbarReltreeClass	Reltree control toolbar target
ToolbarUpdateClass	Form procedure toolbar target

These ToolbarTarget objects convert generic toolbar events into appropriate entity-specific events. There may be zero or more ToolbarTarget objects within a procedure; however, *only one is active* at a time.

The ToolbarClass keeps a list of ToolbarTarget objects so it can forward events and method calls to a particular target.

## ToolbarTarget ABC Template Implementation

Each template that requests global toolbar control instantiates a ToolbarTarget object. The FormVCRControls template's ToolbarTarget object is called ToolBarForm; the RelationTree template's ToolbarTarget object is called REL#:Toolbar, where # is the RelationTree template's instance number; and the BrowseBox's ToolbarTarget object is completely encapsulated within the BrowseClass object and is not referenced in the template generated code. You may see code such as the following in your template generated procedures.

```

Toolbar      ToolbarClass      !declare Toolbar object
CODE
!
ThisWindow.Init PROCEDURE
  SELF.AddItem(Toolbar)          !register Toolbar with WindowManager
  BRW1.AddToolbarTarget(Toolbar) !register BrowseClass as target
  Toolbar.AddTarget(REL1::Toolbar,?RelTree) !register RelTree as target
  SELF.AddItem(ToolbarForm)      !register update form as target

```

## ToolbarTarget Source Files

The ToolbarTarget source code is installed by default to the Clarion \LIBSRC folder. The ToolbarTarget source code and its respective components are contained in:

ABTOOLBA.INC	ToolbarTarget declarations
ABTOOLBA.CLW	ToolbarTarget method definitions

## ToolBarTarget Properties

### ChangeButton (change control number)

#### **ChangeButton SIGNED**

The **ChangeButton** property contains the control number (usually represented by the control's Field Equate Label) of the window control that invokes the change record action for this ToolBarTarget object.

A value of zero (0) disables the toolbar change button.

Implementation: The ToolBarTarget object uses this property to enable or disable the toolbar change button, and as the target control when POSTing certain events. See POST in the *Language Reference* for more information. The ToolBarTarget object POSTs an EVENT:Accepted to the ChangeButton control when the end user CLICKS the toolbar change button.

### Control (window control)

#### **Control SIGNED**

The **Control** property contains the control number (usually represented by the control's Field Equate Label) of the window control associated with this ToolBarTarget object. For ToolBarTarget objects that do not have an associated control (update forms), the Control property may contain any identifying number.

The ToolBarTarget object uses this property as the target control when POSTing certain events. See POST in the *Language Reference*.

The ToolBarClass.AddTarget method sets the value of this property.

Implementation: By convention, update forms have a Control value of negative one (-1).

See Also: ToolBarClass.AddTarget

### **DeleteButton (delete control number)**

#### **DeleteButton    SIGNED**

The **DeleteButton** property contains the control number (usually represented by the control's Field Equate Label) of the window control that invokes the delete record action for this ToolbarTarget object.

A value of zero (0) disables the toolbar delete button.

Implementation:    The ToolbarTarget object uses this property to enable or disable the toolbar delete button, and as the target control when POSTing certain events. See POST in the *Language Reference* for more information. The ToolbarTarget object POSTs an EVENT:Accepted to the DeleteButton control when the end user CLICKS the toolbar delete button.

### **HelpButton (help control number)**

#### **HelpButton    SIGNED**

The **HelpButton** property contains the control number (usually represented by the control's Field Equate Label) of the window control that invokes Windows help for this ToolbarTarget object.

A value of zero (0) disables the toolbar help button.

Implementation:    The ToolbarTarget object uses this property to enable or disable the toolbar help button. The ToolbarTarget object "presses" the help (F1) key when the end user CLICKS the toolbar help button.

## **InsertButton (insert control number)**

**InsertButton**    **SIGNED**

The **InsertButton** property contains the control number (usually represented by the control's Field Equate Label) of the window control that invokes the insert record action for this ToolBarTarget object.

A value of zero (0) disables the toolbar insert button.

Implementation:    The ToolBarTarget object uses this property to enable or disable the toolbar insert button, and as the target control when POSTing certain events. See POST in the *Language Reference* for more information. The ToolBarTarget object POSTs an EVENT:Accepted to the InsertButton control when the end user CLICKS the toolbar insert button.

## **LocateButton(query control number)**

**LocateButton**    **SIGNED**

The **LocateButton** property contains the control number (usually represented by the control's field equate label) of the window control that invokes the query action for this ToolBarTarget object.

A value of zero (0) disables the toolbar Locate button.

Implementation:    The ToolBarTarget object uses this property to enable or disable the toolbar Locate button, and as the target control when POSTing certain events. See POST in the *Language Reference* for more information. The ToolBarTarget object POSTs an EVENT:Accepted to the LocateButton control when the user Presses the toolbar locate button.

The LocateButton control is used when the QBE extension template has been added to a procedure.

See Also:            ToolBarTarget.DisplayButtons

**SelectButton (select control number)****SelectButton    SIGNED**

The **SelectButton** property contains the control number (usually represented by the control's Field Equate Label) of the window control that invokes the select record action for this ToolbarTarget object.

A value of zero (0) disables the toolbar select button.

Implementation:    The ToolbarTarget object uses this property to enable or disable the toolbar select button, and as the target control when POSTing certain events. See POST in the *Language Reference* for more information. The ToolbarTarget object POSTs an EVENT:Accepted to the SelectButton control when the end user CLICKS the toolbar select button.

## ToolBarTarget Methods

### **ToolBarTarget Functional Organization--Expected Use**

As an aid to understanding the ToolBarTarget class, it is useful to recognize that all its methods are virtual. Typically you will not call these methods directly from your program--the ToolbarClass methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

### **Virtual Methods**

---

DisplayButtons	enable appropriate toolbar buttons
TryTakeToolbar	return toolbar control indicator
TakeToolbar	assume control of the toolbar
TakeEvent	convert toolbar events

## **DisplayButtons (enable appropriate toolbar buttons:ToolbarTarget)**

### **DisplayButtons, VIRTUAL**

The **DisplayButtons** method enables and disables the appropriate toolbar buttons for the ToolbarTarget object based on the values of the HelpButton, InsertButton, ChangeButton, DeleteButton, and SelectButton properties.

Implementation: The ToolbarListboxClass.TakeToolbar, ToolbarRelTreeClass.TakeToolbar, and ToolbarUpdateClass.TakeToolbar methods call the DisplayButtons method. The DisplayButtons method appropriately enables and disables toolbar buttons common to all ToolbarTarget objects.

Example:

**MyToolbarListboxClass.DisplayButtons PROCEDURE**

**CODE**

```
PARENT.DisplayButtons           !Call base class DisplayButtons  
!your custom code here
```

See Also: HelpButton, InsertButton, ChangeButton, DeleteButton, SelectButton, TakeToolbar, ToolbarRelTreeClass.TakeToolbar, ToolbarUpdateClass.TakeToolbar

## TakeEvent (convert toolbar events:ToolBarTarget)

**TakeEvent**( [ *vcr* ], *window manager* ), VIRTUAL

---

**TakeEvent** Process toolbar events for this toolbar target.

*vcr* An integer variable to receive the control number of the accepted VCR navigation button. If omitted, the ToolBarTarget object does no "post processing" navigation.

*Windowmanager*

The label of the ToolBarTarget object's WindowManager object. See *Window Manager* for more information.

The **TakeEvent** method handles toolbar events for this toolbar target.

The *vcr* parameter lets the TakeEvent method specify an appropriate subsequent or secondary action. For example, the ToolBarUpdateClass.TakeEvent method (for a FORM), may interpret a *vcr* scroll down as "save and then scroll." The method takes the necessary action to save the item and accomplishes the secondary scroll action by setting the *vcr* parameter.

Implementation: The ToolBarClass.TakeEvent method calls the TakeEvent method for the active ToolBarTarget object. The ToolBarClass.SetTarget method sets the active ToolBarTarget object. The TakeEvent method POSTs an EVENT:Accepted to the appropriate local control (insert, change, delete, help) common to all ToolBarTarget objects.

Example:

```
REL1::Toolbar.TakeEvent PROCEDURE(<*LONG VCR>,WindowManager WM)
CODE
CASE ACCEPTED()
OF Toolbar:Bottom TO Toolbar:Up
  SELF.Control{PROPLIST:MouseDownRow} = CHOICE(SELF.Control)
  EXECUTE(ACCEPTED()-Toolbar:Bottom+1)
  DO REL1::NextParent
  DO REL1::PreviousParent
  DO REL1::NextLevel
  DO REL1::PreviousLevel
  DO REL1::NextRecord
  DO REL1::PreviousRecord
END
OF Toolbar:Insert TO Toolbar>Delete
  SELF.Control{PROPLIST:MouseDownRow} = CHOICE(SELF.Control)
  EXECUTE(ACCEPTED()-Toolbar:Insert+1)
  DO REL1::AddEntry
  DO REL1::EditEntry
  DO REL1::RemoveEntry
END
ELSE
  PARENT.TakeEvent(VCR,ThisWindow)
END
```

See Also:      ToolbarClass.SetTarget, ToolbarClass.TakeEvent

## **TakeToolBar (assume control of the toolbar:ToolBarTarget)**

### **TakeToolBar, VIRTUAL**

The **TakeToolBar** method is a placeholder method to set the toolbar state appropriate to the ToolBarTarget object. This includes setting MSG and TIP attributes, enabling and disabling appropriate buttons, etc.

The TakeToolBar method is a placeholder method for derived classes.

See Also:      ToolBarListBoxClass.TakeToolBar, ToolBarRelTreeClass.TakeToolBar,  
                  ToolBarUpdateClass.TakeToolBar

## **TryTakeToolBar (return toolbar control indicator:ToolBarTarget)**

### **TryTakeToolBar, VIRTUAL**

The **TryTakeToolBar** method is a virtual placeholder method to return a value indicating whether the ToolBarTarget object successfully assumed control of the toolbar. A return value of one (1 or True) indicates success; a value of zero (0 or False) indicates failure to take control of the toolbar.

The TryTakeToolBar method is a placeholder method for derived classes.

Return Data Type:    BYTE

See Also:      ToolBarListBoxClass.TryTakeToolBar, ToolBarUpdateClass.TryTakeToolBar



# ToolBarUpdateClass

## ToolBarUpdateClass Overview

The ToolBarUpdateClass is a ToolBarTarget that handles events for a template generated Form Procedure that is called from a template generated Browse Procedure. See *Procedure Templates--Browse* and *Form* for more information.

## ToolBarUpdateClass Concepts

ToolBarUpdateClass objects implement the event handling specific to a template generated Form Procedure. The Form specific events are primarily the event to complete the Form and save the record (EVENT:Accepted for an OK button). There may be zero or several ToolBarTarget objects within a procedure; however, *only one is active* at a time.

## ToolBarUpdateClass Relationship to Other Application Builder Classes

The ToolBarUpdateClass is derived from the ToolBarTarget class.

The ToolBarClass keeps a list of ToolBarTarget objects (including ToolBarUpdateClass objects) so it can forward events and method calls to a particular target.

## ToolBarUpdateClass ABC Template Implementation

The FormVCRControls extension template generates code to declare a ToolBarUpdateClass object called ToolBarForm, and to register the ToolBarForm object with the procedure's WindowManager.

Once the ToolBarForm is registered with the WindowManager, the WindowManager handles the interaction between the ToolBarClass object and the ToolBarUpdateClass object with no other references in the template generated code.

You can use the FormVCRControl template's prompts to derive from the ToolBarUpdateClass. The templates provide the derived class so you can modify the ToolBarForm's behavior on an instance-by-instance basis.

## ToolBarUpdateClass Source Files

The ToolBarUpdateClass source code is installed by default to the Clarion \LIBSRC folder. The ToolBarUpdateClass source code and its respective components are:

ABTOOLBA.INC	ToolBarUpdateClass declarations
ABTOOLBA.CLW	ToolBarUpdateClass method definitions

## ToolBarUpdateClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a ToolbarClass object and related ToolbarTarget (ToolbarUpdateClass and ToolbarListboxClass) objects.

This example uses a global toolbar to drive a BrowseClass LIST, its child Form procedure, and the Form procedure's secondary BrowseClass LIST.

The program POSTs toolbar events to the active MDI window using the SYSTEM{Prop:Active} property. Then the local ToolbarClass object calls on the active ToolbarTarget object to handle the event.

```

PROGRAM
_AB CDllMode_ EQUATE(0)
_AB CLinkMode_ EQUATE(1)

INCLUDE('ABERROR.INC')
INCLUDE('ABFILE.INC')
INCLUDE('ABWINDOW.INC')
INCLUDE('ABBROWSE.INC')
INCLUDE('ABTOOLBA.INC')
INCLUDE('KEYCODES.CLW')

MAP
Main          PROCEDURE          !contains global toolbar
BrowseCustomers PROCEDURE          !template generated Browse
UpdateCustomer PROCEDURE          !template generated Form
END

GlobalErrors ErrorClass
Access:Customer CLASS(FileManager)
Init          PROCEDURE
END

Relate:Customer CLASS(RelationManager)
Init          PROCEDURE
Kill          PROCEDURE,VIRTUAL
END

Access:Orders CLASS(FileManager)
Init          PROCEDURE
END

Relate:Orders CLASS(RelationManager)
Init          PROCEDURE
Kill          PROCEDURE,VIRTUAL
END

GlobalRequest BYTE(0),THREAD
GlobalResponse BYTE(0),THREAD
VCRRequest    LONG(0),THREAD

Customer      FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,BINDABLE,THREAD
KeyCustNumber KEY(CUS:CustNumber),NOCASE,OPT

```

```

KeyCompany      KEY(CUS:Company),DUP,NOCASE
Record          RECORD,PRE()
CustNumber      LONG
Company         STRING(20)
ZipCode         LONG
                END
                END
Orders          FILE,DRIVER('TOPSPEED'),PRE(ORD),CREATE,BINDABLE,THREAD
KeyOrderNumber KEY(ORD:OrderNumber),NOCASE,OPT,PRIMARY
KeyCustNumber   KEY(ORD:CustNumber),DUP,NOCASE,OPT
Record          RECORD,PRE()
CustNumber      LONG
OrderNumber     SHORT
InvoiceAmount   DECIMAL(7,2)
                END
                END

CODE
GlobalErrors.Init
Relate:Customer.Init
Relate:Orders.Init
Main            !run Application Frame w/ toolbar
Relate:Customer.Kill
Relate:Orders.Kill
GlobalErrors.Kill

Main PROCEDURE          !Application Frame w/ toolbar
Frame APPLICATION('Application'),AT(,,310,210),SYSTEM,MAX,RESIZE,IMM
  MENUBAR
  ITEM('Browse Customers'),USE(?BrowseCustomers)
  END
  TOOLBAR,AT(0,0,,20)      !must use toolbar EQUATES
  BUTTON,AT(4,4),USE(?Toolbar:Top,Toolbar:Top),DISABLE,ICON('VCRFIRST.ICO')
  BUTTON,AT(16,4),USE(?Toolbar:PageUp,Toolbar:PageUp),DISABLE,ICON('VCRPRIOR.ICO')
  BUTTON,AT(28,4),USE(?Toolbar:Up,Toolbar:Up),DISABLE,ICON('VCRUP.ICO')
  BUTTON,AT(40,4),USE(?Toolbar:Down,Toolbar:Down),DISABLE,ICON('VCRDOWN.ICO')
  BUTTON,AT(52,4),USE(?Toolbar:PageDown,Toolbar:PageDown),DISABLE,ICON('VCRNEXT.ICO')
  BUTTON,AT(64,4),USE(?Toolbar:Bottom,Toolbar:Bottom),DISABLE,ICON('VCRLAST.ICO')
  BUTTON,AT(96,4),USE(?Toolbar:Insert,Toolbar:Insert),DISABLE,ICON('INSERT.ICO')
  BUTTON,AT(108,4),USE(?Toolbar:Change,Toolbar:Change),DISABLE,ICON('EDIT.ICO')
  BUTTON,AT(121,4),USE(?Toolbar>Delete,Toolbar>Delete),DISABLE,ICON('DELETE.ICO')
  END
  END

FrameWindow CLASS(WindowManager)
Init          PROCEDURE(),BYTE,PROC,VIRTUAL
TakeAccepted PROCEDURE(),BYTE,PROC,VIRTUAL
                END

CODE
FrameWindow.Run()
FrameWindow.Init PROCEDURE()
ReturnValue   BYTE,AUTO
CODE
ReturnValue = PARENT.Init()

```

```

SELF.FirstField = 1
OPEN(Frame)
SELF.Opened=True
RETURN ReturnValue

FrameWindow.TakeAccepted PROCEDURE()
CODE
CASE ACCEPTED()
OF Toolbar:First TO Toolbar:Last           !post toolbar event to active thread
  POST(EVENT:Accepted,ACCEPTED(),SYSTEM{Prop:Active})
  RETURN Level:Notify
OF ?BrowseCustomers
  START(BrowseCustomers,25000)             !start BrowseCustomers procedure/thread
END
RETURN PARENT.TakeAccepted()

BrowseCustomers PROCEDURE                   !template generated Browse
CustView    VIEW(Customer)
END

CustQ       QUEUE
CUS:CustNumber  LIKE(CUS:CustNumber)
CUS:Company    LIKE(CUS:Company)
CUS:ZipCode    LIKE(CUS:ZipCode)
ViewPosition  STRING(1024)
END

QuickWindow WINDOW('Browse Customers'),AT(,,211,155),IMM,SYSTEM,GRAY,DOUBLE,MDI
LIST,AT(8,6,198,142),USE(?CustList),IMM,HVSCROLL,FROM(CustQ),|
FORMAT('28R(2)|M-ID~C(0)n4@80L(2)|M-Company~36L(2)|M-Zip~@P#####P@')
BUTTON('&Insert'),AT(49,62),USE(?Insert),HIDE
BUTTON('&Change'),AT(98,62),USE(?Change),HIDE,DEFAULT
BUTTON('&Delete'),AT(147,62),USE(?Delete),HIDE
END

BrowseWindow CLASS(WindowManager)         !derive BrowseWindow object
Init        PROCEDURE(),BYTE,PROC,VIRTUAL
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL
Run         PROCEDURE(USHORT Number,BYTE Request),BYTE,PROC,VIRTUAL
END

Toolbar     ToolbarClass                  !declare Toolbar object
BRW1        CLASS(BrowseClass)           !derive BRW1 object from BrowseClass
Q           &CustQ
END

CODE
GlobalResponse = BrowseWindow.Run()

BrowseWindow.Init PROCEDURE()
ReturnValue   BYTE,AUTO
CODE
ReturnValue = PARENT.Init()

```

```

SELF.FirstField = ?CustList
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
SELF.AddItem(Toolbar)           !register Toolbar with BrowseWindow
Relate:Customer.Open
BRW1.Init(?CustList,CustQ.ViewPosition,CustView,CustQ,Relate:Customer,SELF)
OPEN(QuickWindow)
SELF.Opened=True
BRW1.Q &= CustQ
BRW1.AddSortOrder(,CUS:KeyCompany) !set scroll order for Browse AND child Form
BRW1.AddField(CUS:CustNumber,BRW1.Q.CUS:CustNumber)
BRW1.AddField(CUS:Company,BRW1.Q.CUS:Company)
BRW1.AddField(CUS:ZipCode,BRW1.Q.CUS:ZipCode)
BRW1.AskProcedure = 1
BRW1.InsertControl=?Insert
BRW1.ChangeControl=?Change
BRW1.DeleteControl=?Delete
BRW1.AddToolbarTarget(Toolbar)    !BRW1 instantiates a ToolbarListboxClass
SELF.SetAlerts()                 ! object, and makes it a target
RETURN ReturnValue

BrowseWindow.Kill PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()
Relate:Customer.Close
RETURN ReturnValue

BrowseWindow.Run PROCEDURE(USHORT Number,BYTE Request)
CODE
GlobalRequest = Request
UpdateCustomer           !Browse Procedure calls Form Procedure
RETURN GlobalResponse

UpdateCustomer PROCEDURE           !template generated Form Procedure

OrderView    VIEW(Orders)
            END

OrderQ        QUEUE
ORD:OrderNumber    LIKE(ORD:OrderNumber)
ORD:InvoiceAmount  LIKE(ORD:InvoiceAmount)
ViewPosition      STRING(1024)
            END

QuickWindow WINDOW('Update Customer'),AT(,,172,132),IMM,GRAY,DOUBLE,MDI
SHEET,AT(4,4,164,106),USE(?CurrentTab)
TAB('Customer'),USE(?CustomerTab)
PROMPT('&Cust Number:'),AT(8,23),USE(?CustNumber:Prompt)
STRING(@n4),AT(64,23),USE(CUS:CustNumber),RIGHT(1)
PROMPT('&Company:'),AT(8,36),USE(?Company:Prompt)

```

```

ENTRY(@s20),AT(64,36),USE(CUS:Company)
PROMPT('&Zip Code:'),AT(8,52),USE(?Zip:Prompt)
ENTRY(@P#####P),AT(64,52),USE(CUS:ZipCode),RIGHT(1)
END
TAB('Orders'),USE(?OrderTab)
LIST,AT(8,22,156,81),USE(?OrdList),IMM,HVSCROLL,FROM(OrderQ),|
FORMAT('52R(2)|M~Order ID~C(0)@n-7@60D(12)|M~Amount~C(0)@n-10.2@')
END
END
BUTTON('OK'),AT(97,114),USE(?OK),DEFAULT
BUTTON('Cancel'),AT(133,114),USE(?Cancel)
END

```

```

FormWindow CLASS(WindowManager)           !derive FormWindow from WindowManager
Init      PROCEDURE(),BYTE,PROC,VIRTUAL
Kill      PROCEDURE(),BYTE,PROC,VIRTUAL
TakeSelected PROCEDURE(),BYTE,PROC,VIRTUAL
END

```

```

Toolbar      ToolbarClass           !declare Toolbar object
ToolbarForm  ToolbarUpdateClass     !declare ToolbarForm object
OrderBrowse  CLASS(BrowseClass)     !derive OrderBrowse from BrowseClass
Q            &OrderQ
END

```

```

CODE
GlobalResponse = FormWindow.Run()

```

```

FormWindow.Init PROCEDURE()
ReturnValue  BYTE,AUTO
CODE
SELF.Request = GlobalRequest
ReturnValue = PARENT.Init()
SELF.FirstField = ?CustNumber:Prompt
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
SELF.AddItem(?Cancel,RequestCancelled)
Relate:Customer.Open
SELF.Primary &= Relate:Customer
SELF.OkControl = ?OK
IF SELF.PrimeUpdate() THEN RETURN Level:Notify.
OrderBrowse.Init(?OrdList,OrderQ.ViewPosition,OrderView,OrderQ,Relate:Orders,SELF)
OPEN(QuickWindow)
SELF.Opened=True
OrderBrowse.Q &= OrderQ
OrderBrowse.AddSortOrder(,ORD:KeyCustNumber)
OrderBrowse.AddRange(ORD:CustNumber,Relate:Orders,Relate:Customer)
OrderBrowse.AddField(ORD:OrderNumber,OrderBrowse.Q.ORD:OrderNumber)
OrderBrowse.AddField(ORD:InvoiceAmount,OrderBrowse.Q.ORD:InvoiceAmount)
SELF.AddItem(Toolbar)           !Register Toolbar with FormWindow
SELF.AddItem(ToolbarForm)       !Register ToolbarForm with FormWindow

```

```
                                ! (and with FormWindow's Toolbar)
OrderBrowse.AddToolbarTarget(Toolbar)!Instantiate a ToolbarListBoxClass object,
SELF.SetAlerts()                    ! and register with FormWindow's Toolbar
RETURN ReturnValue

FormWindow.Kill PROCEDURE()
ReturnValue  BYTE,AUTO
CODE
  ReturnValue = PARENT.Kill()
  Relate:Customer.Close
  RETURN ReturnValue

FormWindow.TakeSelected PROCEDURE
CODE
  IF FIELD(){PROP:Type}=Create:List
    Toolbar.SetTarget(FIELD())      !make selected list the active Target
  END                                !(FormWindow also auto selects the Target)
  RETURN PARENT.TakeSelected()

Access:Customer.Init PROCEDURE
CODE
  PARENT.Init(Customer,GlobalErrors)
  SELF.FileNameValue = 'Customer'
  SELF.Buffer &= CUS:Record
  SELF.Create = 1
  SELF.AddKey(CUS:KeyCustNumber, 'CUS:KeyCustNumber',1)
  SELF.AddKey(CUS:KeyCompany, 'CUS:KeyCompany',0)
  SELF.AddKey(CUS:KeyZipCode, 'CUS:KeyZipCode',0)

Access:Orders.Init PROCEDURE
CODE
  PARENT.Init(Orders,GlobalErrors)
  SELF.FileNameValue = 'Orders'
  SELF.Buffer &= ORD:Record
  SELF.Create = 1
  SELF.AddKey(ORD:KeyOrderNumber, 'ORD:KeyOrderNumber',1)
  SELF.AddKey(ORD:KeyCustNumber, 'ORD:KeyCustNumber',0)

Relate:Customer.Init PROCEDURE
CODE
  Access:Customer.Init
  PARENT.Init(Access:Customer,1)
  SELF.AddRelation(Relate:Orders,RI:CASCADE,RI:RESTRICT,ORD:KeyCustNumber)
  SELF.AddRelationLink(CUS:CustNumber,ORD:CustNumber)

Relate:Customer.Kill PROCEDURE
CODE
  Access:Customer.Kill
  PARENT.Kill

Relate:Orders.Init PROCEDURE
CODE
  Access:Orders.Init
```

```
PARENT.Init(Access:Orders,1)
SELF.AddRelation(Relate:Customer)
```

```
Relate:Orders.Kill PROCEDURE
CODE
Access:Orders.Kill
PARENT.Kill
```

## ToolBarUpdateClass Properties

The ToolBarUpdateClass inherits all the properties of the ToolbarTarget from which it is derived. See *ToolBarTarget Properties* for more information.

In addition to the inherited properties, the ToolBarUpdateClass contains the following properties.

### **Request (requested database operation)**

**Request**      **BYTE**

The **Request** property indicates for what purpose the ToolBarUpdateClass object's entity is used. The ToolBarUpdateClass uses this value to set appropriate toolbar button TIP attributes and enable and disable the appropriate toolbar buttons.

Implementation:      The TakeToolbar and DisplayButtons methods set the toolbar state based on the value of the Request property. EQUATEs for the Request values are declared in TPLEQU.CLW as follows:

```
InsertRecord    EQUATE (1)    !Add a record
ChangeRecord   EQUATE (2)    !Change the current record
DeleteRecord    EQUATE (3)    !Delete the current record
SelectRecord    EQUATE (4)    !Select the current record
```

See Also:            DisplayButtons, TakeToolbar

**History (enable toolbar history button)****History      BYTE**

The **History** property indicates whether or not to enable the toolbar history (ditto) button for this `ToolbarUpdateClass` object. The `ToolbarUpdateClass` uses this value to set the appropriate toolbar button TIP attribute and enable or disable the appropriate toolbar button.

By convention the history button restores the previous value for a field or record. See *Control Templates--SaveButton* for more information.

Implementation:      The `TakeToolbar` and `DisplayButtons` methods set the toolbar state based on the value of the `History` property. A `History` value of one (1) enables the toolbar history button; a value of zero (0) disables the history button

See Also:              `DisplayButtons`, `TakeToolbar`

## ToolBarUpdateClass Methods

The ToolBarUpdateClass inherits all the methods of the ToolBarTarget from which it is derived. See *ToolBarTarget Methods* for more information.

In addition to (or instead of) the inherited methods, the ToolBarUpdateClass contains the following methods:

### **DisplayButtons (enable appropriate toolbar buttons:ToolBarUpdateClass)**

#### **DisplayButtons, VIRTUAL**

The **DisplayButtons** method enables and disables the appropriate toolbar buttons for the ToolBarUpdateClass object based on the values of the HelpButton, InsertButton, ChangeButton, DeleteButton, and SelectButton properties.

Implementation: The TakeToolBar method calls the DisplayButtons method.

Example:

```

CODE
ToolBar.Init                               !initialize Toolbar object
ToolBar.AddTarget( ToolbarForm, -1 )       !register an Update Form target
ToolBar.AddTarget( REL1::ToolBar, ?RelTree ) !register a RelTree target
ToolBar.SetTarget( -1 )                    !calls DisplayButtons via TakeToolBar
!program code

MyToolBarUpdateClass.DisplayButtons PROCEDURE !a derived class virtual
CODE
ENABLE(ToolBar:History)                   !enable toolbar ditto button
DISABLE(ToolBar:Locate)                   !disable locator button
PARENT.DisplayButtons                     !call base class DisplayButtons
!your custom code here

```

See Also: HelpButton, InsertButton, ChangeButton, DeleteButton, SelectButton, TakeToolBar

**TakeEvent (convert toolbar events:ToolbarUpdateClass)****TakeEvent**( [ *vcr* ], *window manager* ), **VIRTUAL**


---

<b>TakeEvent</b>	Handles toolbar events for the ToolbarUpdateClass object.
<i>vcr</i>	An integer variable to receive the control number of the accepted VCR navigation button. This lets the TakeEvent method specify an appropriate subsequent action. If omitted, the ToolbarUpdateClass object does no "post processing" navigation.
<i>windowmanager</i>	The label of the ToolbarUpdateClass object's WindowManager object. See <i>Window Manager</i> for more information.

The **TakeEvent** method handles toolbar events for the ToolbarUpdateClass object.

The *vcr* parameter lets the TakeEvent method specify an appropriate subsequent or secondary action. For example, the ToolbarUpdateClass.TakeEvent method (for a FORM), may interpret a *vcr* scroll down as "save and then scroll." The method takes the necessary action to save the item and accomplishes the secondary scroll action by setting the *vcr* parameter.

Implementation: The ToolbarClass.TakeEvent method calls the TakeEvent method for the active ToolbarTarget object. The ToolbarClass.SetTarget method sets the active ToolbarTarget object.

Example:

```
ToolbarClass.TakeEvent PROCEDURE(<*LONG VCR>,WindowManager WM)
  CODE
  ASSERT(~SELF.List &= NULL)
  IF RECORDS(SELF.List)
    SELF.List.Item.TakeEvent(VCR,WM)
  END
```

See Also: ToolbarClass.SetTarget, ToolbarClass.TakeEvent

## TakeToolBar (assume control of the toolbar:ToolBarUpdateClass)

### TakeToolBar, VIRTUAL

The **TakeToolBar** method sets the toolbar state appropriate to the ToolBarUpdateClass object.

Implementation: The TakeToolBar method sets appropriate TIP attributes for the toolbar buttons and enables and disables toolbar buttons appropriate for the ToolBarUpdateClass object. The ToolBarClass.SetTarget method and the TryTakeToolBar method call the TakeToolBar method.

Example:

```
CODE
ToolBar.Init                               !initialize Toolbar object
  ToolBar.AddTarget( ToolbarForm, -1 )      !register an Update Form target
  ToolBar.AddTarget( REL1::ToolBar, ?RelTree ) !register a RelTree target
  ToolBar.SetTarget( -1 )                   !calls TakeToolBar
!program code

MyToolBarUpdateClass.TakeToolBar PROCEDURE  !a derived class virtual
CODE
  !your custom code here
  SELF.DisplayButtons                       !enable appropriate buttons
```

See Also:      ToolBarClass.SetTarget, TryTakeToolBar

**TryTakeToolbar (return toolbar control indicator:ToolbarUpdateClass)****TryTakeToolbar, VIRTUAL**

The **TryTakeToolbar** method returns a value indicating whether the ToolbarTarget object successfully assumed control of the toolbar. A return value of one (1 or True) indicates success; a value of zero (0 or False) indicates failure to take control of the toolbar.

Implementation: The ToolbarClass.SetTarget method calls the TryTakeToolbar method. The TryTakeToolbar method calls the TakeToolbar and returns True because, by default, a ToolbarUpdateClass object may always assume toolbar control.

Return Data Type: BYTE

Example:

```
ToolbarClass.SetTarget PROCEDURE(SIGNED Id)
I USHORT,AUTO
Hit USHORT
CODE
  ASSERT(~ (SELF.List &= NULL))
  IF Id                                !set explicitly requested target
    SELF.List.Id = Id
    GET(SELF.List,SELF.List.Id)
    ASSERT (~ERRORCODE())
    SELF.List.Item.TakeToolbar
  ELSE                                  !set a valid target
    LOOP I = 1 TO RECORDS(SELF.List)
      GET(SELF.List,I)
      IF SELF.List.Item.TryTakeToolbar() THEN Hit = I.
    END
    IF Hit THEN GET(SELF.List,Hit).
  END
```

See Also: TakeToolbar, ToolbarClass.SetTarget

---

# TranslatorClass

## TranslatorClass Overview

By default, the ABC Templates, the ABC Library, and the Clarion visual source code formatters generate American English user interfaces. However, Clarion makes it very easy to efficiently produce non-English user interfaces for your application programs.

The TranslatorClass provides very fast runtime translation of user interface text. The TranslatorClass lets you deploy a single application that serves all your customers, regardless of their language preference. That is, you can use the TranslatorClass to display several different user interface languages based on end user input or some other runtime criteria, such as INI file or control file contents.

Alternatively, you can use the Clarion translation files (\*.TRN) to implement a single non-English user interface at compile time.

## TranslatorClass Concepts

The TranslatorClass and the ABUTIL.TRN file provide a way to perform language translation at runtime. That is, you can make your program display one or more non-English user interfaces based on end user input or some other runtime criteria such as INI file or control file contents. You can also use the TranslatorClass to customize a single application for multiple customers. The TranslatorClass operates on all user interface elements including window controls, window titlebars, tooltips, list box headers, and static report controls.

## The ABUTIL.TRN File

---

The ABUTIL.TRN file contains translation pairs for all the user interface text generated by the ABC Templates and the ABC Library. A translation pair is simply two text strings: one text string for which to search and another text string to replace the searched-for text. At runtime, the TranslatorClass applies the translation pairs to each user interface element.

You can directly edit the ABUTIL.TRN file to add additional translation items. We recommend this method for translated text common to several applications. The translation pairs you add to the Translator GROUP declared in ABUTIL.TRN are automatically shared by any application relying on the ABC Library and the ABC Templates.

---

## Translating Custom Text

---

The default ABUTIL.TRN translation pairs do not include any custom text that you apply to your windows and menus. To translate custom text, you simply add translation pairs to the translation process, either at a global level or at a local level according to your requirements. To help identify custom text, the TranslatorClass automatically identifies any untranslated text for you; you need only supply the translation. See ExtractText for more information.

---

## Macro Substitution

---

The TranslatorClass defines and translates macro strings. A TranslatorClass macro is simply text delimited by percent signs (%), such as %mymacro%. You may use a macro within the text on an APPLICATION, WINDOW, or REPORT control or titlebar, or you may use a macro within TranslatorClass translation pairs text.

You define the macro with surrounding percent signs (%), and you define its substitution value with a TranslatorClass translation pair (without percent signs).

This macro substitution capability lets you

- translate a small portion (the macro) of a larger text string
- do multiple levels of translation (a macro substitution value may also contain a macro)

See the Conceptual Example for more information.

## TranslatorClass Relationship to Other Application Builder Classes

The WindowManager, PopupClass, and PrintPreviewClass optionally use the TranslatorClass to translate text at runtime. These classes do not require the TranslatorClass; however, if you want them to do runtime translation, you must include the TranslatorClass in your program. See the Conceptual Example.

## TranslatorClass ABC Template Implementation

The ABC Templates instantiate a global TranslatorClass object for each application that checks the **Enable Run-Time Translation** box on the **Global Properties** dialog. See Template Overview--Application Properties for more information.

The TranslatorClass object is called Translator, and each template-generated procedure calls on the Translator object to translate all text for its APPLICATION, WINDOW or REPORT. Additionally, the template-generated PopupClass objects (ASCIIViewer and BrowseBox templates) and PrintPreviewClass objects (Report template) use the Translator to translate menu text.

**Note:** The ABC Templates use the TranslatorClass to apply user interface text defined at compile time. The templates do not provide a runtime switch between user interface languages.

## TranslatorClass Source Files

The TranslatorClass source code is installed by default to the Clarion \LIBSRC folder. The TranslatorClass source code and its respective components are contained in:

ABUTIL.INC	TranslatorClass declarations
ABUTIL.CLW	TranslatorClass method definitions
ABUTIL.TRN	TranslatorClass default translation pairs

## TranslatorClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a TranslatorClass object.

This example applies both default and custom translations to a "preferences" window. It also collects and stores untranslated text in a file so you don't have to manually collect the text to translate.

```

PROGRAM
INCLUDE('ABUTIL.INC')           !declare TranslatorClass
MAP
END

MyTranslations GROUP           !declare local translations
Items      USHORT(4)           !4 translations pairs
           PSTRING('Company')  ! item 1 text (macro)
           PSTRING('Widget %CoType%') ! item 1 replacement text
           PSTRING('&Sound')     ! item 2 text
           PSTRING('&xSoundx')   ! item 2 replacement text
           PSTRING('&Volume')    ! item 3 text
           PSTRING('&xVolumex')  ! item 3 replacement text
           PSTRING('OK')        ! item 4 text
           PSTRING('xOKx')      ! item 4 replacement text
END

INIMgr      INIClass           !declare INIMgr object
Translator  TranslatorClass    !declare Translator object
CoType     STRING('Inc.')      !default company type
Sound      STRING('ON ')      !default preference value
Volume     BYTE(3)            !default preference value

PWindow WINDOW('%Company% Preferences'),AT(, ,100,35),IMM,SYSTEM,GRAY
           CHECK('&Sound'),AT(8,6),USE(Sound),VALUE('ON','OFF')
           PROMPT('&Volume'),AT(31,19),USE(?VolumePrompt)
           SPIN(@s20),AT(8,20,21,7),USE(Volume),HVSCROLL,RANGE(0,9),STEP(1)
           BUTTON('OK'),AT(57,3,30,10),USE(?OK)
END

CODE
INIMgr.Init('.\MyApp.INI')      !initialize INIMgr object
INIMgr.Fetch('Preferences','CoType',CoType) !get company type, default Inc.
Translator.Init                 !initialize Translator object:
                               ! add default translation pairs
Translator.AddTranslation(MyTranslations) !add local translation pairs
Translator.AddTranslation('CoType',CoType) !add translation pair from INI
Translator.ExtractText='. \MyApp.trn'     !collect user interface text
OPEN(PWindow)
Translator.TranslateWindow       !translate controls & titlebar

```



## TranslatorClass Properties

The TranslatorClass contains the following properties:

### **ExtractText (identify text to translate)**

#### **ExtractText    CSTRING(File:MaxFilePath)**

The **ExtractText** property contains the pathname of a file to receive a list of runtime user interface text to translate. If ExtractText contains a pathname, the TranslatorClass identifies, extracts, and writes the user interface text it encounters at runtime to the named file.

To generate a complete list of text to translate, assign a filename to the ExtractText property, compile and run your application, then open each procedure, menu, and option in the application. When you close the application, the TranslatorClass generates a sorted list of all the untranslated text items. You can then use this information to provide appropriate translations for the untranslated text. See AddTranslation for more information.

For applications that do dynamic text assignments based on data, you may even want to set the ExtractText property when you deploy your application, so you can collect the text that actually appears on end user screens based on the specific ways the end users work and the data they access.

Implementation:    The ExtractText property defaults to blank. A value of blank does not extract untranslated text. A non-blank value extracts the text, and a valid pathname writes the untranslated text to the specified file.

See Also:            AddTranslation

## TranslatorClass Methods

The TranslatorClass contains the following methods:

### AddTranslation (add translation pairs)

```
AddTranslation(      | group      | )
                    | text, translation |
```

---

<b>AddTranslation</b>	Adds translation pairs.
<i>group</i>	The label of a structure that contains one or more <i>text/translation</i> pairs.
<i>text</i>	A string constant, variable, EQUATE, or expression containing user interface text to search for. The TranslatorClass replaces each found <i>text</i> with its corresponding <i>translation</i> .
<i>translation</i>	A string constant, variable, EQUATE, or expression containing the replacement text for the corresponding <i>text</i> .

The **AddTranslation** method adds translation pairs to the runtime translation process.

The *text* is not limited to a single word; it may contain a phrase, or any text string, including TranslatorClass macros (see *TranslatorClass Concepts--Runtime Translation*).

Implementation: The *group* parameter must name a GROUP that *begins* the same as the TranslatorGroup structure declared in ABUTIL.INC:

```
TranslatorGroup GROUP,TYPE
Number          USHORT
                END
```

When you declare a translation *group*, be sure to set the correct number of translation pairs in the GROUP. For example:

```
MyAppTranslator GROUP
Pairs           USHORT(2)           !2 translation pairs
                PSTRING('&Insert')   !begin 1st pair
                PSTRING('&Agregar')  ! end 1st pair
                PSTRING('Insert a new Record') !begin 2nd pair
                PSTRING('Agregar un nuevo Registro') ! end 2nd pair
                END
```

The TranslatorClass uses whole word, case sensitive matching to search for *text*. For example, 'Insert' does not match '&Insert' or 'INSERT' or 'Insert a new Record.'

The Init method uses the AddTranslation method to add the translation pairs declared in ABUTIL.TRN to the translation process.

The various "Translate" methods apply the translation pairs.

Example:

```

MyTranslations GROUP                                !declare local translations
Pairs          USHORT(4)                          !4 translations pairs
               PSTRING('&Sound')                   ! item 1 text
               PSTRING('&xSoundx')                 ! item 1 replacement text
               PSTRING('&Volume')                  ! item 2 text
               PSTRING('&xVolumex')               ! item 2 replacement text
               PSTRING('Preferences')            ! item 3 text
               PSTRING('xPreferencesx')         ! item 3 replacement text
               PSTRING('OK')                     ! item 4 text
               PSTRING('xOKx')                  ! item 4 replacement text
               END
Translator      TranslatorClass                    !declare Translator object
CODE
Translator.Init                               !initialize Translator object
Translator.AddTranslation(MyTranslations)       !add default translation pairs
OPEN(MyWindow)                                !add local translation pairs
Translator.TranslateWindow                     !translate all window controls
                                                ! and window titlebar

```

See Also:      Init, TranslateControl, TranslatedControls, TranslateString, TranslateWindow

## Init (initialize the TranslatorClass object)

### Init

The **Init** method initializes the TranslatorClass object.

Implementation: The Init method uses the TranslatorClass.AddTranslation method to add the translation pairs declared in ABUTIL.TRN to the translation process.

Example:

```
Translator  TranslatorClass      !declare Translator object
CODE
Translator.Init                    !initialize Translator object:
                                   ! with default translation pairs

!program code
Translator.Kill                    !shut down Translator object
```

## Kill (shut down the TranslatorClass object)

### Kill

The **Kill** method frees any memory allocated during the life of the object and does any other required termination code.

Implementation: The Kill method writes out a list of untranslated text strings if the ExtractText property contains a valid INI file pathname.

Example:

```
Translator  TranslatorClass      !declare Translator object
CODE
Translator.Init                    !initialize Translator object:
                                   ! with default translation pairs

!program code
Translator.Kill                    !shut down Translator object
```

**TranslateControl (translate text for a control)****TranslateControl**( *control* [, *window*] ), **VIRTUAL****TranslateControl** Translates text for a control.*control* An integer constant, variable, EQUATE, or expression containing the control number of the control to translate.*window* The label of the APPLICATION, WINDOW, or REPORT to translate. If omitted, TranslateControl operates on the active target.The **TranslateControl** method translates the text for the specified *control*. The AddTranslation method sets the translation values for the control text.

Implementation: The TranslateControl method calls the TranslateString method for the specified control. Where applicable, the TranslateControl method translates MSG attribute text, TIP attribute text, and FORMAT attribute text.

The TranslateControl method does not translate USE variable contents; therefore it does not translate STRING controls that display a variable, nor the contents of ENTRY, SPIN, TEXT, or COMBO controls. You can use the TranslateString method to translate these elements if necessary.

Example:

```

PWindow WINDOW('Preferences'),AT(,,89,34),IMM,SYSTEM,GRAY
  CHECK('&Sound'),AT(8,6),USE(Sound),VALUE('ON','OFF')
  PROMPT('&Volume'),AT(31,19),USE(?VolumePrompt)
  SPIN(@s20),AT(8,20,21,7),USE(Volume),HVSCROLL,RANGE(0,9),STEP(1)
  BUTTON('OK'),AT(57,3,30,10),USE(?OK)
  END
CODE
OPEN(PWindow)
Translator.TranslateControl(?Sound)           !translate Sound check box
Translator.TranslateControl(?VolumePrompt)    !translate Volume prompt
ACCEPT                                         !leave OK button
END                                           ! and window title bar alone

```

See Also: AddTranslation, TranslateString

**TranslateControls (translate text for range of controls)**

**TranslateControls**( *first control*, *last control* [, *window*] ), **VIRTUAL**

---

**TranslateControls** Translates text for a range of controls.

*first control* An integer constant, variable, EQUATE, or expression containing the control number of the first control to translate.

*last control* An integer constant, variable, EQUATE, or expression containing the control number of the last control to translate.

*window* The label of the APPLICATION, WINDOW, or REPORT to translate. If omitted, TranslateControl operates on the active target.

The **TranslateControls** method translates the text for each control between the *first control* and the *last control*, inclusive. The AddTranslation method sets the translation values for the control text.

Implementation: The TranslateControls method calls the TranslateControl method for each control with a USE attribute in the specified range. The TranslateControls method ignores controls with no USE attribute.

Example:

```
Pwindow WINDOW( 'Preferences' ), AT( , , 89, 34 ), IMM, SYSTEM, GRAY
    CHECK( '&Sound' ), AT( 8, 6 ), USE( Sound ), VALUE( 'ON', 'OFF' )
    PROMPT( '&Volume' ), AT( 31, 19 ), USE( ?VolumePrompt )
    SPIN( @s20 ), AT( 8, 20, 21, 7 ), USE( Volume ), HVSCROLL, RANGE( 0, 9 ), STEP( 1 )
    BUTTON( 'OK' ), AT( 57, 3, 30, 10 ), USE( ?OK )
END
CODE
OPEN( Pwindow )
Translator.TranslateControls( ?Sound, ?VolumePrompt ) !translate ?Sound thru ?Volume
ACCEPT !leave OK button untranslated
END
```

See Also: AddTranslation, TranslateControl

**TranslateString (translate text)****TranslateString( *text* ), VIRTUAL**


---

**TranslateString**      Translates a text string.

*text*                  A string constant, variable, EQUATE, or expression containing text to search for.

The **TranslateString** method returns the translation value for the specified *text*. The translation values and macro substitution values are set by the AddTranslation method.

Implementation:      The TranslateString method uses whole word, case sensitive matching to search for *text*. For example, 'Insert' does not match '&Insert' or 'INSERT' or 'Insert a new Record.' If there is no translation value for the specified *text*, TranslateString returns *text*.

The TranslateString method implements the TranslatorClass macro substitution by translating any percent sign (%) delimited text it detects within its own return value.

Return Data Type:    **STRING**

Example:

```
MyVar STRING('Sound')
PWindow WINDOW('Preferences'),AT(,,89,34),IMM,SYSTEM,GRAY
    STRING(@s12),AT(8,30),USE(MyVar)
    BUTTON('OK'),AT(57,3,30,10),USE(?OK)
END
CODE
OPEN(PWindow)
MyVar=Translator.TranslateString(MyVar) !translate USE variable contents
ACCEPT
END
```

See Also:              AddTranslation

**TranslateWindow (translate text for a window)****TranslateWindow( [,window] ), VIRTUAL****TranslateControls** Translates text for each control on the WINDOW.*window* The label of the APPLICATION, WINDOW, or REPORT to translate. If omitted, TranslateControl operates on the active target.

The **TranslateWindow** method translates the text for each control on the active target (APPLICATION, WINDOW, or REPORT). The AddTranslation method sets the translation values for the controls.

Implementation: The TranslateWindow method calls the TranslateControls method, specifying the entire range of controls on the window, except for menus and toolbars.

Example:

```
PWindow WINDOW('Preferences'),AT(,,89,34),IMM,SYSTEM,GRAY
CHECK('&Sound'),AT(8,6),USE(Sound),VALUE('ON','OFF')
PROMPT('&Volume'),AT(31,19),USE(?VolumePrompt)
SPIN(@s20),AT(8,20,21,7),USE(Volume),HVSCROLL,RANGE(0,9),STEP(1)
BUTTON('OK'),AT(57,3,30,10),USE(?OK)
END
CODE
OPEN(PWindow)
Translator.TranslateWindow ! translate all controls
ACCEPT                    ! plus window titlebar
END
```

See Also: AddTranslation, TranslateControls



---

# ViewManager

## ViewManager Overview

The ViewManager class manages a VIEW. The ViewManager gives you easy, reliable access to all the sophisticated power and speed of VIEWS, through its proven objects. So you get this speed and power without reinventing any wheels.

## ViewManager Concepts

The management provided by the ViewManager includes defining and applying multiple sort orders, range limits (key based filters), and filters (non-key based) to the VIEW result set. It also includes opening, buffering, reading, and closing the VIEW. Finally, it includes priming and validating the view's primary file record buffer in anticipation of adding or updating records.

All these services provided by the ViewManager are applied to a VIEW--not a FILE. A VIEW may encompass some or all of the fields in one or more related FILES. The VIEW concept is extremely powerful and perhaps essential in a client-server environment with normalized data. The VIEW lets you access data from several different FILES as though from a single file, and it does so very efficiently. See VIEW in the *Language Reference* for more information.

In addition, the ViewManager supports buffering (some file drivers do not support buffering) which allows the performance of "browse" type procedures to be virtually instantaneous when displaying pages of records already read. Buffering (see BUFFER in the *Language Reference*) can also optimize performance when the file driver is a Client/Server back-end database engine (usually SQL-based), since the file driver can then optimize the calls made to the back-end database for minimum network traffic.

## ViewManager Relationship to Other Application Builder Classes

The ViewManager relies on the FieldPairsClass and the RelationManager to do much of its work. Therefore, if your program instantiates the ViewManager it must also instantiate these other classes. Much of this is automatic when you INCLUDE the ViewManager header (ABFILE.INC) in your program's data section. See Field Pairs Classes and Relation Manager Class for more information. Also, see the Conceptual Example.

Perhaps more significantly, the ViewManager serves as the foundation of the BrowseClass and the ProcessClass. That is, both the BrowseClass and the ProcessClass are derived from the ViewManager.

## **BrowseClass--An Interactive VIEW**

---

The BrowseClass implements an interactive VIEW that includes a visual display of records with scrolling, sorting, searching, and updating capabilities. See Browse Classes for more information.

---

## **ProcessClass--A Non-Interactive VIEW**

---

The ProcessClass implements a batch (non-interactive) VIEW with sorting and updating capability, but no visual display and therefore no scrolling or searching capability. See Process Class for more information.

## **ViewManager ABC Template Implementation**

The ViewManager serves as the foundation to the Browse procedure template, the Report procedure template, and the Process procedure template, because all these templates rely on VIEWS.

The BrowseClass and the ProcessClass are derived from the ViewManager, and the ABC Templates instantiate these derived classes; that is, the templates do not instantiate the ViewManager independently of the BrowseClass or ProcessClass. The Browse procedure template instantiates the BrowseClass, and the Process and Report procedure templates instantiate the ProcessClass.

## **ViewManager Source Files**

The ViewManager source code is installed by default to ..\LIBSRC. The specific ViewManager files and their respective components are:

ABFILE.INC	ViewManager declarations
ABFILE.CLW	ViewManager method definitions

## ViewManager Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a ViewManager object. This example simply establishes a VIEW with a particular sort order, range limit and filter, then processes the result set that fits the range and filter criteria.

```

PROGRAM
  INCLUDE('ABFILE.INC')           !declare ViewManager class
  MAP                             !program map
  END

GlobalErrors ErrorClass          !declare GlobalErrors object
View:Customer ViewManager        !declare View:Customer object

Access:CUSTOMER CLASS(FileManager) !declare Access:Customer object
Init                PROCEDURE
                  END

Relate:CUSTOMER CLASS(RelationManager)!declare Relate:Customer object
Init                PROCEDURE
                  END

CUSTOMER    FILE,DRIVER('TOPSPEED'),PRE(CUS),THREAD,BINDABLE
BYNUMBER    KEY(CUS:CUSTNO),NOCASE,OPT,PRIMARY
Record      RECORD,PRE()
CUSTNO      LONG
NAME        STRING(30)
ZIP         DECIMAL(5)
          END
          END

Customer:View VIEW(CUSTOMER) !declare Customer VIEW
          END

Low        LONG                !low end of range limit
High       LONG(1000)          !high end of range limit
ProgressMsg STRING(60)

ProgressWindow WINDOW('Processing...'),AT(,,215,60),GRAY,TIMER(100)
                STRING(@S60),AT(1,21,210,10),USE(ProgressMsg),CENTER
                BUTTON('Cancel'),AT(87,37,45,14),USE(?Cancel)
                END

```

## CODE

```

GlobalErrors.Init           !initialize GlobalErrors object
Relate:CUSTOMER.Ini         !initialize Relate:Customer object
View:Customer.Init(Customer:View,Relate:CUSTOMER) !initialize View:Customer object
View:Customer.AddSortOrder( CUS:BYNUMBER )      !add sort BYNUMBER
View:Customer.AppendOrder( 'CUS:Name,CUS:ZIP' ) !add secondary sorts
View:Customer.AddRange(CUS:CUSTNO,Low,High)     !add a range limit
View:Customer.SetFilter( 'CUS:ZIP=33066','1' )  !add filter #1
Relate:CUSTOMER.Open       !open customer & related files
OPEN(ProgressWindow)      !open the window
ProgressMsg='Processing...'
```

## ACCEPT

```

CASE EVENT()
OF Event:OpenWindow
  View:Customer.Reset(1)           !open view, apply range & filter
OF Event:Timer
  CASE View:Customer.Next()        !get next view record
  OF Level:Notify                  !if end of file, stop
    POST(EVENT:CloseWindow)
    BREAK
  OF Level:Fatal                   !if fatal error, stop
    POST(EVENT:CloseWindow)
    BREAK
  END
  CUS:ZIP=33065                    !process the record
  IF Relate:CUSTOMER.Update()      !update customer & related files
    BREAK
  ELSE
    ProgressMsg = CLIP(CUS:Name)&' zip changed to '&CUS:ZIP
    DISPLAY(ProgressMsg)
  END
END
IF FIELD() = ?Cancel              !if user cancelled, stop
  IF EVENT() = Event:Accepted
    POST(Event:CloseWindow)
  END
END
END
Relate:CUSTOMER.Close             !close customer & related files
View:CUSTOMER.Kill               !shut down View:Customer object
Relate:CUSTOMER.Kill             !shut down Relate:Customer object
GlobalErrors.Kill                !shut down GlobalErrors object
```

```
Access:CUSTOMER.Init PROCEDURE
CODE
PARENT.Init(Customer,GlobalErrors)
SELF.FileNameValue = 'CUSTOMER.TPS'
SELF.Buffer &= CUS:Record
SELF.AddKey(CUS:BYNUMBER, 'CUS:BYNUMBER',1)
SELF.LazyOpen = False
```

```
Relate:CUSTOMER.Init PROCEDURE
CODE
Access:CUSTOMER.Init
PARENT.Init(Access:CUSTOMER,1)
```

## ViewManager Properties

The ViewManager properties include references to the specific view being managed, as well as several flags or switches that tell the ViewManager how to manage the referenced view.

The references are to the VIEW, the primary FILE's RelationManager object, and the VIEW's sort information. These references allow the otherwise generic ViewManager object to process a specific view.

The processing switches include buffering parameters that allow asynchronous read-ahead buffering of pages and saving pages of already read records. This buffering provides instant response for procedures displaying pages of records already read, and can also minimize network traffic for Client/Server programs by reducing packets.

Each of these properties is fully described below.

### Order (sort, range-limit, and filter information)

#### **Order &SortOrder, PROTECTED**

The **Order** property is a reference to a structure that contains the sort, range, and filter information for the managed VIEW. The ViewManager methods use this information to sort, range limit, and filter the VIEW result set.

Several ViewManager methods affect the contents of the Order property, including AddSortOrder, AddRange, AppendOrder, and SetFilter. The SetOrder method overrides a particular sort order, and the SetSort method determines which sort order is current for the underlying VIEW.

Implementation: The Order property is a reference to QUEUE declared in ABFILE.INC:

```

FilterQueue QUEUE,TYPE
ID          STRING(30)      !sorted to indicate priority
Filter      &STRING        !filter expression
            END

SortOrder   QUEUE,TYPE     !sort & filter information
Filter      &FilterQueue   !ANDed list of filter expressions
FreeElement ANY           !the Free key element
LimitType   BYTE          !range limit type flag
MainKey     &KEY           !the main KEY
Order       &STRING        !ORDER expression list
RangeList   &BufferedPairsClass !list of fields in range limit
            END

```

See Also: AddSortOrder, AddRange, AppendOrder, SetFilter, SetOrder, SetSort

## PagesAhead (buffered pages)

### **PagesAhead USHORT**

The **PagesAhead** property controls automatic record set buffering for the managed view (see *BUFFER* in the *Language Reference*). Some file drivers do not support buffering. PagesAhead specifies the number of additional "pages" of records to read ahead of the currently displayed page.

Implementation: The Init method sets the PagesAhead property to zero (0). The Open method implements the buffering specified by the PagesAhead, PagesBehind, PageSize, and TimeOut properties.

See Also: Init, Open, PagesBehind, PageSize, TimeOut

## PagesBehind (buffered pages)

### **PagesBehind USHORT**

The **PagesBehind** property controls automatic record set buffering for the managed view (see *BUFFER* in the *Language Reference*). Some file drivers do not support buffering. PagesBehind specifies the number of "pages" of already read records to save.

Implementation: The Init method sets the PagesBehind property to two (2). The Open method implements the buffering specified by the PagesAhead, PagesBehind, PageSize, and TimeOut properties.

See Also: Init, Open, PagesAhead, PageSize, TimeOut

**PageSize (buffer page size)****PageSize      USHORT**

The **PageSize** property controls automatic record set buffering for the managed view (see *BUFFER* in the *Language Reference*). Some file drivers do not support buffering. PageSize specifies the number of records in a buffer "page."

Implementation:      The Init method sets the PageSize property to twenty(20). The Open method implements the buffering specified by the PagesAhead, PagesBehind, PageSize, and TimeOut properties.

See Also:              Init, Open, PagesAhead, PagesBehind, TimeOut

**Primary (the primary file RelationManager )****Primary              &RelationManager, PROTECTED**

The **Primary** property is a reference to the RelationManager object for the managed VIEW's primary file. The ViewManager methods use this property to enforce relational integrity constraints among related files within the managed VIEW.

The ViewManager.Init method sets the value of the Primary property.

See Also:              Init

## **SavedBuffers (saved record buffers)**

**SavedBuffers    &BuffersQueue, PROTECTED**

The **SavedBuffers** property contains references to saved copies of the record buffer for the managed view. The saved record images may be used to detect changes by other workstations, to implement cancel operations, etc.

Implementation:    The SaveBuffers method stores a copy of the current Buffer contents into the SavedBuffers property.

                    The RestoreBuffers method releases memory allocated by the SaveBuffers method. Therefore, to prevent a memory leak, each call to SaveBuffers should be paired with a corresponding call to RestoreBuffers.

                    SavedBuffers is a reference to a QUEUE declared in ABFILE.INC as follows:

```
BuffersQueue QUEUE,TYPE    !Saved records
Id            USHORT        !Handle to recognize saved instance
FM            &FileManager !Reference to the FileManager Class
                              END
```

See Also: SaveBuffers, Restore Buffers

## **TimeOut (buffered pages freshness)**

**TimeOut**      **USHORT**

The **TimeOut** property controls automatic record set buffering for the managed view (see *BUFFER* in the *Language Reference*). Some file drivers do not support buffering.

TimeOut specifies the number of seconds the buffered records are considered "trustworthy" in a network environment. If the TimeOut period has expired, the VIEW fills a request for records from the backend database rather than from the buffer.

Implementation:      The Init method sets the TimeOut property to sixty (60). The Open method implements the buffering specified by the PagesAhead, PagesBehind, PageSize, and TimeOut properties.

See Also:              Init, Open, PagesAhead, PagesBehind, PageSize

## **View (the managed VIEW)**

**View**      **&VIEW**

The **View** property is a reference to the managed VIEW. The View property simply identifies the managed VIEW for the various ViewManager methods.

The ViewManager.Init method sets the value of the View property.

See Also:              Init

## ViewManager Methods

The ViewManager contains the following methods.

### ViewManager Functional Organization--Expected Use

As an aid to understanding the ViewManager, it is useful to organize its methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the ViewManager methods.

#### Non-Virtual Methods

---

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the ViewManager object
AddRange	add a range limit to the active sort order
AddSortOrder	add a sort order
AppendOrder	refine the active sort order
Kill <sub>v</sub>	Shut down the ViewManager object

##### **Mainstream Use:**

Open <sub>v</sub>	open the VIEW
Next <sub>v</sub>	get the next element
Previous <sub>v</sub>	get the previous element
PrimeRecord	prepare a record for adding
ValidateRecord <sub>v</sub>	validate the current element
SetFilter <sub>v</sub>	specify a filter for the active sort order
SetSort <sub>v</sub>	set the active sort order
Close <sub>v</sub>	close the VIEW

##### **Occasional Use:**

SetOrder <sub>v</sub>	replace the active sort order
UseView	use LazyOpen files

<sub>v</sub> These methods are also Virtual.

---

**Virtual Methods**

---

Typically, you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Open	open the VIEW
Next	get the next element
Previous	get the previous element
Reset	reset the VIEW position
SetSort	set the active sort order
SetFilter	specify a filter for the active sort order
SetOrder	replace the active sort order
ApplyFilter	range limit and filter the result set
ApplyOrder	sort the result set
ApplyRange	range limit & filter the result set
ValidateRecord	validate the current element
GetFreeElementName	return the free element field name
GetFreeElementPosition	return the free element field position
Close	close the VIEW
Kill	shut down the ViewManager object

**AddRange (add a range limit)**

```
AddRange( field |[ min limit [ max limit ] | )
           | primaryrelation, parentrelation |
```

---

<b>AddRange</b>	Specifies a sort-specific range limit.
<i>field</i>	The label of the field to limit. This need not be a component of a KEY or INDEX, but VIEW performance is substantially faster if it is.
<i>min limit</i>	A constant, variable, EQUATE, or expression that specifies the value, or the lower end of a range of values, to which the <i>field</i> is limited. If omitted, the <i>field</i> is limited to its current value.
<i>max limit</i>	A constant, variable, EQUATE, or expression that specifies the upper end of an inclusive range of values to which the <i>field</i> is limited. The lower end of the inclusive range is specified by <i>min limit</i> . If <i>max limit</i> is omitted, the <i>field</i> is limited to the value of <i>min limit</i> .
<i>primaryrelation</i>	The label of the RelationManager object for the managed VIEW's primary file. This limits all available linking fields to their current values in the corresponding parent file fields.
<i>parentrelation</i>	The label of the RelationManager object for the primary file's parent file. The ViewManager uses this object to get the limiting values from the parent file for a file-relationship range limit.

The **AddRange** method specifies a sort-specific range limit that may be applied to the VIEW when the range limit's sort order is active. When the range limit is applied, only those records whose *field* contains the specified value(s) are included in the result set. You may specify only one range limit per sort order.

Implementation: The AddSortOrder method adds a sort order. The ApplyRange method applies the active sort order's range limit. The SetSort method sets the active sort order.

AddRange ignores the *field* parameter when the *primaryrelation* parameter is present.

Example:

```
MyView.AddSortOrder(ORD:ByCustomer)           !sort by customer no
MyView.AddRange(ORD:CustNo,Relate:Orders,Relate:Customer) !range limit by parent file
MyView.AddSortOrder(ORD:ByOrder)             !sort by order no
MyView.AddRange(ORD:OrderNo)                 !range limit by current
                                              !value of ORD:OrderNo
```

See Also: AddSortOrder, ApplyRange, SetSort

**AddSortOrder (add a sort order)****AddSortOrder( [key] ), PROC**


---

**AddSortOrder** Specifies a sort order for the ViewManager object.

*key*                    The label of the primary file KEY on which to sort. If omitted, the ViewManager processes in record order.

The **AddSortOrder** method specifies a sort order for the ViewManager object and returns a number identifying the sequence in which the sort order was added.

Only one sort order is active at a time. The SetSort method sets the active sort order based on the sequence numbers returned by AddSortOrder.

Implementation:    You may specify multiple sort orders by calling AddSortOrder multiple times. The first call to AddSortOrder returns one (1), the second call returns two (2), etc.

Return Data Type:    BYTE

Example:

```

CustSort = MyView.AddSortOrder(ORD:ByCustomer)           !sort by customer no
MyView.AddRange(ORD:CustNo,Relate:Orders,Relate:Customer) !range limit by parent file
OrderSort = MyView.AddSortOrder(ORD:ByOrder)            !sort by order no
MyView.AddRange(ORD:OrderNo)                            !range limit by current
                                                         !value of ORD:OrderNo

!program code
IF MyView.SetSort(CustSort)                             !set active sort order
  DISPLAY                                               !if changed, refresh
END

```

See Also:            SetSort

## AppendOrder (refine a sort order)

**AppendOrder**( *expression list* ), **VIRTUAL**

---

**AppendOrder** Refines the active sort order for the ViewManager object.

*expression list* A string constant, variable, EQUATE, or expression that contains an ORDER expression list. See the *Language Reference--ORDER* for more information.

The **AppendOrder** method refines or extends the active sort order for the ViewManager object.

The SetSort method sets the active sort order.

Implementation: The ViewManager implements sort orders with the VIEW's ORDER attribute. The AppendOrder method appends the *expression list* to the active sort order's expression list. You do not need to prepend a comma or other separator to the *expression list*.

Example:

```
MyView.AddSortOrder(ORD:ByCustomer)           !sort by customer no  
MyView.AppendOrder('CUST:CustName')         !and customer name
```

See Also: AddSortOrder, SetSort

## ApplyFilter (range limit and filter the result set)

### ApplyFilter, VIRTUAL

The **ApplyFilter** method applies the range limits and filter for the active sort order to the managed VIEW. The filter applies starting with the next read.

The AddSortOrder and SetSort methods set the active sort order. The SetFilter method sets filter expression.

Implementation: The ViewManager implements range limits and filters with the VIEW's FILTER attribute. See the *Language Reference--FILTER* for more information.

Example:

```
MyView.AddSortOrder(ORD:ByCustomer)           !sort by customer no
MyView.AddRange(ORD:CustNo,Relate:Orders,Relate:Customer) !range limit by parent file
MyView.SetFilter( '(CUST:Name>'T')' )         !set customer name filter
!program code
MyView.ApplyFilter                             !apply the filter
MyView.Next()                                 !get next subject to filter
```

See Also: SetFilter, SetSort

## ApplyOrder (sort the result set)

### ApplyOrder, VIRTUAL

The **ApplyOrder** method applies the active sort order to the managed VIEW. The order applies starting with the next read from the VIEW.

The AddSortOrder method sets the available sort orders. The SetSort method sets the active sort order.

Implementation: The ViewManager implements sort orders with the VIEW's ORDER attribute. See the *Language Reference--ORDER* for more information.

Example:

<code>MyView.AddSortOrder(ORD:ByCustomer)</code>	<code>!sort by customer no</code>
<code>!program code</code>	
<code>MyView.ApplyOrder</code>	<code>!apply the order</code>
<code>MyView.Next()</code>	<code>!get next in specified order</code>

See Also: AddSortOrder, SetSort

## ApplyRange (conditionally range limit and filter the result set)

### ApplyRange, VIRTUAL, PROC

The **ApplyRange** method applies the range limits and calls the ApplyFilter method if the range limits have changed. The ApplyRange method returns a value indicating whether or not a change occurred. A return value of one (1 or True) indicates a change; a return value of zero (0 or False) indicates no change.

The AddRange method specifies the range limits for the ViewManager object. The SetSort method sets the active sort order.

Implementation: The ApplyRange method applies range limits and filters with the ApplyFilter method.

Return Data Type: BYTE

Example:

```
MyView.AddSortOrder(ORD:ByCustomer)           !sort by customer no
MyView.AddRange(ORD:CustNo,Relate:Orders,Relate:Customer) !range limit by parent file
!program code
MyView.ApplyRange                             !apply the range limit
MyView.Next()                                 !get next, subject to range
```

See Also: SetSort ,AddRange, ApplyFilter

## Close (close the view)

### **Close ( <force> ), VIRTUAL**

The **Close** method closes the managed VIEW.

Normally, a VIEW is closed by the **Close** method, providing that it was opened with the Open method. The *force* flag is used to close a VIEW that was not originally opened by the ViewManagers's Open method

Example:

```
MyView.AddSortOrder(ORD:ByCustomer)           !sort by customer no
MyView.AddRange(ORD:CustNo,Relate:Orders,Relate:Customer) !range limit by parent file
MyView.Open                                     !open the view
!program code
MyView.Close                                     !close the view
```

## GetFirstSortField (return first field of current sort)

### **GetFirstSortField**

The **GetFirstSortField** method returns the first field that contains the current sort. If there is no current sort active, the **GetFirstSortField** method returns a NULL.

Implementation: The BrowseClass uses the GetFirstSortField method to prime the BrowseClass.SetLocatorField method.

Return Data Type: ANY

## **GetFreeElementName (return free key element name)**

### **GetFreeElementName**

The **GetFreeElementName** method returns the fully qualified field name of the first sort field in the active sort order that is not limited to a single value by the applied range limit. For example, consider a VIEW sorted by Customer, Order, and Item, with the Customer field range limited to its current value. The free element is the Order field. But remove the range limit, and the free element is the Customer field.

The AddSortOrder method sets the key/sort order for the VIEW. The SetSort method sets the active sort order. The AddRange method adds range limits.

Implementation: The FilterLocatorClass uses the GetFreeElementName method to refresh the window.

Return Data Type: STRING

Example:

```
BuildFilter PROCEDURE(STRING filter)
FieldName CSTRING(100)
CODE
  FieldName = MyView.GetFreeElementName()           !get filterable field name
  MyView.SetFilter(FieldName&'[1] = ''&filter[1]&'') !set a filter expression
  MyView.ApplyFilter()                             !apply the filter expression
```

See Also: AddRange, AddSortOrder, SetSort

## GetFreeElementPosition (return free key element position)

### **GetFreeElementPosition, PROTECTED, VIRTUAL**

The **GetFreeElementPosition** method returns the position of the first sort field in the active sort order that is not limited to a single value by the applied range limit. For example, consider a VIEW sorted by Customer, Order, and Item, with the Customer field range limited to its current value. The free element is the Order field. But remove the range limit, and the free element is the Customer field.

The AddSortOrder method sets the key/sort order for the VIEW. The SetSort method sets the active sort order. The AddRange method adds range limits.

Implementation: The BrowseClass.TakeKey method uses the GetFreeElementPosition method to reposition the VIEW based on the fixed key elements. The GetFreeElementName method uses the GetFreeElementPosition method to find the free element name.

Return Data Type: BYTE

Example:

#### **BrowseClass.TakeKey PROCEDURE**

```
!method code
IF SELF.Sort.Locator.TakeKey()
  Handled = 1
  SELF.Reset(SELF.GetFreeElementPosition())
  SELF.ResetQueue(Reset:Done)
ELSE
  SELF.ListControl{PROP:SelStart} = SELF.CurrentChoice
END
```

See Also: GetFreeElementName, BrowseClass.TakeKey

**Init (initialize the ViewManager object)**

**Init**( *view*, *primaryrelation* [, *order* ] )

---

<b>Init</b>	Initializes the ViewManager object.
<i>view</i>	The label of the managed VIEW.
<i>primaryrelation</i>	The label of the RelationManager object for the <i>view</i> 's primary file.
<i>order</i>	A structure containing the sort, range limit, and filter information for the managed VIEW. If omitted, the Init method supplies an empty SortOrder structure that may be set up with AddSortOrder, AppendOrder, SetOrder, AddRange, and SetFilter methods.

The **Init** method initializes the ViewManager object.

Implementation: The Init method sets the values of the Order, PagesAhead, PagesBehind, PageSize, Primary, and View properties.

The *order* parameter allows derived classes, such as the BrowseClass, to add additional sort information to their underlying views.

By passing the Order property from another ViewManager object or the Sort property from a BrowseClass object as the *order* parameter, you can implement several objects with similar sorts, filters, and range limits.

Example:

```
MyView.Init(OrderView,Relate:Order)!initialize the ViewManager
MyView.Open                          !open the view
!program code
MyView.Close                          !close the view
MyView.Kill                           !shut down the ViewManager
```

See Also: Order, Primary, View, PagesAhead, PagesBehind, PageSize

**Kill (shut down the ViewManager object)****Kill, VIRTUAL**

The **Kill** method shuts down the ViewManager object by freeing any memory allocated during the life of the object and executing any other required termination code.

Example:

```
MyView.Init(OrderView,Relate:Order)           !initialize the ViewManager
MyView.AddSortOrder(ORD:ByCustomer)          !sort by customer no
MyView.AddRange(ORD:CustNo,Relate:Orders,Relate:Customer) !range limit by parent file
MyView.Open                                  !open the view
!program code
MyView.Close                                 !close the view
MyView.Kill                                  !shut down the ViewManager
```

**Next (get the next element)****Next, VIRTUAL**

The **Next** method gets the next VIEW element, subject to the applied sort order, range limit, and filter, and returns a value indicating its success or failure.

If Next succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns Level:Notify or Level:Fatal depending on the error encountered. See *Error Class* for more information on severity levels.

Implementation: The Next method uses the ValidateRecord method to validate records that are not filtered out.

Return Data Type: BYTE

Example:

```
CASE MyView.Next()           !try to get the next record
OF Level:Benign              !& check for success
  !process the record
OF Level:Notify              !& check for failure
  !write error log
OF Level:Fatal               !& check for fatality
  POST(Event:CloseWindow)
  BREAK
END
```

See Also: ValidateRecord

**Open (open the view)****Open, VIRTUAL**

The **Open** method opens the managed VIEW.

Implementation: The Open method opens the view *and* applies the active sort order and filter with the ApplyOrder and ApplyFilter methods. The Open method applies the buffering specified by the PagesAhead, PagesBehind, PageSize, and TimeOut properties.

Example:

```
MyView.AddSortOrder(ORD:ByCustomer)           !sort by customer no
MyView.AddRange(ORD:CustNo,Relate:Orders,Relate:Customer) !range limit by parent file
MyView.Open                                     !open the view
!program code
MyView.Close                                   !close the view
```

See Also: ApplyFilter, ApplyOrder, PagesAhead, PagesBehind, PageSize, TimeOut

## Previous (get the previous element)

### Previous, VIRTUAL

The **Previous** method gets the previous VIEW element, subject to the applied sort order, range limit, and filter, and returns a value indicating its success or failure.

Implementation: If Previous succeeds, it returns Level:Benign (declared in ABERROR.INC). If it fails, it returns Level:Notify or Level:Fatal depending on the error encountered. See *Error Class* for more information on severity levels.

The Previous method uses the ValidateRecord method to validate records that are not filtered out.

Return Data Type: BYTE

Example:

```
CASE MyView.Previous()      !try to get the previous record
OF Level:Benign            !& check for success
  !process the record
OF Level:Notify            !& check for failure
  !write error log
OF Level:Fatal             !& check for fatality
  POST(Event:CloseWindow)
  BREAK
END
```

See Also: ValidateRecord

## PrimeRecord (prepare a record for adding:ViewManager)

**PrimeRecord**( [*suppress clear*] ), **VIRTUAL**

---

**PrimeRecord** Prepares the VIEW's primary file record buffer to add a new record.

*suppress clear* An integer constant, variable, EQUATE, or expression that indicates whether or not to clear the record buffer. A value of zero (0 or False) clears the buffer; a value of one (1 or True) does not clear the buffer. If omitted, *suppress clear* defaults to zero (0).

The **PrimeRecord** method prepares the VIEW's primary file record buffer with initial values to add a new record.

Implementation: The PrimeRecord method uses the primary file's FileManager.PrimeRecord method to prime the record. Then it uses any applicable range limit information to prime other fields. The *suppress clear* parameter lets you clear or retain any other values in the record buffer.

Example:

```
CASE FIELD()  
OF ?InsertButton           !on insert button  
  CASE EVENT()  
  OF EVENT:Accepted        !if insert clicked  
    MyView.PrimeRecord     !prime the record for adding  
    !insert the new record  
  END  
END
```

See Also: FileManager.PrimeRecord

**Reset (reset the view position)****Reset( [ *number* ] ), VIRTUAL**


---

<b>Reset</b>	Resets the VIEW position.
<i>number</i>	An integer constant, variable, EQUATE, or expression that specifies the start position based on the contents of the first <i>number</i> components of the applicable ORDER attribute. If omitted, Reset positions the VIEW to the first element in theVIEW's result set.

The **Reset** method resets the VIEW position to the beginning of the result set specified by the VIEW's applied sort order, range limit and filter. The *number* parameter further refines the position by considering the *contents* of the first *number* expressions in the active sort order.

For example, consider a VIEW sorted by Customer where Customer's value is ten(10). If *number* is omitted, Reset positions to the element with the lowest Customer value, regardless of Customer's value. However, if *number* is one (1), Reset positions to the first element with a Customer value of ten (10).

Implementation: The Reset method calls the Open method and SETs the managed VIEW. See the *Language Reference--SET* for more information.

Example:

```
View:Customer.Init(Customer:View,Relate:CUSTOMER) !initialize View:Customer object
View:Customer.AddSortOrder( CUS:BYNUMBER ) !add sort BYNUMBER
View:Customer.AddRange(CUS:CUSTNO,Low,High) !add a range limit
View:Customer.SetFilter( 'CUS:ZIP=33064','1' ) !add filter #1
Relate:CUSTOMER.Open !open customer & related files
View:Customer.Reset !open view, apply range & filter
IF View:Customer.Next() !get first view record
  HALT !if no records, stop
END
```

See Also: Open

## **RestoreBuffers (restore VIEW file buffers)**

**RestoreBuffers( ), VIRTUAL**

---

**RestoreBuffers** Restores the contents of a VIEW's file buffers.

The **RestoreBuffers** method restores the current file buffer's contents specified by the view from an internal queue as defined by the *SavedBuffers* property.

See Also: SaveBuffers

## **SaveBuffers (save VIEW file buffers)**

**SaveBuffers( ), VIRTUAL**

---

**SaveBuffers** Saves the contents of a VIEW's file buffers.

The **SaveBuffers** method saves the current file buffer's contents specified by the view to an internal queue defined by the *SavedBuffers* property.

See Also: RestoreBuffers

**SetFilter (add, change, or remove active filter)****SetFilter**( *expression* [, *id* ] ), VIRTUAL

---

<b>SetFilter</b>	Specifies a filter for the active sort order.
<i>expression</i>	A string constant, variable, EQUATE, or expression that contains a FILTER expression. See <i>FILTER</i> in the <i>Language Reference</i> for more information. If <i>expression</i> is null ("), SetFilter deletes any existing filter with same <i>id</i> .
<i>id</i>	A string constant, variable, EQUATE, or expression that uniquely identifies (and prioritizes) the filter so you can apply multiple filter conditions, and so you can replace or remove filter conditions with subsequent calls to SetFilter. If omitted, the filter gets a default <i>id</i> so that subsequent calls to SetFilter with no <i>id</i> replace the filter <i>expression</i> set by prior calls to SetFilter with no <i>id</i> .

The **SetFilter** method specifies a filter for the active sort order. When the filter is applied, the view only includes those elements whose *expression* evaluates to true.

The *id* parameter lets you specify multiple filter *expressions* or replace a specific *expression* by its *id*. If you set several *expressions*, each with a unique *id*, then all those *expressions* must evaluate to true to include an item in the result set.

The ViewManager evaluates the *expressions* in *id* order, so it is efficient to prioritize *expressions* most likely to fail; for example:

```
MyView.SetFilter('TaxPayer=True','9Tax')      !low priority expression
MyView.SetFilter('LotteryWinner=True','1Lot') !high priority expression
!evaluates as: (LotteryWinner=True) AND (TaxPayer=True)
```

The ApplyFilter and ApplyRange methods apply the active sort order's filter. The SetSort method sets the active sort order.

Implementation: The ViewManager uses the *id* to indicate the priority of the *expression*. The priority is implemented by sorting the list of filter expressions by the *id*. The *id* is truncated after 30 characters. If omitted, *id* defaults to '5 Standard' which specifies a medium priority filter that is replaced by any subsequent calls to SetFilter with *id* omitted (or '5 Standard') and with the same active sort order.

Each call to SetFilter with a unique *id* parameter adds to the filter expression for the active sort order. Multiple expressions added in this fashion are joined with the boolean AND operator.

The SetFilter method adds the filter *id* and *expression* to the Order property.

Example:

```
MyView.AddSortOrder(ORD:ByOrder)           !order no. sort (1)
MyView.SetFilter('(ORD:OrdNo=CUST:OrdNo)', '1OrderNo') !filter on OrderNo
MyView.SetFilter('(ORD:Date=&TODAY()&)', '1Date')    !AND on date. Date test applied
                                                !first because it sorts first
MyView.AddSortOrder(ORD:ByName)           !customer name sort (2)
MyView.SetFilter('CUST:Name[1]='A')        !filter on cust name

!program code
MyView.SetSort(2)                          !sort by customer name
MyView.SetFilter('CUST:Name[1]='J')        !new filter on cust name
                                                !replaces prior name filter
```

See Also:      AddSortOrder, Order

## SetOrder (replace a sort order)

**SetOrder**( *expression list* ), **VIRTUAL**

---

**SetOrder** Replaces the active sort order.

*expression list* A string constant, variable, EQUATE, or expression that contains an ORDER attribute expression list. See the *Language Reference--ORDER* for more information.

The **SetOrder** method replaces the active sort order for the ViewManager object.

The SetSort method sets the active sort order.

Implementation: The ViewManager implements sort orders with the VIEW's ORDER attribute. The SetOrder method replaces the active sort order's expression list with the *expression list*.

Example:

```
MyView.AddSortOrder(ORD:ByCustomer)           !sort by customer no
!program code
MyView.SetOrder(CUST:CustName)                 !sort by customer name
```

See Also: SetSort



## UseView (use LazyOpen files)

### UseView, PROTECTED

The **UseView** method notifies ABC Library objects that the files in the managed view whose opening was delayed by the LazyOpen property are about to be used.

Implementation: The Init and Open methods call the UseView method. The UseView method calls FileManager.UseFile for each file in the managed view.

Example:

```
ViewManager.Open PROCEDURE
CODE
IF ~SELF.Opened
  ASSERT(RECORDS(SELF.Order))
  SELF.UseView()           !really open files
  OPEN(SELF.View)
  IF ERRORCODE()
    SELF.Primary.Me.Throw(Msg:ViewOpenFailed)
  END
  BUFFER(SELF.View,SELF.PageSize,SELF.PagesBehind,SELF.PagesAhead,SELF.TimeOut)
  SELF.Opened = 1
  SELF.ApplyOrder
  SELF.ApplyFilter
END
```

See Also: Init, Open, FileManager.LazyOpen, FileManager.UseFile

**ValidateRecord (validate an element)****ValidateRecord, VIRTUAL**

The **ValidateRecord** method validates the current VIEW element and returns a value indicating whether or not the data is valid. A return value of zero (0) indicates the item is valid; any other value indicates the item is invalid.

Implementation: The ValidateRecord is a virtual placeholder for derived class methods.

The Next and Previous methods call the ValidateRecord method.

Return values are declared in ABFILE.INC as follows:

```
ITEMIZE(0),PRE(Record)
OK          EQUATE      !Record passes range and filter
OutOfRange EQUATE      ! Record fails range test
Filtered   EQUATE      ! Record fails filter tests
END
```

Return Data Type: BYTE

Example:

```
ViewManager.Next PROCEDURE
CODE
LOOP
NEXT(SELF.View)
IF ERRORCODE()
IF ERRORCODE() = BadRecErr
RETURN Level:Notify
ELSE
SELF.Primary.Me.Throw(Msg:AbortReading)
RETURN Level:Fatal
END
ELSE
CASE SELF.ValidateRecord()
OF Record:OK
RETURN Level:Benign
OF Record:OutOfRange
RETURN Level:Notify
END
END
END
```

See Also: Next, Previous

# WindowComponent Interface

## WindowComponent Overview

The WindowComponent interface is used with the WindowManager to provide an program efficient way of the window's components to communicate easily with the WindowManager. Methods common to all components are included with the interface, including event handling (TakeEvent), initialization and refreshing the components and window itself (Reset, Update), and general housekeeping (Buffer Save and Restore, Kill).

## WindowComponent Concepts

The WindowComponent interface defines a set of common methods an object must implement in order for the object to plug into the window.

Although all of the classes that implement the WindowComponent interface must implement all of the methods, that does not mean that all need to do something.

## Relationship to Other Application Builder Classes

The BrowseClass, FileDropClass, BrowseToolbarClass, FormVCRClass, HistHandlerClass, and RecipientControl all implement the WindowComponent interface.

## WindowComponent Source Files

The WindowComponent source code is installed by default to the Clarion \LIBSRC folder. The specific WindowComponent source code and their respective components are contained in:

ABWINDOW.INC	WindowComponent interface declaration
ABBROWSE.CLW	BrowseClass.WindowComponent method definitions
ABDROPS.CLW	FileDropClass.WindowComponent method definitions
ABDST.CLW	RecipientControl.WindowComponent method definitions
ABERROR.CLW	HistHandler.WindowComponent method definitions
ABTOOLBA.CLW	BrowseToolbarClass.WindowComponent method definitions
ABVCRFRM.CLW	FormVCRClass.WindowComponent method definitions

## WindowComponent Methods

### **WindowComponent Methods**

The WindowComponent interface defines the following methods.

### **Kill(shutdown the parent object)**

#### **Kill**

The **Kill** method releases any memory allocated during the life of the object and performs any other required termination code.

#### **BrowseClass Implementation:**

The Kill method calls the BrowseClass.Kill method to terminate the BrowseClass object.

#### **FileDropClass Implementation:**

The Kill method calls the FileDropClass.Kill method to terminate the FileDropClass object.

See Also: BrowseClass.Kill, FileDropClass.Kill

### **PrimaryBufferRestored(confirm restore of primary buffer)**

#### **PrimaryBufferRestored()**

The **PrimaryBufferRestored** method notifies the Window component that the buffer has been restored successfully. This allows synchronization of the restored buffer with other Window Manager threads.

#### **Implementation:**

The **PrimaryBufferRestored** method is called after the SELF.Primary.Me.RestoreBuffer() method to notify the Window Component that the restore is completed.

See Also: PrimaryBufferSaved  
PrimaryBufferRestoreRequired

### **PrimaryBufferRestoreRequired(flag restore of primary buffer)**

#### **PrimaryBufferRestoreRequired(),BYTE**

The **PrimaryBufferRestoreRequired** method returns TRUE (1) if the buffer pointer of the Window Component has been changed. This allows synchronization of the saved buffer with other Window Manager threads.

#### **Implementation:**

The **PrimaryBufferRestoreRequired** method calls the SELF.Primary.Me.RestoreBuffer method to restore the primary buffer in the WindowManager's ResetBuffers method.

See Also: PrimaryBufferSaveRequired  
PrimaryBufferRestored

## **PrimaryBufferSaved(confirm save of primary buffer)**

### **PrimaryBufferSaved**

The **PrimaryBufferSaved** method notifies the Window component that the buffer has been saved successfully. This allows synchronization of the saved buffer with other Window Manager threads.

#### **Implementation:**

The **PrimaryBufferSaved** method is called after the SELF.Primary.Me.SaveBuffer() method to notify the Window Component that the save is completed.

See Also: PrimaryBufferRestored  
PrimaryBufferSaveRequired

## **PrimaryBufferSaveRequired(flag save of primary buffer)**

### **PrimaryBufferSaveRequired(),BYTE**

The **PrimaryBufferSaveRequired** method returns TRUE (1) if the buffer pointer of the Window Component has been deleted. This allows synchronization of the saved buffer with other Window Manager threads.

#### **Implementation:**

The **PrimaryBufferSaveRequired** method calls the SELF.Primary.Me.SaveBuffer() method to save the primary buffer in the WindowManager's ResetBuffers method.

See Also: PrimaryBufferSaved  
PrimaryBufferRestoreRequired

**Reset(reset object's data)****Reset(*forcereset*)**

<b>Reset</b>	Resets the object's data.
<i>forcereset</i>	A numeric constant, variable, EQUATE, or expression that indicates whether to reset the object's data. A value of one (1 or True) unconditionally resets the object's data; a value of zero (0 or False) only resets the objects data as circumstances require

The **Reset** method resets the object's data if needed or if *forcereset* is TRUE.

**BrowseClass Implementation:**

The Reset method calls the BrowseClass.ResetSort method to reapply the active sort order to the BrowseClass object. For more information see the BrowseClass.ResetSort section.

**FileDropClass Implementation:**

The Reset method calls the FileDropClass.ResetQueue method to fill or refill the File Drop control template's display queue.

See Also: BrowseClass.ResetSort, FileDropClass.ResetQueue

**ResetRequired(determine if screen refresh needed)****ResetRequired**

The **ResetRequired** method determines whether the objects data needs to be refreshed. A TRUE return value indicates a refresh occurred and a screen redraw is necessary.

**BrowseClass Implementation:**

The ResetRequired method calls the BrowseClass.ApplyRange method to apply the reset fields and range limits and refresh the Browse Box control list if necessary. A TRUE return value indicates a screen redraw is needed.

**FileDropClass Implementation:**

The ResetRequired method calls the FileDropClass's ApplyRange method inherited from the ViewManager. This method applies the range limits and filters. A TRUE return value indicates a screen redraw is needed.

**Return Data Type:** BYTE

**See Also:** BrowseClass.ApplyRange, ViewManager.FileDropClass.ApplyRange

## **SetAlerts(alert keystrokes for window component)**

### **SetAlerts**

The **SetAlerts** method alerts standard keystrokes for the control associated with the window component's object.

#### **BrowseClass Implementation:**

The SetAlerts method alerts standard keystrokes for the Browse Box control and for any associated locator controls.

#### **FileDropClass Implementation:**

The SetAlerts method is not implemented for the FileDropClass window component.

## **TakeEvent(process the current ACCEPT loop event)**

### **TakeEvent**

The **TakeEvent** method processes the current ACCEPT loop event.

#### **BrowseClass Implementation:**

The TakeEvent method calls the BrowseClass.TakeEvent method to process the current ACCEPT loop event for the BrowseClass object. The method returns a Level:Benign value. For more information see the BrowseClass.TakeEvent section.

#### **FileDropClass Implementation:**

The TakeEvent method processes the current ACCEPT loop event for the FileDropClass object. The method returns a Level:Benign value. For more information see the FileDropClass.TakeEvent section.

**Return Data Type:** BYTE

**See Also:** BrowseClass.TakeEvent, FileDropClass.TakeEvent

## **Update(get VIEW data for the selected item)**

### **Update**

The **Update** method regets the selected item from the VIEW in order to update the record on disk.

#### **BrowseClass Implementation:**

The Update method calls the BrowseClass.UpdateViewRecord method to reread the selected record from the VIEW. For more information see the BrowseClass.UpdateViewRecord.

#### **FileDropClass Implementation:**

The Update method is not implemented for the FileDropClass window component.

See Also: BrowseClass.UpdateViewRecord

## **UpdateWindow(update window controls)**

### **UpdateWindow**

The **UpdateWindow** method updates the controls on the window based upon certain conditions set by the WindowComponent object.

#### **BrowseClass Implementation:**

The UpdateWindow method calls the BrowseClass.UpdateWindow method to refresh the window controls based upon determined conditions from the BrowseBox. For more information see the BrowseClass.UpdateViewRecord.

#### **FileDropClass Implementation:**

The UpdateWindow method is not implemented for the FileDropClass window component.

See Also: BrowseClass.UpdateWindow

# WindowResizeClass

## WindowResizeClass Overview

The WindowResizeClass lets the end user resize windows that have traditionally been fixed in size due to the controls they contain (List boxes, entry controls, buttons, nested controls, etc.). The WindowResizeClass *intelligently* repositions the controls, resizes the controls, or both, when the end user resizes the window.

## WindowResizeClass Concepts

The intelligent repositioning is accomplished by recognizing there are many different types of controls that each have unique repositioning *and* resizing requirements. The WindowResizeClass also recognizes that controls are often nested, and considers whether a given control's coordinates are more closely related to the window's coordinates or to another control's coordinates. That is, intelligent repositioning correctly identifies each control's parent. See SetParentControl for more information on the parent concept.

The intelligent repositioning includes several overall strategies that apply to all window controls, as well as custom per-control strategies for resizing and repositioning individual controls. The overall strategies include:

Surface	Makes the most of the available pixels by positioning other controls to maximize the size of LIST, SHEET, PANEL, and IMAGE controls. We recommend this strategy for template generated windows.
Spread	Maintains the design-time look and feel of the window by applying a strategy specific to each control type. For example, BUTTON sizes are not changed but their positions are tied to the nearest window edge. In contrast, LIST sizes <i>and</i> positions are scaled in proportion to the window.
Resize	Rescales all controls in proportion to the window.

See SetStrategy for more information on resizing strategies for individual controls.

**Note:** To allow window resizing you must set the WINDOW's frame type to Resizable. We also recommend adding the MAX attribute. See The Window Formatter--The Window Properties Dialog in the *User's Guide* for more information on these settings.

## WindowResizeClass Relationship to Other Application Builder Classes

The WindowResizeClass is independent of the other Application Builder Classes. It does not rely on other ABC classes, nor do other ABC classes rely on it.

## WindowResizeClass ABC Template Implementation

The ABC Templates instantiate a WindowResizeClass object for each WindowResize template in the application, typically one for each procedure that manages a window. The templates may also derive a class from the WindowResizeClass. The derived class (and its object) is called Resizer. The ABC Templates provide the derived class so you can use the WindowResize template **Classes** tab to easily modify the Resizer's behavior on an instance-by-instance basis.

The object instantiated from the derived class is called Resizer. This object supports all the functionality specified in the WindowResize template. See Other Templates--Window Resize for more information on the template implementation of this class.

## WindowResizeClass Source Files

The WindowResizeClass source code is installed by default to the Clarion \LIBSRC folder. The WindowResizeClass source code and its respective components are contained in:

ABRESIZE.INC	WindowResizeClass declarations
ABRESIZE.CLW	WindowResizeClass method definitions

## WindowResizeClass Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a WindowResizeClass object. This example illustrates the Surface strategy plus some custom strategies for specific controls. The program does nothing except present a window with a typical variety of controls.

```

PROGRAM
  INCLUDE('ABRESIZE.INC')           !declare WindowResizeClass
  MAP
  END
Resizer  WindowResizeClass         !declare Resizer object

ClientQ  QUEUE,PRE(CLI)            !declare LIST QUEUE
Name     STRING(20)
State    STRING(2)
        END
!WINDOW needs IMM & RESIZE
window  WINDOW('Client Information'),AT(, ,185,100),IMM,GRAY,MAX,RESIZE
        SHEET,AT(3,3,180,78),USE(?Sheet1)
        TAB('Client List'),USE(?ListTab)
        LIST,AT(10,20,165,55),USE(?List1),FROM(ClientQ),|
        FORMAT('87L~Name~@s20@8L~State Code~@s2@')
        END
        TAB('Client Logo'),USE(?LogoTab)
        IMAGE('SV.gif'),AT(50,35),USE(?CLI:Logo)
        END
        END
        PROMPT('Locate:'),AT(7,87),USE(?LocatorPrompt)
        ENTRY(@s20),AT(33,86,61,12),USE(CLI:Name)
        BUTTON('Restore'),AT(110,84),USE(?Restore)
        BUTTON('Close'),AT(150,84),USE(?Close)
        END
CODE
OPEN(window)
window{PROP:MinWidth}=window{PROP:Width}      !set window's minimum width
window{PROP:MinHeight}=window{PROP:Height}    !set window's minimum height
Resizer.Init(AppStrategy:Surface)             !initialize Resizer object
Resizer.SetStrategy(?LocatorPrompt, |        !set control specific strategy:
  Resize:FixLeft+Resize:FixBottom,Resize:LockSize) ! at bottom left & fixed size
Resizer.SetStrategy(?CLI:Name, |            !set control specific strategy:
  Resize:FixLeft+Resize:FixBottom,Resize:LockHeight)! at bottom left & fixed height
ACCEPT
CASE EVENT()
OF EVENT:CloseWindow                         !on close window,
  Resizer.Kill                               ! shut down Resizer object
OF EVENT:Sized                               !on sized window,

```

---

```
    Resizer.Resize                ! resize & reposition controls
END                               ! applying above strategies
CASE ACCEPTED()
OF ?Restore
    Resizer.RestoreWindow        !restore window to initial size
OF ?Close
    POST(Event:CloseWindow)
END
END
```

## WindowResizeClass Properties

The WindowResizeClass contains the following properties.

### AutoTransparent (optimize redraw)

**AutoTransparent**      **BYTE**

The **AutoTransparent** property indicates whether controls that support it are made transparent (TRN attribute) during the resize process. Transparent controls result in less flicker and shadow and smoother resizing, and avoids a Windows bug on some windows.

A value of one (1) makes controls transparent; a value of zero (0) does not.

### DeferMoves (optimize resize)

**DeferMoves**      **BYTE**

The **DeferMoves** property indicates whether to defer control movement until the end of the ACCEPT loop (see *PROP:DeferMove* in the *Language Reference*). This lets the runtime library perform all control movement at once, resulting in a cleaner, "snappier" resize, and avoids a Windows bug on some windows.

A value of one (1) defers control movement; a value of zero (0) does not.

## WindowResizeClass Methods

The WindowResizeClass contains the methods listed below.

### **WindowResizeClass Functional Organization--Expected Use**

As an aid to understanding the WindowResizeClass, it is useful to organize the various WindowResizeClass methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the WindowResizeClass methods.

#### **Non-Virtual Methods**

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Init	initialize the WindowResizeClass object
Kill	shut down the WindowResizeClass object

##### **Mainstream Use:**

Resize <sub>v</sub>	resize and reposition all controls
---------------------	------------------------------------

##### **Occasional Use:**

SetParentControl	set control's parent
SetStrategy	set control's resize strategy

<sub>v</sub> These methods are also Virtual.

#### **Virtual Methods**

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

SetParentDefaults	set all controls' parents
RestoreWindow	restore window to initial size
GetParentControl	return control's parent
Resize	resize and reposition all controls

**GetParentControl (return parent control)****GetParentControl( *control* ), VIRTUAL**


---

**GetParentControl** Returns the parent for a window *control*.

*control* An integer constant, variable, EQUATE, or expression containing a control number. The Resize method rescales the *control* based on the coordinates of the parent.

The **GetParentControl** method returns the parent for a window *control*. A return value of zero indicates the WINDOW is the parent. Otherwise, the return value is the field equate of another window control.

The SetParentDefaults method intelligently sets the appropriate parent for all the window controls, and the SetParentControl method sets the parent for a single control. The Resize method rescales the *control* based on the coordinates of the parent.

Return Data Type: SIGNED

Example:

```

window WINDOW( 'Nested Controls' ), AT( , , 165, 97 ), IMM, GRAY, MAX, RESIZE
GROUP( 'OuterGroup' ), AT( 5, 3, 154, 92 ), USE( ?OuterGroup ), BOXED
BUTTON( 'Button 1' ), AT( 14, 23 ), USE( ?Button1 )
ENTRY( @s20 ), AT( 60, 24 ), USE( Entry1 )
GROUP( 'InnerGroup' ), AT( 11, 49, 141, 38 ), USE( ?InnerGroup ), BOXED
CHECK( 'Check 1' ), AT( 32, 64 ), USE( Check1 )
CHECK( 'Check 2' ), AT( 91, 64 ), USE( Check2 )
. . .
CODE
OPEN( window )
Resizer. Init( AppStrategy: Spread )           !initialize Resizer object
Resizer. SetParentDefaults                     !set parents for all controls
Resizer. SetParentControl( ?Button1, ?OuterGroup ) !override parent for a control
Resizer. SetParentControl( ?Check1, ?InnerGroup ) !override parent for a control
Resizer. SetParentControl( ?Check2, ?InnerGroup ) !override parent for a control

```

See Also: [Resize](#), [SetParentControl](#), [SetParentDefaults](#)

**GetPositionStrategy (return position strategy for a control type)**

**GetPositionStrategy**( *control type* [, *strategy* ] )

**GetPositionStrategy**

Returns the repositioning strategy for a *control type*.

*control type* An integer constant, variable, EQUATE, or expression indicating the type of control (BUTTON, ENTRY, LIST, etc.).

*strategy* An integer constant, variable, EQUATE, or expression indicating the overall strategy for resizing and repositioning all the controls on the window. If omitted, *strategy* defaults to the strategy specified by the Init method.

The **GetPositionStrategy** method returns the appropriate repositioning strategy for a particular *control type* based on the overall *strategy*.

Implementation: The Reset method calls the GetPositionStrategy method to set the position strategy for dynamically created controls.

EQUATES for the *control type* parameter are declared in EQUATES.CLW. Each control type EQUATE is prefixed with CREATE:.

EQUATES for the return value are declared in ABRESIZE.INC. Each strategy EQUATE is prefixed with Resize:.

Example:

```
GET( SELF.ControlQueue, SELF.ControlQueue.ID)      !get control resize info
IF ERRORCODE()                                     !if no control info, add it
  SELF.ControlQueue.Type=FieldCounter{PROP:Type} ! set control type
  SELF.ControlQueue.ParentID=0                     ! set parent
  SELF.ControlQueue.HasChildren=False              ! set children
  SELF.ControlQueue.ID=FieldCounter                ! set ID
  GetSizeInfo(FieldCounter, SELF.ControlQueue.Pos)! set coordinates
                                                    ! set resize strategies
  SELF.ControlQueue.PositionalStrategy=SELF.GetPositionStrategy(SELF.ControlQueue.Type)
  SELF.ControlQueue.ResizeStrategy=SELF.GetResizeStrategy(SELF.ControlQueue.Type)
  ADD( SELF.ControlQueue, SELF.ControlQueue.ID)    ! add control info
  ASSERT( ~ERRORCODE() )
END
```

See Also: Init, Reset

**GetResizeStrategy (return resize strategy for a control type)**

**GetResizeStrategy**( *control type* [, *strategy* ] )

**GetResizeStrategy**

Returns the resizing strategy for a *control type*.

*control type* An integer constant, variable, EQUATE, or expression indicating the type of control (BUTTON, ENTRY, LIST, etc.).

*strategy* An integer constant, variable, EQUATE, or expression indicating the overall strategy for resizing and repositioning all the controls on the window. If omitted, *strategy* defaults to the strategy specified by the Init method.

The **GetResizeStrategy** method returns the appropriate resizing strategy for a particular *control type* based on the overall *strategy*.

Implementation: The Reset method calls the GetResizeStrategy method to set the resizing strategy for dynamically created controls.

EQUATES for the *control type* parameter are declared in EQUATES.CLW. Each control type EQUATE is prefixed with CREATE:.

EQUATES for the return value are declared in ABRESIZE.INC. Each strategy EQUATE is prefixed with Resize:.

Return Data Type: USHORT

Example:

```
GET( SELF.ControlQueue, SELF.ControlQueue.ID)      !get control resize info
IF ERRORCODE()                                     !if no control info, add it
  SELF.ControlQueue.Type=FieldCounter{PROP:Type}! set control type
  SELF.ControlQueue.ParentID=0                     ! set parent
  SELF.ControlQueue.HasChildren=False              ! set children
  SELF.ControlQueue.ID=FieldCounter                ! set ID
  GetSizeInfo(FieldCounter, SELF.ControlQueue.Pos)! set coordinates
  ! set resize strategies
  SELF.ControlQueue.PositionalStrategy=SELF.GetPositionStrategy( SELF.ControlQueue.Type)
  SELF.ControlQueue.ResizeStrategy=SELF.GetResizeStrategy( SELF.ControlQueue.Type)
  ADD( SELF.ControlQueue, SELF.ControlQueue.ID)    ! add control info
  ASSERT( ~ERRORCODE() )
END
```

See Also: Init, Reset

**Init (initialize the WindowResizeClass object)**

**Init**( [*strategy*] [,*minimum size*] [,*maximum size*] )

---

<b>Init</b>	Initializes the WindowResizeClass object.
<i>strategy</i>	An integer constant, variable, EQUATE, or expression indicating the overall strategy for resizing and repositioning all the controls on the window. If omitted, <i>strategy</i> defaults to <code>AppStrategy:Resize</code> , which rescales all controls in proportion to the parent.
<i>minimum size</i>	An integer constant, variable, EQUATE, or expression indicating the minimum size of the window. A value of one (1) sets the minimum window size to its design size. If omitted, <i>minimum size</i> defaults to zero (0), which indicates no minimum.
<i>maximum size</i>	An integer constant, variable, EQUATE, or expression indicating the minimum size of the window. A value of one (1) sets the maximum window size to its design size. If omitted, <i>maximum size</i> defaults to zero (0), which indicates no maximum.

The **Init** method initializes the WindowResizeClass object and sets the overall strategy for resizing and repositioning window controls. You can use the SetStrategy method to override the overall strategy for individual controls.

Implementation: The Init method adds the IMM attribute to the WINDOW.

If the *strategy* parameter is present, Init applies a strategy to each control based on the parameter value. If the *strategy* parameter is absent, Init applies the default strategy to each control. The default *strategy* is to rescale all control coordinates (x, y, width, and height) proportionally with the parent.

The parent may be the WINDOW containing the control, or it may be another control on the WINDOW. The SetParentControl and SetParentDefaults methods determine the parent for a given control.

The *strategy* parameter EQUATEs are declared in RESIZE.INC as follows:

```

ITEMIZE(0),PRE(AppStrategy)
Resize EQUATE !Rescale all controls proportionally
Spread EQUATE !Preserve design time look & feel
Surface EQUATE !Maximize available pixels
END

```

The purpose and effect of these strategies are:

Resize	Scales all window coordinates by the same amount as the parent, thus preserving the relative sizes and positions of all controls. This is the default strategy.
Surface	Makes the most of the available pixels by positioning other controls to maximize the size of LIST, SHEET, PANEL, and IMAGE controls.
Spread	Preserves the design-time look and feel of the window by applying the following strategies by control type:
BUTTON	Horizontal and Vertical position (X and Y coordinates) are "fixed" relative to the nearest parent border; width and height are unchanged.
RADIO	Horizontal and vertical position are scaled with the parent, but width and height are unchanged.
CHECK	Horizontal and vertical position are scaled with the parent, but width and height are unchanged.
ENTRY	Width, horizontal and vertical position are scaled with the parent, but height is unchanged.
COMBO+DROP	Width, horizontal and vertical position are scaled with the parent, but height is unchanged.
LIST+DROP	Width, horizontal and vertical position are scaled with the parent, but height is unchanged.
SPIN	Width, horizontal and vertical position are scaled with the parent, but height is unchanged.
Other	All coordinates are scaled with the parent.

**Tip:** Even though LIST and COMBO controls may be resized, the column widths within them are not resized. However, the right-most column does expand or contract depending on the available space.

Example:

```
OPEN(window)
Resizer.Init(AppStrategy:Surface)!initialize Resizer object
ACCEPT
CASE EVENT()
  OF EVENT:CloseWindow           !on close window,
    Resizer.Kill                 ! shut down Resizer object
  OF EVENT:Sized                 !on sized window,
    Resizer.Resize              ! resize & reposition controls
END
END
```

See Also:      SetParentControl, SetParentDefaults, SetStrategy

## Kill (shut down the WindowResizeClass object)

### Kill

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code.

Example:

```
OPEN(window)
Resizer.Init(AppStrategy:Surface)      !initialize Resizer object
ACCEPT
CASE EVENT()
OF EVENT:CloseWindow                  !on close window,
  Resizer.Kill                         ! shut down Resizer object
OF EVENT:Sized                         !on sized window,
  Resizer.Resize                       ! resize & reposition controls
END
END
```

## Reset (resets the WindowResizeClass object)

### Reset, VIRTUAL

The **Reset** method resets the WindowResizeClass object to conform to the window in its present state.

Implementation: The Init method calls the Reset method. The Reset method stores the initial coordinates for the window and its controls. The WindowResizeClass object uses the stored coordinates to restore the window, establish parent-child relationships between controls, etc.

Example:

```
ThisWindow.Init PROCEDURE()  
ReturnValue      BYTE,AUTO  
CODE  
!procedure code  
Resizer.Init(AppStrategy:Surface,Resize:SetMinSize)  
SELF.AddItem(Resizer)  
Resizer.AutoTransparent=True  
Resizer.SetParentDefaults  
INIMgr.Fetch('BrowseMembers',QuickWindow)  
Resizer.Resize      !Resize needed if window altered by INIMgr  
Resizer.Reset       !Reset needed if window altered by INIMgr  
SELF.SetAlerts()  
RETURN ReturnValue
```

See Also:     Init

**Resize (resize and reposition controls)****Resize, VIRTUAL, PROC**

The **Resize** method resizes and repositions each window control by applying the specified strategy to each control, and returns a value indicating whether ACCEPT loop processing is complete and should stop.

Resize returns Level:Benign to indicate processing of the event (typically EVENT:Sized) should continue normally; it returns Level:Notify to indicate processing is completed for the event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

The Init method and the SetStrategy method determine the strategies to apply to each control. All resizing strategies consider the new coordinates of the each control's "parent." By default, the WINDOW is the parent of each control. However, you may designate any control as the parent of any other control with the SetParentControl method.

Return Data Type: BYTE

Example:

```

OPEN(window)
Resizer.Init(AppStrategy:Surface)           !init Resizer-general strategy
Resizer.SetStrategy(?CloseButton, |       !set control specific strategy:
  Resize:FixRight+Resize:FixBottom,Resize:LockSize) ! at bottom right & fixed
size
ACCEPT
CASE EVENT()
OF EVENT:CloseWindow           !on close window,
  Resizer.Kill                 ! shut down Resizer object
OF EVENT:Sized                 !on sized window,
  Resizer.Resize               ! resize & reposition controls
END
END

```

See Also: Init, SetStrategy, SetParentControl

## RestoreWindow (restore window to initial size)

### RestoreWindow, VIRTUAL

The **RestoreWindow** method restores the window and all its controls to their sizes in effect when the Init method executed.

Example:

```
OPEN(window)
Resizer.Init(AppStrategy:Surface)      !init Resizer overall strategy
ACCEPT
CASE EVENT()
OF EVENT:CloseWindow
  Resizer.Kill                          ! shut down Resizer object
OF EVENT:Sized
  Resizer.Resize                        ! resize & reposition controls
END
CASE ACCEPTED()
OF ?RestoreButton
  Resizer.RestoreWindow                 !restore window to original spec
END
END
```

See Also:        Init

**SetParentControl (set parent control)****SetParentControl**( *control* [,*parent*] )**SetParentControl** Sets the *parent* for a window *control*.

*control* An integer constant, variable, EQUATE, or expression containing a control number. The Resize method rescales the *control* based on the coordinates of the *parent*.

*parent* An integer constant, variable, EQUATE, or expression containing a control number. The Resize method rescales the *control* based on the coordinates of the *parent*. If omitted, *parent* defaults to the WINDOW.

The **SetParentControl** method sets the *parent* for a window *control*. The Resize method rescales the *control* based on the coordinates of the *parent*.

This lets you rescale a particular control based upon a related control's coordinates rather than on the window's coordinates. This is appropriate when the strategy applied to the parent control causes it to be scaled disproportionately from the window. For example, controls within a GROUP structure whose size is "locked" may be rescaled to fit the GROUP's coordinates rather than the window's coordinates.

The SetParentDefaults method intelligently sets the appropriate parent for each window control so you only need to use SetParentControl if SetParentDefaults sets an inappropriate parent. The GetParentControl method returns the parent control number for a control.

Example:

```

window WINDOW('Nested Controls'),AT(,,165,97),IMM,GRAY,MAX,RESIZE
  GROUP('OuterGroup'),AT(5,3,154,92),USE(?OuterGroup),BOXED
  BUTTON('Button 1'),AT(14,23),USE(?Button1)
  ENTRY(@s20),AT(60,24),USE(Entry1)
  GROUP('InnerGroup'),AT(11,49,141,38),USE(?InnerGroup),BOXED
  CHECK('Check 1'),AT(32,64),USE(Check1)
  CHECK('Check 2'),AT(91,64),USE(Check2)
END
END
CODE
OPEN(window)
Resizer.Init(AppStrategy:Spread)           !initialize Resizer object
Resizer.SetParentDefaults                   !set parents for all controls
Resizer.SetParentControl(?Button1,?OuterGroup) !override parent for a control
Resizer.SetParentControl(?Check1,?InnerGroup) !override parent for a control
Resizer.SetParentControl(?Check2,?InnerGroup) !override parent for a control

```

See Also:      GetParentControl, Resize, SetParentDefaults

## SetParentDefaults (set default parent controls)

### SetParentDefaults, VIRTUAL

The **SetParentDefaults** method intelligently sets the appropriate parent for each window control. The **Resize** method rescales each control based on the coordinates of its parent.

This lets you rescale a particular control based upon a related control's coordinates rather than on the window's coordinates. This is appropriate when the strategy applied to the parent control causes it to be scaled disproportionately from the window. For example, controls within a **GROUP** structure whose size is "locked" may be rescaled to fit the **GROUP**'s coordinates rather than the window's coordinates.

You may use the **SetParentControl** method to set the parent for a single control.

Implementation: The **SetParentDefaults** method considers each control's coordinates. If the control's coordinates fall within the coordinates of another control, the **SetParentDefaults** method sets the "outer" control as the parent of the "inner" control.

The **Init** method calls the **SetParentDefaults** method when the resize strategy is **AppStrategy:Surface**.

Example:

```

window WINDOW('Nested Controls'),AT(,,165,97),IMM,GRAY,MAX,RESIZE
  GROUP('OuterGroup'),AT(5,3,154,92),USE(?OuterGroup),BOXED
  BUTTON('Button 1'),AT(14,23),USE(?Button1)
  ENTRY(@s20),AT(60,24),USE(Entry1)
  GROUP('InnerGroup'),AT(11,49,141,38),USE(?InnerGroup),BOXED
  CHECK('Check 1'),AT(32,64),USE(Check1)
  CHECK('Check 2'),AT(91,64),USE(Check2)
  END
  END
END
CODE
OPEN(window)
Resizer.Init(AppStrategy:Spread)           !initialize Resizer object
Resizer.SetParentDefaults                  !set parents for all controls
Resizer.SetParentControl(?Button1,?OuterGroup) !override parent for a control
Resizer.SetParentControl(?Check1,?InnerGroup) !override parent for a control
Resizer.SetParentControl(?Check2,?InnerGroup) !override parent for a control

```

See Also: [Resize](#), [SetParentControl](#)

**SetStrategy (set control resize strategy)**

**SetStrategy**( [[*control*] ,*position strategy*, *size strategy*] )

| *source control*, *target control* |

---

<b>SetStrategy</b>	Sets the <i>position strategy</i> and the <i>size strategy</i> to apply to a control.
<i>control</i>	An integer constant, variable, EQUATE, or expression containing a control number. If omitted, the SetStrategy method applies <i>position strategy</i> and <i>size strategy</i> to all controls on the WINDOW.
<i>position strategy</i>	An integer constant, variable, EQUATE, or expression indicating the position strategy to apply to the <i>control</i> .
<i>size strategy</i>	An integer constant, variable, EQUATE, or expression indicating the size strategy to apply to the <i>control</i> .
<i>source control</i>	An integer constant, variable, EQUATE, or expression identifying the control whose <i>position strategy</i> and <i>size strategy</i> are applied to the <i>target control</i> .
<i>target control</i>	An integer constant, variable, EQUATE, or expression identifying the control whose <i>position strategy</i> and <i>size strategy</i> are copied from the <i>source control</i> .

The **SetStrategy** method sets the *position strategy* and the *size strategy* to apply to a window *control* or controls. The Resize method applies the specified strategies.

Implementation: EQUATEs for the *position strategy* and the *size strategy* parameters are declared in ABRESIZE.INC as follows. To apply two or more strategies, simply add them together.

```

!Resize strategies
Resize:Resize           EQUATE(0000b) !rescale height & width
Resize:LockWidth        EQUATE(0001b) !locks width
Resize:LockHeight       EQUATE(0010b) !locks height
Resize:LockSize         EQUATE(0011b) !locks height & width
Resize:ConstantRight    EQUATE(0100b) !locks right edge, moves left
Resize:ConstantBottom   EQUATE(1000b) !locks bottom edge, moves top

!Reposition Strategies - Horizontal position
Resize:Reposition       EQUATE(0000h) !rescale X & Y
Resize:LockXPos         EQUATE(0001h) !locks left edge (absolute)
Resize:FixRight         EQUATE(0002h) !fixes right edge (relative)
Resize:FixLeft          EQUATE(0003h) !fixes left edge (relative)
Resize:FixXCenter       EQUATE(0004h) !fixes horizontal center (relative)
Resize:FixNearestX     EQUATE(0005h) !FixRight or FixLeft

```

```
!Reposition Strategies - Vertical position
Resize:LockYPos      EQUATE(0100h) !locks top edge (absolute)
Resize:FixBottom    EQUATE(0200h) !fixes bottom edge (relative)
Resize:FixTop       EQUATE(0300h) !fixes top edge (relative)
Resize:FixYCenter   EQUATE(0400h) !fixes vertical center (relative)
Resize:FixNearestY EQUATE(0500h) !FixTop or FixBottom
```

Example:

```
window WINDOW('Client Information'),AT(,,185,100),IMM,GRAY,MAX,RESIZE
  SHEET,AT(3,3,180,78),USE(?Sheet1)
  TAB('Client List'),USE(?ListTab)
  LIST,AT(10,20,165,55),USE(?List1),FROM(ClientQ),|
  FORMAT('87L~Name~@s20@8L~State Code~@s2@')
  END
  TAB('Client Logo'),USE(?LogoTab)
  IMAGE,AT(10,20,165,55),USE(?CLI:Logo)
  END
  END
  PROMPT('Locate: '),AT(7,87),USE(?LocatorPrompt)
  ENTRY(@s20),AT(33,86,61,12),USE(CLI:Name)
  BUTTON('Close'),AT(150,84),USE(?Close)
  END
CODE
OPEN(window)
Resizer.Init(AppStrategy:Surface)           !init Resizer overall strategy
Resizer.SetStrategy(?LocatorPrompt, |      !set control specific strategy:
Resize:FixLeft+Resize:FixBottom,Resize:LockSize) ! at bottom left & fixed size
Resizer.SetStrategy(?CLI:Name, |          !set control specific strategy:
Resize:FixLeft+Resize:FixBottom,Resize:LockHeight)! at bottom left & fixed height
```

See Also:      Resize

---

# WindowManager

## WindowManager Overview

The WindowManager class declares a Window Manager that provides highly structured, consistent, flexible, and convenient processing for Clarion window procedures. The WindowManager class is actually a window *procedure* manager. This includes almost every template generated procedure, including Process and Report procedures.

## WindowManager Concepts

---

### A Structured Window Procedure Manager

---

The WindowManager object initializes the procedure, runs the procedure by handling all ACCEPT loop events for the WINDOW, then shuts down the procedure. The WindowManager handles events primarily by forwarding the events to other ABC Library objects for processing.

The WindowManager is a fairly generic base class and therefore handles events and processes that are common across most Windows applications. For an example of a process-specific WindowManager implementation, see Print Preview Class and Report Manager Class.

---

### Implements Update Procedure Policy

---

In addition to its function as a general purpose window procedure manager, the WindowManager may be configured to implement a variety of options for update procedures--window procedures that support record inserts, changes, and deletes. The WindowManager carries out the specified options for these update procedures (forms).

---

### Integrated with other ABC Library Objects

---

The WindowManager is closely integrated with several other ABC Library objects; in particular, the BrowseClass, ToolbarClass, FileDropClass, and FileDropComboClass objects. These objects register their presence with each other, set each other's properties, and call each other's methods to accomplish their goals.

These integrated objects could override the WindowManager's methods (such as TakeAccepted) to perform their jobs; however, because the WindowManager is programmed to understand these ABC objects, once they are registered (AddItem), the WindowManager drives them directly according to their documented interfaces.

## Encapsulated Event Processing

---

The WindowManager provides separate virtual methods to group the handling of all ACCEPT loop events into logical, convenient containers (virtual methods), so that, should you need to implement custom (non-default) event handling, you can implement your changes within the relatively small scope of the specific virtual method that implements the default event handling you wish to change. This logical grouping of window event handling is as follows:

TakeEvent	(handle <b>all</b> events)
TakeWindowEvent	(handle <b>all</b> non-field events--do default processing for common non-field e
TakeAccepted	(do default EVENT:Accepted processing)
TakeRejected	(do default EVENT:Rejected processing)
TakeSelected	(do default EVENT:Selected processing)
TakeNewSelection	(do default EVENT:NewSelection processing)
TakeCompleted	(do default EVENT:Completed processing)
TakeCloseEvent	(do default EVENT:Close processing)
TakeFieldEvent	(handle <b>all</b> field events--do custom processing for field events)

## WindowManager ABC Template Implementation

The ABC Templates *derive* a class from the WindowManager class for *each* procedure that drives an interactive window, including Report and Process procedures. The derived class is called ThisWindow, and its methods and behavior can be modified on the Window Behavior Classes tab.

The ABC Templates generate virtual methods as needed to provide procedure specific initialization, event handling, and shut down.

## WindowManager Relationship to Other Application Builder Classes

The WindowManager is closely integrated with several other ABC Library objects--in particular, the BrowseClass, FileDropClass, FileDropComboClass, and ToolbarClass objects. These objects register their presence with the WindowManager, set each other's properties, and call each other's methods as needed to accomplish their respective goals.

The BrowseClass uses the WindowManager to refresh the window as needed. Therefore, if your program instantiates the BrowseClass, it must also instantiate the WindowManager. Much of this is automatic when you INCLUDE the BrowseClass header (ABBROWSE.INC) in your program's data section. See the Conceptual Example and see Browse Class for more information.

The WindowManager serves as the foundation of the PrintPreviewClass and the ReportManager. That is, both the PrintPreviewClass and the ReportManager are derived from the WindowManager, because both derived classes manage a window procedure.

---

**PrintPreviewClass--Print Preview Window Manager**

---

The PrintPreviewClass implements a full featured print preview window. See Print Preview Class for more information.

---

**ReportManager--Progress Window Manager**

---

The ReportManager implements a progress window that monitors and displays the status of a report. See Report Manager Class for more information.

**WindowManager Source Files**

The WindowManager source code is installed by default to the Clarion \LIBSRC folder. The WindowManager source code and its respective components are contained in:

ABWINDOW.INC	WindowManager declarations
ABWINDOW.CLW	WindowManager method definitions

## WindowManager Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a WindowManager and related objects. This example performs repetitive inserts to a Customer file and also adds phone numbers for each customer to a related Phones file. It uses the WindowManager to call a procedure to validate the customer's state code against a States file.

Note that the WindowManager is aware of other ABC objects, such as BrowseClass objects, Toolbar objects, FileDrop objects, etc. This example shows the interaction between the WindowManager object and a FileManager object and a BrowseClass object.

```
AddCustomer  PROGRAM

    INCLUDE('ABWINDOW.INC')           !declare WindowManager
    INCLUDE('ABFILE.INC')             !declare File,View&Relation Mgrs
    INCLUDE('ABBROWSE.INC')          !declare BrowseClass

    MAP

    SelectState PROCEDURE             !procedure to validate State
    END

    GlobalErrors  ErrorClass          !declare GlobalErrors object
    GlobalRequest BYTE(0),THREAD      !inter procedure communication
    GlobalResponse BYTE(0),THREAD    !inter procedure communication
    VCRRequest    LONG(0),THREAD      !inter procedure communication

    Customer      FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,THREAD
    BYNUMBER      KEY(CUS:CUSTNO),NOCASE,OPT,PRIMARY
    Record        RECORD,PRE()
    CUSTNO        LONG
    Name          STRING(30)
    State         STRING(2)
                END
                END

    Phones        FILE,DRIVER('TOPSPEED'),PRE(PH),CREATE,THREAD
    IDKEY         KEY(PH:ID),DUP,NOCASE
    Record        RECORD,PRE()
    ID           LONG
    NUMBER        STRING(20)
                END
                END

    State         FILE,DRIVER('TOPSPEED'),PRE(ST),CREATE,THREAD
    StateCodeKey  KEY(ST:STATECODE),NOCASE,OPT
    Record        RECORD,PRE()
```

```

STATECODE      STRING(2)
STATENAME      STRING(20)
              END
              END
Access:State    CLASS(FileManager)      !declare Access:State object
Init           PROCEDURE
              END
Relate:State    CLASS(RelationManager)   !declare Relate:State object
Init           PROCEDURE
              END
Access:Customer CLASS(FileManager)      !declare Access:Customer object
Init           PROCEDURE
              END
Relate:Customer CLASS(RelationManager)   !declare Relate:Customer object
Init           PROCEDURE
              END
Access:Phones   CLASS(FileManager)      !declare Access:Phones object
Init           PROCEDURE
              END
Relate:Phones   CLASS(RelationManager)   !declare Relate:Phones object
Init           PROCEDURE
              END

PhoneView VIEW(Phones)                  !declare Phones VIEW
              END

PhoneQ  QUEUE                            !declare PhoneQ for browse list
PH:ID   LIKE(PH:ID)
PH:NUMBER LIKE(PH:NUMBER)
ViewPos STRING(512)
              END

CUS:Save LIKE(CUS:RECORD),STATIC        !declare save area for Cus ditto key

CUSWindow WINDOW('Add Customer'),AT(,,146,128),IMM,SYSTEM,GRAY
SHEET,AT(4,4,136,102),USE(?CurrentTab)
  TAB('General'),USE(?GeneralTab)      !General tab
  PROMPT('ID:'),AT(8,35),USE(?CUSTNO:Prompt)
  ENTRY(@n-14),AT(42,35,41,10),USE(CUS:CUSTNO),RIGHT(1)
  PROMPT('Name:'),AT(8,49),USE(?NAME:Prompt)
  ENTRY(@s30),AT(42,49,90,10),USE(CUS:NAME)! Customer Name
  PROMPT('State:'),AT(8,63),USE(?State:Prompt)
  ENTRY(@s2),AT(42,63,40,10),USE(CUS:State)! Customer State
  END
  TAB('Phones'),USE(?PhoneTab)         !Phones tab
  LIST,AT(8,20,128,63),USE(?PhoneList),IMM,HVSCROLL,FROM(PhoneQ),|
  FORMAT(' 38R(2)|M~ID~C(0)@n-14@80L(2)|M~NUMBER~@s20@')
  BUTTON('&Insert'),AT(8,87),USE(?Insert)

```

```

        BUTTON('&Change'),AT(53,87),USE(?Change)
        BUTTON('&Delete'),AT(103,87),USE(?Delete)
    END
END
BUTTON('OK'),AT(68,110),USE(?OK),DEFAULT
BUTTON('Cancel'),AT(105,110),USE(?Cancel)
END

```

```

ThisWindow CLASS(WindowManager)           !declare derived ThisWindow object
Init        PROCEDURE(),BYTE,PROC,VIRTUAL !procedure specific initialization
Kill        PROCEDURE(),BYTE,PROC,VIRTUAL !procedure specific shut down
Run         PROCEDURE(USHORT Number,BYTE Request),BYTE,PROC,VIRTUAL !run a procedure
TakeAccepted PROCEDURE(),BYTE,PROC,VIRTUAL !non-default EVENT:Accepted handling
END

```

```

PhBrowse CLASS(BrowseClass)              !declare PhBrowse object
Q         &PhoneQ                         !which works with ThisWindow object
END

```

## CODE

```

    ThisWindow.Run()                       !run the program / procedure
                                           !(Init, Ask, Kill)
ThisWindow.Init PROCEDURE()               !setup and "program" ThisWindow
ReturnValue     BYTE,AUTO
CODE
GlobalErrors.Init           !initialize GlobalErrors object
Relate:Customer.Init       !initialize Relate:Customer object
Relate:State.Init          !initialize Relate:State object
Relate:Phones.Init         !initialize Relate:Phones object
ReturnValue = PARENT.Init() !call base class WindowManager.Init
Relate:Customer.Open       !open Customer & related files
Relate:State.Open          !open State & related files
                           !Program ThisWindow object:
SELF.Request = InsertRecord ! insert records only
SELF.FirstField = ?CUSTNO:Prompt! CustNo is firstfield for ThisWindow
SELF.VCRRequest &= VCRRequest ! set VCRRequest for ThisWindow
SELF.Errors &= GlobalErrors ! set error handler for ThisWindow
SELF.HistoryKey = 734      ! set ditto key (CTRL')
SELF.AddHistoryFile(CUS:Record,CUS:Save) ! set ditto file
SELF.AddHistoryField(?CUS:CUSTNO,1)      ! set ditto (restorable) field
SELF.AddHistoryField(?CUS:NAME,2)        ! set ditto (restorable) field
SELF.AddHistoryField(?CUS:State,3)       ! set ditto (restorable) field
SELF.AddUpdateFile(Access:Customer)      ! register FileManager with ThisWindow
SELF.Primary &= Relate:Customer          ! register RelationMgr with ThisWindow
SELF.AddItem(?Cancel,RequestCancelled)   ! set action for Cancel button
SELF.InsertAction = Insert:Batch         ! set insert action (repetitive)
SELF.OkControl = ?OK                    ! set OK button

```

```

IF SELF.PrimeUpdate() THEN RETURN Level:Notify. !prepare record for add
OPEN(CUSWindow)                               !open the window
SELF.Opened=True                               ! flag it as open
                                                !Program PhBrowse object, including
                                                ! registering ThisWindow (SELF)
PhBrowse.Init(?PhoneList,PhoneQ.ViewPos,PhoneView,PhoneQ,Relate:Phones,SELF)
PhBrowse.Q &= PhoneQ
PhBrowse.AddSortOrder(,PH:IDKEY)
PhBrowse.AddRange(PH:ID,Relate:Phones,Relate:Customer)
PhBrowse.AddField(PH:ID,PhBrowse.Q.PH:ID)
PhBrowse.AddField(PH:NUMBER,PhBrowse.Q.PH:NUMBER)
PhBrowse.InsertControl=?Insert
PhBrowse.ChangeControl=?Change
PhBrowse.DeleteControl=?Delete
SELF.SetAlerts()                               !alert keys for ThisWindow
RETURN ReturnValue

ThisWindow.Kill PROCEDURE()                    !shut down ThisWindow
ReturnValue    BYTE,AUTO
CODE
ReturnValue = PARENT.Kill()                    !call base class WindowManager.Kill
Relate:Customer.Close                         !close Customer & related files
Relate:State.Close                            !close State & related files
Relate:Customer.Kill                          !shut down Relate:Customer object
Relate:State.Kill                             !shut down Relate:State object
Relate:Phones.Kill                            !shut down Relate:Phones object
GlobalErrors.Kill                             !shut down GlobalErrors object
RETURN ReturnValue

ThisWindow.Run PROCEDURE(USHORT Number, BYTE Request)!call other procedures
ReturnValue    BYTE,AUTO
CODE
GlobalRequest = Request                       !set inter procedure request
EXECUTE Number                                !run specified procedure
    SelectState
END
ReturnValue = GlobalResponse                   !set inter procedure response
RETURN ReturnValue

ThisWindow.TakeAccepted PROCEDURE()            !EVENT:Accepted handling
ReturnValue    BYTE,AUTO
Looped BYTE
CODE
LOOP
    IF Looped THEN RETURN Level:Notify ELSE Looped = 1. !allow CYCLE to work
    ReturnValue = PARENT.TakeAccepted()          !do standard EVENT:Accepted
    CASE ACCEPTED()                             !do special EVENT:Accepted
    OF ?CUS:State                               ! on State field

```

```
ST:STATECODE = CUS:State           ! lookup State code
IF Access:State.Fetch(ST:StateCodeKey) ! if not found
  IF SELF.Run(1,SelectRecord) = RequestCompleted ! let user select one
    CUS:State = ST:STATECODE           ! set selected state
  ELSE                                  !if user didn't select one
    SELECT(?CUS:State)                 !focus on State field
    CYCLE                               !start over
  END
END
ThisWindow.Reset()                  !reset ThisWindow if needed
END
RETURN ReturnValue
END
```

## WindowManager Properties

The WindowManager contains the following properties.

### **AutoRefresh (reset window as needed flag)**

#### **AutoRefresh BYTE**

The **AutoRefresh** property determines whether the WindowManager automatically resets the window and its associated objects whenever it detects a change. The WindowManager checks for changes after it processes each event. A value of one (1 or True) automatically resets the window; a value of zero (0 or False) does not automatically reset the window.

AutoRefresh is particularly useful when resetting a BrowseClass object changes a field which is a range-limit of another BrowseClass object.

Implementation: The Init method sets the AutoRefresh property to one. The TakeEvent method implements the action specified by AutoRefresh by calling the Reset method only if any registered BrowseClass objects have changed.

The AddItem method registers BrowseClass objects with the WindowManager.

See Also: AddItem, Init, Reset

### **AutoToolbar (set toolbar target on new tab selection)**

#### **AutoToolbar BYTE**

The **AutoToolbar** property determines how the WindowManager sets the ToolbarTarget. A value of one (1 or True) uses the ToolbarClass object to set the appropriate ToolbarTarget whenever a new TAB is selected; a value of zero (0 or False) uses the current ToolbarTarget.

Implementation: The Init method sets the AutoToolbar property to True. The TakeNewSelection method implements the action specified by AutoToolbar by calling ToolbarClass.SetTarget if the control selected is a SHEET.

See Also: Init, ToolbarClass.SetTarget, ToolbarTargetClass

## CancelAction (response to cancel request)

### CancelAction BYTE

The **CancelAction** property indicates the WindowManager action to take when the end user "Cancels" the window with changes pending. Valid actions are:

Cancel:Cancel	immediate abandon (no confirmation)
Cancel:Save	immediate save (no confirmation)
Cancel:Save+Cancel:Query	offer to save or abandon
Cancel:Cancel+Cancel:Query	offer to resume editing or abandon

Implementation: The Init method sets the CancelAction property to Cancel:Save + Cancel:Query. The TakeCloseEvent method carries out the action specified by the CancelAction property.

CancelAction EQUATEs are declared in ABWINDOW.INC as follows:

```
ITEMIZE,PRE(Cancel)
Cancel EQUATE(0)
Save EQUATE(1)
Query EQUATE(2)
END
```

See Also: Init, TakeCloseEvent, Request, Response

## **ChangeAction (response to change request)**

### **ChangeAction BYTE**

The **ChangeAction** property whether change is a valid action for an update procedure. A value of one (1 or True) indicates the procedure may change (write) records; a value of zero (0 or False) indicates the procedure may not change records.

Implementation: The Init method sets the ChangeAction property to one (1).

See Also: Init

## **Dead (shut down flag)**

### **Dead BYTE, PROTECTED**

The **Dead** property indicates whether the WindowManager should shut down. The WindowManager uses this property to undertake a normal shut down at the earliest opportunity. A value of one (1 or True) indicates the WindowManager should shut down; a value of zero (0 or False) indicates the WindowManager should continue.

Implementation: The Kill method sets the Dead property to True.

See Also: Kill

## DeleteAction (response to delete request)

### DeleteAction BYTE

The **DeleteAction** property indicates the WindowManager action to take when the end user requests to delete a record. Valid actions are:

Delete:None	delete not allowed
Delete:Warn	confirm delete with message
Delete:Form	confirm delete with update form
Delete:Auto	immediate delete (no confirmation)

Implementation: The Init method sets the DeleteAction property to Delete:Warn. The PrimeUpdate method carries out the action specified by the DeleteAction property.

DeleteAction EQUATEs are declared in ABWINDOW.INC as follows:

```
ITEMIZE,PRE(Delete)
None EQUATE
Warn EQUATE
Form EQUATE
Auto EQUATE
END
```

See Also: Init, TakeCloseEvent, Request, Response

## Errors (ErrorClass object)

### **Errors &ErrorClass**

The **Errors** property is a reference to the ErrorClass object that handles unexpected conditions for the WindowManager. In an ABC Template generated program, the ErrorClass object is called GlobalErrors.

Implementation: The WindowManagerClass does not initialize the Errors property. Your derived Init method should initialize the Errors property. See the Conceptual Example.

## FilesOpened(files opened by procedure)

### **FilesOpened BYTE, PROTECTED**

The **FilesOpened** property is a flag used to keep track of files opened by the WindowManager class.

Implementation: The FilesOpened property is set to TRUE in the Init method if there are files to be opened by the procedure. This property is examined in the Kill method to determine whether files need to be closed by the procedure.

## FirstField (first window control)

### **FirstField SIGNED**

The **FirstField** property contains the control number (field equate) of the window control that initially receives focus when the window displays.

Implementation: The WindowManagerClass does not initialize the FirstField property. Your derived Init method should initialize the FirstField property. See the Conceptual Example.

## **ForcedReset (force reset flag)**

### **ForcedReset BYTE**

The **ForcedReset** property indicates whether the WindowManager should unconditionally reset itself. A value of zero (0 or False) allows a conditional reset (reset only if circumstances demand, for example, when the end user invokes a new BrowseBox sort order or invokes a BrowseBox locator); a value of one (1 or True) forces an unconditional reset.

Implementation: The Reset method carries out the action specified by the ForcedReset property.

See Also: Reset

## **HistoryKey (restore field key)**

### **HistoryKey SIGNED**

The **HistoryKey** property enables "save/restore field history" and sets the keystroke which restores a form field's prior saved value. When the end user presses the specified key, the WindowManager restores the field with focus from the previously processed record.

Implementation: The WindowManagerClass does not initialize the HistoryKey property. Your derived Init method should initialize the HistoryKey property if your window uses a history key. See the *Conceptual Example*.

The AddHistoryFile method names the file and record buffers from which fields are saved and restored. AddHistoryField associates specific fields from the history file with their corresponding WINDOW controls. The SaveHistory method saves a copy of the history fields. The RestoreField method restores the contents of a specific control.

Keystroke EQUATEs are declared in \LIBSRC\KEYCODES.CLW.

See Also: AddHistoryField, AddHistoryFile, RestoreField, SaveHistory

## InsertAction (response to insert request)

**InsertAction** **BYTE**

The **InsertAction** property indicates the WindowManager action to take when the end user "Inserts" a record. Valid actions are:

Insert:None	use the default insert action (Insert:Caller)
Insert:Caller	return to calling procedure
Insert:Batch	immediately allow another insert
Insert:Query	offer to return or do another insert

Implementation: The Init method sets the InsertAction property to Insert:Caller. The TakeCompleted method carries out the action specified by the InsertAction property.

The AddUpdateFile method registers files involved in batch adds.

InsertAction EQUATEs are declared in ABWINDOW.INC as follows:

```
ITEMIZE,PRE(Insert)
None EQUATE
Caller EQUATE
Batch EQUATE
Query EQUATE
END
```

See Also: AddUpdateFile, TakeCompleted, Init, Request, Response

## LastInsertedPosition (hold position of last inserted record)

**LastInsertedPosition** **STRING(1024), PROTECTED**

The **LastInsertedPosition** property contains the position of the last record added.

Implementation: The WindowManagerClass clears the LastInsertedPosition property in the Ask method. The property is updated by the File Manager's Position method after a Insert:Batch update.

## **OKControl (window acceptance control--OK button)**

**OKControl**      **SIGNED**

The **OKControl** property contains the control number (field equate) of the window control that indicates end user acceptance of the window--typically the OK button. The WindowManager uses this property to close the window, or to initiate control and record validation if changes are pending.

Implementation:      The WindowManagerClass does not initialize the OKControl property. Your derived Init method should initialize the OKControl property. See the Conceptual Example.

## **Opened (window opened flag)**

**Opened**                      **BYTE**

The **Opened** property indicates whether the WindowManager's WINDOW has been opened. A value of one (1 or True) indicates the WINDOW is open; a value of zero (0 or False) indicates the WINDOW is not opened. You can use this property to control tasks (such as resizing, or saving and restoring window coordinates) that require the WINDOW to be opened or closed.

Implementation:      The WindowManagerClass does not set the Opened property. Your derived Init method should set it. See the Conceptual Example.

See Also:              Init

## OriginalRequest (original database request)

**OriginalRequest**                      **BYTE**

The **OriginalRequest** property indicates the database action for which the procedure was originally called. The WindowManager uses this property to make appropriate processing decisions with regard to priming records, saving or abandoning changes, etc. Valid requests are:

InsertRecord  
ChangeRecord  
DeleteRecord  
SelectRecord

Implementation:    The Init method sets the OriginalRequest property to equal the Request property. EQUATEs for the OriginalRequest and Request properties are declared in \LIBSRC\TPLEQU.CLW as follows:

```
InsertRecord    EQUATE (1)    !Add a record
ChangeRecord   EQUATE (2)    !Change the current record
DeleteRecord    EQUATE (3)    !Delete the current record
SelectRecord    EQUATE (4)    !Select a record
```

See Also:            Init, Request

## Primary (RelationManager object)

**Primary**                              **&RelationManager**

The **Primary** property is a reference to the RelationManager object for the WindowManager's primary file. The WindowManager uses this property to carry out inserts, changes and deletes.

Implementation:    The WindowManagerClass does not initialize the Primary property. Your derived Init method should initialize the Primary property if the procedure does database updates. See the Conceptual Example.

## Request (database request)

**Request**      **BYTE**

The **Request** property indicates the database action the procedure is handling. The WindowManager uses this property to make appropriate processing decisions with regard to priming records, saving or abandoning changes, etc. Valid requests are:

InsertRecord  
ChangeRecord  
DeleteRecord  
SelectRecord

Implementation:      The WindowManagerClass does not set the Request property. Your derived Init method should immediately set the Request property. The WindowManagerClass.Init method sets the OriginalRequest property equal to the Request property to preserve its initial value. See the Conceptual Example.

EQUATEs for the OriginalRequest and Request properties are declared in \LIBSRC\TPLEQU.CLW as follows:

```
InsertRecord    EQUATE (1)   !   Add a record to table  
ChangeRecord   EQUATE (2)   !   Change the current record  
DeleteRecord   EQUATE (3)   !   Delete the current record  
SelectRecord   EQUATE (4)   !   Select the current record
```

See Also:      Init, OriginalRequest

## **ResetOnGainFocus (gain focus reset flag)**

### **ResetOnGainFocus    BYTE**

The **ResetOnGainFocus** property indicates whether the WindowManager should unconditionally reset itself when the window receives focus. A value of zero (0 or False) allows a conditional reset (reset only if changes demand, for example, when the end user invokes a new BrowseBox sort order or invokes a BrowseBox locator); a value of one (1 or True) forces an unconditional reset (reset regardless of circumstances).

Implementation:    The ResetOnGainFocus property defaults to zero (0). The TakeWindowEvent method carries out the action specified by the ResetOnGainFocus property by optionally setting the ForcedReset property to True when the window loses focus.

See Also:            ForcedReset

## **Resize (WindowResize object)**

### **Resize &WindowResizeClass**

The **Resize** property is a reference to the WindowResizeClass object that handles window resizing events.

## Response (response to database request)

### **Response     BYTE**

The **Response** property indicates the WindowManager's response to the original database request (indicated by the OriginalRequest property). The WindowManager uses this property to make appropriate processing decisions with regard to priming records, saving or abandoning changes, etc.

The SetResponse method sets the value of the Response property and exits the procedure.

Implementation:     EQUATEs for the Response property are declared in \LIBSRC\TPLEQU.CLW as follows:

```
RequestCompleted         EQUATE (1)                     ! Update Completed
RequestCancelled         EQUATE (2)                     ! Update Aborted
```

See Also:            OriginalRequest, SetResponse

## Saved (copy of primary file record buffer)

### **Saved   USHORT, PROTECTED**

The **Saved** property locates a copy of the WindowManager's primary file record buffer. The WindowManager uses this property to detect pending changes to the record, and to restore the record if necessary.

The SetSaved method sets the value of the Saved property.

Implementation:     The WindowManager uses the FileManager.SaveBuffer, FileManager.RestoreBuffer, and FileManager.EqualBuffer methods (through its Primary property) to manipulate the Saved property.

See Also:            FileManager.SaveBuffer, FileManager.RestoreBuffer, FileManager.EqualBuffer

**Translator (TranslatorClass object:WindowManager)****Translator    &TranslatorClass**

The **Translator** property is a reference to the TranslatorClass object for the WindowManager. The WindowManager uses this property to translate window text to the appropriate language.

The AddItem method sets the value of the Translator property.

Implementation:    The WindowManagerClass does not initialize the Translator property. The WindowManager only invokes the Translator if the Translator property is not null. Your derived Init method should initialize the Translator property if translation is needed. See the Conceptual Example.

See Also:            AddItem

**VCRRequest (delayed scroll request)****VCRRequest    &LONG**

The **VCRRequest** property is a reference to a variable identifying a scroll request made simultaneously with a database operation request. The WindowManager uses this property to carry out the scroll request after it completes the database operation.

For example, when the end user changes fields on a form then presses the Insert button, he simultaneously requests to save the changes and to scroll to the next record. The WindowManager completes the change request , and only then does it handle the scroll request.

Implementation:    EQUATEs for the VCRRequest property are declared in \LIBSRC\ABTOOLBA.INC as follows:

```

ITEMIZE,PRE(VCR)
Forward        EQUATE(Toolbar:Down)    !EQUATE(1)
Backward      EQUATE(Toolbar:Up)       !EQUATE(2), etc..
PageForward   EQUATE(Toolbar:PageDown)
PageBackward  EQUATE(Toolbar:PageUp)
First         EQUATE(Toolbar:Top)
Last          EQUATE(Toolbar:Bottom)
Insert        EQUATE(Toolbar:Insert)
None          EQUATE(0)
END

```

## WindowManager Methods

The WindowManager contains the following methods.

### WindowManager Functional Organization--Expected Use

As an aid to understanding the WindowManager, it is useful to organize its various methods into two large categories according to their expected use--the Non-Virtual and the virtual methods. This organization reflects what we believe is typical use of the WindowManager methods.

#### Non-Virtual Methods

The Non-Virtual methods, which you are likely to call fairly routinely from your program, can be further divided into three categories:

##### **Housekeeping (one-time) Use:**

Run <sub>v</sub>	run this procedure
Init <sub>v</sub>	initialize the WindowManager object
AddHistoryField	add restorable control and field
AddHistoryFile	add restorable history file
AddItem	program the WindowManager object
AddUpdateFile	register batch add files
Kill <sub>v</sub>	shut down the WindowManager object

##### **Mainstream Use:**

**None**

##### **Occasional Use:**

Run <sub>v</sub>	run another procedure
SaveHistory	save history fields for later restoration
PostCompleted	a virtual to prime fields

<sub>v</sub> These methods are also Virtual.

---

**Virtual Methods**

---

Typically you will not call these methods directly--the Non-Virtual methods call them. However, we anticipate you will often want to override these methods, and because they are virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init	initialize the WindowManager object
Ask	display window and process its events
Kill	shut down the WindowManager object
Open	a virtual to execute on EVENT:OpenWindow
PrimeFields	a virtual to prime fields
PrimeUpdate	update or prepare for update
Reset	reset the window and registered items
RestoreField	restore field to last saved value
Run	run this procedure or another procedure
SetAlerts	alert window control keystrokes
SetResponse	OK or Cancel the window
TakeAccepted	a virtual to process EVENT:Accepted
TakeCompleted	a virtual to complete an update form
TakeCloseEvent	a virtual to Cancel the window
TakeEvent	a virtual to process all events
TakeFieldEvent	a virtual to process field events
TakeNewSelection	a virtual to process EVENT:NewSelection
TakeRejected	a virtual to process EVENT:Rejected
TakeSelected	a virtual to process EVENT:Selected
TakeWindowEvent	a virtual to process non-field events
Update	Prepare records for writing to disk

**AddHistoryField (add restorable control and field)****AddHistoryField( control, field )**


---

**AddHistoryField** Adds a history field to the WindowManager object.

*control* An integer constant, variable, EQUATE, or expression containing the control number of the control whose contents to restore from the *field*. This is the field equate number of the control.

*field* An integer constant, variable, EQUATE, or expression containing the position of the field within the history file's record layout. The field is identified by its position in the FILE declaration. A value of one (1) indicates the first field, two (2) indicates the second field, etc. See WHAT and WHERE in the *Language Reference* for more information.

The **AddHistoryField** method adds a history field to the WindowManager object. AddHistoryField associates a window control with its corresponding database field or column, so the WindowManager can restore the control's contents when the end user invokes the history key (or FrameBrowseControl ditto button).

Implementation: The AddHistoryFile method names the file and record buffers from which fields are saved and restored. The AddHistoryField method associates specific fields from the history file with their corresponding WINDOW controls. The SaveHistory method saves a copy of the history fields. The RestoreField method restores the contents of a specific control.

Example:

```
ThisWindow.Init PROCEDURE()
  CODE
  !procedure code
  SELF.HistoryKey = CtrlR
  SELF.AddHistoryFile(CLI:Record,History::CLI:Record)
  SELF.AddHistoryField(?CLI:Name,2)
  SELF.AddHistoryField(?CLI:StateCode,3)
```

See Also: AddHistoryFile, HistoryKey, RestoreField, SaveHistory

## AddHistoryFile (add restorable history file)

**AddHistoryFile**( *record buffer*, *save buffer* )

---

**AddHistoryFile** Adds a history file to the WindowManager object.

*record buffer*    The label of the history file's RECORD.

*save buffer*     The label of a STATIC variable declared LIKE(*record buffer*). The WindowManager saves to and restores from this variable.

The **AddHistoryFile** method adds a history file to the WindowManager object. AddHistoryFile sets the file's record buffer and a corresponding save buffer so the WindowManager can restore from the save buffer when the end user invokes the history key (or FrameBrowseControl ditto button).

Implementation:    The AddHistoryFile method names the file and record buffers from which fields are saved and restored. The AddHistoryField method associates specific fields from the history file with their corresponding WINDOW controls. The SaveHistory method saves a copy of the history fields. The RestoreField method restores the contents of a specific control.

Example:

```
ThisWindow.Init PROCEDURE()  
  CODE  
  !procedure code  
  SELF.HistoryKey = CtrlR  
  SELF.AddHistoryFile(CLI:Record,History::CLI:Record)  
  SELF.AddHistoryField(?CLI:Name,2)  
  SELF.AddHistoryField(?CLI:StateCode,3)
```

See Also:            AddHistoryFile, HistoryKey, RestoreField, SaveHistory

**AddItem (program the WindowManager object)**

```

AddItem(| class           | )
           | WindowComponent |
           | control, response |

```

---

**AddItem** Adds specific functionality to the WindowManager.

*class* The label of one of the following objects: BrowseClass, ToolbarClass, ToolbarUpdateClass, TranslatorClass, or WindowResizeClass

*WindowComponent* The label of a WindowComponent interface.

*control* An integer constant, variable, EQUATE, or expression containing the control number of the control whose acceptance invokes the *response*— typically OK and Cancel buttons.

*response* An integer constant, variable, EQUATE, or expression indicating the action to register when the *control* is accepted.

The **AddItem** method registers another ABC Library object with the WindowManager object to add the object's specific functionality to the WindowManager. The AddItem method also registers an interface with the Window Manager's Component List.

Implementation: The TakeAccepted method assigns the *response* value to the Response property when the *control* is accepted. EQUATEs for the *response* parameter are declared in \LIBSRC\TPLEQU.CLW as follows:

```

RequestCompleted EQUATE (1) !Update Completed
RequestCancelled EQUATE (2) !Update Aborted

```

See Also: Response, WindowManager.TakeAccepted

## AddUpdateFile (register batch add files)

### **AddUpdateFile**( *file manager* )

---

**AddUpdateFile** Registers FileManager objects with the WindowManager object.

*file manager* The label of the FileManager object for the file.

The **AddUpdateFile** method registers FileManager objects with the WindowManager object, for files whose record buffers must be saved and restored to support batch (repetitive) adds.

Implementation: The WindowManager uses the update file's FileManager to save and restore the file's buffer.

The InsertAction property specifies batch adds.

Example:

```
ThisWindow.Init PROCEDURE()  
  CODE  
  !procedure code  
  SELF.AddUpdateFile(Access:Client)  
  !procedure code
```

See Also: InsertAction

**Ask (display window and process its events:WindowManager)****Ask, VIRTUAL**

The **Ask** method displays the window and processes its events.

Implementation: The Run method calls the Ask method only if the Init method returns Level:Benign. Ask RETURNS immediately if the Dead property is True. The Kill method sets the Dead property to True, so calling the Kill method before the Ask method has the effect of shutting down the window procedure before Ask displays the WINDOW.

The Ask method implements the ACCEPT loop for the window and calls the TakeEvent method to handle all events. The ACCEPT loop continues until TakeEvent RETURNS Level:Fatal.

**Tip: To shut down the window procedure while the Ask method is running, RETURN Level:Fatal from any of the "Take" methods.**

The ACCEPT loop CYCLES when TakeEvent returns Level:Notify.

**Tip: To immediately stop processing for an event (including stopping resizing and alerted keys), RETURN Level:Notify from any of the "Take" methods.**

Example:

```
WindowManager.Run PROCEDURE
CODE
IF ~SELF.Init()
  SELF.Ask
END
SELF.Kill
```

```
WindowManager.Ask PROCEDURE
CODE
IF SELF.Dead THEN RETURN .
CLEAR(SELF.LastInsertedPosition)
ACCEPT
CASE SELF.TakeEvent()
  OF Level:Fatal
    BREAK
  OF Level:Notify
    CYCLE !Not as dopey at it looks, it is for 'short-stopping' certain events
  END
END
```

See Also: Dead, Init, Kill, Run, TakeEvent

## ChangeRecord(execute change record process)

### ChangeRecord, VIRTUAL

The **ChangeRecord** method performs the necessary database change or update operations when called. **ChangeRecord** returns Level:Benign to indicate successful a change operation.

Implementation: The ChangeRecord method is called by the TakeCompleted method when Request is set to ChangeRecord.

Return Data Type: BYTE

Example:

```
WindowManager.TakeCompleted PROCEDURE
CODE
SELF.SaveHistory()
CASE SELF.Request
OF InsertRecord
RETURN SELF.InsertAction()
OF ChangeRecord
RETURN SELF.ChangeAction()
OF DeleteRecord
RETURN SELF.DeleteAction()
OF SaveRecord
CASE SELF.OriginalRequest
OF InsertRecord
RETURN SELF.SaveOnInsertAction()
OF ChangeRecord
RETURN SELF.SaveOnChangeAction()
END
END
RETURN Level:Benign
```

See Also:

TakeCompleted

Request

InsertRecord

DeleteRecord

## DeleteRecord(execute delete record process)

### DeleteRecord, VIRTUAL

The **DeleteRecord** method performs the necessary database delete operations when called. **DeleteRecord** returns Level:Benign to indicate successful a delete operation.

Implementation: The DeleteRecord method is called by the TakeCompleted method when Request is set to DeleteRecord.

Return Data Type: BYTE

Example:

```
WindowManager.TakeCompleted PROCEDURE
CODE
SELF.SaveHistory()
CASE SELF.Request
OF InsertRecord
RETURN SELF.InsertAction()
OF ChangeRecord
RETURN SELF.ChangeAction()
OF DeleteRecord
RETURN SELF.DeleteAction()
OF SaveRecord
CASE SELF.OriginalRequest
OF InsertRecord
RETURN SELF.SaveOnInsertAction()
OF ChangeRecord
RETURN SELF.SaveOnChangeAction()
END
END
RETURN Level:Benign
```

See Also:

TakeCompleted

Request

ChangeRecord

InsertRecord

## Init (initialize the WindowManager object)

### Init, VIRTUAL, PROC

The **Init** method initializes the WindowManager object. Init returns Level:Benign to indicate normal initialization.

The Init method both "programs" the WindowManager object and initializes the overall procedure.

The WindowManager may be configured to implement a variety of options regarding update windows (forms). You can use the Init method to configure form behavior by setting the Request, InsertAction, ChangeAction, and DeleteAction properties.

The WindowManager is closely integrated with several other ABC Library objects. You can use the Init method to register these other objects with the WindowManager by calling the AddItem method. The objects can then set each other's properties and call each other's methods as needed to accomplish their respective goals.

Implementation: Typically, the Init method is paired with the Kill method, performing the converse of the Kill method tasks.

The Run method calls the Init method.

Return value EQUATEs are declared in ABERROR.INC.

**Tip: To prevent the Ask method from starting, RETURN Level:Notify from the Init method.**

Return Data Type: BYTE

Example:

```
MyWindowManager.Run PROCEDURE
CODE
IF SELF.Init() = Level:Benign
    SELF.Ask
END
SELF.Kill
```

```
ThisWindow.Init PROCEDURE()
CODE
SELF.Request = GlobalRequest
PARENT.Init()
SELF.FirstField = ?Browse:1
SELF.VCRRequest &= VCRRequest
```

```

SELF.Errors &= GlobalErrors
SELF.AddItem(Toolbar)
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
SELF.AddItem(?Close,RequestCancelled)
Relate:Client.Open
FilesOpened = True
OPEN(QuickWindow)
SELF.Opened=True
Resizer.Init(AppStrategy:Surface,Resize:SetMinSize)
SELF.AddItem(Resizer)
Resizer.AutoTransparent=True
BRW1.Init|
(?Browse:1,Queue:Browse:1.Position,BRW1::View:Browse,Queue:Browse:1,Relate:Client,SELF
BRW1.Q &= Queue:Browse:1
BRW1::Sort1:StepClass.Init(+ScrollSort:AllowAlpha,ScrollBy:Runtime)
BRW1.AddSortOrder(BRW1::Sort1:StepClass,CLI:NameKey)
BRW1.AddLocator(BRW1::Sort1:Locator)
BRW1::Sort1:Locator.Init(,CLI:Name,1,BRW1)
BRW1.AddField(CLI:Name,BRW1.Q.CLI:Name)
BRW1.AddField(CLI:StateCode,BRW1.Q.CLI:StateCode)
BRW1.AddField(CLI:ID,BRW1.Q.CLI:ID)
BRW1.InsertControl=?Insert:2
BRW1.ChangeControl=?Change:2
BRW1.DeleteControl=?Delete:2
BRW1.AddToolbarTarget(Toolbar)
BRW1.AskProcedure = 1
SELF.SetAlerts()
RETURN Level:Benign

```

See Also:      AddItem, Ask, Kill, Run

## InsertRecord (execute insert record activity)

### InsertRecord, VIRTUAL

The **InsertRecord** method performs the necessary database insert operations when called. **InsertRecord** returns Level:Benign to indicate successful a insert operation.

Implementation: The TakeCompleted method calls the InsertRecord method when Request is set to InsertRecord.

Return Data Type: BYTE

Example:

```
WindowManager.TakeCompleted PROCEDURE
CODE
SELF.SaveHistory()
CASE SELF.Request
OF InsertRecord
RETURN SELF.InsertAction()
OF ChangeRecord
RETURN SELF.ChangeAction()
OF DeleteRecord
RETURN SELF.DeleteAction()
OF SaveRecord
CASE SELF.OriginalRequest
OF InsertRecord
RETURN SELF.SaveOnInsertAction()
OF ChangeRecord
RETURN SELF.SaveOnChangeAction()
END
END
RETURN Level:Benign
```

See Also:

TakeCompleted

Request

ChangeRecord

DeleteRecord

## Kill (shut down the WindowManager object)

### Kill, VIRTUAL, PROC

The **Kill** method frees any memory allocated during the life of the object and performs any other required termination code. Kill returns a value to indicate the status of the shut down.

Implementation: Kill sets the Dead property to True and returns Level:Benign to indicate a normal shut down. If the Dead property is already set to True, Kill returns Level:Notify to indicate it is taking no additional action.

Typically, the Kill method is paired with the Init method, performing the converse of the Init method tasks.

The Run method calls the Kill method.

Return value EQUATEs are declared in ABERROR.INC.

Return Data Type: BYTE

Example:

```
ThisWindow.Kill PROCEDURE()  
CODE  
IF PARENT.Kill() THEN RETURN Level:Notify.  
IF FilesOpened  
    Relate:Defaults.Close  
END  
IF SELF.Opened  
    INIMgr.Update('Main',AppFrame)  
END  
GlobalResponse = CHOOSE(LocalResponse=0,RequestCancelled,LocalResponse)
```

See Also: Dead, Init, Run

## Open (a virtual to execute on EVENT:OpenWindow--WindowManager)

### Open, VIRTUAL

The **Open** method prepares the window for display. It is designed to execute on window opening events such as EVENT:OpenWindow and EVENT:GainFocus.

Implementation: The Open method invokes the Translator if present and calls the Reset method to reset the WINDOW.

The TakeWindowEvent method calls the Open method.

Example:

```
ThisWindow.TakeWindowEvent  PROCEDURE
CODE
CASE EVENT( )
  OF EVENT:OpenWindow
    IF ~BAND(SELF.Inited,1)
      SELF.Open
    END
  OF EVENT:GainFocus
    IF BAND(SELF.Inited,1)
      SELF.Reset
    ELSE
      SELF.Open
    END
  END
END
RETURN Level:Benign
```

```
ThisWindow.Open  PROCEDURE
CODE
  IF ~SELF.Translator&=NULL
    SELF.Translator.TranslateWindow
  END
  SELF.Reset
  SELF.Inited = BOR(SELF.Inited,1)
```

See Also: Reset, TakeWindowEvent

## PostCompleted (initiates final Window processing)

### PostCompleted

The **PostCompleted** method initiates final or closedown processing for the window. This process is typically initiated with an "OK" button. The actual processing depends on the type of window defined.

Implementation: The TakeAccepted method calls the PostCompleted method. The ToolbarUpdateClass.TakeEvent also calls PostCompleted. The PostCompleted method initiates AcceptAll mode for update Forms (see SELECT in the *Language Reference* for more information) and POSTs an EVENT:Completed for all other windows.

Example:

```
WindowManager.TakeAccepted PROCEDURE
I LONG,AUTO
A SIGNED,AUTO
CODE
A = ACCEPTED()
IF ~SELF.Toolbar &= NULL
  SELF.Toolbar.TakeEvent(SELF.VCRRequest,SELF)
  IF A = Toolbar:History
    SELF.RestoreField(FOCUS())
  END
END
LOOP I = 1 TO RECORDS(SELF.Buttons)
  GET(SELF.Buttons,I)
  IF SELF.Buttons.Control = A
    SELF.SetResponse(SELF.Buttons.Action)
    RETURN Level:Notify
  END
END
IF SELF.OkControl AND SELF.OkControl = A
  SELF.PostCompleted
END
RETURN Level:Benign
```

See Also: OKControl, TakeAccepted

## PrimeFields (a virtual to prime form fields)

### PrimeFields, VIRTUAL

The **PrimeFields** method is a virtual placeholder method to prime fields for adding a record. PrimeFields is called *after* the FileManager.PrimeRecord method to allow update form specific field priming.

Example:

```
ThisWindow.PrimeFields PROCEDURE  
CODE  
CLI:StateCode = 'FL'  
PARENT.PrimeFields
```

## PrimeUpdate (update or prepare for update)

### PrimeUpdate, VIRTUAL, PROC

The **PrimeUpdate** method prepares the record buffer for entering the update form ACCEPT loop. For actions that can be completed without the ACCEPT loop, PrimeUpdate prevents the ACCEPT loop from executing by returning an appropriate value.

PrimeUpdate returns Level:Benign to indicate the record buffer is ready and the update form's ACCEPT loop should execute.

PrimeUpdate returns Level:Fatal to indicate the ACCEPT loop should not execute, either because the record buffer could not be primed, or because PrimeUpdate completed the requested operation and no further action is necessary.

Implementation: The PrimeUpdate method primes the record buffer for inserts, deletes the record for automatic deletes, and saves a copy of the record buffer in all cases.

Return value EQUATEs are declared in ABERROR.INC.

Return Data Type: BYTE

Example:

```
ThisWindow.Init PROCEDURE()  
CODE  
!procedure code  
IF SELF.PrimeUpdate() THEN RETURN Level:Fatal .  
OPEN(ClientFormWindow)  
SELF.SetAlerts()  
RETURN Level:Benign
```

---

**RemoveItem(remove WindowComponent object)****RemoveItem**(*WindowComponent*)

---

**RemoveItem** Unregisters a *WindowComponent* object.

*WindowComponent* A reference to a *WindowComponent* interface.

The **RemoveItem** method removes a *WindowComponent* from the *WindowManager* object. The object was initially registered with the *WindowManager* by the *AddItem* method.

See Also: [WindowComponent Interface](#), [WindowManager](#)

**Reset (reset the window for display)****Reset**( [*force reset* ] ), VIRTUAL**Reset** Resets the WindowManager object.

*force reset* A numeric constant, variable, EQUATE, or expression that indicates whether to conditionally or unconditionally reset the window. A value of one (1 or True) unconditionally resets the window; a value of zero (0 or False) only resets the window if circumstances require, such as a new sort on browse object or a changed reset field on a browse object. If omitted, *force reset* defaults to zero (0).

The **Reset** method resets the WindowManager object and any registered (AddItem) objects. A *force reset* value of one (1 or True) unconditionally resets all the objects and should therefore be used sparingly to enhance performance.

Implementation: The Reset method calls the ResetSort and UpdateWindow methods for each BrowseClass object registered by the AddItem method. It calls the ResetQueue method for each FileDropClass object registered by the AddItem method.

The Open, TakeWindowEvent, and TakeNewSelection methods all call the Reset method.

Example:

```

ThisWindow.TakeWindowEvent PROCEDURE
CODE
CASE EVENT()
OF EVENT:GainFocus
  IF BAND(SELF.Inited,1)
    SELF.Reset
  ELSE
    SELF.Open
  END
OF EVENT:Sized
  IF BAND(SELF.Inited,2)
    SELF.Reset
  ELSE
    SELF.Inited = BOR(SELF.Inited,2)
  END
END
END
RETURN Level:Benign

```

See Also: AutoRefresh, Open, ResetOnGainFocus, TakeNewSelection, TakeWindowEvent, BrowseClass.AddResetField, BrowseClass.ResetSort, BrowseClass.UpdateWindow

## RestoreField (restore field to last saved value)

### RestoreField( *control* ), VIRTUAL

---

**RestoreField** Restores the contents of the specified control.

*control* An integer constant, variable, EQUATE, or expression containing the control number of the control whose contents to restore. This is the field equate number of the control.

The **RestoreField** method restores the contents of the specified control to the value it contained when the record was last saved. The RestoreField only works if the HistoryKey property is set.

Implementation: The AddHistoryFile method names the file and record buffers from which fields are saved and restored. The AddHistoryField method associates specific fields from the history file with their corresponding WINDOW controls. The SaveHistory method saves a copy of the history fields. The RestoreField method restores the contents of a specific control.

Example:

```
WindowManager.TakeAccepted PROCEDURE
A SIGNED,AUTO
CODE
A = ACCEPTED()
IF ~SELF.Toolbar &= NULL
    SELF.Toolbar.TakeEvent(SELF.VCRRequest,SELF)
    IF A = Toolbar:History
        SELF.RestoreField(FOCUS())
    END
END
!procedure code
```

See Also: AddHistoryField, AddHistoryFile, HistoryKey, SaveHistory

**Run (run this procedure or a subordinate procedure)****Run( [ *number, request* ] ), VIRTUAL, PROC**


---

<b>Run</b>	Run this procedure, or run the specified subordinate procedure.
<i>number</i>	An integer constant, variable, EQUATE, or expression identifying the subordinate procedure to run. A value of one (1) runs the first procedure, two (2) runs the second procedure, etc. Typically, this is the procedure's position within an EXECUTE structure. If omitted, Run executes the normal WindowManager Init-Ask-Kill sequence.
<i>request</i>	An integer constant, variable, EQUATE, or expression identifying the action (insert, change, delete, select) the subordinate procedure takes. If omitted, Run executes the normal WindowManager Init-Ask-Kill sequence.

The **Run** method executes the normal WindowManager Init-Ask-Kill sequence, or it runs the specified subordinate procedure on the same thread. Run returns a value indicating whether it completed or cancelled the requested operation.

**Run**  
Executes the normal WindowManager Init-Ask-Kill sequence.

**Run(*number, request*)**  
A virtual placeholder method to execute a procedure identified by *number*. This allows other objects and template generated code to invoke subordinate WindowManager procedures by number rather than by name. The procedure runs on the same thread as the calling procedure.

Return Data Type: **BYTE**

Implementation: Return value EQUATEs are declared in \LIBSRC\TPLEQU.CLW as follows:

```
RequestCompleted EQUATE (1) !Update Completed
RequestCancelled EQUATE (2) !Update Cancelled
```

Example:

```
!procedure data
CODE
ThisWindow.Run                !normal Init-Ask-Kill sequence

ThisWindow.TakeAccepted PROCEDURE()
CODE
!procedure code
IF SELF.Run(1,SelectRecord) = RequestCompleted !run a procedure on this thread
  CLI:StateCode = ST:StateCode
ELSE
  SELECT(?CLI:StateCode)
```

```
    CYCLE
  END

BrowseClass.Ask PROCEDURE(BYTE Request)
  CODE
  !procedure code
  Response=SELF.Window.Run(SELF.AskProcedure,Request) !run a procedure on this thread

ThisWindow.Run PROCEDURE                                !do Init-Ask-Kill sequence
  CODE
  IF SELF.Init() = Level:Benign
    SELF.Ask
  END
  SELF.Kill
  RETURN GlobalResponse

ThisWindow.Run PROCEDURE(USHORT Number,BYTE Request) !run a subordinate procedure
  CODE
  GlobalRequest = Request
  EXECUTE Number
  SelectStates
  UpdatePhones
  END
  RETURN GlobalResponse
```

See Also:      Init, Ask, Kill

## **SaveHistory (save history fields for later restoration)**

### **SaveHistory, PROTECTED**

The **SaveHistory** method saves a copy of the fields named by the AddHistoryField method for later restoration by the RestoreField method.

Implementation: The AddHistoryFile method names the file and record buffers from which fields are saved and restored. The AddHistoryField method associates specific fields from the history file with their corresponding WINDOW controls. The SaveHistory method saves a copy of the history fields. The RestoreField method restores the contents of a specific control.

Example:

```
WindowManager.TakeCompleted PROCEDURE
CODE
SELF.SaveHistory
CASE SELF.Request
OF InsertRecord
DO InsertAction
OF ChangeRecord
DO ChangeAction
OF DeleteRecord
DO DeleteAction
END
```

See Also: RestoreField AddHistoryField, AddHistoryFile, HistoryKey,

## **SaveOnChangeAction(execute change record process and remain active)**

### **SaveOnChangeAction, VIRTUAL**

The **SaveOnChangeAction** method performs the necessary database change or update operations when called. **SaveOnChangeAction** returns Level:Benign to indicate a successful change operation.

Implementation: The **SaveOnChangeAction** method is called by the **TakeCompleted** method when **Request** is set to **SaveRecord** and **OriginalRequest** is set to **ChangeRecord**.

Return Data Type: BYTE

Example:

```
WindowManager.TakeCompleted PROCEDURE
CODE
SELF.SaveHistory()
CASE SELF.Request
OF InsertRecord
RETURN SELF.InsertAction()
OF ChangeRecord
RETURN SELF.ChangeAction()
OF DeleteRecord
RETURN SELF.DeleteAction()
OF SaveRecord
CASE SELF.OriginalRequest
OF InsertRecord
RETURN SELF.SaveOnInsertAction()
OF ChangeRecord
RETURN SELF.SaveOnChangeAction()
END
END
RETURN Level:Benign
```

See Also:

TakeCompleted

Request

## **SaveOnInsertAction(execute insert record activity and remain active)**

### **SaveOnInsertAction, VIRTUAL**

The **SaveOnInsertAction** method performs the necessary database insert operations when called. **SaveOnInsertAction** returns Level:Benign to indicate a successful insert operation.

Implementation: The SaveOnInsertAction method is called by the TakeCompleted method when Request is set to SaveRecord and OriginalRequest is set to InsertRecord.

Return Data Type: BYTE

Example:

```
WindowManager.TakeCompleted PROCEDURE
CODE
SELF.SaveHistory()
CASE SELF.Request
OF InsertRecord
RETURN SELF.InsertAction()
OF ChangeRecord
RETURN SELF.ChangeAction()
OF DeleteRecord
RETURN SELF.DeleteAction()
OF SaveRecord
CASE SELF.OriginalRequest
OF InsertRecord
RETURN SELF.SaveOnInsertAction()
OF ChangeRecord
RETURN SELF.SaveOnChangeAction()
END
END
RETURN Level:Benign
```

See Also:

TakeCompleted

Request

## SetAlerts (alert window control keystrokes)

### SetAlerts, VIRTUAL

The **SetAlerts** method alerts any required keystrokes for the window's controls, including keystrokes required by the window's history key, browse lists, and locators.

Implementation: The SetAlerts method calls the BrowseClass.SetAlerts method for each BrowseClass object added by the AddItem method. SetAlerts also ALERTs the HistoryKey keystroke for each AddHistoryField control.

Note that the alerted keystrokes are associated only with the specific affected controls, such as a LIST or ENTRY. The keystrokes are not alerted for the WINDOW. See ALERT in the *Language Reference* for more information.

Example:

```
ThisWindow.Init PROCEDURE()  
  CODE  
  !procedure code  
  SELF.SetAlerts()  
  RETURN Level:Benign
```

See Also: AddHistoryField, HistoryKey, BrowseClass.SetAlerts

**SetResponse (OK or Cancel the window)****SetResponse( *response* ), VIRTUAL**


---

**SetResponse** Initiates standard "OK" or "Cancel" processing.

*response* An integer constant, variable, EQUATE, or expression indicating the WindowManager's response (OK or Cancel) to the requested operation.

The **SetResponse** method initiates standard "OK" or "Cancel" processing for the procedure. That is, it registers the procedure's result (completed or cancelled) and triggers the normal procedure shut down.

Implementation: The TakeAccepted method calls the SetResponse method. SetResponse sets the Response property and POSTs an EVENT:CloseWindow. If the *response* is RequestCancelled, SetResponse also sets the VCRRequest property to VCR:None.

EQUATEs for the response parameter are declared in \LIBSRC\TPLEQU.CLW. as follows:

```
RequestCompleted EQUATE (1) !Update Completed
RequestCancelled EQUATE (2) !Update Aborted
```

Example:

```
WindowManager.TakeAccepted PROCEDURE
I LONG,AUTO
A SIGNED,AUTO
CODE
A = ACCEPTED()
!procedure code
LOOP I = 1 TO RECORDS(SELF.Buttons)
GET(SELF.Buttons,I)
IF SELF.Buttons.Control = A
SELF.SetResponse(SELF.Buttons.Action)
RETURN Level:Notify
END
END
!procedure code
RETURN Level:Benign
```

See Also: Request, Response

**TakeAccepted (a virtual to process EVENT:Accepted--WindowManager)****TakeAccepted, VIRTUAL, PROC**

The **TakeAccepted** method processes EVENT:Accepted events for the window's controls, and returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeAccepted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: TakeAccepted carries out HistoryKey and 2 parameter AddItem actions.

Return values are declared in ABERROR.INC.

The TakeEvent method calls the TakeAccepted method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I USHORT,AUTO
CODE
IF ~FIELD()
  RVal = SELF.TakeWindowEvent()
  IF RVal THEN RETURN RVal.
END
CASE EVENT()
OF EVENT:Accepted; RVal = SELF.TakeAccepted()
OF EVENT:Rejected; RVal = SELF.TakeRejected()
OF EVENT:Selected; RVal = SELF.TakeSelected()
OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
OF EVENT:Completed; RVal = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
  RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
IF FIELD()
  RVal = SELF.TakeFieldEvent()
END
RETURN RVal
```

See Also: AddItem, HistoryKey, TakeEvent

## TakeCloseEvent (a virtual to Cancel the window)

### TakeCloseEvent, VIRTUAL, PROC

The **TakeCloseEvent** method processes EVENT:CloseWindow and EVENT:CloseDown events for the window and returns a value indicating whether window ACCEPT loop processing is complete and should stop.

TakeCloseEvent implements the default processing when the end user cancels an update form (presses the Cancel button). The actual process depends on the value of various WindowManager properties, including Request, Response, CancelAction, OriginalRequest, etc.

TakeCloseEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeEvent method calls the TakeCloseEvent method. The TakeCloseEvent method undoes any processing rendered invalid by the form cancellation (for example, deleting a dummy autoincremented record that is no longer needed).

Return values are declared in ABERROR.INC.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I USHORT,AUTO
CODE
IF ~FIELD()
  RVal = SELF.TakeWindowEvent()
  IF RVal THEN RETURN RVal.
END
CASE EVENT()
OF EVENT:Accepted; RVal = SELF.TakeAccepted()
OF EVENT:Rejected; RVal = SELF.TakeRejected()
OF EVENT:Selected; RVal = SELF.TakeSelected()
OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
OF EVENT:Completed; RVal = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
  RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
```

```
IF FIELD()  
  RVal = SELF.TakeFieldEvent()  
END  
RETURN RVal
```

See Also:      [CancelAction](#), [Request](#), [Response](#), [OriginalRequest](#), [TakeEvent](#)

## **TakeCompleted (a virtual to complete an update form)**

### **TakeCompleted, VIRTUAL, PROC**

The **TakeCompleted** method processes the EVENT:Completed event for the window and returns a value indicating whether window ACCEPT loop processing is complete and should stop.

TakeCompleted implements the default processing when the end user accepts an update form (presses the OK button). The actual process depends on the value of various WindowManager properties, including Request, InsertAction, VCRRequest, etc.

TakeCompleted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeCompleted method calls the SaveHistory method, then completes the requested action (insert, change, or delete), subject to various validation constraints. That is the FileManager object validates form fields and does concurrency checking, and the RelationManager object enforces any referential constraints.

TakeCompleted sets the Response property and POSTs an EVENT:CloseWindow when appropriate.

Return values are declared in ABERROR.INC.

The TakeEvent method calls the TakeCompleted method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
Rval BYTE(Level:Benign)
I    USHORT,AUTO
CODE
IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
END
CASE EVENT()
OF EVENT:Accepted;RVal = SELF.TakeAccepted()
OF EVENT:Rejected;RVal = SELF.TakeRejected()
OF EVENT:Selected;RVal = SELF.TakeSelected()
OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
OF EVENT:Completed;RVal = SELF.TakeCompleted()
```

```
    OF EVENT:CloseWindow OROF EVENT:CloseDown
      RVal = SELF.TakeCloseEvent()
    END
  IF RVal THEN RETURN RVal.
  IF FIELD()
    RVal = SELF.TakeFieldEvent()
  END
  RETURN RVal
```

See Also:      InsertAction, Request, Response, TakeEvent, VCRRequest

## TakeEvent (a virtual to process all events:WindowManager)

### TakeEvent, VIRTUAL, PROC

The **TakeEvent** method processes all window events and returns a value indicating whether ACCEPT loop processing is complete and should stop. TakeEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: Return values are declared in ABERROR.INC.

The Ask method calls the TakeEvent method.

Return Data Type: BYTE

Example:

```
WindowManager.Ask PROCEDURE
CODE
IF SELF.Dead THEN RETURN .
CLEAR(SELF.LastInsertedPosition)
ACCEPT
CASE SELF.TakeEvent()
OF Level:Fatal
BREAK
OF Level:Notify
CYCLE !Not as dopey at it looks, it is for 'short-stopping' certain events
END
END
```

See Also: Ask

## TakeFieldEvent (a virtual to process field events:WindowManager)

### TakeFieldEvent, VIRTUAL, PROC

The **TakeFieldEvent** method is a virtual placeholder to process all field-specific/control-specific events for the window. It returns a value indicating whether window process is complete and should stop. TakeFieldEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: Return values are declared in ABERROR.INC.

The TakeEvent method calls the TakeFieldEvent method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I   USHORT,AUTO
CODE
IF ~FIELD()
  RVal = SELF.TakeWindowEvent()
  IF RVal THEN RETURN RVal.
END
CASE EVENT()
OF EVENT:Accepted;   RVal = SELF.TakeAccepted()
OF EVENT:Rejected;  RVal = SELF.TakeRejected()
OF EVENT:Selected;   RVal = SELF.TakeSelected()
OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
OF EVENT:Completed;  RVal = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
  RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
IF FIELD()
  RVal = SELF.TakeFieldEvent()
END
RETURN RVal
```

See Also: Ask

**TakeNewSelection (a virtual to process EVENT:NewSelection)****TakeNewSelection, VIRTUAL, PROC**

The **TakeNewSelection** method processes EVENT:NewSelection events for the window's controls and returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeNewSelection returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: TakeNewSelection resets the WindowManager when the end user selects a new TAB.

Return values are declared in ABERROR.INC.

The TakeEvent method calls the TakeNewSelection method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;    RVal = SELF.TakeRejected()
    OF EVENT:Selected;    RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;   RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also: TakeEvent

## TakeRejected (a virtual to process EVENT:Rejected)

### TakeRejected, VIRTUAL, PROC

The **TakeRejected** method processes EVENT:Rejected events for the window's controls and returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeRejected returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: TakeRejected sounds the audible alarm and returns focus to the offending (rejected) control.

Return values are declared in ABERROR.INC.

The TakeEvent method calls the TakeRejected method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I USHORT,AUTO
CODE
IF ~FIELD()
RVal = SELF.TakeWindowEvent()
IF RVal THEN RETURN RVal.
END
CASE EVENT()
OF EVENT:Accepted; RVal = SELF.TakeAccepted()
OF EVENT:Rejected; RVal = SELF.TakeRejected()
OF EVENT:Selected; RVal = SELF.TakeSelected()
OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
OF EVENT:Completed; RVal = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
IF FIELD()
RVal = SELF.TakeFieldEvent()
END
RETURN RVal
```

See Also: TakeEvent

**TakeSelected (a virtual to process EVENT:Selected)****TakeSelected, VIRTUAL, PROC**

The **TakeSelected** method is a virtual placeholder to process EVENT:Selected events for the window's controls. It returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeSelected returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: Return values are declared in ABERROR.INC.

The TakeEvent method calls the TakeSelected method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I   USHORT,AUTO
    CODE
    IF ~FIELD()
        RVal = SELF.TakeWindowEvent()
        IF RVal THEN RETURN RVal.
    END
    CASE EVENT()
    OF EVENT:Accepted;   RVal = SELF.TakeAccepted()
    OF EVENT:Rejected;   RVal = SELF.TakeRejected()
    OF EVENT:Selected;   RVal = SELF.TakeSelected()
    OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
    OF EVENT:Completed;  RVal = SELF.TakeCompleted()
    OF EVENT:CloseWindow OROF EVENT:CloseDown
        RVal = SELF.TakeCloseEvent()
    END
    IF RVal THEN RETURN RVal.
    IF FIELD()
        RVal = SELF.TakeFieldEvent()
    END
    RETURN RVal
```

See Also: TakeEvent

**TakeWindowEvent (a virtual to process non-field events:WindowManager)****TakeWindowEvent, VIRTUAL, PROC**

The **TakeWindowEvent** method processes all non-field events for the window and returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeWindowEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: TakeWindowEvent implements standard handling of EVENT:OpenWindow (Open method), EVENT:LoseFocus, EVENT:GainFocus (Reset method), and EVENT:Sized (WindowResizeClass.Resize method).

Return values are declared in ABERROR.INC.

The TakeEvent method calls the TakeWindowEvent method.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I USHORT,AUTO
CODE
IF ~FIELD()
  RVal = SELF.TakeWindowEvent()
  IF RVal THEN RETURN RVal.
END
CASE EVENT()
OF EVENT:Accepted; RVal = SELF.TakeAccepted()
OF EVENT:Rejected; RVal = SELF.TakeRejected()
OF EVENT:Selected; RVal = SELF.TakeSelected()
OF EVENT:NewSelection; RVal = SELF.TakeNewSelection()
OF EVENT:Completed; RVal = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
  RVal = SELF.TakeCloseEvent()
END
IF RVal THEN RETURN RVal.
IF FIELD()
  RVal = SELF.TakeFieldEvent()
END
RETURN RVal
```

See Also: Open, Reset, TakeEvent, WindowResizeClass.Resize

**Update (prepare records for writing to disk)****Update, VIRTUAL**

The **Update** method prepares the WindowManager's FILE and VIEW records for writing to disk by synchronizing buffer contents with their corresponding screen values. The Update method also arms automatic optimistic concurrency checking so an eventual write (PUT) to disk returns an error if another user changed the data since it was retrieved.

Implementation: The Update method calls BrowseClass.UpdateViewRecord for each BrowseClass object added by the AddItem method.

Example:

```

ThisWindow.TakeAccepted PROCEDURE()
Looped BYTE
CODE
LOOP
  IF Looped
    RETURN Level:Notify
  ELSE
    Looped = 1
  END
  PARENT.TakeAccepted()
  CASE ACCEPTED()
  OF ?Expand
    ThisWindow.Update
    ?CusTree{PropList:MouseDownRow} = CHOICE(?CusTree)
    DO REL1::ExpandAll
  OF ?Contract
    ThisWindow.Update
    ?CusTree{PropList:MouseDownRow} = CHOICE(?CusTree)
    DO REL1::ContractAll
  OF ?Insert
    ThisWindow.Update
    ?CusTree{PropList:MouseDownRow} = CHOICE(?CusTree)
    DO REL1::AddEntry
  OF ?Change
    ThisWindow.Update
    ?CusTree{PropList:MouseDownRow} = CHOICE(?CusTree)
    DO REL1::EditEntry
  OF ?Delete
    ThisWindow.Update
    ?CusTree{PropList:MouseDownRow} = CHOICE(?CusTree)
    DO REL1::RemoveEntry
  END
RETURN Level:Benign

```

**Index:**

- \_Print (print rich text control contents)..... 305
- \_Print print a Crystal Report .....283
- ABC Template Implementation.....338, 957, 1099, 1167
- ACCEPT ..... 1332
- ActiveInvisible ..... 150
  - BrowseClass ..... 150
- ActiveInvisible (obscured browse list action) ..... 150
- AddBreak ..... 123
- AddControl ..... 463
- AddEditControl ..... 174
  - BrowseClass ..... 174
- AddEditControl (specify custom edit-in-place class) ..... 174
- AddErrors ..... 502
  - ErrorClass ..... 502
- AddErrors (add or override recognized errors) ..... 502
- AddField ..... 175, 579
  - BrowseClass ..... 175
  - FileDropClass ..... 579
- AddField (specify a FILE/QUEUE field pair) ..... 175
- AddField (specify display fields) ..... 579
- AddField(track fields in a structure) ..... 604
- AddHistory (update History structure) ..... 503
- AddHistoryField ..... 1328
  - WindowManagerClass ..... 1328
- AddHistoryField (add restorable control and field) ..... 1328
- AddHistoryFile ..... 1329
  - WindowManagerClass ..... 1329
- AddHistoryFile (add restorable history file) ..... 1329
- AddItem ..... 104, 256, 540, 835, 1118, 1330
  - AsciiViewerClass ..... 104
  - FieldPairsClass ..... 540
  - PopupClass ..... 835
  - QueryClass ..... 913
  - StepCustomClass ..... 1118
  - WindowManagerClass ..... 1330
- AddItem (add a step marker) ..... 1118
- AddItem (add field to query) ..... 913
- AddItem (add menu item) ..... 835
  - AddItem (maintain the columninfo structure) ..... 316
  - AddItem (maintain the namequeue structure) ..... 327
  - AddItem (program the AsciiViewer object) ..... 104
  - AddItem (program the WindowManager object) ..... 1330
  - AddItem(program the BrowseClass object) ..... 176
  - AddItem(program the ReportManager object) ..... 1032
  - AddItemEvent ..... 837
    - PopupClass ..... 837
  - AddItemEvent (set menu item action) ..... 837
  - AddItemMimic ..... 838
    - PopupClass ..... 838
  - AddItemMimic (tie menu item to a button) ..... 838
  - AddKey ..... 605
    - FileManagerClass ..... 605
  - AddKey (set the file's keys) ..... 605
  - AddLocator ..... 176
    - BrowseClass ..... 176
  - AddLocator (specify a locator) ..... 758
  - AddLocator (specify a locator) ..... 176
  - AddLogFile (maintain log file structure) ... 317
  - AddMask ..... 1090
    - SelectFileClass ..... 1090
  - AddMask (add file dialog file masks) ..... 1090
  - AddMenu ..... 839
    - PopupClass ..... 839
  - AddPair ..... 238, 541
    - BufferedPairsClass ..... 239
    - FieldPairsClass ..... 541
  - AddPair (add a field pair) ..... 238, 541
  - AddRange ..... 1255
    - ViewManagerClass ..... 1255
  - AddRange (add a range limit) ..... 1255
  - AddRecord (add a record filedrop queue) ..... 561
  - AddRelation ..... 1000
    - RelationManagerClass ..... 1000
  - AddRelation (set a file relationship) ..... 1000
  - AddRelationLink ..... 1002

- AddResetField ..... 177  
     BrowseClass ..... 177  
 AddResetField (set a field to monitor for changes) ..... 177  
 AddSortOrder ..... 178, 1256  
     BrowseClass ..... 178  
     ViewManagerClass ..... 1256  
 AddSortOrder (add a sort order) ..... 1256  
 AddSortOrder (specify a browse sort order) ..... 178  
 AddSubMenu ..... 840  
     PopupClass ..... 840  
 AddTarget ..... 1179  
     ToolBarClass ..... 1179  
 AddTarget (register toolbar driven entity) ..... 1179  
 AddThread (maintains the triggerqueue) .327  
 AddToolBarTarget ..... 179  
     BrowseClass ..... 179  
 AddToolBarTarget (set the browse toolbar) ..... 179  
 AddTranslation ..... 1235, 1236  
     TranslatorClass ..... 1235, 1236  
 AddTranslation (add translation pairs) .. 1235  
 AddUpdateField ..... 580  
     FileDropClass ..... 580  
 AddUpdateField (specify field assignments) ..... 580  
 AddUpdateFile ..... 1331  
     WindowManagerClass ..... 1331  
 AddUpdateFile (register batch add files) ..... 1331  
 AddValue  
     EditMultiSelectClass ..... 437  
 AddValue (prime the MultiSelect dialog) .437  
 Again ..... 454  
 AliasedFile ..... 594  
     FileManagerClass ..... 594  
 AliasedFile (the primary file) ..... 594  
 AlinkLookup (associative link lookup).... 1160  
 AllowReset ..... 576  
 AllowUnfilled ..... 151  
     BrowseClass ..... 151  
 AllowUnfilled (display filled list) ..... 151  
 AllowUserZoom ..... 866  
     PrintPreviewClass ..... 866  
 AllowUserZoom (allow any zoom factor) .866  
 AllText ..... 721  
 AppendOrder ..... 1257  
     ViewManagerClass ..... 1257  
 AppendOrder (refine a sort order) ..... 1257  
 ApplyFilter ..... 1258  
     ViewManagerClass ..... 1258  
 ApplyFilter (range limit and filter the result set) ..... 1258  
 ApplyOrder ..... 1259  
     ViewManagerClass ..... 1259  
 ApplyOrder (sort the result set) ..... 1259  
 ApplyRange ..... 180, 1260  
     BrowseClass ..... 180  
     ViewManagerClass ..... 1260  
 ApplyRange (conditionally range limit and filter the result set) ..... 1260  
 ApplyRange (refresh browse based on resets and range limits) ..... 180  
 Arrow ..... 454  
 ArrowAction ..... 152  
     BrowseClass ..... 152  
 ArrowAction (edit-in-place action on arrow key) ..... 152  
 ASCIIFile ..... 64  
     ASCIIFileClass ..... 64  
 ASCIIFile (the ASCII file) ..... 64  
 ASCIIFileClass ..... 97, 98  
 ASCIIFileClass Functional Organization—  
     Expected Use ..... 65  
 ASCIIFileClass Overview ..... 61  
 ASCIIPrintClass ..... 97, 98  
 ASCIIPrintClass Overview ..... 79  
 ASCIISearchClass ..... 97, 98  
 ASCIISearchClass Overview ..... 87  
 ASCIIViewerClass ..... 97, 98, 99  
 AsciiViewerClass Functional Organization—  
     Expected Use ..... 102  
 ASCIIViewerClass Overview ..... 97  
 Ask  
     ASCIIPrintClass ..... 83  
     ASCIISearchClass ..... 92  
     BrowseClass ..... 181  
     FileDropComboClass ..... 561  
     PopupClass ..... 841  
     QueryClass ..... 915  
     QueryFormClass ..... 938  
     ReportManagerClass ..... 1032

- SelectFileClass ..... 1091
- ASK.. 83, 92, 181, 183, 561, 841, 964, 1032, 1091, 1332
- Ask (a virtual to accept query criteria) ..... 915
- Ask (add a record to the lookup file)..... 561
- Ask (display the popup menu) ..... 841
- Ask (display window and process its events) ..... 1032
- Ask (display Windows file dialog) ..... 1091
- Ask (solicit print specifications) ..... 83
- Ask (solicit query criteria  
QueryListClass) ..... 964
- Ask (solicit query criteria) ..... 938
- Ask (solicit search specifications)..... 92
- Ask (update selected browse item) ..... 181
- AskGotoLine ..... 105
- ASCIIViewerClass..... 105
- AskGotoLine (go to user specified line)... 105
- AskPage ..... 872
- PrintPreviewClass ..... 872
- AskPreview ..... 1033
- ReportManagerClass ..... 1033
- AskPreview (preview or print the report) 1033
- AskPrintPages ..... 873
- AskProcedure ..... 153
- BrowseClass ..... 153
- AskProcedure (update procedure) .. 153, 556
- AskRecord
- BrowseClass ..... 183
- AskThumbnails ..... 875
- PrintPreviewClass ..... 875
- AskThumbnails (prompt for new thumbnail configuration) ..... 875
- AssignBufferToLeft ..... 240
- BufferedPairsClass ..... 240
- AssignBufferToLeft (copy from "buffer" fields to "left" fields) ..... 240
- AssignBufferToRight..... 241
- BufferedPairsClass ..... 241
- AssignBufferToRight (copy from "buffer" fields to "right" fields) ..... 241
- AssignLeftToBuffer ..... 242
- BufferedPairsClass ..... 242
- AssignLeftToBuffer (copy from "left" fields to "buffer" fields)..... 242
- AssignLeftToRight ..... 543
- FieldPairsClass ..... 543
- AssignLeftToRight (copy from "left" fields to "right" fields) ..... 543
- AssignRightToBuffer ..... 243
- BufferedPairsClass ..... 243
- AssignRightToBuffer (copy from "right" fields to "buffer" fields) ..... 243
- AssignRightToLeft ..... 544
- FieldPairsClass ..... 544
- AssignRightToLeft (copy from "right" fields to "left" fields) ..... 544
- Attribute..... 1022
- autoincrement ..... 607, 645, 648
- AutoRefresh ..... 1313
- WindowManagerClass ..... 1313
- AutoRefresh (reset window as needed flag) ..... 1313
- AutoToolbar ..... 1313
- WindowManagerClass ..... 1313
- AutoToolbar (set toolbar target on new tab selection) ..... 1313
- AutoTransparent ..... 1289
- WindowResizeClass ..... 1289
- AutoTransparent (optimize redraw) ..... 1289
- background processes
- ReportManagerClass ..... 1029
- BC ..... 137
- BC (browse class)..... 137
- BeforeChange (update audit log file before file change)..... 318, 769
- BeginRefresh ..... 721
- begins with
- Filter Locator ..... 683
- BindFields ..... 606
- FileManagerClass ..... 606
- BindFields (bind fields when file is opened) ..... 606
- BreakMan..... 1022
- Browse ..... 1190
- initial position..... 167
- ToolBarListBoxClass ..... 1190
- Browse (BrowseClass object)..... 226, 1190
- BrowseBox
- reset ..... 173
- BrowseClass ..... 145, 146, 147
- methods..... 172
- properties ..... 150

BrowseClass Functional Organization-- Expected Use.....	172	CalcPopup .....	724
BrowseClass Methods.....	172	CalcPopupAdd2 .....	725
BrowseClass Overview.....	145	CancelAction.....	1314
BrowseClass Properties .....	150	WindowManagerClass .....	1314
BrowseEIPManagerClass.....	131	CancelAction (response to cancel request) .....	1314
BrowseEIPManagerClass Concepts .....	132	CancelAutoInc .....	607, 608, 609, 1004
BrowseEIPManagerClass Properties .....	137	FileManagerClass .....	607
BrowseEIPManagerClass Source Files ..	133	RelationManagerClass.....	1004
BrowseEIPManagerClass--ABC Template Implementation .....	133	CancelAutoInc (undo autoincrement) ....	1004
BrowseEIPManagerClass--Conceptual Example .....	134	CancelAutoInc (undo PrimeAutoInc) .....	607
BrowseEIPManagerClass--Functional Organization--Expected Use.....	138	CancelPrintReport .....	1034
BrowseEIPManagerClass--Overview .....	131	CanRedo (check for redo data) .....	293
BrowseEIPManagerClass--Relationship to Other Application Builder Classes .....	133	Caption (window title) .....	824
BrowseQueue Concepts.....	221	CaseSensitiveValue (case sensitive flag)	895
BrowseQueue Source Files.....	221	ChangeAction .....	1315
BrowseToolBarClass ABC Template Implementation .....	225	WindowManagerClass .....	1315
BrowseToolBarClass Concepts .....	225	ChangeAction (response to change request) .....	1315
BrowseToolBarClass Overview .....	225	ChangeButton .....	1205
BrowseToolBarClass Source Files .....	225	ToolBarTargetClass.....	1205
Buffer		ChangeButton (change control number)	1205
FileManagerClass .....	595	ChangeControl.....	153
BUFFER .....	595	BrowseClass .....	153
Buffer (the record buffer) .....	595	ChangeField (virtual method for managing field changes).....	770
buffer management .....	535	ChangeFontStyle (set current font style) .	293
BufferedPairsClass		ChangeRecord.....	1333
methods .....	236	CheckChanges(check record for changes) .....	328
properties .....	235	.....	328
BufferedPairsClass Functional Organization—Expected Use .....	236	CheckPair(check field pairs for changes)	328
BufferedPairsClass Methods .....	236	Children (reference to child group controls) .....	754
BufferedPairsClass Overview.....	233	Choice(returns current selection number) .....	773
BufferedPairsClass Properties .....	235	.....	754
Buffers .....	595	Chosen (current browse queue element)	754
FileManagerClass .....	595	ClearColumn .....	139, 464
Buffers (saved record buffers) .....	595	ClearColumn (reset column property values EIPManagerClass) .....	464
Button (toolbar buttons FEQ values) .....	227	ClearColumn (reset column property values) .....	139
ButtonTypes (standard windows buttons)	824	ClearKey .....	610
CalcBestPositionNodeText.....	722	FileManagerClass .....	610
CalcCurrentGraph .....	722	ClearKey (clear specified key components) .....	610
CalcCurrentNode .....	723	ClearKeycode .....	833
CalcGraph.....	723	PopupClass .....	833

- ClearKeycode (clear KEYCODE character) .....833
- ClearLeft .....545
  - FieldPairsClass .....545
- ClearLeft (clear each "left" field) .....545
- ClearQuery .....916
- ClearQuery ( remove loaded query ) .....916
- ClearRight .....546
  - FieldPairsClass .....546
- ClearRight (clear each "right" field) .....546
- ClickPress (forward control) .....755
- Close
  - FileManagerClass .....612
  - RelationManagerClass.....1005
  - ViewManagerClass .....1261
- CLOSE.....612, 1005, 1261
- Close (close a file and any related files)1005
- Close (close standarderrorlog file) .....1101
- Close (close the file) .....612
- Close (close the view) .....1261
- Close (initiate close of log file).....532
- CloseHelp (close HTML help file) .....1161
- COLUMN .....454
- ConfirmPages .....866
- ConstantClass Functional Organization—
  - Expected Use .....255
- ConstantClass Overview .....249
- Construct (initialize FuzzyClass object)...688
- Construct (initialize StandardErrorLogClass object) .....1101
- Control .....813, 1205
  - LocatorClass .....813
  - ToolbarTargetClass .....1205
- Control (the locator control number).....813
- Control (window control).....1205
- ControlBase (base control number) .....755
- ControlNumber (number of controls) .....755
- Controls .....1105
  - StepClass .....1105
- Controls (the StepClass sort sequence)1105
- CREATE .....596
- CreateControl .....357, 447
  - EditCheckClass.....392, 404
  - EditClass .....345
  - EditColorClass .....375, 414
  - EditFontClass.....425
- CreateControl (a virtual to create the edit control) .....345
- CreateControl (create the edit-in-place CHECK control).....366
- CreateControl (create the edit-in-place COMBO control).....384
- CreateControl (create the edit-in-place control) .....375, 414, 425
- CreateControl (create the edit-in-place DROPLIST control) .....392
- CreateControl (create the edit-in-place ENTRY control) .....404
- CreateControl (create the edit-in-place SPIN control) .....357
- CreateHeader (create log file header records) .....318
- Crystal8 Class .....265
- Crystal8 Class Properties .....266
- CurrentPage .....867
  - PrintPreviewClass .....867
- CurrentPage (the selected report page) ..867
- cwRTF ABC Template Implementation ...290
- cwRTF Class Concepts .....289
- cwRTF Overview .....289
- cwRTF Properties .....291
- cwRTF Relationship to Other Application
  - Builder Classes .....290
- cwRTF Source Files .....290
- Database Operations
  - FileManagerClass .....601
- DbAuditManager ABC Template
  - Implementation.....313
- DbAuditManager Methods .....316
- DbAuditManager Properties .....314
- DbAuditManager Source Files.....313
- DbChangeManager ABC Template
  - Implementation.....325
- DbChangeManager Methods.....327
- DbChangeManager Overview .....325
- DbChangeManager Properties .....326
- DbChangeManager Source Files .....325
- DbLogFileManager ABC Template
  - Implementation.....333
- DbLogFileManager Methods .....335
- DbLogFileManager Overview .....333
- DbLogFileManager Properties.....334
- DbLogFileManager Source Files .....333

Dead .....	1315	DisplayButtons .....	1180, 1191, 1201, 1210, 1225
WindowManagerClass .....	1315	ToolBarClass .....	1180
Dead (shut down flag) .....	1315	ToolBarListBoxClass .....	1191
DefaultCategory (error category).....	495	ToolBarReltreeClass .....	1201
DefaultDirectory .....	1088	ToolBarTargetClass.....	1210
SelectFileClass .....	1088	ToolBarUpdateClass .....	1225
DefaultDirectory (initial path) .....	1088	DisplayButtons (enable appropriate toolbar buttons) .....	1180, 1191, 1201, 1210, 1225
DefaultFile.....	1088	DisplayPage.....	106
SelectFileClass .....	1088	ASCIIViewerClass.....	106
DefaultFile (initial filename/filemask).....	1088	DisplayPage (display new page) .....	106
DefaultFill.....	576	Draw.....	728
FileDropClass .....	576	DrawGraph.....	728
DeferMoves .....	1289	DrawReport.....	729
WindowResizeClass .....	1289	DrawWallpaper .....	729
DeferMoves (optimize resize).....	1289	DrillDown.....	730
DeferOpenReport .....	1023	ECON (current state of entry completion).....	556
ReportManagerClass .....	1023	EditCheckClass .....	359
DeferOpenReport (defer open) .....	1023	CreateControl.....	366, 392, 404
DeferWindow .....	1023	methods.....	365
Delete		properties .....	364
RelationManagerClass.....	1006	EditCheckClass ABC Template	
DELETE.....	917, 1006	Implementation.....	359
Delete ( remove saved query ).....	917	EditCheckClass Concepts .....	359
Delete (delete record subject to referential constraints) .....	1006	EditCheckClass Conceptual Example .....	361
DeleteAction .....	1316	EditCheckClass Functional Organization—	
WindowManagerClass .....	1316	Expected Use .....	365
DeleteAction (response to delete request) .....	1316	EditCheckClass Methods.....	365
DeleteButton.....	1206	EditCheckClass Overview .....	359
DeleteButton (delete control number) ...	1206	EditCheckClass Properties.....	364
DeleteControl.....	154	EditCheckClass Relationship to Other	
BrowseClass .....	154	Application Builder Classes.....	359
Deleted (return record status).....	613	EditCheckClass Source Files .....	359
DeletelImageQueue .....	876	EditClass.....	337
Deleteltem .....	842	CreateControl.....	345
PopupClass.....	842	FEQ .....	343
Deleteltem (remove menu item).....	842	Init.....	346
DeleteRecord.....	1334	Kill.....	347
DeleteRecord (delete a record) .....	614	methods.....	346
Destruct (automatic destructor) .....	615	SetAlerts.....	347, 393
Destruct (remove the		TakeEvent .....	349
StandardErrorLogClass object) .....	1101	EditClass Concepts .....	337
DiagramNameText .....	726, 727	EditClass Conceptual Example .....	339
DiagramText .....	726, 727	EditClass Overview.....	337
DISPLAY.....	877	EditClass Properties .....	343
		EditClass Source Files.....	338

- EditColorClass ..... 367
  - CreateControl..... 375, 414
  - properties ..... 373
  - TakeEvent ..... 376, 415
  - Title ..... 373
- EditColorClass ABC Template
  - Implementation ..... 367
- EditColorClass Concepts..... 367
- EditColorClass Conceptual Example ..... 369
- EditColorClass Functional Organization--
  - Expected Use..... 374
- EditColorClass Properties ..... 373
- EditColorClass Relationship to Other
  - Application Builder Classes ..... 367
- EditColorClass Source Files..... 368
- EditColorClass Overview ..... 367
- EditDropComboClass ABC Template
  - Implementation ..... 377
- EditDropComboClass Concepts..... 377
- EditDropComboClass Conceptual Example ..... 379
- EditDropComboClass Functional Organization..... 383
- EditDropComboClass Methods ..... 383
- EditDropComboClass Overview ..... 377
- EditDropComboClass Properties ..... 382
- EditDropComboClass Source Files ..... 378
- EditDropListClass ..... 385
  - properties ..... 390
- EditDropListClass ABC Template
  - Implementation ..... 385
- EditDropListClass Concepts..... 385
- EditDropListClass Conceptual Example.. 387
- EditDropListClass Functional Organization--
  - Expected Use..... 391
- EditDropListClass Overview ..... 385
- EditDropListClass Properties..... 390
- EditDropListClass Relationship to Other
  - Application Builder Classes ..... 385
- EditDropListClass Source Files ..... 385
- EditEntryClass ..... 397
  - methods ..... 403
  - properties ..... 402
- EditEntryClass ABC Template
  - Implementation ..... 398
- EditEntryClass Concepts..... 397
- EditEntryClass Conceptual Example..... 398
- EditEntryClass Functional Organization--
  - Expected Use ..... 403
- EditEntryClass Methods ..... 403
- EditEntryClass Overview ..... 397
- EditEntryClass Properties..... 402
- EditEntryClass Relationship to Other
  - Application Builder Classes..... 397
- EditEntryClass Source Files ..... 398
- EditFileClass ..... 405
  - FileMask ..... 411
  - FilePattern ..... 411
  - properties ..... 411
  - Title ..... 412
- EditFileClass ABC Template Implementation ..... 405
- EditFileClass Concepts..... 405
- EditFileClass Conceptual Example ..... 407
- EditFileClass Functional Organization--
  - Expected Use ..... 413
- EditFileClass Overview ..... 405
- EditFileClass Properties ..... 411
- EditFileClass Relationship to Other
  - Application Builder Classes..... 405
- EditFileClass Source Files..... 406
- EditFontClass ..... 417
  - CreateControl ..... 425
  - methods..... 424
  - properties ..... 423
  - TakeEvent ..... 426
  - Title..... 423
- EditFontClass ABC Template
  - Implementation..... 417
- EditFontClass Concepts ..... 417
- EditFontClass Conceptual Example ..... 419
- EditFontClass Functional Organization--
  - Expected Use ..... 424
- EditFontClass Methods..... 424
- EditFontClass Overview ..... 417
- EditFontClass Properties ..... 423
- EditFontClass Relationship to Other
  - Application Builder Classes..... 417
- EditFontClass Source Files ..... 418
- EditList ..... 155
  - BrowseClass ..... 155
- EditList (list of edit-in-place controls)..... 155
- EditMultiSelectClass ..... 427
  - AddValue..... 437

CreateControl.....	438	EndReport.....	1035
methods .....	435	Enter .....	455
properties .....	434	Enter (edit-in-place action on enter key).....	455
Reset.....	438	EnterAction .....	156
TakeAction .....	439, 440	BrowseClass .....	156
TakeEvent.....	442	EnterAction (edit-in-place action on enter	
Title .....	434	key).....	156
EditMultiSelectClass ABC Template		Entries.....	1117
Implementation .....	428	StepCustomClass .....	1117
EditMultiSelectClass Concepts .....	427	Entries (expected data distribution) .....	1117
EditMultiSelectClass Conceptual Example		EntryCompletion .....	557
.....	429	FileDropComboClass.....	557
EditMultiSelectClass Functional		EntryCompletion (automatic fill-ahead flag)	
Organization--Expected Use.....	435	.....	557
EditMultiSelectClass Methods.....	435	EntryLocatorClass	
EditMultiSelectClass Overview.....	427	methods.....	482
EditMultiSelectClass Properties .....	434	properties .....	481
EditMultiSelectClass Relationship to Other		EntryLocatorClass Methods.....	482
Application Builder Classes .....	427	EntryLocatorClass Overview .....	477
EditMultiSelectClass Source Files.....	428	EntryLocatorClass Properties .....	481
EditSpinClass .....	351	EQ.....	455
EditSpinClass -- Relationship to Other		Equal.....	547
Application Builder Classes .....	351	FieldPairsClass .....	547
EditSpinClass Concepts.....	351	Equal (return 1 if all pairs are equal) .....	547
EditSpinClass Methods .....	356	Equal(checks for equal before and after	
EditSpinClass Properties.....	355	values).....	329
EditSpinClass Source Files .....	351	EqualBuffer .....	616
EditSpinClass--ABC Template		FileManagerClass .....	616
Implementation .....	351	EqualBuffer (detect record buffer changes)	
EditSpinClass--Conceptual Example .....	352	.....	616
EditSpinClass--Functional Organization—		EqualLeftBuffer .....	244
Expected Use.....	356	BufferedPairsClass .....	244
EditSpinClass--Overview.....	351	EqualLeftBuffer (compare "left" fields to	
EditTextClass Overview .....	443	"buffer" fields).....	244
EditTextClass Methods.....	446	EqualLeftRight .....	548
EIP .....	155	FieldPairsClass .....	548
EIPManagerClass.....	449	EqualLeftRight (return 1 if all pairs are	
EIPManagerClass Concepts .....	449	equal) .....	548
EIPManagerClass Source Files .....	450	EqualRightBuffer.....	245
EIPManagerClass--ABC Template		BufferedPairsClass .....	245
Implementation .....	450	EqualRightBuffer (compare "right" fields to	
EIPManagerClass--Conceptual Example	451	"buffer" fields).....	245
EIPManagerClass--Functional Organization-		Err (errorclass object) .....	824
-Expected Use .....	461	Err (errorclass obejct) .....	765
EIPManagerClass--Overview .....	449	ErrorClass	
EIPManagerClass--Relationship to Other		methods.....	500, 501
Application Builder Classes .....	449	properties .....	495

- ErrorClass Functional Organization--
  - Expected Use.....500
- ErrorClass Overview.....489
- ErrorClass Properties .....495
- ErrorClass Source Files.....823
- ErrorLog (errorlog interface).....495
- ErrorLogInterface Concepts .....531
- ErrorLogInterface Methods.....532
- ErrorLogInterface Source Files .....531
- ErrorMgr.....64
  - ASCIIFileClass.....64
- ErrorMgr (ErrorClass object) .....64
- Errors .....496, 1317
  - ErrorClass .....496
  - WindowManagerClass .....1317
- Errors (ErrorClass object).....315, 1317
- Errors (recognized error definitions).....496
- eShowSBonFirstThread .....710
- eSumYMax .....710
- ExtractText.....1234
  - TranslatorClass.....1234
- ExtractText (identify text to translate) ....1234
- FEQ .....346
  - EditClass .....343
- FEQ (the edit-in-place control number) ...343
- Fetch .....184, 617, 789, 790
  - BrowseClass .....184
  - FileManagerClass .....617
  - INIClass.....790
- Fetch (get a page of browse items).....184
- Fetch (get a specific record by key value)
  - .....617
- Fetch (get INI file entries) .....789
- FetchFeq (retrieve button feq).....826
- FetchField.....791
  - INIClass.....791
- FetchField (return comma delimited INI file value) .....791
- FetchQueue.....792
  - INIClass.....792
- FetchQueue (get INI file queue entries) ..792
- FetchRecord (retrieve selected record)...758
- FetchStdButton (determine button pressed)
  - .....826
- FieldName .....497
- FieldPairsClass.....535, 536, 537
  - properties .....538
- FieldPairsClass Functional Organization--
  - Expected Use.....539
- FieldPairsClass Overview.....535
- FieldPairsClass Properties .....538
- Fields .....456
- Fields (managed fields) .....456
- FILE .....596
- File Manager.....589
- FileDropClass
  - methods.....577
  - properties .....576
- FileDropClass Functional Organization--
  - Expected Use .....577
- FileDropClass Methods .....577
- FileDropClass Overview .....571
- FileDropClass Properties.....576
- FileDropComboClass
  - methods.....559
  - Overview .....551
  - properties .....556
- FileDropComboClass Functional Organization--Expected Use .....559
- FileDropComboClass Methods.....559
- FileDropComboClass Properties .....556
- FileManager
  - methods.....601
  - properties .....594
- FileManager Functional Organization--
  - Expected Use .....602
- FileManager Overview.....589
- FileManager Properties .....594
- FileManagerClass
  - Database Operations .....601
  - Interactive Database Operations .....601
  - Silent Database Operations .....601
- FileMask
  - EditFileClass .....411
- FileMgr .....82, 91
  - AsciiPrintClass .....82
  - ASCIISearchClass .....91
- FileName.....497, 597
  - FileManagerClass .....597
  - INIClass.....788
- FileName (variable filename).....597
- FileNameValue .....598
  - FileManagerClass .....598
- FileNameValue (constant filename) .....598

FilesOpened(files opened by procedure)	FuzzyClass ABC Template Implementation
..... 1317	.....687
FilterLocatorClass	FuzzyClass Source Files .....687
methods ..... 684	FuzzyClassClass Properties.....688
properties .....683	GetAcross (number of horizontal grids)...759
FilterLocatorClass Methods.....684	GetButtonFeq(returns a field equate label)
FilterLocatorClass Overview.....679	.....948
FilterLocatorClass Properties .....683	GetCategory (retrieve error category) .....503
FilterReset .....817	GetClickPress (forward click control).....759
Find.....90	GetComponents.....618
ASCIISearchClass .....90	FileManagerClass .....618
Find (search constraints).....90	GetComponents (return the number of key
FindNearbyNodes.....731	components) .....618
FirstField	GetControl(returns control number).....774
WindowManagerClass ..... 1317	GetDOSFilename .....67
FIRSTFIELD ..... 1317	ASCIIFileClass .....67
FirstField (first window control)..... 1317	GetDOSFilename (let end user select file) 67
Flags ..... 1089	GetDown (number of vertical grids ) .....759
SelectFileClass ..... 1089	GetEdit .....465
Flags (file dialog behavior) ..... 1089	GetEdit (identify edit-in-place field).....465
FlatButtons (use flat button style) .....296	GetEOF .....619
FldsEIP .....972	FileManagerClass .....619
FldsEIP (reference to the EditDropListClass)	GetEOF (return end of file status) .....619
.....972	GetError .....503, 620
FloatRight .....683	ErrorClass .....503
FilterLocatorClass .....683	FileManagerClass .....620
FloatRight ("contains" or "begins with" flag)	GetError (return the current error ID).....620
.....683	GetErrorcode .....504
FocusLoss .....456	ErrorClass .....504
FocusLoss ( action on loss of focus) .....456	GetField .....621
FocusLossAction .....157	FileManagerClass .....621
BrowseClass .....157	GetField (return a reference to a key
FocusLossAction (edit-in-place action on	component) .....621
lose focus).....157	GetFieldName.....623
Font (apply font attributes).....297	FileManagerClass .....623
ForcedReset	GetFieldName (return a key component field
WindowManagerClass ..... 1318	name) .....623
ForcedReset (force reset flag)..... 1318	GetFieldPicture(get field picture) .....625
ForceRefresh ..... 1318	GetFields(get number of fields) .....625
FormatLine .....66	GetFieldType(get field type) .....625
ASCIIFileClass .....66	GetFilename .....68
FormatLine (a virtual to format text) .....66	ASCIIFileClass .....68
FreeElement .....813	GetFilename (return the filename).....68
LocatorClass .....813	GetFirstSortField.....1261
FreeElement (the locator's first free key	GetFreeElementName .....1262
element) .....813	ViewManagerClass .....1262
Functional Organization--Expected Use..344	

- GetFreeElementName (return free key element name) ..... 1262
- GetFreeElementPosition ..... 1263
  - ViewManagerClass ..... 1263
- GetFreeElementPosition (return free key element position)..... 1263
- GetHelpFile (get help file name)..... 1161
- GetItemChecked..... 844
  - PopupClass..... 844
- GetItemChecked (return toggle item status) ..... 844
- GetItemEnabled..... 845
  - PopupClass..... 845
- GetItemEnabled (return item status) ..... 845
- GetItems ..... 846
- GetItems(returns number of entries) ..... 774
- GetLastLineNo..... 69
  - ASCIIFileClass ..... 69
- GetLastLineNo (return last line number) ... 69
- GetLastSelection ..... 847
  - PopupClass..... 847
- GetLastSelection (return selected item) .. 847
- GetLimit
  - QueryClass ..... 920
- GetLimit (get searchvalues)..... 920
- GetLine ..... 70
  - ASCIIFileClass ..... 70
- GetLine (return line of text)..... 70
- GetMouse ..... 732
- GetName ..... 626
  - FileManagerClass ..... 626
- GetName (return the filename)..... 626
- GetNbFiles(returns number of children) 1007
- GetNbRelations(returns number of relations) ..... 1007
- GetParentControl..... 1291
  - WindowResizeClass ..... 1291
- GetParentControl (return parent control) ..... 1291
- GetPercentile .... 71, 1106, 1119, 1127, 1141
  - ASCIIFileClass ..... 71
  - StepClass ..... 1106
  - StepCustomClass ..... 1119
  - StepLongClass ..... 1127
  - StepRealClass ..... 1141
  - StepStringClass ..... 1153
- GetPercentile (convert file position to percentage) ..... 71
- GetPercentile (return a value's percentile) ..... 1106, 1119, 1127, 1141, 1153
- GetPosition (retrieve group control position) ..... 760
- GetPositionStrategy ..... 1292
  - WindowResizeClass ..... 1292
- GetPositionStrategy (return position strategy for a control type) ..... 1292
- GetProcedureName ..... 504
- GetProcedureName (return procedure name ) ..... 504
- GetQueueMatch ..... 562
  - FileDropComboClass ..... 562
- GetQueueMatch (locate a list item) ..... 562
- GetRelation(returns reference to relation manager) ..... 1008
- GetRelationType(returns relation type) . 1008
- GetResizeStrategy ..... 1293
  - WindowResizeClass ..... 1293
- GetResizeStrategy (return resize strategy for a control type) ..... 1293
- GetShadow(return shadow value) ... 482, 815
- GetText (copy text to variable)..... 298
- GetTopic (get current topic name)..... 1161
- GetValue ..... 1107, 1120, 1128, 1142
  - StepClass ..... 1107
  - StepCustomClass ..... 1120
  - StepLongClass ..... 1128
  - StepRealClass ..... 1142
  - StepStringClass ..... 1154
- GetValue (return a percentile's value) .. 1107, 1120, 1128, 1142, 1154
- GetValueFromField..... 733
- GetValueFromStatusBar..... 733
- GetVisible(returns visibility of control) .... 774
- GraphClass Overview ..... 709
- GraphClass Source Files..... 709
- GridClass ABC Template Implementation ..... 753
- GridClass Methods ..... 758
- GridClass Overview ..... 753
- GridClass Properties..... 754
- GridClass Source Files ..... 753
- GroupColor (background color of group fields)..... 756

GroupControl (GROUP control number) .756	History.....1224
GroupTitle (title of group element).....756	ToolbarUpdateClass .....1224
gShowDiagramName .....711	History (enable toolbar history button)...1224
gShowDiagramNameV .....712	History (error history structure).....765
gShowMouse .....713	HistoryHandler (windowcomponent
gShowMouseX .....714	interface) .....825
gShowMouseY .....715	HistoryKey.....1318
gShowNodeName .....716	WindowManagerClass .....1318
gShowNodeNameV .....717	HistoryKey (restore field key).....1318
gShowNodeValue .....718	HistoryMsg (initialize the message window)
gShowNodeValueX .....719	.....505
gShowNodeValueY .....720	hRTFWindow(RTF control handle).....291
HasCancelButton display cancel button on	Icon (icon for image control) .....824
report preview .....269	IDbChangeAudit Concepts .....769
HasCloseButton display close button on	IDbChangeAudit Methods.....769
report preview .....270	IDbChangeAudit Source Files .....769
HasExportButton display export button on	IfGroupField (determine if current control is
report preview .....271	a GROUP) .....760
HasLaunchButton display launch button on	IListControl Concepts .....773
report preview .....272	IListControl Methods.....773
HasNavigationControls display navigation	IListControl Source Files.....773
controls on report preview.....273	ImageToWMF .....734
HasPrintButton display print button on report	IncrementalLocatorClass
preview.....274	methods.....782
HasPrintSetupButton display print setup	properties .....781
button on report preview .....275	IncrementalLocatorClass Methods .....782
HasProgressControls display progress	IncrementalLocatorClass Overview.....777
controls on report preview.....276	IncrementalLocatorClass Properties .....781
HasRefreshButton display refresh button on	INIClass .....785, 786, 787
report preview .....277	methods.....789
HasSearchButton display search button on	properties .....788
report preview .....278	INIClass Methods .....789
HasThumb .....158	INIClass Properties.....788
BrowseClass .....158	Init 72, 84, 93, 107, 108, 140, 185, 246, 257,
HasZoomControl display zoom control on	466, 483, 506, 548, 563, 564, 581, 627,
report preview .....279	628, 734, 793, 816, 848, 878, 899, 965,
HelpButton .....1206	975, 987, 1009, 1036, 1092, 1108, 1121,
ToolbarTargetClass .....1206	1181, 1237, 1264, 1294, 1335
HelpButton (help control number) .....1206	ASCIIFileClass .....72
HideSelect .....158	ASCIIPrintClass .....84
High .....1126, 1140	ASCIISearchClass .....93
StepLongClass.....1126	ASCIIViewerClass.....107, 108
StepRealClass .....1140	BrowseClass .....185
High (upper boundary).....1126, 1140	BufferedPairsClass .....246
HistHandlerClass Methods .....767	ConstantClass.....257
HistHandlerClass Properties .....765	EditClass .....346
HistHandlerClass Source Files.....765	EntryLocatorClass.....483

ErrorClass .....	506	Init (initialize the ErrorClass object) .....	506
FieldPairsClass .....	548	Init (initialize the FieldPairsClass object) .....	548
FileDropClass .....	581	Init (initialize the FileDropClass object) .....	581
FileDropComboClass .....	563	Init (initialize the FileDropComboClass object) .....	563
INIClass .....	793, 794	Init (initialize the GridClass object) .....	761
LocatorClass .....	816	Init (initialize the HistHandlerClass object) .....	767
PopupClass .....	848	Init (initialize the INIClass object) .....	793
PrintPreviewClass .....	878	Init (initialize the LocatorClass object) .....	816
ProcessClass .....	899, 900	Init (initialize the MsgBoxClass object) .....	827
QueryClass .....	921	Init (initialize the PopupClass object) .....	848
QueryFormClass .....	939	Init (initialize the PrintPreviewClass object) .....	878
QueryFormVisual .....	949	Init (initialize the QueryClass object) .....	921
SelectFileClass .....	1092	Init (initialize the QueryFormClass object) .....	939
StepClass .....	1108	Init (initialize the QueryFormVisual object) .....	949
StepCustomClass .....	1121	Init (initialize the QueryListClass object) .....	965
StepStringClass .....	1155	Init (initialize the QueryListVisual object) .....	975
ToolBarClass .....	1181	Init (initialize the QueryVisual object) .....	987
TranslatorClass .....	1237	Init (initialize the SelectFileClass object) .....	1092
ViewManagerClass .....	1264	Init (initialize the StepClass object) .....	1108
WindowManagerClass .....	1335	Init (initialize the StepCustomClass object) .....	1121
WindowResizeClass .....	1294	Init (initialize the ToolbarClass object) .....	1181
Init (initialize TextWindow object) .....	1169	Init (initialize the TranslatorClass object) .....	1237
Init (initialize FuzzyClass object) .....	688	Init (initialize the ViewManager object) .....	1264
Init (initialize HTML Help object) .....	1162	Init (initialize the WindowManager object) .....	1335
Init (initialize the ASCIIPrintClass object) .....	84	Init (initialize the WindowResizeClass object) .....	1294
Init (initialize the ASCIISearchClass object) .....	93	Init initialize Crystal8 object .....	280
Init (initialize the ASCIIViewerClass object) .....	107	Init(initialize the StandardBehavior object) .....	1097
Init (initialize the BrowseClass object) .....	185	InitBrowse (initialize the BrowseToolBarClass update buttons) .....	228
Init (initialize the BrowseEIPManagerClass object) .....	140	InitControls .....	466
Init (initialize the BrowseToolBarClass object) .....	228	InitMisc (initialize the BrowseToolBarClass miscellaneous buttons) .....	229
Init (initialize the BufferedPairsClass object) .....	246	InitSort (initialize locator values) .....	186
Init (initialize the ConstantClass object) .....	257	InitSyncPair .....	576
Init (initialize the cwRTF object) .....	299	InitVCR (initialize the BrowseToolBarClass VCR buttons) .....	230
Init (initialize the DbAuditManager object) .....	319		
Init (initialize the DbChangeManager object) .....	329		
Init (initialize the DbLogFileManager object) .....	335		
Init (initialize the EditClass object) .....	346		
Init (initialize the EntryLocatorClass object) .....	483		

InPageList.....	879	RelationManagerClass.....	1010
Insert.....	457, 629	ReportManagerClass.....	1037
FileManagerClass.....	629	StepClass.....	1109
Insert (add a new record).....	629	StepCustomClass.....	1122
Insert (placement of new record).....	457	StepStringClass.....	1156
Insert(add entry to LIST queue).....	223	ToolBarClass.....	1181
InsertAction.....	1319	TranslatorClass.....	1237
WindowManagerClass.....	1319	WindowManagerClass.....	1338
InsertAction (response to insert request) .....	1319	WindowResizeClass.....	1297
InsertButton.....	1207	Kill (perform any necessary termination code).....	507, 827
ToolBarTargetClass.....	1207	Kill (shut down DbAuditManger object) ...	319
InsertButton (insert control number).....	1207	Kill (shut down DbChangeManger object) .....	330
InsertControl.....	159	Kill (shut down the ASCIIFileClass object)	73
BrowseClass.....	159	Kill (shut down the ASCIIViewerClass object).....	109
InsertRecord.....	1337	Kill (shut down the BrowseClass object) .	187
Interactive Database Operations		Kill (shut down the BrowseEIPManagerClass object) .....	141
FileManagerClass.....	601	Kill (shut down the BufferedPairsClass object).....	247
Interactivity.....	735	Kill (shut down the ConstantClass object) .....	259
IsDirty (indicates modified data).....	300	Kill (shut down the csRTF object).....	301
IsOverNode.....	735	Kill (shut down the EIPManagerClass object) .....	
IsSkelActive.....	761	Kill (shut down the FieldPairsClass object) .....	549
KeepVisible.....	1024	Kill (shut down the FileDropClass object)	582
KeyToOrder.....	630	Kill (shut down the PopupClass object) ...	848
FileManagerClass.....	630	Kill (shut down the PrintPreviewClass object).....	880
KeyToOrder (return ORDER expression for a key).....	630	Kill (shut down the ProcessClass object)	901
KeyValid (check for valid keystroke) .....	565	Kill (shut down the QueryClass object)....	922
KeyWordLookup (lookup keyword) .....	1163	Kill (shut down the QueryFormClass object) .....	940
Kill .... 73, 109, 110, 141, 187, 247, 259, 467, 507, 549, 582, 632, 736, 848, 880, 901, 966, 988, 1010, 1037, 1109, 1122, 1156, 1181, 1237, 1265, 1297, 1338		Kill (shut down the QueryListClass object) .....	966
ASCIIViewerClass.....	109	Kill (shut down the QueryVisual object) ...	988
BrowseClass.....	187	Kill (shut down the RelationManager object) .....	1010
BufferedPairsClass.....	247	Kill (shut down the ReportManager object) .....	1037
ConstantClass.....	259	Kill (shut down the StepClass object) ....	1109
EditClass.....	347	Kill (shut down the StepCustomClass object) .....	1122
ErrorClass.....	507		
FieldPairsClass.....	549		
FileDropClass.....	582		
FileManagerClass.....	632		
PopupClass.....	848		
PrintPreviewClass.....	880		
ProcessClass.....	901		
QueryClass.....	922		
QueryFormClass.....	940		

- Kill (shut down the StepStringClass object) ..... 1156
- Kill (shut down the ToolbarClass object) ..... 1181
- Kill (shut down the TranslatorClass object) ..... 1237
- Kill (shut down the WindowManager object) ..... 1338
- Kill (shut down the WindowResizeClass object) ..... 1297
- Kill (shutdown FuzzyClass object) ..... 688
- Kill (shutdown TextWindow object) ..... 1169
- Kill (shutdown the FileManager object) ... 632
- Kill (shutdown the GridClass object) ..... 762
- Kill (shutdown the QueryListVisual object) ..... 976
- Kill (shutdown the TagHTMLHelp object) ..... 1163
- Kill shut down Crystal8 object ..... 281
- Kill(shutdown the parent object) ..... 1278
- LastColumn ..... 458
- LastColumn (previous edit-in-place column) ..... 458
- LastInsertedPosition ..... 1319
- WindowManager ..... 1319
- LazyOpen ..... 599
- FileManagerClass ..... 599
- LazyOpen (delay file open until access).. 599
- LBColumns (number of listbox columns). 766
- LeftIndent (indent the current or selected paragraph) ..... 301
- Level ..... 490
- Benign ..... 490
- Cancel ..... 490
- Fatal ..... 490
- Notify ..... 490
- Program ..... 490
- User ..... 490
- LFM (DbLogFileManager object) ..... 315
- LimitTextSize (limit amount of text) ..... 302
- LineCounter ..... 91
- ASCIISearchClass ..... 91
- List
- FieldPairsClass ..... 538
- LIST ..... 538
- List (recognized field pairs) ..... 538
- ListControl ..... 458
- ListControl (listbox control number) ..... 458
- ListLinkingFields ..... 1011
- RelationManagerClass ..... 1011
- ListLinkingFields (map pairs of linked fields) ..... 1011
- ListQueue ..... 159
- BrowseClass ..... 159
- Load ..... 260
- ConstantClass ..... 260
- Loaded ..... 159
- BrowseClass ..... 159
- Loaded (browse queue loaded flag) ..... 159
- LoadField (load rich text data from field) .302
- LocateButton(query control number) ..... 1207
- LocatorClass ..... 679, 811, 812
- methods ..... 815
- properties ..... 813
- LocatorClass Methods ..... 815
- LocatorClass Overview ..... 811
- LocatorClass Properties ..... 813
- LockRecover ..... 599
- FileManagerClass ..... 599
- LockRecover (/RECOVER wait time parameter) ..... 599
- LookupMode ..... 1150
- StepStringClass ..... 1150
- Low ..... 1126, 1140
- StepLongClass ..... 1126
- StepRealClass ..... 1140
- Low (lower boundary) ..... 1126, 1140
- Match (find query matches) ..... 689
- Maximize
- PrintPreviewClass ..... 867
- MAXIMIZE ..... 867
- Maximize (number of pages displayed horizontally) ..... 867
- Me ..... 998
- RelationManagerClass ..... 998
- Me (the primary file's FileManager object) ..... 998
- Message
- ErrorClass ..... 508
- MESSAGE ..... 508
- MessageBox ..... 510
- ErrorClass ..... 510
- MessageBox (display error message to window) ..... 510

- MessageText ..... 499  
 MouseText ..... 736  
 MouseXText ..... 736  
 MouseYText ..... 737  
 Msg ..... 509  
   ErrorClass ..... 509  
 Msg (initiate error message destination) .509  
 MsgBoxClass Methods ..... 826  
 MsgBoxClass Overview ..... 823  
 MsgBoxClass Properties ..... 824  
 MsgRVal (message box return value) ..... 825  
 NameQueue (pointer into trigger queue). 326  
 Naming Conventions and Dual Approach to  
   Database Operations ..... 601  
 Next  
   ASCIISearchClass ..... 94  
   BrowseClass ..... 188  
   ConstantClass ..... 262  
   FileManagerClass ..... 633  
   ProcessClass ..... 902  
   ReportManagerClass ..... 1038  
 NEXT ..... 94, 188, 262, 468, 633, 902, 1038,  
   1265  
 Next (copy next constant item to targets) 262  
 Next (find next line containing search text) 94  
 Next (get next element) ..... 902  
 Next (get next record in sequence) ..... 633  
 Next (get next report record) ..... 1038  
 Next (get the next browse item) ..... 188  
 Next (get the next edit-in-place field) ..... 468  
 Next (load all constant items to file or queue) .....  
 NoCase  
   LocatorClass ..... 814  
 NOCASE ..... 814  
 NoCase (case sensitivity flag) ..... 814  
 NodeNameText ..... 737  
 NodeText ..... 738  
 NodeTipText ..... 738  
 NodeValueText ..... 739  
 NodeXText ..... 739  
 NodeYText ..... 740  
 OKControl ..... 1320  
   WindowManagerClass ..... 1320  
 OKControl (window acceptance control--OK  
   button) ..... 1320  
 OnChange (update audit log file after a  
   record change) ..... 320, 771  
 OnDelete (update audit log file when a  
   record is deleted) ..... 320  
 OnFieldChange (virtual method for each  
   field change) ..... 321  
 OnInsert (update audit log file when a record  
   is added) ..... 322  
 Open  
   FileManagerClass ..... 634  
   PrintPreviewClass ..... 881  
   RelationManagerClass ..... 1012  
   ViewManagerClass ..... 1266  
   WindowManagerClass ..... 1339  
 OPEN ..... 634, 881, 1012, 1039, 1266, 1339  
 Open (a virtual to execute on EVENT  
   OpenWindow) ..... 1339  
 Open (initiate open of log file) ..... 533  
 Open (open a file and any related files). 1012  
 Open (open standarderrorlog file) ..... 1102  
 Open (open the file) ..... 634  
 Open (open the view) ..... 1266  
 Open (prepare preview window for display)  
   ..... 881  
 Opened ..... 1320  
   WindowManagerClass ..... 1320  
 Opened (file opened flag) ..... 334, 802  
 Opened (window opened flag) ..... 1320  
 OpenLogFile (open the audit log file) ..... 322  
 OpenMode ..... 600  
   FileManagerClass ..... 600  
 OpenMode (file access/sharing mode) ..... 600  
 OpenReport ..... 1040, 1060  
   ReportManagerClass ..... 1040  
 OpenReport (prepare report for execution)  
   ..... 1040  
 OpsEIP ..... 972  
 OpsEIP (reference to the EditDropListClass)  
   ..... 972  
 Order  
   ViewManagerClass ..... 1248  
 ORDER ..... 1248  
 Order (sort  
   range-limit  
   and filter information) ..... 1248  
 OriginalRequest ..... 1321  
   WindowManagerClass ..... 1321  
 OriginalRequest (original database request)  
   ..... 1321

- OutputFileQueue ..... 1024  
 Overview ..... 313, 687  
 PageDown ..... 111  
   ASCIIViewerClass ..... 111  
 PageDown (scroll down one page) ..... 111  
 PagesAcross ..... 867  
   PrintPreviewClass ..... 867  
 PagesAcross (number of pages displayed horizontally) ..... 867  
 PagesAhead ..... 1249  
   ViewManagerClass ..... 1249  
 PagesAhead (buffered pages) ..... 1249  
 PagesBehind ..... 1249  
   ViewManagerClass ..... 1249  
 PagesBehind (buffered pages) ..... 1249  
 PagesDown ..... 868  
   PrintPreviewClass ..... 868  
 PagesDown (number of vertical thumbnails) ..... 868  
 PageSize ..... 1250  
   ViewManagerClass ..... 1250  
 PageSize (buffer page size) ..... 1250  
 PagesToPrint ..... 868  
 PageUp ..... 112  
   ASCIIViewerClass ..... 112  
 PageUp (scroll up one page) ..... 112  
 Paste (paste text from clipboard) ..... 305  
 Percentile ..... 895  
   ProcessClass ..... 895  
 Percentile (portion of process completed) ..... 895  
 Popup ..... 740  
   ASCIIViewerClass ..... 100  
   BrowseClass ..... 160  
 POPUP ..... 100, 160  
 PopupAsk ..... 741  
 PopupClass ..... 829, 830, 831  
   methods ..... 834  
   properties ..... 833  
 PopupClass Functional Organization--  
   Expected Use ..... 834  
 PopupClass Methods ..... 834  
 PopupClass Overview ..... 829  
 PopupClass Properties ..... 833  
 Position  
   FileManagerClass ..... 635  
 POSITION ..... 635  
 Position (return the current record position) ..... 635  
 PostCompleted ..... 1340  
   WindowManagerClass ..... 1340  
 PostCompleted (initiates final Window processing) ..... 1340  
 PostEvent ..... 742  
 PostNewSelection ..... 190  
   BrowseClass ..... 190  
 PostNewSelection (post an EVENT NewSelection to the browse list) ..... 190  
 Preview  
   ReportManagerClass ..... 1025  
 PREVIEW ..... 1025  
 Preview (PrintPreviewClass object) ..... 1025  
 Preview preview a Crystal Report ..... 282  
 PreviewQueue ..... 1025  
   ReportManagerClass ..... 1025  
 PreviewQueue (report metafile pathnames) ..... 1025  
 Previous  
   BrowseClass ..... 191  
   FileManagerClass ..... 644  
   ViewManagerClass ..... 1267  
 PREVIOUS ..... 191, 644, 1267  
 Previous (get previous record in sequence) ..... 644  
 Previous (get the previous browse item) .191  
 Previous (get the previous element) ..... 1267  
 Primary  
   ViewManagerClass ..... 1250  
   WindowManagerClass ..... 1321  
 PRIMARY ..... 1250, 1321  
 Primary (RelationManager object) ..... 1321  
 Primary (the primary file RelationManager ) ..... 1250  
 PrimeAutoInc ..... 645, 646  
   FileManagerClass ..... 645  
 PrimeAutoInc (prepare an autoincremented record for adding) ..... 645  
 PrimeFields ..... 647, 1341  
   FileManagerClass ..... 647  
   WindowManagerClass ..... 1341  
 PrimeFields (a virtual to prime fields) ..... 647  
 PrimeFields (a virtual to prime form fields) ..... 1341  
 PrimeRecord ..... 648, 1268

FileManagerClass .....	648	QueryFromVisual .....	946
ViewManagerClass .....	1268	QFC (reference to the QueryFormClass)	946
PrimeRecord (prepare a record for adding)		QFC (reference to the QueryListClass) ...	972
.....	648, 1268	QKCurrentQuery .....	910
PrimeUpdate .....	1342	QKCurrentQuery ( popup menu choice )	910
WindowManagerClass .....	1342	QKIcon .....	910
PrimeUpdate (update or prepare for update)		QKIcon ( icon for popup submenu ).....	910
.....	1342	QKMenulcon .....	910
Printer		QKMenulcon ( icon for popup menu ) ....	910
ASCIIViewerClass.....	100	QKSupport .....	911
PRINTER .....	100	QKSupport ( quickqbe flag) .....	911
PrintGraph .....	744	Query retrieve or set the SQL data query	284
PrintLines.....	85	QueryClass .....	905
ASCIIPrintClass .....	85	AddItem .....	913
PrintLines (print or preview specified lines)		Ask .....	915
.....	85	GetFilter.....	918
PrintPreview .....	82	GetLimit.....	920
AsciiPrintClass .....	82	Init.....	921
PrintPreviewClass .861, 862, 863, 864, 1017		Kill.....	922
methods .....	871	methods.....	912
properties .....	866	properties .....	910
PrintPreviewClass Functional Organization--		Reset .....	923
Expected Use.....	871	SetLimit .....	926
PrintPreviewClass Methods.....	871	QueryClass ABC Template Implementation	
PrintPreviewClass Overview .....	861	.....	906
PrintPreviewClass Properties .....	866	QueryClass Concepts .....	905
PrintReport .....	1041	QueryClass Conceptual Example.....	907
Process.....	1026	QueryClass Functional Organization--	
ReportManagerClass .....	1026	Expected Use .....	912
Process (ProcessClass object) .....	1026	QueryClass Methods .....	912
ProcessArc .....	803	QueryClass Overview .....	905
ProcessChord.....	805	QueryClass Properties.....	910
ProcessClass		QueryClass Relationship to Other	
methods .....	897	Application Builder Classes.....	905
properties .....	895	QueryClass Source Files .....	906
ProcessClass Functional Organization--		QueryControl (query button).....	1026
Expected Use.....	897	QueryFormClass.....	931
ProcessClass Methods.....	897	Ask .....	938
ProcessClass Overview.....	891	Init.....	939
ProcessClass Properties .....	895	Kill.....	940
ProcessResultFiles.....	1042	methods.....	937
PText .....	896	properties .....	936
ProcessClass .....	896	QueryFormClass ABC Template	
PText (progress control number).....	896	Implementation.....	932
QC .....	986	QueryFormClass Concepts .....	931
QC (reference to the QueryClass).....	986	QueryFormClass Conceptual Example ...	933
QFC .....	972		

- QueryFormClass Functional Organization--
  - Expected Use.....937
- QueryFormClass Methods.....937
- QueryFormClass Overview .....931
- QueryFormClass Properties.....936
- QueryFormClass Relationship to Other
  - Application Builder Classes .....931
- QueryFormClass Source Files .....932
- QueryFormVisual.....941
  - Init .....949
  - QFC.....946
  - TakeAccepted .....952
  - TakeCompleted.....953
- QueryFormVisual ABC Template
  - Implementation .....941
- QueryFormVisual Concepts .....941
- QueryFormVisual Conceptual Example ..943
- QueryFormVisual Functional Organization--
  - Expected Use.....947
- QueryFormVisual Overview.....941
- QueryFormVisual Relationship to Other
  - Application Builder Classes .....941
- QueryFormVisual Source Files .....942
- QueryListClass .....957
- QueryListClass Concepts .....957
- QueryListClass Methods .....963
- QueryListClass Properties.....962
- QueryListClass Source Files .....958
- QueryListClass--Conceptual Example ....959
- QueryListClass--Functional Organization--
  - Expected Use.....963
- QueryListClass--Overview.....957
- QueryListClass--Relationship to Other
  - Application Builder Classes .....957
- QueryListVisual .....967
- QueryListVisual Concepts .....967
- QueryListVisual Methods.....974
- QueryListVisual Properties .....972
- QueryListVisual Source Files .....967
- QueryListVisual--ABC Template
  - Implementation .....967
- QueryListVisual--Conceptual Example....969
- QueryListVisual--Functional Organization--
  - Expected Use.....974
- QueryListVisual--Overview .....967
- QueryListVisual--Relationship to Other
  - Application Builder Classes .....967
- QueryVisualClass Overview .....985
- QueryVisualClass Methods .....987
- QueryVisualClass Properties.....986
- QuickScan.....163
  - BrowseClass .....163
- READONLY .....343
- ReadOnly ( edit-in-place control is read-only).....343
- RealList .....235
  - BufferedPairsClass .....235
- RealList (recognized field pairs) .....235
- RECORDS .....192
- Records(return number of records) .....223
- RecordsProcessed .....896
  - ProcessClass .....896
- RecordsProcessed (number of elements processed).....896
- RecordsToProcess .....896
  - ProcessClass .....896
- RecordsToProcess (number of elements to process).....896
- Redo (reapply action) .....306
- referential integrity
  - enforcement of .....993
- Refresh .....745
- refresh/redisplay ABC BrowseBoxes.....172
- RelationManager .....993, 994, 995
  - properties .....998
- RelationManager Functional Organization--
  - Expected Use.....999
- RelationManager Overview .....993
- RelationManager Properties.....998
- Relationship to Other Application Builder Classes.....221, 225, 313, 325, 333, 337, 377, 531, 687, 753, 769, 773, 1095, 1159, 1167, 1277
- RemoveDuplicatesFlag (remove duplicate data) .....557
- RemoveErrors.....511
  - ErrorClass .....511
- RemoveErrors (remove or restore recognized errors) .....511
- RemoveItem(remove WindowComponent object).....1343
- Replace (find and replace search).....306
- Report
  - ReportManagerClass .....1027

- REPORT ..... 1027  
 Report (the managed REPORT) ..... 1027  
 ReportManager Concepts ..... 1017  
 ReportManager Functional Organization--  
   Expected Use ..... 1031  
 ReportManager Methods ..... 1031  
 ReportManager Overview ..... 1017  
 ReportManager Properties ..... 1022  
 ReportManagerClass  
   methods ..... 1031  
   properties ..... 1022  
 ReportTarget ..... 1027  
 Repost ..... 459  
 Repost (event synchronization) ..... 459  
 RepostField ..... 459  
 RepostField (event synchronization field) 459  
 REQ ..... 460  
 Req (database request) ..... 460  
 Request ..... 1223, 1322  
   ToolBarUpdateClass ..... 1223  
   WindowManagerClass ..... 1322  
 Request (database request) ..... 1322  
 Request (requested database operation)  
   ..... 1223  
 Reset  
   ASCIIFileClass ..... 74  
   ASCIIViewerClass ..... 113  
   ConstantClass ..... 263  
   EditMultiSelectClass ..... 438  
   FilterLocatorClass ..... 817  
   QueryClass ..... 923  
   ViewManagerClass ..... 1269  
   WindowManagerClass ..... 1344  
   WindowResizeClass ..... 1298  
 RESET . 74, 113, 263, 903, 989, 1269, 1298,  
 1344  
 Reset ( reset the dialog for display  
   QueryVisualClass ) ..... 989  
 Reset (reset the ASCIIFileClass object) .... 74  
 Reset (reset the ASCIIViewerClass object)  
   ..... 113  
 Reset (reset the locator for next search) .817  
 Reset (reset the object to the beginning of  
   the constant data) ..... 263  
 Reset (reset the QueryClass object) ..... 923  
 Reset (reset the view position) ..... 1269  
 Reset (reset the window for display) ..... 1344  
 Reset (resets the WindowResizeClass  
   object)..... 1298  
 reset ABC BrowseBoxes ..... 172  
 Reset(reset object's data) ..... 1281  
 ResetButton (synchronize a toolbar control  
   with a corresponding browse control) ..231  
 ResetColumn ..... 469  
 ResetColumn (reset edit-in-place object to  
   selected field) ..... 469  
 ResetFromAsk ..... 193  
   BrowseClass ..... 193, 194  
 ResetFromAsk (reset browse after update)  
   ..... 193  
 ResetFromBrowse(synchronize toolbar  
   controls with browse controls).....232  
 ResetFromBuffer ..... 195  
   BrowseClass ..... 195  
 ResetFromBuffer (fill queue starting from  
   record buffer)..... 195  
 ResetFromFile ..... 196  
   BrowseClass ..... 196  
 ResetFromFile (fill queue starting from file  
   POSITION) ..... 196  
 ResetFromQuery ..... 950, 977  
 ResetFromQuery ( reset the  
   QueryFormVisual object ) ..... 950  
 ResetFromQuery ( reset the QueryList  
   Visual object ) ..... 977  
 ResetFromView ..... 197  
   BrowseClass ..... 197  
 ResetFromView (reset browse from current  
   result set)..... 197  
 ResetHistory(clear History structure)..... 512  
 ResetOnGainFocus ..... 1323  
   WindowManagerClass ..... 1323  
 ResetOnGainFocus (gain focus reset flag)  
   ..... 1323  
 ResetQueue..... 198, 566, 583  
   BrowseClass ..... 198  
   FileDropClass ..... 583  
   FileDropComboClass ..... 566  
 ResetQueue (fill filedrop queue)..... 583  
 ResetQueue (fill or refill queue) ..... 198  
 ResetQueue (refill the filedrop queue)..... 566  
 ResetRequired(determine if screen refresh  
   needed) ..... 1282  
 ResetResets ..... 199

- BrowseClass ..... 199
- ResetResets (copy the Reset fields) ..... 199
- ResetSort..... 200
  - BrowseClass ..... 200
- ResetSort (apply sort order to browse) ... 200
- Resize ..... 745
  - WindowResizeClass ..... 1299
- RESIZE ..... 1299
- Resize (resize and reposition controls) . 1299
- Resize (WindowResize object)..... 1323
- ResizeControls (used internally) ..... 306
- Resizer..... 986
- Resizer (reference to the
  - WindowResizeClass QueryVisualClass)
    - ..... 986
- Response..... 1324
  - WindowManagerClass ..... 1324
- Response (response to database request)
  - ..... 1324
- Restore ..... 849, 924
  - PopupClass..... 849
- Restore ( retrieve saved query ) ..... 924
- Restore (restore a saved menu)..... 849
- RestoreBuffer ..... 650
- RestoreBuffers..... 1270
- RestoreField ..... 1345
  - WindowManagerClass ..... 1345
- RestoreField (restore field to last saved
  - value) ..... 1345
- RestoreFile ..... 651
  - FileManagerClass ..... 651
- RestoreFile (restore a previously saved file
  - state) ..... 651
- RestoreWindow ..... 1300
  - WindowResizeClass ..... 1300
- RestoreWindow (restore window to initial
  - size)..... 1300
- RetainRow ..... 164
  - BrowseClass ..... 164
- ReturnFromDrillDown ..... 746
- RightIndent (indent the current or selected
  - paragraph) ..... 307
- Root ..... 1151
  - StepStringClass ..... 1151
- Root (the static portion of the step) ..... 1151
- Run
  - WindowManagerClass ..... 1346
- RUN ..... 470, 1346, 1347
- Run (run the EIPManager)..... 470
- Run (run this procedure or a subordinate
  - procedure) ..... 1346
- Save ..... 850, 925, 1012
  - PopupClass ..... 850
  - RelationManagerClass ..... 1012
- Save ( save a query ) ..... 925
- Save (copy the current record and any
  - related records) ..... 1012
- Save (save a menu for restoration) ..... 850
- SaveAsGraph ..... 747
- SaveBuffer ..... 652
  - FileManagerClass ..... 652
- SaveBuffer (save a copy of the record
  - buffer) ..... 652
- SaveBuffers ..... 1251, 1270
- Saved ..... 1324
  - WindowManagerClass ..... 1324
- Saved (copy of primary file record buffer)
  - ..... 1324
- SaveField (save rich text data to field) .... 307
- SaveFile ..... 653
  - FileManagerClass ..... 653
- SaveFile (save rich text data to file) ..... 308
- SaveFile (save the current file state) ..... 653
- SaveGraph..... 747
- SaveHistory..... 1348
  - WindowManagerClass ..... 1348
- SaveHistory (save history fields for later
  - restoration) ..... 1348
- SaveOnChangeAction ..... 1349
- SaveOnInsertAction ..... 1350
- ScrollEnd..... 201
  - BrowseClass ..... 201
- ScrollEnd (scroll to first or last item) ..... 201
- ScrollOne ..... 202
  - BrowseClass ..... 202
- ScrollOne (scroll up or down one item) ... 202
- ScrollPage..... 203
  - BrowseClass ..... 203
- ScrollPage (scroll up or down one page). 203
- Searcher ..... 101
  - ASCIIViewerClass ..... 101
- SeekForward ..... 460
- SeekForward (get next field flag)..... 460
- SelColor (color of selected element) ..... 757

SeIE (ending edit position).....	1168	SetAlias.....	1013
Selectable (element selectable flag) .....	757	RelationManagerClass.....	1013
SelectButton .....	1208	SetAlias (set a file alias) .....	1013
ToolBarTargetClass .....	1208	SetCategory (set error category) .....	512
SelectButton (select control number) ....	1208	SetChoice(change selected entry) .....	775
SelectControl .....	164	SetControl(change selected entry) .....	775
BrowseClass .....	164	SetDefault .....	748
SelectFileClass		SetDefaultPages.....	885
properties .....	1088	SetDirtyFlag (set modified flag) .....	309
SelectFileClass Overview.....	1085	SetDynamicControlsAttributes.....	1044
SelectFileClass Properties .....	1088	SetEnabled .....	819
Selecting.....	165	LocatorClass .....	819
BrowseClass .....	165	SetEnabled (enable or disable the locator control) .....	819
SelectionFormula retrieve or set the Crystal formula .....	285	SetError.....	654
SelectText (select characters).....	308	FileManagerClass .....	654
SelectWholeRecord.....	165	SetError (save the specified error and underlying error state) .....	654
SeIS (starting edit position).....	1168	SetErrors.....	513, 655
Set		ErrorClass .....	513
ConstantClass.....	264	SetErrors (save the error state) .....	513
EntryLocatorClass.....	484	SetFatality.....	514
LocatorClass .....	818	ErrorClass .....	514
StepLocatorClass.....	1135	SetFatality (set severity level for a particular error).....	514
SET.....	264, 484, 818, 1135	SetField.....	515
Set (restart the locator).....	484, 818, 1135	ErrorClass .....	515
Set (set the constant data to process).....	264	SetField (set the substitution value of the %Field macro) .....	515
SetAlerts .....	204, 782, 818, 978, 1351	SetFile.....	516
BrowseClass .....	204	ErrorClass .....	516
EditClass .....	347, 393	SetFile (set the substitution value of the %File macro) .....	516
IncrementalLocatorClass .....	782	SetFilter.....	1271
LocatorClass .....	818	SetFM (determine log file status).....	323
WindowManagerClass .....	1351	SetFocus (give rich text control focus) ....	309
SetAlerts (alert keystrokes for list and locator controls) .....	204	SetHelpFile (set the current HTML Help file name) .....	1164
SetAlerts (alert keystrokes for the edit control		SetId.....	518
QueryListVisual).....	978	ErrorClass .....	518
SetAlerts (alert keystrokes for the edit control) .....	393	SetId (make a specific error current) .....	518
SetAlerts (alert keystrokes for the LIST control) .....	782, 818	SetINIManager.....	882
SetAlerts (alert window control keystrokes) .....	1351	PrintPreviewClass .....	882
SetAlerts (initialize and create child controls) .....	762	SetINIManager (save and restore window coordinates) .....	882
SetAlerts(alert keystrokes for window component) .....	1283	SetItemCheck .....	852
		PopupClass.....	852

- SetItemCheck (set toggle item status) .... 852  
SetItemEnable ..... 853  
    PopupClass..... 853  
SetItemEnable (set item status) ..... 853  
SetKey ..... 656  
    FileManagerClass ..... 656  
SetKey (set current key) ..... 656  
SetLevel..... 854  
    PopupClass..... 854  
SetLevel (set menu item level) ..... 854  
SetLimit..... 1110, 1129, 1143, 1157  
    QueryClass ..... 926, 927  
    StepClass..... 1110  
    StepLongClass..... 1129  
    StepRealClass ..... 1143  
    StepStringClass ..... 1157  
SetLimit (set search values) ..... 926  
SetLimit (set smooth data distribution) . 1110,  
    1129, 1143, 1157  
SetLimitNeeded ..... 1111, 1158  
    StepClass..... 1111  
    StepStringClass ..... 1158  
SetLimitNeeded (return static/dynamic  
    boundary flag) ..... 1158  
SetLine..... 75, 114  
    ASCIIFileClass ..... 75  
    ASCIIViewerClass..... 114  
SetLine (a virtual to position the file) ..... 75  
SetLine (position to specific line) ..... 114  
SetLineRelative ..... 115  
    ASCIIViewerClass..... 115  
SetLineRelative (move n lines)..... 115  
SetLocatorField ..... 205  
SetLocatorFromSort ..... 205  
SetMask..... 1093  
    SelectFileClass ..... 1093  
SetMask (set file dialog file masks) ..... 1093  
SetName..... 657  
    FileManagerClass ..... 657  
SetName (set current filename) ..... 657  
SetOption (set fuzzymatch options) ..... 690  
SetOrder ..... 1273  
    ViewManagerClass ..... 1273  
SetOrder (replace a sort order) ..... 1273  
SetParentControl ..... 1301  
SetParentDefaults ..... 1302  
    WindowResizeClass ..... 1302  
SetParentDefaults (set default parent  
    controls)..... 1302  
SetPercentile ..... 76  
    ASCIIFileClass ..... 76  
SetPercentile (set file to relative position) .76  
SetPosition  
    PrintPreviewClass ..... 883  
SETPOSITION..... 883  
SetPosition (set initial preview window  
    coordinates) ..... 883  
SetProcedureName ..... 519  
SetProcedureName ( stores procedure  
    names..... 519  
SetProgressLimits..... 903  
SetQueueRecord ..... 206, 584  
    BrowseClass ..... 206  
    FileDropClass..... 584  
SetQueueRecord (copy data from file buffer  
    to queue buffer) ..... 206, 584  
SetQuickPopup ..... 928  
SetQuickPopup ( add QuickQBE to browse  
    popup ) ..... 928  
SetQuickScan ..... 1014  
    RelationManagerClass..... 1014  
SetQuickScan (enable QuickScan on a file  
    and any related files) ..... 1014  
SetReadOnly ..... 348, 394  
SetReadOnly (set edit control to read-only  
    EditDropClass) ..... 394  
SetReadOnly (set edit control to read-only)  
    ..... 348  
SetReportTarget ..... 1043  
SetResponse ..... 1352  
    WindowManagerClass ..... 1352  
SetResponse (OK or Cancel the window)  
    ..... 1352  
SetShadow..... 485  
SetShadow (update shadow value) ..... 819  
SetShadow(set shadow value) ..... 485  
SetSort ..... 207, 1274  
    BrowseClass ..... 207  
    ViewManagerClass ..... 1274  
SetSort (apply a sort order to the browse)  
    ..... 207  
SetSort (set the active sort order)..... 1274  
SetStaticControlsAttributes ..... 1043  
SetStrategy ..... 1303

WindowResizeClass .....	1303	ShowToolBarTips show tips on preview	
SetStrategy (set control resize strategy)	1303	window toolbar .....	288
SetTarget		ShowTopic (display a help topic) .....	1166
ToolBarClass .....	1182	Silent Database Operations	
SETTARGET .....	1182	FileManagerClass .....	601
SetTarget (sets the active target) .....	1182	SkipHeldRecords .....	600
SetText .....	855, 951	FileManagerClass .....	600
PopupClass .....	855	SkipHeldRecords (HELD record switch) ..	600
SetText ( set prompt text QueryFormVisual )		SkipPreview .....	1027
.....	951	ReportManagerClass .....	1027
SetText (place text into rich text control) .	310	SkipPreview (print rather than preview) .	1027
SetText (set menu item text) .....	855	Sort	
SetThread (read triggerqueue) .....	330	BrowseClass .....	166
SetTopic (set the current HTML Help file		SORT .....	166
topic) .....	1165	Sort (browse sort information) .....	166
SetTranslator .....	116, 857	SortChars .....	1151
ASCIIViewerClass .....	116	StepStringClass .....	1151
PopupClass .....	857	SortChars (valid sort characters) .....	1151
SetTranslator (set run-time translator) ..	116,	StandardBehavior Class Concepts .....	1095
857		StandardBehavior Methods .....	1097
Setup .....	95	StandardBehavior Overview .....	1095
ASCIISearchClass .....	95	StandardBehavior Properties .....	1096
Setup (set search constraints) .....	95	StandardBehavior Source Files .....	1095
SetupAdditionalFeqs (initialize additional		StandardErrorLogClass Overview .....	1099
control properties) .....	828	StandardErrorLogClass Properties .....	1100
SetViewPosition(set VIEW position) .....	223	StandardErrorLogClass Source Files ...	1099
SetZoom .....	884	StartAtCurrent .....	167
SetZoomPercentile		BrowseClass .....	167
PrintPreviewClass .....	884	StartAtCurrent (initial browse position) ...	167
SetZoomPercentile (set user or standard		StepClass .....	1103, 1104
zoom factor) .....	884	properties .....	1105
Shadow .....	481	StepClass Overview .....	1103
EntryLocatorClass .....	481	StepClass Properties .....	1105
Shadow (the search value) .....	481	StepCustomClass .....	1113, 1114, 1115
ShowControl (hide/unhide RTF control) ..	311	GetPercentile .....	1153
ShowDocumentTips show tips on		GetValue .....	1154
docuement in the preview window .....	286	Init .....	1155
ShowIndex (open the HTML Help index tab)		methods .....	1118
.....	1165	properties .....	1117
ShowOnField .....	748	StepCustomClass Methods .....	1118
ShowOnStatusBar .....	749	StepCustomClass Overview .....	1113
ShowReportControls show print controls	287	StepCustomClass Properties .....	1117
ShowSearch (open the HTML Help search		StepLocatorClass	
tab) .....	1166	methods .....	1135
ShowTOC (open the HTML Help contents		properties .....	1134
tab) .....	1166	StepLocatorClass Methods .....	1135
ShowToolBarTips .....	288	StepLocatorClass Overview .....	1131

- StepLocatorClass Properties ..... 1134
- StepLongClass ..... 1123, 1124
  - methods ..... 1127
  - properties ..... 1126
- StepLongClass Methods ..... 1127
- StepLongClass Overview ..... 1123
- StepLongClass Properties ..... 1126
- StepRealClass ..... 1137, 1138
  - methods ..... 1141
  - properties ..... 1140
- StepRealClass Methods ..... 1141
- StepRealClass Overview ..... 1137
- StepRealClass Properties ..... 1140
- StepStringClass ..... 1145, 1146, 1147, 1148
  - methods ..... 1153
  - properties ..... 1150
- StepStringClass Methods ..... 1153
- StepStringClass Overview ..... 1145
- StepStringClass Properties ..... 1150
- Style (font style) ..... 825
- SubsString ..... 520
  - ErrorClass ..... 520
- SubsString (resolves error message macros) ..... 520
- SyncGroup (initialize GROUP field properties) ..... 762
- SyncImageQueue ..... 885
- TAB ..... 460
- Tab (action on a tab key) ..... 460
- TabAction ..... 168
  - BrowseClass ..... 168
- TabAction (edit-in-place action on tab key) ..... 168
- TagHTMLHelp ABC Template Implementation ..... 1159
- TagHTMLHelp Class Concepts ..... 1159
- TagHTMLHelp Methods ..... 1160
- TagHTMLHelp Source Files ..... 1159
- TagHTMLHelpOverview ..... 1159
- Take ..... 929
- Take ( process QuickQBE popup menu choice ) ..... 929
- Take (update the log file) ..... 533
- TakeAcceptAll ..... 470
- TakeAccepted 348, 485, 684, 820, 886, 979, 1353
  - LocatorClass ..... 820
  - PrintPreviewClass ..... 886
  - QueryFormVisual ..... 952
  - WindowManager ..... 1353
- TakeAccepted (a virtual to process EVENT Accepted) ..... 1353
- TakeAccepted (handle query dialog EVENT Accepted events
  - QueryFormVisual) ..... 952
  - Accepted events) ..... 979
- TakeAccepted (handle query dialog EVENT Accepted events) ..... 990
- TakeAccepted (process accepted event) ..... 567, 828
- TakeAccepted (process an accepted locator value) ..... 820
- TakeAccepted (process EVENT Accepted events) ..... 886
- TakeAccepted (process window controls) ..... 1170
- TakeAcceptedLocator ..... 208
  - BrowseClass ..... 208
- TakeAcceptedLocator (apply an accepted locator value) ..... 208
- TakeAccepted ..... 990
- TakeAction ..... 471
  - EditMultiSelectClass ..... 439
- TakeAction (process MultiSelect dialog action) ..... 439
- TakeAction (process edit-in-place action) ..... 471
- TakeBenign ..... 521
  - ErrorClass ..... 521
- TakeBenign (process benign error) ..... 521
- TakeCloseEvent ..... 1045, 1354
  - ReportManagerClass ..... 1045
  - WindowManagerClass ..... 1354
- TakeCloseEvent (a virtual to Cancel the window) ..... 1354
- TakeCloseEvent (a virtual to process EVENT CloseWindow) ..... 1045
- TakeCompleted ..... 142, 472, 980, 1356
  - QueryFormVisual ..... 953
- TakeCompleted (complete the query dialog QueryFormVisual) ..... 953
- TakeCompleted (complete the query dialog) ..... 980
- TakeCompleted (process completion of edit

- EIPManagerClass).....472
- TakeCompleted (process completion of edit)  
..... 142
- TakeError.....522
- ErrorClass .....522
- TakeError (process specified error).....522
- TakeEvent..... 117, 209, 395, 448, 473, 567,  
585, 749, 887, 981, 1183, 1192, 1211,  
1212, 1226, 1358
- ASCIIViewerClass.....117
- BrowseClass .....209
- EditClass .....349
- EditColorClass .....376, 415
- EditFontClass.....426
- EditMultiSelectClass .....442
- FileDropClass .....585
- FileDropComboClass.....567
- PrintPreviewClass .....887
- ToolBarClass.....1183
- ToolBarListBoxClass .....1192
- ToolBarTargetClass .....1211
- ToolBarUpdateClass .....1226
- WindowManagerClass .....1358
- TakeEvent (process edit-in-place events  
  QueryListVisual).....981
- TakeEvent (process edit-in-place events)  
.....349, 415
- TakeEvent (process window specific  
  events) .....473
- TakeEvent (a virtual to process all events)  
.....1358
- TakeEvent (convert toolbar events) .....1192,  
1211, 1226
- TakeEvent (process ACCEPT loop event)  
.....117
- TakeEvent (process all events).....887
- TakeEvent (process the current ACCEPT  
  loop event) .....209, 567, 585, 763
- TakeEvent (process toolbar event) .....1183
- TakeEvent (process window events).....767
- TakeEvent(process the current ACCEPT  
  loop event) .....1283
- TakeEvent(process the current event) ....232
- TakeEventofParent.....750
- TakeFatal.....523
- ErrorClass .....523
- TakeFatal (process fatal error) .....523
- TakeFieldEvent..... 474, 888, 954, 982, 991,  
1359
- PrintPreviewClass .....888
- WindowManager .....1359
- TakeFieldEvent (process field specific  
  events).....474
- TakeFieldEvent (a virtual to process field  
  events QueryFormVisual) .....954
- TakeFieldEvent (a virtual to process field  
  events QueryListVisual) .....982
- TakeFieldEvent (a virtual to process field  
  events QueryVisualClass).....991
- TakeFieldEvent (a virtual to process field  
  events)..... 888, 1359
- TakeFocusLoss .....475
- TakeFocusLoss (a virtual to process loss of  
  focus).....475
- TakeKey..... 210, 486, 783, 820, 1136
- BrowseClass .....210
- IncrementalLocatorClass .....783
- LocatorClass .....820
- StepLocatorClass .....1136
- TakeKey (process an alerted keystroke) 210,  
783, 820, 1136
- TakeNewSelection..... 211, 476, 568, 586
- BrowseClass .....211
- FileDropClass.....586
- FileDropComboClass .....568
- WindowManagerClass .....1360
- TakeNewSelection (reset edit-in-place  
  column  
  EIPManagerClass) .....476
- TakeNewSelection (reset edit-in-place  
  column).....143
- TakeNewSelection (a virtual to process  
  EVENT  
  NewSelection) .....1360
- TakeNewSelection (process a new  
  selection).....211
- TakeNewSelection (process EVENT  
  NewSelection events) .....568, 586
- TakeNoRecords .....1046
- ReportManagerClass .....1046
- TakeNoRecords (process empty report)1046
- TakeNotify.....524
- ErrorClass .....524
- TakeNotify (process notify error) .....524

- TakeOther .....525
  - ErrorClass .....525
- TakeOther (process other error).....525
- TakeProgram .....526
  - ErrorClass .....526
- TakeProgram (process program error)...526
- TakeRecord .....904
  - ProcessClass .....904
- TakeRecord (a virtual to process each report record).....904
- TakeRecord(process each record).....1046
- TakeRejected .....1361
  - WindowManagerClass .....1361
- TakeRejected (a virtual to process EVENT Rejected).....1361
- TakeScroll.....212
  - BrowseClass .....212
- TakeScroll (process a scroll event) .....212
- TakeSelected.....1362
  - WindowManagerClass .....1362
- TakeSelected (a virtual to process EVENT Selected).....1362
- TakeToolbar .....1202, 1213, 1227
  - ToolbarListBoxClass .....1193
  - ToolbarReltreeClass .....1202
  - ToolbarTargetClass .....1213
  - ToolbarUpdateClass .....1227
- TakeToolbar (assume control of the toolbar) .....1193
- TakeToolbar (assume control of the toolbar) .....1202, 1213, 1227
- TakeUser .....527
  - ErrorClass .....527
- TakeUser (process user error) .....527
- TakeVCRScroll .....213
  - BrowseClass .....213
- TakeVCRScroll (process a VCR scroll event) .....213
- TakeWindowEvent.....889, 992, 1047, 1363
  - PrintPreviewClass .....889
  - ReportManager .....1047
- TakeWindowEvent (process non-field events) .....889
- TakeWindowEvent (a virtual to process non-field events).....1047
- TargetSelector .....1028
- TargetSelectorCreated .....1028
- TerminatorField.....253
- TerminatorInclude.....253
- TerminatorValue (end of data marker) ....254
- TestLen .....1152
  - StepStringClass .....1152
- TestLen (length of the static step portion) .....1152
- TextWindowClass Concepts.....1167
- TextWindowClass Methods .....1169
- TextWindowClass Overview .....1167
- TextWindowClass Properties.....1168
- TextWindowClass Source Files .....1167
- Throw .....528, 659
  - ErrorClass .....528
  - FileManagerClass .....659
- Throw (process specified error).....528
- Throw (pass an error to the error handler for processing).....659
- ThrowFile .....529
  - ErrorClass .....529
- ThrowFile (set value of %File then process error).....529
- ThrowMessage .....530, 660
  - ErrorClass .....530
  - FileManagerClass .....660
- ThrowMessage (pass an error and text to the error handler) .....660
- TimeOut .....1252
  - ViewManagerClass .....1252
- TimeOut (buffered pages freshness).....1252
- TimeSlice .....1029
  - ReportManagerClass .....1029
- TimeSlice (report resource usage) .....1029
- Title .....445
  - EditColorClass .....373
  - EditFileClass .....412
  - EditFontClass .....423
  - EditMultiSelectClass .....434
- Title (color dialog title text).....373
- Title (font dialog title text).....423, 434
- Title (text dialog title text).....445
- Toolbar
  - BrowseClass .....169
- TOOLBAR.....169
- Toolbar (browse Toolbar object).....169
- ToolbarClass. 1171, 1172, 1173, 1174, 1176 methods.....1178

properties .....	1178	TranslatorClass .....	1241
ToolbarClass Functional Organization--		TranslateWindow (translate text for a	
Expected Use.....	1178	window) .....	1241
ToolbarClass Methods.....	1178	Translator.....	82, 91, 1325
ToolbarClass Overview .....	1171	AsciiPrintClass .....	82
ToolbarItem .....	170	ASCIISearchClass .....	91
BrowseClass .....	170	WindowManager .....	1325
ToolbarItem (browse ToolbarTarget object)		Translator (TranslatorClass object) .....	1325
.....	170	TranslatorClass.....	1229, 1230, 1231, 1232
ToolbarListboxClass .....	1185	methods.....	1235
methods .....	1191	properties .....	1234
properties .....	1190	TranslatorClass Methods.....	1235
ToolbarListboxClass Methods .....	1191	TranslatorClass Overview.....	1229
ToolbarListboxClass Overview .....	1185	TranslatorClass Properties .....	1234
ToolbarListboxClass Properties .....	1190	TriggerQueue (pointer to BFP for field	
ToolbarReltreeClass		changes).....	326
methods .....	1201	TryFetch.....	662, 795
properties .....	1200	FileManagerClass .....	662
ToolbarReltreeClass Methods .....	1201	INIClass.....	795
ToolbarReltreeClass Overview.....	1195	TryFetch (get a value from the INI file) ....	795
ToolbarReltreeClass Properties .....	1200	TryFetch (try to get a specific record by key	
ToolbarTarget .....	1203, 1204	value).....	662
ToolbarTarget Functional Organization--		TryFetchField .....	796
Expected Use.....	1209	INIClass.....	796
ToolbarTarget Overview .....	1203	TryFetchField (return comma delimited INI	
ToolbarUpdateClass .....	1215, 1220	file value) .....	796
methods .....	1225	TryInsert.....	663
properties .....	1223	TryNext .....	664
ToolbarUpdateClass Methods .....	1225	FileManagerClass .....	664
ToolbarUpdateClass Overview.....	1215	TryNext (try to get next record in sequence)	
ToolbarUpdateClass Properties .....	1223	.....	664
ToolTip.....	750	TryOpen .....	665
TopLine.....	101	FileManagerClass .....	665
ASCIIViewerClass.....	101	TryOpen (try to open the file).....	665
ToShowValues .....	751	TryPrevious.....	666
TranslateControl .....	1238	FileManagerClass .....	666
TranslatorClass.....	1238	TryPrevious (try to get previous record in	
TranslateControl (translate text for a control)		sequence).....	666
.....	1238	TryPrimeAutoInc .....	667
TranslateControls .....	1239	FileManagerClass .....	667
TranslatorClass.....	1239	TryPrimeAutoInc (try to prepare an	
TranslateControls (translate text for range of		autoincremented record for adding) ....	667
controls) .....	1239	TryReget .....	669
TranslateString .....	1240	TryTakeToolbar .....	1194, 1213, 1228
TranslatorClass.....	1240	ToolbarListBoxClass .....	1194
TranslateString (translate text) .....	1240	ToolbarTargetClass.....	1213
TranslateWindow.....	1241	ToolbarUpdateClass .....	1228

- TryTakeToolBar (return toolbar control indicator) ..... 1194, 1213, 1228
- TryUpdate ..... 669  
    FileManagerClass ..... 669
- TryUpdate (try to change the current record) ..... 669
- TryValidateField(validate field contents) .670
- Txt (field equate number) ..... 1168
- Undo (undo action) ..... 311
- UniquePosition (check queue for duplicate record by key position)..... 569
- Update  
    EntryLocatorClass..... 487  
    FileManagerClass ..... 671  
    INIClass..... 798  
    RelationManagerClass..... 1015
- UPDATE ..... 487, 671, 797, 798, 1015, 1364
- Update (change the current record) ..... 671
- Update (update record subject to referential constraints) ..... 1015
- Update (update the audit log file buffer) .. 331
- Update (update the locator control and free elements) ..... 487
- Update (write INI file entries) ..... 797
- Update(FileManager)..... 1328
- Update(get VIEW data for the selected item) ..... 1284
- Update(update entry in LIST queue) ..... 224
- UpdateBuffer ..... 214  
    BrowseClass ..... 214
- UpdateBuffer (copy selected item from queue buffer to file buffer)..... 214
- UpdateControl (file update trigger) ..... 757
- UpdateControl(updates the edit-in-place entry control) ..... 983
- UpdateControlEvents ..... 757
- UpdateFields ..... 955, 983
- UpdateFields ( process query values ) .. 955, 983
- UpdateQuery (set default query interface) ..... 215
- UpdateResets ..... 216  
    BrowseClass ..... 216
- UpdateResets (copy reset fields to file buffer)..... 216
- UpdateThumb ..... 217  
    BrowseClass ..... 217
- UpdateThumb (position the scrollbar thumb) ..... 217
- UpdateThumbFixed ..... 218  
    BrowseClass ..... 218
- UpdateThumbFixed (position the scrollbar fixed thumb) ..... 218
- UpdateViewRecord ..... 219  
    BrowseClass ..... 219
- UpdateViewRecord (get view data for the selected item)..... 219
- UpdateWindow ..... 220, 487, 685, 821  
    BrowseClass ..... 220  
    EntryLocatorClass ..... 487  
    FilterLocatorClass ..... 685  
    LocatorClass ..... 821
- UpdateWindow (apply the search criteria) ..... 685
- UpdateWindow (redraw the locator control with its current value) ..... 821
- UpdateWindow (redraw the locator control) ..... 487
- UpdateWindow (update display variables to match browse)..... 220
- UpdateWindow(update window controls) ..... 1284
- UseField ..... 558  
    FileDropComboClass ..... 558
- UseField (COMBO USE variable) ..... 558
- UseFile ..... 672  
    FileManagerClass ..... 672
- UseFile (use LazyOpen file) ..... 672
- UseLogout (transaction framing flag) ..... 998
- UserPercentile ..... 868  
    PrintPreviewClass ..... 868
- UserPercentile (custom zoom factor) ..... 868
- UseView ..... 1275  
    ViewManagerClass ..... 1275
- UseView (use LazyOpen files) ..... 1275
- ValidateField ..... 674  
    FileManagerClass ..... 674
- ValidateField (validate a field)..... 674
- ValidateFields ..... 675  
    FileManagerClass ..... 675
- ValidateFields (validate a range of fields)675
- ValidateFieldServer(validate field contents) ..... 676
- ValidateLine ..... 77

- ASCIIFileClass .....77  
 ValidateLine (a virtual to implement a filter)  
 .....77  
 ValidateRecord .....587, 677, 1276  
   FileDropClass .....587  
   FileManagerClass .....677  
   ViewManagerClass .....1276  
 ValidateRecord (a virtual to validate records)  
 .....587  
 ValidateRecord (validate all fields) .....677  
 ValidateRecord (validate an element) ...1276  
 ValueEIP(reference to QEditEntryClass) 973  
 VCRRequest.....1325  
   WindowManagerClass .....1325  
 VCRRequest (delayed scroll request) ...1325  
 View  
   ViewManagerClass .....1252  
 VIEW .....1252  
 View (the managed VIEW) .....1252  
 ViewHistory (initiates the view of the current  
 errors).....530  
 ViewManager.....814, 1243, 1244, 1245  
   LocatorClass .....814  
   methods .....1253  
   properties .....1248  
 ViewManager (the locator's ViewManager  
 object) .....814  
 ViewManager Functional Organization--  
   Expected Use.....1253  
 ViewManager Methods .....1253  
 ViewManager Overview.....1243  
 ViewManager Properties .....1248  
 ViewMenu .....859  
   PopupClass.....859  
 ViewMenu (popup menu debugger) .....859  
 VLBProc (retrieve LIST and error history  
 information.) .....768  
 WaitCursor.....1029  
 Who(returns field name) .....224  
 Win (reference to window).....766, 825  
 Window  
   BrowseClass .....171  
 WINDOW .....171, 911  
 Window ( browse window QueryClass ) ..911  
 Window (WindowManager object) .....227  
 WindowComponent Concepts .....1277  
 WindowComponent Methods.....1278  
 WindowComponent Overview .....1277  
 WindowComponent Source Files .....1277  
 WindowManager.... 1305, 1306, 1307, 1308,  
   1310, 1311  
   methods.....1326  
   properties .....1313  
 WindowManager Functional Organization--  
   Expected Use .....1326  
   WindowManager Methods .....1326  
   WindowManager Overview .....1305  
   WindowManager Properties .....1313  
 WindowPosSet .....869  
   PrintPreviewClass .....869  
 WindowPosSet (use a non-default initial  
 preview window position) .....869  
 WindowResizeClass ..... 1285, 1286, 1287  
   methods.....1290  
   properties .....1289  
 WindowResizeClass Functional  
   Organization--Expected Use .....1290  
 WindowResizeClass Methods .....1290  
 WindowResizeClass Overview .....1285  
 WindowResizeClass Properties.....1289  
 WindowSizeSet.....869  
   PrintPreviewClass .....869  
 WindowSizeSet (use a non-default initial  
 preview window size) .....869  
 WindowTitle .....1089  
   SelectFileClass .....1089  
 WindowTitle (file dialog title text) .....1089  
 WMFParser.....1029  
 Zoom.....1030  
   ReportManagerClass .....1030  
 Zoom (initial report preview magnification)  
   .....1030  
 ZoomIndex .....870  
   PrintPreviewClass .....870  
 ZoomIndex (index to applied zoom factor)  
   .....870



