

***CLARION 5***

**Application  
Handbook**

**COPYRIGHT 1997, 1998 by TopSpeed Corporation  
All rights reserved.**

This publication is protected by copyright and all rights are reserved by TopSpeed Corporation. 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 TopSpeed Corporation.

This publication supports Clarion 5. It is possible that it may contain technical or typographical errors. TopSpeed Corporation provides this publication “as is,” without warranty of any kind, either expressed or implied.

**TopSpeed Corporation**  
150 East Sample Road  
Pompano Beach, Florida 33064  
(954) 785-4555

**Trademark Acknowledgements:**

TopSpeed® is a registered trademark of TopSpeed Corporation.

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.

# **TABLE OF CONTENTS**

<b>FOREWORD</b>	<b>37</b>
Welcome	37
<b>Documentation Conventions</b>	<b>38</b>
Typeface Conventions .....	38
Keyboard Conventions .....	38
Other Conventions .....	38
 <b>PART I—APPLICATION BUILDER CLASS TEMPLATES .....</b>	 <b>39</b>
<b>1 - TEMPLATE OVERVIEW</b>	<b>41</b>
<b>What is a Template</b>	<b>41</b>
Clarion Templates and Application Builder Class (ABC) Templates .....	42
ABC Templates and Code Generation .....	42
ABC Templates and the ABC Library .....	43
Browse-Form Application Paradigm .....	45
<b>ABC Templates and SQL</b>	<b>46</b>
<b>Global ABC Template Settings</b>	<b>47</b>
General Tab Options .....	47
File Control Tab Options .....	49
Individual File Overrides Tab Options .....	53
External Module Options Tab .....	54
Classes Tab Options—Global .....	54
<b>Classes Tab Options—Local</b>	<b>60</b>
<b>ABC Compliant Classes</b>	<b>64</b>
<b>Global ABC Embed Points</b>	<b>65</b>
<b>Using ABC Templates to Derive Classes</b>	<b>67</b>
Why the Templates Derive Classes .....	67
Deriving with Embed Points .....	68
Deriving with Classes Tab .....	68
 <b>2 - WIZARDS AND UTILITY TEMPLATES</b>	 <b>69</b>
<b>Code Generation Wizards</b>	<b>69</b>
<b>Application Wizards</b>	<b>70</b>

Quick Start Wizard .....	70
Application Wizard .....	72
<b>Procedure Wizards</b>	<b>76</b>
Browse Wizard .....	76
Form Wizard .....	78
Report Wizard .....	80
<b>Dictionary Print Wizard</b>	<b>82</b>
<b>Optimizing the Wizards</b>	<b>83</b>
File Options .....	83
Alias Options .....	83
Field Options .....	83
Key Options .....	84
Relation Options .....	84
Naming Conventions .....	84
Using Default Window Controls .....	85
<b>3 - PROCEDURE TEMPLATES</b>	<b>87</b>
<b>Overview</b>	<b>87</b>
Procedures and Procedure Templates .....	87
Procedures as Containers .....	88
Inter-Procedure Communication .....	89
<b>Window Procedure Templates</b>	<b>90</b>
Window Template .....	90
Browse Template .....	92
Form Template .....	93
Frame Template .....	95
Menu Template .....	97
Process Template .....	97
Report Template .....	101
Splash Template .....	106
Viewer Template .....	107
<b>Other Procedure Templates</b>	<b>109</b>
External Template .....	109
Source Template .....	109
<b>4 - CONTROL TEMPLATES</b>	<b>111</b>
<b>Overview</b>	<b>111</b>
Adding Control Templates .....	111

<b>Read-Only Browse Templates</b>	<b>112</b>
ASCIIViewControl .....	112
ASCIIPrintButton .....	113
ASCIISearchButton .....	113
<b>Read-Write Browse Templates</b>	<b>114</b>
BrowseBox Overview .....	114
Scrolling with a Page-loaded BrowseBox .....	115
BrowseBox Options .....	116
BrowsePrintButton .....	127
BrowsePublishButton .....	128
BrowseQueryButton .....	129
BrowseSelectButton .....	132
BrowseToolboxButton .....	132
BrowseUpdateButtons .....	133
RelationTree Overview .....	136
RelationTree Options .....	138
RelationTreeUpdateButtons .....	140
<b>Other Window Control Templates</b>	<b>141</b>
CancelButton .....	141
CloseButton .....	141
DOSFileLookup .....	141
FieldLookupButton .....	144
FileDrop .....	144
FileDropCombo .....	150
FrameBrowseControl .....	155
PauseButton .....	157
SaveButton .....	158
<b>Report Control Templates</b>	<b>162</b>
ReportDateStamp .....	162
ReportTimeStamp .....	163
ReportPageNumber .....	163
<b>5 - CODE AND EXTENSION TEMPLATES</b>	<b>165</b>
<b>Code Templates</b>	<b>165</b>
CallABCMethod .....	165
CallProcedureAsLookup .....	166
CloseCurrentWindow .....	166
ControlValueValidation .....	166

DisplayPopupMenu .....	167
InitiateThread .....	169
LookupNonRelatedRecord .....	169
ResizeSetStrategy .....	170
SelectToolBarTarget .....	172
SetABCProperty .....	173
SetProperty .....	173
<b>Extension Templates</b> .....	<b>174</b>
AsciiViewInListBox .....	174
DateTimeDisplay .....	175
ExtendProgressWindow .....	176
FormVCRControls .....	177
RecordValidation .....	178
ReportChildFiles .....	178
WindowResize .....	180

## **PART II—APPLICATION BUILDER CLASS LIBRARY ..... 185**

### **6 - ABC LIBRARY OVERVIEW 187**

#### **About This Part 187**

#### **Application Builder Class (ABC) Library 187**

  Class Libraries Generally ..... 187

  Application Builder Classes—The ABCs of Rapid Application Development . 187

  ABC Library and the ABC Templates ..... 191

#### **ABC Coding Conventions 193**

  Method Names ..... 193

  Where to Initialize & Kill Objects ..... 194

  Return Values ..... 194

  PRIVATE (undocumented) Items ..... 195

#### **Documentation Conventions 196**

  Reference Item and Syntax Diagram ..... 196

### **7 - ASCIIFILECLASS 199**

#### **Overview 199**

  Relationship to Other Application Builder Classes ..... 199

  ABC Template Implementation ..... 199

  ASCIIFileClass Source Files ..... 200

  Conceptual Example ..... 200

<b>AsciiFileClass Properties</b>	<b>202</b>
ASCIIFile (the ASCII file) .....	202
ErrorMgr (ErrorClass object) .....	202
OpenMode (file access/sharing mode) .....	202
<b>AsciiFileClass Methods</b>	<b>203</b>
Functional Organization—Expected Use .....	203
FormatLine (a virtual to format text) .....	204
GetDOSFilename (let end user select file) .....	205
GetFilename (return the filename) .....	206
GetLastLineNo (return last line number) .....	206
GetLine (return line of text) .....	207
GetPercentile (convert file position to percentage) .....	208
Init (initialize the ASCIIFileClass object) .....	209
Kill (shut down the ASCIIFileClass object) .....	210
Reset (reset the ASCIIFileClass object) .....	211
SetLine (a virtual to position the file) .....	212
SetPercentile (set file to relative position) .....	213
ValidateLine (a virtual to implement a filter) .....	214
<b>8 - ASCIIPRINTCLASS</b>	<b>215</b>
<b>Overview</b>	<b>215</b>
Relationship to Other Application Builder Classes .....	215
ABC Template Implementation .....	215
ASCIIPrintClass Source Files .....	216
Conceptual Example .....	216
<b>AsciiPrintClass Properties</b>	<b>218</b>
FileMgr (AsciiFileClass object) .....	218
PrintPreview (print preview switch) .....	218
Translator (TranslatorClass object) .....	218
<b>AsciiPrintClass Methods</b>	<b>219</b>
Ask (solicit print specifications) .....	219
Init (initialize the ASCIIPrintClass object) .....	219
PrintLines (print or preview specified lines) .....	220
<b>9 - ASCIISEARCHCLASS</b>	<b>221</b>
<b>Overview</b>	<b>221</b>
Relationship to Other Application Builder Classes .....	221
ABC Template Implementation .....	221

ASCIISearchClass Source Files .....	222
Conceptual Example .....	222
<b>AsciiSearchClass Properties</b>	<b>224</b>
Find (search constraints) .....	224
FileMgr (AsciiFileClass object) .....	224
LineCounter (current line number) .....	224
Translator (TranslatorClass object) .....	225
<b>AsciiSearchClass Methods</b>	<b>226</b>
Ask (solicit search specifications) .....	226
Init (initialize the ASCIISearchClass object) .....	227
Next (find next line containing search text) .....	227
Setup (set search constraints) .....	228
<b>10 - ASCIIVIEWERCLASS</b>	<b>229</b>
<b>Overview</b>	<b>229</b>
Relationship to Other Application Builder Classes .....	230
ABC Template Implementation .....	230
ASCIIViewerClass Source Files .....	230
Conceptual Example .....	231
<b>AsciiViewerClass Properties</b>	<b>232</b>
Popup (PopupClass object) .....	232
Printer (ASCIIPrintClass object) .....	232
Searcher (ASCIISearchClass object) .....	232
TopLine (first line currently displayed) .....	233
<b>AsciiViewerClass Methods</b>	<b>234</b>
Functional Organization—Expected Use .....	234
AddItem (program the AsciiViewer object) .....	236
AskGotoLine (go to user specified line) .....	237
DisplayPage (display new page) .....	237
Init (initialize the ASCIIViewerClass object) .....	238
Kill (shut down the ASCIIViewerClass object) .....	240
PageDown (scroll down one page) .....	241
PageUp (scroll up one page) .....	241
Reset (reset the ASCIIViewerClass object) .....	242
SetLine (position to specific line) .....	243
SetLineRelative (move n lines) .....	244
SetTranslator (set run-time translator) .....	245
TakeEvent (process ACCEPT loop event) .....	246

<b>11 - BROWSECLASS</b>	<b>247</b>
<b>Overview</b>	<b>247</b>
BrowseClass Concepts .....	247
Relationship to Other Application Builder Classes .....	247
ABC Template Implementation .....	248
BrowseClass Source Files .....	248
Conceptual Example .....	249
<b>BrowseClass Properties</b>	<b>251</b>
ActiveInvisible (obscured browse list action) .....	251
AllowUnfilled (display filled list) .....	251
ArrowAction (edit-in-place action on arrow key) .....	252
AskProcedure (update procedure) .....	252
ChangeControl (change/edit button) .....	253
DeleteControl (delete button) .....	253
EditList (list of edit-in-place controls) .....	254
EnterAction (edit-in-place action on enter key) .....	254
Fields (managed fields) .....	255
FocusLossAction (edit-in-place action on lose focus) .....	255
HasThumb (vertical scroll bar flag) .....	256
HideSelect (hide select button) .....	256
InsertControl (add/insert button) .....	256
ListControl (browse LIST control) .....	257
ListQueue (browse data queue) .....	257
Loaded (queue loaded flag) .....	257
Popup (popup menu manager) .....	257
PrintControl (print button) .....	258
PrintProcedure (print procedure) .....	258
Query (ad hoc query manager) .....	258
QueryControl (query button) .....	259
QueryShared (query scope flag) .....	259
QuickScan (buffered reads flag) .....	260
RetainRow (highlight bar refresh behavior) .....	260
SelectControl (select button) .....	261
Selecting (select mode only flag) .....	261
Sort (browse sort information) .....	262
StartAtCurrent (initial browse position) .....	263
TabAction (edit-in-place action on tab key) .....	263
Toolbar (browse Toolbar object) .....	264

ToolbarItem (browse ToolbarTarget object) .....	264
ToolControl (toolbox button) .....	265
Window (WindowManager object) .....	265
<b>BrowseClass Methods</b> .....	<b>266</b>
Functional Organization—Expected Use .....	266
AddEditControl (specify custom edit-in-place class) .....	268
AddField (specify a FILE/QUEUE field pair) .....	269
AddLocator (specify a locator) .....	270
AddResetField (set a field to monitor for changes) .....	271
AddSortOrder (specify a browse sort order) .....	272
AddToolbarTarget (set the browse toolbar) .....	273
ApplyRange (refresh browse based on resets and range limits) .....	274
Ask (update selected browse item) .....	275
AskRecord (edit-in-place selected browse item) .....	276
Fetch (get a page of browse items) .....	278
Init (initialize the BrowseClass object) .....	279
Kill (shut down the BrowseClass object) .....	280
Next (get the next browse item) .....	281
PostNewSelection (post an EVENT:NewSelection to the browse list) .....	281
Previous (get the previous browse item) .....	282
Records (return the number of browse queue items) .....	282
ResetFromAsk (reset browse after update) .....	283
ResetFromBuffer (fill queue starting from record buffer) .....	285
ResetFromFile (fill queue starting from file POSITION) .....	286
ResetFromView (reset browse from current result set) .....	287
ResetQueue (fill or refill queue) .....	288
ResetResets (copy the Reset fields) .....	289
ResetSort (apply sort order to browse) .....	290
ScrollEnd (scroll to first or last item) .....	291
ScrollOne (scroll up or down one item) .....	292
ScrollPage (scroll up or down one page) .....	293
SetAlerts (alert keystrokes for list and locator controls) .....	294
SetQueueRecord (copy data from file buffer to queue buffer) .....	294
SetSort (apply a sort order to the browse) .....	295
TakeAcceptedLocator (apply an accepted locator value) .....	296
TakeEvent (process the current ACCEPT loop event) .....	297
TakeKey (process an alerted keystroke) .....	298
TakeLocate (collect and apply ad hoc query) .....	298
TakeNewSelection (process a new selection) .....	299

TakeScroll (process a scroll event) .....	300
TakeVCRScroll (process a VCR scroll event) .....	301
UpdateBuffer (copy selected item from queue buffer to file buffer) .....	302
UpdateQuery (set default query interface) .....	303
UpdateResets (copy reset fields to file buffer) .....	304
UpdateThumb (position the scrollbar thumb) .....	304
UpdateThumbFixed (position the scrollbar fixed thumb) .....	305
UpdateViewRecord (get view data for the selected item) .....	305
UpdateWindow (update display variables to match browse) .....	306

## **12 - BUFFEREDPAIRSCLASS 307**

### **Overview 307**

BufferedPairsClass Concepts .....	307
Relationship to Other Application Builder Classes .....	307
ABC Template Implementation .....	307
BufferedPairsClass Source Files .....	308
Conceptual Example .....	308

### **BufferedPairsClass Properties 309**

RealList (recognized field pairs) .....	309
---	-----

### **BufferedPairsClass Methods 310**

Functional Organization—Expected Use .....	310
AddPair (add a field pair) .....	312
AssignBufferToLeft (copy from “buffer” fields to “left” fields) .....	313
AssignBufferToRight (copy from “buffer” fields to “right” fields) .....	313
AssignLeftToBuffer (copy from “left” fields to “buffer” fields) .....	314
AssignRightToBuffer (copy from “right” fields to “buffer” fields) .....	314
EqualLeftBuffer (compare “left” fields to “buffer” fields) .....	315
EqualRightBuffer (compare “right” fields to “buffer” fields) .....	315
Init (initialize the BufferedPairsClass object) .....	316
Kill (shut down the BufferedPairsClass object) .....	316

## **13 - CONSTANTCLASS 317**

### **Overview 317**

ConstantClass Concepts .....	317
Relationship to Other Application Builder Classes .....	318
ABC Template Implementation .....	318
ConstantClass Source Files .....	319
Conceptual Example .....	319

<b>ConstantClass Properties</b>	<b>320</b>
TerminatorValue (end of data marker) .....	320
<b>ConstantClass Methods</b>	<b>321</b>
Functional Organization—Expected Use .....	321
AddItem (set constant datatype and target variable) .....	322
Init (initialize the ConstantClass object) .....	323
Kill (shut down the ConstantClass object) .....	324
Next (load all constant items to file or queue) .....	325
Next (copy next constant item to targets) .....	326
Reset (reset the object to the beginning of the constant data) .....	327
Set (set the constant data to process) .....	328
<b>14 - EDITCHECKCLASS</b>	<b>329</b>
<b>Overview</b>	<b>329</b>
EditCheckClass Concepts .....	329
Relationship to Other Application Builder Classes .....	329
ABC Template Implementation .....	329
EditCheckClass Source Files .....	330
Conceptual Example .....	330
<b>EditCheckClass Properties</b>	<b>333</b>
<b>EditCheckClass Methods</b>	<b>334</b>
Functional Organization—Expected Use .....	334
CreateControl (create the edit-in-place CHECK control) .....	335
<b>15 - EDITCLASS</b>	<b>337</b>
<b>Overview</b>	<b>337</b>
EditClass Concepts .....	337
Relationship to Other Application Builder Classes .....	337
ABC Template Implementation .....	338
EditClass Source Files .....	338
Conceptual Example .....	339
<b>EditClass Properties</b>	<b>342</b>
FEQ (the edit-in-place control number) .....	342
<b>EditClass Methods</b>	<b>343</b>
Functional Organization—Expected Use .....	343
CreateControl (a virtual to create the edit control) .....	344
Init (initialize the EditClass object) .....	345
Kill (shut down the EditClass object) .....	345

SetAlerts (alert keystrokes for the edit control) .....	346
TakeEvent (process edit-in-place events) .....	347
<b>16 - EDITCOLORCLASS</b>	<b>349</b>
<b>Overview</b>	<b>349</b>
EditColorClass Concepts .....	349
Relationship to Other Application Builder Classes .....	349
ABC Template Implementation .....	350
EditColorClass Source Files .....	350
Conceptual Example .....	350
<b>EditColorClass Properties</b>	<b>353</b>
Title (color dialog title text) .....	353
<b>EditColorClass Methods</b>	<b>354</b>
Functional Organization—Expected Use .....	354
CreateControl (create the edit-in-place control) .....	355
TakeEvent (process edit-in-place events) .....	356
<b>17 - EDITDROPLISTCLASS</b>	<b>357</b>
<b>Overview</b>	<b>357</b>
EditDropListClass Concepts .....	357
Relationship to Other Application Builder Classes .....	357
ABC Template Implementation .....	357
EditDropListClass Source Files .....	358
Conceptual Example .....	358
<b>EditDropListClass Properties</b>	<b>361</b>
<b>EditDropListClass Methods</b>	<b>362</b>
Functional Organization—Expected Use .....	362
CreateControl (create the edit-in-place DROPLIST control) .....	363
SetAlerts (alert keystrokes for the edit control) .....	364
<b>18 - EDITENTRYCLASS</b>	<b>365</b>
<b>Overview</b>	<b>365</b>
EditEntryClass Concepts .....	365
Relationship to Other Application Builder Classes .....	365
ABC Template Implementation .....	366
EditEntryClass Source Files .....	366
Conceptual Example .....	366

<b>EditEntryClass Properties</b>	<b>369</b>
<b>EditEntryClass Methods</b>	<b>370</b>
Functional Organization—Expected Use .....	370
CreateControl (create the edit-in-place ENTRY control) .....	371
<b>19 - EDITFILECLASS</b>	<b>373</b>
<b>Overview</b>	<b>373</b>
EditFileClass Concepts .....	373
Relationship to Other Application Builder Classes .....	373
ABC Template Implementation .....	374
EditFileClass Source Files .....	374
Conceptual Example .....	374
<b>EditFileClass Properties</b>	<b>377</b>
FileMask (file dialog behavior) .....	377
FilePattern (file dialog filter) .....	377
Title (file dialog title text) .....	378
<b>EditFileClass Methods</b>	<b>379</b>
Functional Organization—Expected Use .....	379
CreateControl (create the edit-in-place control) .....	380
TakeEvent (process edit-in-place events) .....	381
<b>20 - EDITFONTCLASS</b>	<b>383</b>
<b>Overview</b>	<b>383</b>
EditFontClass Concepts .....	383
Relationship to Other Application Builder Classes .....	383
ABC Template Implementation .....	384
EditFontClass Source Files .....	384
Conceptual Example .....	384
<b>EditFontClass Properties</b>	<b>388</b>
Title (font dialog title text) .....	388
<b>EditFontClass Methods</b>	<b>389</b>
Functional Organization—Expected Use .....	389
CreateControl (create the edit-in-place control) .....	390
TakeEvent (process edit-in-place events) .....	391
<b>21 - EDITMULTISELECTCLASS</b>	<b>393</b>
<b>Overview</b>	<b>393</b>
EditMultiSelectClass Concepts .....	393

Relationship to Other Application Builder Classes .....	394
ABC Template Implementation .....	394
EditMultiSelectClass Source Files .....	394
Conceptual Example .....	395
<b>EditMultiSelectClass Properties</b>	<b>399</b>
Available (multi-select dialog available items queue) .....	399
Selected (multi-select dialog selected items queue) .....	399
FilePattern (multi-select dialog file pattern text) .....	399
Title (multi-select dialog title text) .....	399
<b>EditMultiSelectClass Methods</b>	<b>400</b>
Functional Organization—Expected Use .....	400
AddValue (prime the MultiSelect dialog) .....	402
CreateControl (create the edit-in-place control) .....	403
Reset (reset the EditMultiSelectClass object) .....	403
TakeAction (process MultiSelect dialog action) .....	404
TakeEvent (process edit-in-place events) .....	407
<b>22 - ENTRYLOCATORCLASS</b>	<b>409</b>
<b>Overview</b>	<b>409</b>
EntryLocatorClass Concepts .....	409
Relationship to Other Application Builder Classes .....	409
ABC Template Implementation .....	409
EntryLocatorClass Source Files .....	410
Conceptual Example .....	410
<b>EntryLocatorClass Properties</b>	<b>412</b>
Shadow (the search value) .....	412
<b>EntryLocatorClass Methods</b>	<b>413</b>
Init (initialize the EntryLocatorClass object) .....	413
Set (restart the locator) .....	414
TakeAccepted (process an accepted locator value) .....	414
TakeKey (process an alerted keystroke) .....	415
Update (update the locator control and free elements) .....	416
UpdateWindow (redraw the locator control) .....	416
<b>23 - ERROR CLASS</b>	<b>417</b>
<b>Overview</b>	<b>417</b>
ErrorClass Source Files .....	417
Multiple Customizable Levels of Error Treatment .....	417

Predefined Windows and Database Errors .....	418
Dynamic Extensibility of Errors .....	418
ABC Template Implementation .....	418
Relationship to Other Application Builder Classes .....	419
Macro Expansion .....	419
Multi-Language Capability .....	420
Conceptual Example .....	421
<b>ErrorClass Properties</b> .....	<b>422</b>
Errors (recognized error definitions) .....	422
FieldName (field that produced the error) .....	423
FileName (file that produced the error) .....	423
MessageText (custom error message text) .....	423
<b>ErrorClass Methods</b> .....	<b>424</b>
Functional Organization—Expected Use .....	424
AddErrors (add or override recognized errors) .....	425
Init (initialize the ErrorClass object) .....	426
Kill (perform any necessary termination code) .....	426
Message (display an error message) .....	427
RemoveErrors (remove or restore recognized errors) .....	428
SetErrors (save the error state) .....	429
SetFatality (set severity level for a particular error) .....	430
SetField (set the substitution value of the %Field macro) .....	431
SetFile (set the substitution value of the %File macro) .....	431
SetId (make a specific error current) .....	432
SubsString (resolves error message macros) .....	433
TakeBenign (process benign error) .....	434
TakeError (process specified error) .....	435
TakeFatal (process fatal error) .....	436
TakeNotify (process notify error) .....	437
TakeOther (process other error) .....	438
TakeProgram (process program error) .....	439
TakeUser (process user error) .....	440
Throw (process specified error) .....	441
ThrowFile (set value of %File, then process error) .....	442
ThrowMessage (set value of %Message, then process error) .....	443

## **24 - FIELDPAIRSCLASS** **445**

<b>Overview</b> .....	<b>445</b>
FieldPairsClass Concepts .....	445

Relationship to Other Application Builder Classes .....	446
ABC Template Implementation .....	446
FieldPairsClass Source Files .....	446
Conceptual Example .....	447
<b>FieldPairsClass Properties</b> .....	<b>448</b>
List (recognized field pairs) .....	448
<b>FieldPairsClass Methods</b> .....	<b>449</b>
Functional Organization—Expected Use .....	449
AddItem (add a field pair from one source field) .....	450
AddPair (add a field pair) .....	451
AssignLeftToRight (copy from “left” fields to “right” fields) .....	452
AssignRightToLeft (copy from “right” fields to “left” fields) .....	453
ClearLeft (clear each “left” field) .....	454
ClearRight (clear each “right” field) .....	455
Equal (return 1 if all pairs are equal) .....	456
EqualLeftRight (return 1 if all pairs are equal) .....	456
Init (initialize the FieldPairsClass object) .....	457
Kill (shut down the FieldPairsClass object) .....	457
 <b>25 - FILEDROPCLASS</b> .....	 <b>459</b>
<b>Overview</b> .....	<b>459</b>
Future FileDropClasses .....	459
FileDropClass Concepts .....	459
Relationship to Other Application Builder Classes .....	459
ABC Template Implementation .....	460
FileDropClass Source Files .....	460
Conceptual Example .....	461
<b>FileDropClass Properties</b> .....	<b>464</b>
DefaultFill (initial display value) .....	464
InitSyncPair (initial list position) .....	464
<b>FileDropClass Methods</b> .....	<b>465</b>
Functional Organization—Expected Use .....	465
AddField (specify display fields) .....	466
AddUpdateField (specify field assignments) .....	467
Init (initialize the FileDropClass object) .....	468
Kill (shut down the FileDropClass object) .....	469
ResetQueue (fill filedrop queue) .....	470
SetQueueRecord (copy data from file buffer to queue buffer) .....	471

TakeEvent (process the current ACCEPT loop event) .....	471
TakeNewSelection (process EVENT:NewSelection events) .....	472
ValidateRecord (a virtual to validate records) .....	473

## **26 - FILEDROPCOMBOCLASS 475**

### **Overview 475**

Future File DropCombo Classes .....	475
FileDropComboClass Concepts .....	475
Relationship to Other Application Builder Classes .....	475
ABC Template Implementation .....	476
FileDropComboClass Source Files .....	476
Conceptual Example .....	477

### **FileDropComboClass Properties 480**

EntryCompletion (automatic fill-ahead flag) .....	480
UseField (COMBO USE variable) .....	480

### **FileDropComboClass Methods 481**

Functional Organization—Expected Use .....	481
Ask (add a record to the lookup file) .....	483
GetQueueMatch (locate a list item) .....	484
Init (initialize the FileDropComboClass object) .....	485
ResetQueue (refill the filedrop queue) .....	487
TakeEvent (process the current ACCEPT loop event) .....	488
TakeNewSelection (process EVENT:NewSelection events) .....	489

## **27 - FILEMANAGER 491**

### **Overview 491**

Dual Approach to Database Operations .....	491
Relationship to Other Application Builder Classes .....	492
FileManager and Threaded Files .....	492
ABC Template Implementation .....	492
FileManager Source Files .....	493
Conceptual Example .....	494

### **FileManager Properties 496**

AliasedFile (the primary file) .....	496
Buffer (the record buffer) .....	496
Buffers (saved record buffers) .....	497
Create (create file switch) .....	497
File (the managed file) .....	498

FileName (variable filename) .....	498
FileNameValue (constant filename) .....	499
LazyOpen (delay file open until access) .....	500
LockRecover (/RECOVER wait time parameter) .....	500
OpenMode (file access/sharing mode) .....	501
SkipHeldRecords (HELD record switch) .....	501
<b>FileManager Methods</b> .....	<b>502</b>
Naming Conventions and Dual Approach to Database Operations .....	502
Functional Organization—Expected Use .....	503
AddKey (set the file's keys) .....	505
BindFields (bind fields when file is opened) .....	506
CancelAutoInc (undo PrimeAutoInc) .....	507
ClearKey (clear specified key components) .....	509
Close (close the file) .....	511
EqualBuffer (detect record buffer changes) .....	512
Fetch (get a specific record by key value) .....	513
GetComponents (return the number of key components) .....	514
GetEOF (return end of file status) .....	515
GetError (return the current error ID) .....	516
GetField (return a reference to a key component) .....	517
GetFieldName (return a key component field name) .....	518
GetName (return the filename) .....	519
Init (initialize the FileManager object) .....	520
Insert (add a new record) .....	521
KeyToOrder (return ORDER expression for a key) .....	522
Kill (shutdown the FileManager object) .....	523
Next (get next record in sequence) .....	524
Open (open the file) .....	525
Position (return the current record position) .....	526
Previous (get previous record in sequence) .....	527
PrimeAutoInc (prepare an autoincremented record for adding) .....	528
PrimeFields (a virtual to prime fields) .....	530
PrimeRecord (prepare a record for adding) .....	531
RestoreBuffer (restore a previously saved record buffer) .....	533
RestoreFile (restore a previously saved file state) .....	534
SaveBuffer (save a copy of the record buffer) .....	535
SaveFile (save the current file state) .....	536
SetError (save the specified error and underlying error state) .....	537

SetKey (set current key) .....	537
SetName (set current filename) .....	538
Throw (pass an error to the error handler for processing) .....	539
ThrowMessage (pass an error and text to the error handler) .....	540
TryFetch (try to get a specific record by key value) .....	541
TryInsert (try to add a new record) .....	542
TryNext (try to get next record in sequence) .....	543
TryOpen (try to open the file) .....	544
TryPrevious (try to get previous record in sequence) .....	545
TryPrimeAutoInc (try to prepare an autoincremented record for adding) .....	546
TryReget (try to get a specific record by position) .....	548
TryUpdate (try to change the current record) .....	548
Update (change the current record) .....	549
UseFile (use LazyOpen file) .....	549
ValidateField (validate a field) .....	550
ValidateFields (validate a range of fields) .....	551
ValidateRecord (validate all fields) .....	552

## **28 - FILTERLOCATORCLASS 553**

### **Overview 553**

FilterLocatorClass Concepts .....	553
Relationship to Other Application Builder Classes .....	554
ABC Template Implementation .....	554
FilterLocatorClass Source Files .....	554
Conceptual Example .....	555

### **FilterLocatorClass Properties 557**

FloatRight (“contains” or “begins with” flag) .....	557
---	-----

### **FilterLocatorClass Methods 558**

TakeAccepted (process an accepted locator value) .....	558
UpdateWindow (apply the search criteria) .....	559

## **29 - INCREMENTALLOCATORCLASS 561**

### **Overview 561**

IncrementalLocatorClass Concepts .....	561
Relationship to Other Application Builder Classes .....	561
ABC Template Implementation .....	562
IncrementalLocatorClass Source Files .....	562
Conceptual Example .....	563

<b>IncrementalLocatorClass Properties</b>	<b>565</b>
<b>IncrementalLocatorClass Methods</b>	<b>565</b>
SetAlerts (alert keystrokes for the LIST control) .....	565
TakeKey (process an alerted keystroke) .....	566
<b>30 - INICLASS</b>	<b>567</b>
<b>Overview</b>	<b>567</b>
INI Class Concepts .....	567
Relationship to Other Application Builder Classes .....	567
ABC Template Implementation .....	567
INI Class Source Files .....	568
Conceptual Example .....	568
<b>INIClass Properties</b>	<b>569</b>
FileName .....	569
<b>INIClass Methods</b>	<b>570</b>
Fetch (get INI file entries) .....	570
FetchField (return comma delimited INI file value) .....	572
FetchQueue (get INI file queue entries) .....	573
Init (initialize the INIClass object) .....	574
TryFetch (get a value from the INI file) .....	575
TryFetchField (return comma delimited INI file value) .....	576
Update (write INI file entries) .....	577
<b>31 - LOCATORCLASS</b>	<b>579</b>
<b>Overview</b>	<b>579</b>
LocatorClass Concepts .....	579
Relationship to Other Application Builder Classes .....	579
ABC Template Implementation .....	580
LocatorClass Source Files .....	580
<b>LocatorClass Properties</b>	<b>581</b>
Control (the locator control number) .....	581
FreeElement (the locator's first free key element) .....	581
NoCase (case sensitivity flag) .....	581
ViewManager (the locator's ViewManager object) .....	582
<b>LocatorClass Methods</b>	<b>583</b>
Init (initialize the LocatorClass object) .....	583
Reset (reset the locator for next search) .....	584

Set (restart the locator) .....	584
SetAlerts (alert keystrokes for the LIST control) .....	585
SetEnabled (enable or disable the locator control) .....	585
TakeAccepted (process an accepted locator value) .....	586
TakeKey (process an alerted keystroke) .....	586
UpdateWindow (redraw the locator control with its current value) .....	586

## **32 - POPUPCLASS**

## **587**

### **Overview**

### **587**

PopupClass Concepts .....	587
Relationship to Other Application Builder Classes .....	587
ABC Template Implementation .....	587
PopupClass Source Files .....	588
Conceptual Example .....	588

### **PopupClass Properties**

### **590**

ClearKeycode (clear KEYCODE character) .....	590
--	-----

### **PopupClass Methods**

### **591**

Functional Organization—Expected Use .....	591
AddItem (add menu item) .....	592
AddItemEvent (set menu item action) .....	594
AddItemMimic (tie menu item to a button) .....	595
AddMenu (add a menu) .....	596
AddSubMenu (add submenu) .....	598
Ask (display the popup menu) .....	600
DeleteItem (remove menu item) .....	601
GetItemChecked (return toggle item status) .....	602
GetItemEnabled (return item status) .....	603
GetLastSelection (return selected item) .....	603
Init (initialize the PopupClass object) .....	604
Kill (shut down the PopupClass object) .....	604
Restore (restore a saved menu) .....	605
Save (save a menu for restoration) .....	606
SetIcon (set menu item icon) .....	607
SetItemCheck (set toggle item status) .....	608
SetItemEnable (set item status) .....	609
SetLevel (set menu item level) .....	609
SetText (set menu item text) .....	610
SetToolbox (include item on toolbox) .....	611

SetTranslator (set run-time translator) .....	612
Toolbox (display the popup toolbox) .....	613
ViewMenu (popup menu debugger) .....	613

### **33 - PRINTPREVIEWCLASS 615**

#### **Overview 615**

PrintPreviewClass Concepts .....	615
Relationship to Other Application Builder Classes .....	616
ABC Template Implementation .....	616
PrintPreviewClass Source Files .....	616
Zoom Configuration .....	617
Conceptual Example .....	617

#### **PrintPreviewClass Properties 620**

AllowUserZoom (allow any zoom factor) .....	620
CurrentPage (the selected report page) .....	620
Maximize (number of pages displayed horizontally) .....	620
PagesAcross (number of pages displayed horizontally) .....	621
PagesDown (number of vertical thumbnails) .....	621
UserPercentile (custom zoom factor) .....	621
WindowPosSet (use a non-default initial preview window position) .....	621
WindowSizeSet (use a non-default initial preview window size) .....	622
ZoomIndex (index to applied zoom factor) .....	622

#### **PrintPreviewClass Methods 623**

Functional Organization—Expected Use .....	623
AskPage (prompt for new report page) .....	624
AskThumbnails (prompt for new thumbnail configuration) .....	625
Display (preview the report) .....	626
Init (initialize the PrintPreviewClass object) .....	628
Kill (shut down the PrintPreviewClass object) .....	628
Open (prepare preview window for display) .....	629
SetINIManager (save and restore window coordinates) .....	630
SetPosition (set initial preview window coordinates) .....	631
SetZoomPercentile (set user or standard zoom factor) .....	632
TakeAccepted (process EVENT:Accepted events) .....	633
TakeEvent (process all events) .....	634
TakeFieldEvent (a virtual to process field events) .....	635
TakeWindowEvent (process non-field events) .....	636

<b>34 - PROCESSCLASS</b>	<b>637</b>
<b>Overview</b>	<b>637</b>
ProcessClass Concepts .....	637
Relationship to Other Application Builder Classes .....	637
ABC Template Implementation .....	637
ProcessClass Source Files .....	638
Conceptual Example .....	638
<b>ProcessClass Properties</b>	<b>641</b>
ChildRead (portion of process completed) .....	641
Percentile (portion of process completed) .....	641
PText (progress control number) .....	642
RecordsProcessed (number of elements processed) .....	642
RecordsToProcess (number of elements to process) .....	642
<b>ProcessClass Methods</b>	<b>643</b>
Functional Organization—Expected Use .....	643
AddItem (add a child viewmanager) .....	645
Init (initialize the ProcessClass object) .....	646
Kill (shut down the ProcessClass object) .....	648
Next (get next element) .....	649
Reset (position to the first element) .....	650
SetProgressLimits (calibrate the progress monitor) .....	650
TakeRecord (a virtual to process each record) .....	651
<b>35 - QUERYCLASS</b>	<b>653</b>
<b>Overview</b>	<b>653</b>
QueryClass Concepts .....	653
Relationship to Other Application Builder Classes .....	653
ABC Template Implementation .....	654
QueryClass Source Files .....	654
Conceptual Example .....	654
<b>QueryClass Properties</b>	<b>657</b>
<b>QueryClass Methods</b>	<b>658</b>
Functional Organization—Expected Use .....	658
AddItem (add field to query) .....	659
Ask (a virtual to accept query criteria) .....	660
GetFilter (return filter expression) .....	661
GetLimit (get searchvalues) .....	663

Init (initialize the QueryClass object) .....	664
Kill (shut down the QueryClass object) .....	664
Reset (reset the QueryClass object) .....	665
SetLimit (set search values) .....	666
<b>36 - QUERYFORMCLASS</b>	<b>669</b>
<b>Overview</b>	<b>669</b>
QueryFormClass Concepts .....	669
Relationship to Other Application Builder Classes .....	669
ABC Template Implementation .....	669
QueryFormClass Source Files .....	670
Conceptual Example .....	670
<b>QueryFormClass Properties</b>	<b>673</b>
<b>QueryFormClass Methods</b>	<b>674</b>
Functional Organization—Expected Use .....	674
Ask (solicit query criteria) .....	675
Init (initialize the QueryFormClass object) .....	676
Kill (shut down the QueryFormClass object) .....	677
<b>37 - QUERYFORMVISUAL</b>	<b>679</b>
<b>Overview</b>	<b>679</b>
QueryFormVisual Concepts .....	679
Relationship to Other Application Builder Classes .....	679
ABC Template Implementation .....	679
QueryFormVisual Source Files .....	680
Conceptual Example .....	680
<b>QueryFormVisual Properties</b>	<b>683</b>
QFC (reference to the QueryFormClass) .....	683
<b>QueryFormVisual Methods</b>	<b>684</b>
Functional Organization—Expected Use .....	684
Init (initialize the QueryFormVisual object) .....	685
TakeAccepted (handle query dialog EVENT:Accepted events) .....	686
TakeCompleted (complete the query dialog) .....	687
<b>38 - RELATIONMANAGER</b>	<b>689</b>
<b>Overview</b>	<b>689</b>
Relation Manager Concepts and Conventions .....	689
ABC Template Implementation .....	690

Relationship to Other Application Builder Classes .....	690
RelationManager Source Files .....	691
Conceptual Example .....	692
<b>RelationManager Properties</b> .....	<b>695</b>
Me (the primary file's FileManager object) .....	695
UseLogout (transaction framing flag) .....	695
<b>RelationManager Methods</b> .....	<b>696</b>
Functional Organization—Expected Use .....	696
AddRelation (set a file relationship) .....	697
AddRelationLink (set linking fields for a relationship) .....	699
CancelAutoInc (undo autoincrement) .....	700
Close (close a file and any related files) .....	701
Delete (delete record subject to referential constraints) .....	702
Init (initialize the RelationManager object) .....	703
Kill (shut down the RelationManager object) .....	704
ListLinkingFields (map pairs of linked fields) .....	705
Open (open a file and any related files) .....	706
Save (copy the current record and any related records) .....	706
SetAlias (set a file alias) .....	707
SetQuickScan (enable QuickScan on a file and any related files) .....	708
Update (update record subject to referential constraints) .....	709
 <b>39 - REPORTMANAGER</b> .....	 <b>711</b>
<b>Overview</b> .....	<b>711</b>
ReportManager Concepts .....	711
Relationship to Other Application Builder Classes .....	711
ABC Template Implementation .....	711
ReportManager Source Files .....	712
Conceptual Example .....	712
<b>ReportManager Properties</b> .....	<b>715</b>
DeferOpenReport (defer open) .....	715
DeferWindow (defer progress window display) .....	715
KeepVisible (persistent progress window) .....	716
Preview (PrintPreviewClass object) .....	716
PreviewQueue (report metafile pathnames) .....	717
Process (ProcessClass object) .....	717
Report (the managed REPORT) .....	717
SkipPreview (print rather than preview) .....	718
TimeSlice (report resource usage) .....	718

WaitCursor (defer progress window display) .....	719
Zoom (initial report preview magnification) .....	719
<b>ReportManager Methods</b> .....	<b>720</b>
Functional Organization—Expected Use .....	720
Ask (display window and process its events) .....	721
AskPreview (preview or print the report) .....	721
Init (initialize the ReportManager object) .....	722
Kill (shut down the ReportManager object) .....	723
Next (get next report record) .....	724
Open (a virtual to execute on EVENT:OpenWindow) .....	725
OpenReport (prepare report for execution) .....	726
TakeCloseEvent (a virtual to process EVENT:CloseWindow) .....	727
TakeNoRecords (process empty report) .....	728
TakeWindowEvent (a virtual to process non-field events) .....	729
<b>40 - SELECTFILECLASS</b> .....	<b>731</b>
<b>Overview</b> .....	<b>731</b>
SelectFileClass Concepts .....	731
Relationship to Other Application Builder Classes .....	731
ABC Template Implementation .....	731
SelectFileClass Source Files .....	731
Conceptual Example .....	732
<b>SelectFileClass Properties</b> .....	<b>733</b>
DefaultDirectory (initial path) .....	733
DefaultFile (initial filename/filemask) .....	733
Flags (file dialog behavior) .....	733
WindowTitle (file dialog title text) .....	733
<b>SelectFileClass Methods</b> .....	<b>734</b>
AddMask (add file dialog file masks) .....	734
Ask (display Windows file dialog) .....	735
Init (initialize the SelectFileClass object) .....	736
SetMask (set file dialog file masks) .....	737
<b>41 - STEPCLASS</b> .....	<b>739</b>
<b>Overview</b> .....	<b>739</b>
StepClass Concepts .....	739
Relationship to Other Application Builder Classes .....	739
ABC Template Implementation .....	740
StepClass Source Files .....	740

<b>StepClass Properties</b>	<b>741</b>
Controls (the StepClass sort sequence) .....	741
<b>StepClass Methods</b>	<b>742</b>
GetPercentile (return a value's percentile) .....	742
GetValue (return a percentile's value) .....	742
Init (initialize the StepClass object) .....	743
Kill (shut down the StepClass object) .....	743
SetLimit (set smooth data distribution) .....	744
SetLimitNeeded (return static/dynamic boundary flag) .....	744
<b>42 - STEPCUSTOMCLASS</b>	<b>745</b>
<b>Overview</b>	<b>745</b>
StepCustomClass Concepts .....	745
Relationship to Other Application Builder Classes .....	745
ABC Template Implementation .....	746
StepCustomClass Source Files .....	746
Conceptual Example .....	747
<b>StepCustomClass Properties</b>	<b>749</b>
Entries (expected data distribution) .....	749
<b>StepCustomClass Methods</b>	<b>750</b>
AddItem (add a step marker) .....	750
GetPercentile (return a value's percentile) .....	751
GetValue (return a percentile's value) .....	752
Init (initialize the StepCustomClass object) .....	753
Kill (shut down the StepCustomClass object) .....	753
<b>43 - STEPLocatorCLASS</b>	<b>755</b>
<b>Overview</b>	<b>755</b>
StepLocatorClass Concepts .....	755
Relationship to Other Application Builder Classes .....	755
ABC Template Implementation .....	756
StepLocatorClass Source Files .....	756
Conceptual Example .....	757
<b>StepLocatorClass Properties</b>	<b>759</b>
<b>StepLocatorClass Methods</b>	<b>760</b>
Set (restart the locator) .....	760
TakeKey (process an alerted keystroke) .....	760

<b>44 - STEPLONGCLASS</b>	<b>761</b>
<b>Overview</b>	<b>761</b>
StepLongClass Concepts .....	761
Relationship to Other Application Builder Classes .....	761
ABC Template Implementation .....	761
StepLongClass Source Files .....	762
Conceptual Example .....	762
<b>StepLongClass Properties</b>	<b>764</b>
Low (lower boundary) .....	764
High (upper boundary) .....	764
<b>StepLongClass Methods</b>	<b>765</b>
GetPercentile (return a value's percentile) .....	765
GetValue (return a percentile's value) .....	766
SetLimit (set smooth data distribution) .....	767
<b>45 - STEPREALCLASS</b>	<b>769</b>
<b>Overview</b>	<b>769</b>
StepRealClass Concepts .....	769
Relationship to Other Application Builder Classes .....	769
ABC Template Implementation .....	769
StepRealClass Source Files .....	770
Conceptual Example .....	770
<b>StepRealClass Properties</b>	<b>772</b>
Low (lower boundary) .....	772
High (upper boundary) .....	772
<b>StepRealClass Methods</b>	<b>773</b>
GetPercentile (return a value's percentile) .....	773
GetValue (return a percentile's value) .....	774
SetLimit (set smooth data distribution) .....	775
<b>46 - STEPSTRINGCLASS</b>	<b>777</b>
<b>Overview</b>	<b>777</b>
StepStringClass Concepts .....	777
Relationship to Other Application Builder Classes .....	777
ABC Template Implementation .....	778
StepStringClass Source Files .....	778
Conceptual Example .....	779

<b>StepStringClass Properties</b>	<b>781</b>
LookupMode (expected data distribution) .....	781
Root (the static portion of the step) .....	782
SortChars (valid sort characters) .....	782
TestLen (length of the static step portion) .....	783
<b>StepStringClass Methods</b>	<b>784</b>
GetPercentile (return a value's percentile) .....	784
GetValue (return a percentile's value) .....	785
Init (initialize the StepStringClass object) .....	786
Kill (shut down the StepStringClass object) .....	787
SetLimit (set smooth data distribution) .....	787
SetLimitNeeded (return static/dynamic boundary flag) .....	788
<b>47 -TOOLBARCLASS</b>	<b>789</b>
<b>Overview</b>	<b>789</b>
ToolbarClass Concepts .....	789
Relationship to Other Application Builder Classes .....	790
ABC Template Implementation .....	790
Toolbar Class Source Files .....	791
Conceptual Example .....	792
<b>ToolbarClass Methods</b>	<b>795</b>
Functional Organization—Expected Use .....	795
AddTarget (register toolbar driven entity) .....	796
DisplayButtons (enable appropriate toolbar buttons) .....	796
Init (initialize the ToolbarClass object) .....	797
Kill (shut down the ToolbarClass object) .....	797
SetTarget (sets the active target) .....	798
TakeEvent (process toolbar event) .....	799
<b>48 - TOOLBARLISTBOXCLASS</b>	<b>801</b>
<b>Overview</b>	<b>801</b>
ToolbarListboxClass Concepts .....	801
Relationship to Other Application Builder Classes .....	801
ABC Template Implementation .....	801
ToolbarListboxClass Source Files .....	801
Conceptual Example .....	802
<b>ToolbarListboxClass Properties</b>	<b>805</b>
Browse (BrowseClass object) .....	805

<b>ToolbarListBoxClass Methods</b>	<b>806</b>
DisplayButtons (enable appropriate toolbar buttons) .....	806
TakeEvent (convert toolbar events) .....	807
TakeToolbar (assume control of the toolbar) .....	808
TryTakeToolbar (return toolbar control indicator) .....	809
<b>49 - TOOLBARRELTREECLASS</b>	<b>811</b>
<b>Overview</b>	<b>811</b>
ToolbarReltreeClass Concepts .....	811
Relationship to Other Application Builder Classes .....	811
ABC Template Implementation .....	811
Toolbar ToolbarReltreeClass Source Files .....	811
Conceptual Example .....	812
<b>ToolbarReltreeClass Properties</b>	<b>815</b>
<b>ToolbarReltreeClass Methods</b>	<b>815</b>
DisplayButtons (enable appropriate toolbar buttons) .....	815
TakeToolbar (assume control of the toolbar) .....	816
<b>50 - TOOLBARTARGET</b>	<b>817</b>
<b>Overview</b>	<b>817</b>
ToolbarTarget Concepts .....	817
Relationship to Other Application Builder Classes .....	818
ABC Template Implementation .....	818
ToolbarTarget Source Files .....	818
<b>ToolbarTarget Properties</b>	<b>819</b>
ChangeButton (change control number) .....	819
Control (window control) .....	819
DeleteButton (delete control number) .....	820
HelpButton (help control number) .....	820
InsertButton (insert control number) .....	820
SelectButton (select control number) .....	821
<b>ToolbarTarget Methods</b>	<b>822</b>
Functional Organization—Expected Use .....	822
DisplayButtons (enable appropriate toolbar buttons) .....	822
TakeEvent (convert toolbar events) .....	823
TakeToolbar (assume control of the toolbar) .....	824
TryTakeToolbar (return toolbar control indicator) .....	824

<b>51 - TOOLBARUPDATECLASS</b>	<b>825</b>
<b>Overview</b>	<b>825</b>
ToolbarUpdateClass Concepts .....	825
Relationship to Other Application Builder Classes .....	825
ABC Template Implementation .....	825
ToolbarUpdateClass Source Files .....	826
Conceptual Example .....	826
<b>ToolbarUpdateClass Properties</b>	<b>832</b>
Request (requested database operation) .....	832
History (enable toolbar history button) .....	832
<b>ToolbarUpdateClass Methods</b>	<b>833</b>
DisplayButtons (enable appropriate toolbar buttons) .....	833
TakeEvent (convert toolbar events) .....	834
TakeToolbar (assume control of the toolbar) .....	835
TryTakeToolbar (return toolbar control indicator) .....	836
<b>52 - TRANSLATORCLASS</b>	<b>837</b>
<b>Overview</b>	<b>837</b>
TranslatorClass Concepts .....	837
Relationship to Other Application Builder Classes .....	838
ABC Template Implementation .....	838
TranslatorClass Source Files .....	839
Conceptual Example .....	840
<b>TranslatorClass Properties</b>	<b>841</b>
ExtractText (identify text to translate) .....	841
<b>TranslatorClass Methods</b>	<b>842</b>
AddTranslation (add translation pairs) .....	842
Init (initialize the TranslatorClass object) .....	844
Kill (shut down the TranslatorClass object) .....	844
TranslateControl (translate text for a control) .....	845
TranslateControls (translate text for range of controls) .....	846
TranslateProperty (translate textual control property) .....	847
TranslateString (translate text) .....	848
TranslateWindow (translate text for a window) .....	849
<b>53 - VIEWMANAGER</b>	<b>851</b>
<b>Overview</b>	<b>851</b>
ViewManager Concepts .....	851

Relationship to Other Application Builder Classes .....	851
ABC Template Implementation .....	852
ViewManager Source Files .....	852
Conceptual Example .....	853
<b>ViewManager Properties</b> .....	<b>855</b>
Order (sort, range-limit, and filter information) .....	855
PagesAhead (buffered pages) .....	856
PagesBehind (buffered pages) .....	856
PageSize (buffer page size) .....	856
Primary (the primary file RelationManager ) .....	857
TimeOut (buffered pages freshness) .....	857
View (the managed VIEW) .....	857
<b>ViewManager Methods</b> .....	<b>858</b>
Functional Organization—Expected Use .....	858
AddRange (add a range limit) .....	860
AddSortOrder (add a sort order) .....	861
AppendOrder (refine a sort order) .....	862
ApplyFilter (range limit and filter the result set) .....	862
ApplyOrder (sort the result set) .....	863
ApplyRange (conditionally range limit and filter the result set) .....	863
Close (close the view) .....	864
GetFreeElementName (return free key element name) .....	864
GetFreeElementPosition (return free key element position) .....	865
Init (initialize the ViewManager object) .....	866
Kill (shut down the ViewManager object) .....	867
Next (get the next element) .....	867
Open (open the view) .....	868
Previous (get the previous element) .....	868
PrimeRecord (prepare a record for adding) .....	869
Reset (reset the view position) .....	870
SetFilter (add, change, or remove active filter) .....	871
SetOrder (replace a sort order) .....	873
SetSort (set the active sort order) .....	874
UseView (use LazyOpen files) .....	875
ValidateRecord (validate an element) .....	876
 <b>54 - WINDOWMANAGER</b> .....	 <b>877</b>
<b>Overview</b> .....	<b>877</b>
WindowManager Concepts .....	877

ABC Template Implementation .....	879
Relationship to Other Application Builder Classes .....	879
WindowManager Source Files .....	880
Conceptual Example .....	881
<b>WindowManager Properties</b> .....	<b>885</b>
AutoRefresh (reset window as needed flag) .....	885
AutoToolbar (set toolbar target on new tab selection) .....	885
CancelAction (response to cancel request) .....	886
ChangeAction (response to change request) .....	886
Dead (shut down flag) .....	887
DeleteAction (response to delete request) .....	887
Errors (ErrorClass object) .....	888
FirstField (first window control) .....	888
ForcedReset (force reset flag) .....	888
HistoryKey (restore field key) .....	889
InsertAction (response to insert request) .....	889
OKControl (window acceptance control—OK button) .....	890
Opened (window opened flag) .....	890
OriginalRequest (original database request) .....	891
Primary (RelationManager object) .....	891
Request (database request) .....	892
ResetOnGainFocus (gain focus reset flag) .....	892
Response (response to database request) .....	893
Saved (copy of primary file record buffer) .....	893
Translator (TranslatorClass object) .....	894
VCRRequest (delayed scroll request) .....	894
<b>WindowManager Methods</b> .....	<b>895</b>
Functional Organization—Expected Use .....	895
AddHistoryField (add restorable control and field) .....	897
AddHistoryFile (add restorable history file) .....	898
AddItem (program the WindowManager object) .....	899
AddUpdateFile (register batch add files) .....	901
Ask (display window and process its events) .....	902
Init (initialize the WindowManager object) .....	903
Kill (shut down the WindowManager object) .....	905
Open (a virtual to execute on EVENT:OpenWindow) .....	906
PostCompleted (initiates final Window processing) .....	907
PrimeFields (a virtual to prime form fields) .....	908
PrimeUpdate (update or prepare for update) .....	908

Reset (reset the window for display) .....	909
RestoreField (restore field to last saved value).....	910
Run (run this procedure or a subordinate procedure) .....	911
SaveHistory (save history fields for later restoration) .....	913
SetAlerts (alert window control keystrokes) .....	914
SetResponse (OK or Cancel the window) .....	915
TakeAccepted (a virtual to process EVENT:Accepted) .....	916
TakeCloseEvent (a virtual to Cancel the window) .....	917
TakeCompleted (a virtual to complete an update form) .....	918
TakeEvent (a virtual to process all events) .....	920
TakeFieldEvent (a virtual to process field events).....	921
TakeNewSelection (a virtual to process EVENT:NewSelection).....	922
TakeRejected (a virtual to process EVENT:Rejected) .....	923
TakeSelected (a virtual to process EVENT:Selected) .....	924
TakeWindowEvent (a virtual to process non-field events) .....	925
Update (prepare records for writing to disk) .....	926

## **55 - WINDOWRESIZECLASS 927**

<b>Overview</b>	<b>927</b>
WindowResizeClass Concepts .....	927
Relationship to Other Application Builder Classes .....	928
ABC Template Implementation .....	928
WindowResizeClass Source Files .....	928
Conceptual Example .....	929
<b>WindowResizeClass Properties</b>	<b>930</b>
AutoTransparent (optimize redraw) .....	930
DeferMoves (optimize resize) .....	930
<b>WindowResizeClass Methods</b>	<b>931</b>
Functional Organization—Expected Use .....	931
GetParentControl (return parent control) .....	932
GetPositionStrategy (return position strategy for a control type) .....	933
GetResizeStrategy (return resize strategy for a control type) .....	934
Init (initialize the WindowResizeClass object) .....	935
Kill (shut down the WindowResizeClass object) .....	937
Reset (resets the WindowResizeClass object) .....	937
Resize (resize and reposition controls).....	938
RestoreWindow (restore window to initial size) .....	939
SetParentControl (set parent control) .....	940
SetParentDefaults (set default parent controls) .....	941

SetPosition (calculate control coordinates) ..... 942  
SetStrategy (set control resize strategy) ..... 943

**INDEX**

**945**

# ***FOREWORD***

## ***Welcome***

Welcome to the Clarion Application Handbook! This book is designed to be your every day reference to the tools you use most often during application development—the ABC Templates and ABC Library.

Once you've become familiar with the Clarion development environment, through *Getting Started*, *Learning Clarion* and the *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 ABC Templates and Application Builder Class methods.

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

There are two parts to the Application Handbook:

### **Part I—Application Builder Class Templates**

Detailed discussions of the ABC Templates and all their prompts. This section tells you when to use a particular template and points out the startling flexibility and versatility of the ABC Templates in this product.

### **Part II—Application Builder Class Library**

In-depth discussions of the ABC Library. This section 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> .
SMALL 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.
LETTER GOTHIC	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:** **Special Tips, Notes, and Warnings—information that is not immediately evident from the topic explanation.**



Indicates vital information. If you read nothing else, read this.

# PART I

---

## APPLICATION BUILDER CLASS TEMPLATES



# 1 - TEMPLATE OVERVIEW

## What is a Template

Clarion templates are highly configurable, interactive, interpreted, code generation scripts. A template typically prompts you for information then generates a custom set of source code based on your responses. In addition to its prompts, many templates also add source code embed points to your application—points at which you can supply custom source code that is integrated into the template generated code. You may want to think of the template prompts as a way to define the static (compile time) characteristics of a program or procedure, and the embedded source as a way to define the changing (runtime) characteristics of a program or procedure.

### Template Prompts

A template typically prompts you for information at design time. The Application Generator interprets the template and presents a dialog with all the template's prompts. You fill in the prompts, with assistance from the on-line help, to define the static (compile time) characteristics of your program or procedure. For example, fill in the Record Filter prompt to establish a filter for a BrowseBox template.

### Template Embed Points

In addition to its prompts, many templates also add source code embed points to your application or procedure—points at which you can supply custom source code that is integrated into the template generated code. You can use these embed points to define the changing (runtime) characteristics of a program or procedure. For example, embed source code to hide a related listbox when there are no related records to display. See *Application Generator—Embedded Source Code* in the *User's Guide* for more information on using embed points.

### Template Benefits

Templates promote code reuse and centralized maintenance of code. They provide many of the same benefits of object oriented programming, especially reusability. In addition, templates can compliment and enhance the use of object oriented code by providing easy-to-use wrappers for complex objects. The ABC Templates and ABC Library are a prime example of this synergistic relationship between templates and objects.

### Template Flexibility

You can modify templates to your specifications and store your modifications in the Template Registry. See the *User's Guide—Maintaining*

*Your Templates* for more information. You may also add third party templates and use them in addition to, and along with, the Clarion templates. You may write your own templates too. The Template Language is documented in the *Programmer's Guide* and in the on-line help.

## **Clarion Templates and Application Builder Class (ABC) Templates**

---

Clarion ships with several classes (sets) of templates or template chains. By default, the templates are installed to the \CLARION5\TEMPLATE directory. In addition, the ABC Templates are preregistered when you install Clarion. See *Registering Templates* in the *User's Guide* for more information.

### **ABC Templates**

The ABC Templates are preregistered when you install Clarion. They are the latest templates that use the most advanced code generation capabilities, including generation of object oriented code—code that relies on the Application Builder Class (ABC) Library. ABC Templates include:

ABChain.tpl    **Class ABC - Application Builder Class Templates**  
ABWizard.tpl   **Class ABC Wizards - Clarion Wizard Templates**

### **Clarion (Compatibility) Templates**

In addition to the ABC Templates, Clarion 5 includes the latest Clarion for Windows 2.00x Templates. These templates are not preregistered. They are included for backward compatibility only. We do not recommend these templates for new application development because the ABC Templates are the focus of TopSpeed's ongoing development efforts. The Clarion Templates include:

CW.tpl        **Class Clarion - Clarion Release Templates**  
Wizard.tpl   **Class Wizards - Clarion Wizard Templates**

## **ABC Templates and Code Generation**

---

Clarion's ABC Templates generate source code for you in several ways. The various templates in this package generate everything from single statements to entire procedures and application programs:

### **Class ABC Wizards**

#### **Quick Start Wizard**

Generates a one-file data dictionary *and* an entire application program for viewing, searching, updating, and printing the data.

### **Application Wizard**

Generates an entire application program, including a main menu and subordinate procedures for viewing, searching, updating, and printing data based on an *existing* data dictionary with one *or more* related or unrelated files.

### **Procedure Wizards**

Generate *data* oriented procedures (data browsing, data entry, and reports) based on specific file descriptions in a data dictionary. The generated code accommodates the defined file relationships, by including *multiple procedures as needed* to support both primary and related file updates and validation.

## **Class ABC**

### **Procedure Templates**

Generate a *single* task oriented *or* data oriented procedure (menus, splash screens, data entry, reports, etc.).

### **Control Templates**

Generate the source code to declare one or more window controls *and* to manage the controls, by loading data in and out of the controls, scrolling and selecting the data, etc.

### **Code Templates**

Generate a variety of *task* oriented source code statements at a *single location* that you specify.

### **Extension Templates**

Generate a variety of *task* oriented source code statements at one *or more preset locations* as needed to accomplish the extension's task. Extensions may apply to a single procedure or to an entire application.

Part I of the *Application Handbook* (this part) describes all the Clarion ABC Templates and provides instructions and suggestions for completing their prompts.

## **ABC Templates and the ABC Library**

---

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 *Classes Tab Options (Global)* for more information on configuring the global level interaction between the ABC Templates and the ABC Library. See *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
```

The data declarations instantiate objects from the ABC Library, and the executable statements reference the instantiated objects. The various Application Builder 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>
GlobalErrors	ErrorClass
INIMgr	INIClass
Access: <i>file</i>	FileManager
Relate: <i>file</i>	RelationManager
ThisWindow	WindowManager, ReportManager
BRWn	BrowseClass
BRWn::Sortn:Locator	LocatorClass
BRWn::Sortn:StepClass	StepClass
EditInPlace:: <i>field</i>	EditClass
Popup	PopupClass
Resizer	WindowResizeClass
Toolbar	ToolbarClass
ToolbarForm	ToolbarUpdateClass
RELn::Toolbar	ToolbarReltreeClass
ThisReport	ProcessClass
Previewer	PrintPreviewClass
ThisProcess	ProcessClass
ProgressMgr	StepClass
FDBn	FileDropClass
FDCBn	FileDropComboClass
ViewerN	ASCIIViewerClass
FileLookupN	SelectFileClass
Translator	TranslatorClass

## Browse-Form Application Paradigm

---

There are many different ways to structure a database program and its procedures. By default, Clarion's Wizards and Procedure templates (both ABC and Clarion templates) use a multi-threaded Browse-Form paradigm for the database programs and procedures they generate.

### **Multi-threading**

Multi-threaded programs are the Windows standard, because multiple execution threads allow end users to control their programs by selecting the program or process they need, when they need it. The end users control their programs; the programs do not control the end users.

### **Browse-Form**

Clarion's Browse-Form paradigm uses Browsers (windows with sortable, scrollable, searchable, selectable lists of data), Forms (windows with a single updatable database record), and Reports to organize and present database information to end users. In Clarion's Browse-Form paradigm, the Form not only displays the primary file's record, it also displays related records from other files in the form of child Browsers. The Browse may be enhanced with edit-in-place functionality to provide a more concise user interface with fewer levels of complexity.

### **Browse-Form and Normalized Data**

The beauty of this Browse-Form paradigm is that it works well with any normalized database. Therefore, it can be reliably applied in many situations and it results in a consistent, comfortable, recognizable, tried-and-true (pretested) program for infinite varieties of data. This makes the Browse-Form approach the ideal one for general purpose database programming.

If you need a different application paradigm, then you may want to use individual procedure, control, code, and extension templates to build your application. For example, lots of Browsers can result in lots of network traffic in a client/server application; in that circumstance, a Scrolling-Form paradigm can eliminate Browsers and reduce network traffic.

## ABC Templates and SQL

The ABC Templates are more “SQL Friendly” than the Clarion Templates. That is, the ABC Templates are more efficient (they refresh from disk only when absolutely necessary or when explicitly requested to do so); they can be optimized for use with SQL databases (see *Classes Tab Options—Global—BrowseClass Configuration*); they take advantage of TopSpeed’s Accelerator technology (see *BUFFER in the Language Reference* for more information); and they allow file-loaded browses which eliminate costly backward paging (see *Control Templates—BrowseBox*).

The ABC Templates are more strict in their implementation of some Data Dictionary settings. In particular, the Clarion Templates ignore the “case insensitive key” setting for SQL databases, but the ABC Templates do not.

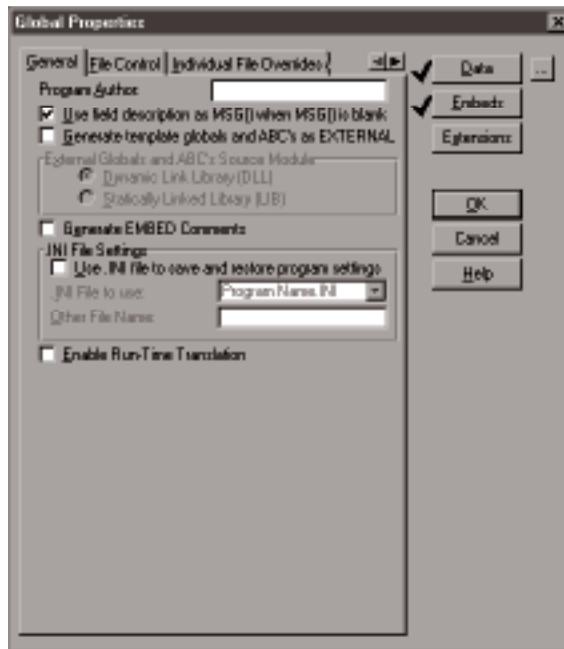
**Note:** Use caution when clearing the Case Sensitive box in your SQL based Data Dictionary. Unlike the Clarion Templates, the ABC Templates enforce caseless keys which can drastically slow down your application.

The ABC Templates apply the SQL UCASE(keyfield) command if you request “caseless” keys. Most SQL backends ignore keys and do a full table scan when a key element is the subject of a scalar function such as UCASE. Therefore, clearing the **Case Sensitive** box in your Data Dictionary results in a UCASE(keyfield) in your generated SQL Statement which forces a full table scan (ignoring any indexes) and drastically slows down your application.

## Global ABC Template Settings

You can specify a number of template settings that apply to your entire application, including file handling defaults, use of .INI files, global variables, and embedded source code. These “global” settings are done primarily through the **Global Properties** dialog.

These global prompts are generated and processed by the Class ABC Application template. Each prompt has a default setting that is appropriate for most applications. In many cases, you can simply use the default setting.



The buttons in the **Global Properties** dialog (**Data**, **Embeds**, and **Extensions**) are provided by the Application Generator, not by the ABC Application Template. See *Global Embed Points* in this chapter and *Global Application Settings* in the *User's Guide* for more information on these buttons.

## General Tab Options

The following options are available on the **General** tab of the **Global Properties** dialog:

### Program Author

Optionally identify the developer or developing organization.

### Use field description as MSG() when MSG() is blank

Check this box to use the Data Dictionary field description as the

default status bar message for each field in your application. See *MSG* in the *Language Reference*.

#### **Generate Template global data and ABC's as EXTERNAL**

Adds the EXTERNAL attribute to the global variable declarations generated by the templates, and the DLL attribute to any CLASS declarations generated by the templates. This means your program relies on an external library to allocate memory for these variables and objects, and to export them so your program can access them.

You should add the EXTERNAL and DLL attributes to get the same effect for any global variables or classes you declare. See the *Language Reference* for more information on these attributes.

**Note:** If you create a program that consists of more than one AppGen created DLL, you should check the Generate Global Data as EXTERNAL box for all the applications except one. See the *User's Guide—Development and Deployment Strategies*.

#### **External Globals and ABC's Source Module**

Specify whether the external library is dynamically or statically linked.

This sets the *flag* parameter of the DLL attribute for template generated class declarations. See the *Language Reference* for more information on the DLL attribute.

#### **Generate EMBED Comments**

Check this box to generate identifying comments surrounding your embedded source code. If you check this box, you should also check the **Enable embed commenting** box in the **Application Options** dialog (choose **Setup** ► **Application Options, Generation** tab) to optimize the comment generation.

#### **Enable Run-Time Translation**

Generates code to translate window text based on the translation strings defined by default in the UTILITY.TRN file. See *Translator Class* for more information.

### **.INI File Support**

The Clarion and ABC Templates support .INI (standard windows initialization) files. These are ASCII text files that store information for an application between sessions.

One use for the .INI file is to store the user's preferred window positions for the next session. Another use is to save program configuration settings between sessions. Clarion's procedure templates let you do both automatically when you enable .INI file support. See *Procedure Templates* for more information.

## To enable automatic .INI file support

1. Select the **General** tab in the **Global Properties** dialog.
2. Check the **Use .INI file to save and restore program settings** box.
3. Specify the .INI file name and its path.

To specify the same file name as the executable, with an .INI extension, choose **Program Name.INI** from the **.INI file to use** drop-down list. This places the INI file in the Windows System directory.

To specify a different name, choose **Other** in the drop-down list, then fill in the **Other File Name** field. You may specify a full pathname, no path, or a path of (.) to generate the INI file as shown below.

Other File Name	resulting INI file location
c:\Programs\Payroll.ini	c:\Programs\Payroll.ini
\Programs\Payroll.cfg	current drive:\Programs\Payroll\Payroll.cfg
Payroll.ini	windows system directory\Payroll.ini
.\Payroll.ini	current directory\Payroll.ini

4. Press the **OK** button to close the **Global Properties** dialog.

See GETINI and PUTINI in the *Language Reference*, and see *INIClass* and *Procedure Templates* in this book for more information.

**Tip:** If your application requires dozens or even hundreds of variables to store from session to session, don't put them in an .INI file, use a control file and normal file I/O instead. Retrieving a variable from an INI file is relatively slow. Also, if you need to hide the information from the end user, remember that INI files are text files, and are easily accessible.

## Saving Global Data Between Sessions

Once you've enabled INI support, the ABC Templates automatically save and restore the values of designated global variables. This provides a simple mechanism for saving and reapplying end user preferences or program configuration options.

1. Press the **Data** button in the **Global Properties** dialog to define your global variables.
2. Press the **Preserve** button in the **Global Properties** dialog to designate selected variables to automatically save and restore.

## File Control Tab Options

The **Global Properties** dialog lets you *override* some of the settings in your data dictionary (see *Dictionary Editor* in the *User's Guide* for more information). You can also define how your procedures access files. You can

specify these file attributes for *all* files, or for *each* file individually. To access these features, select the **File Control** tab.

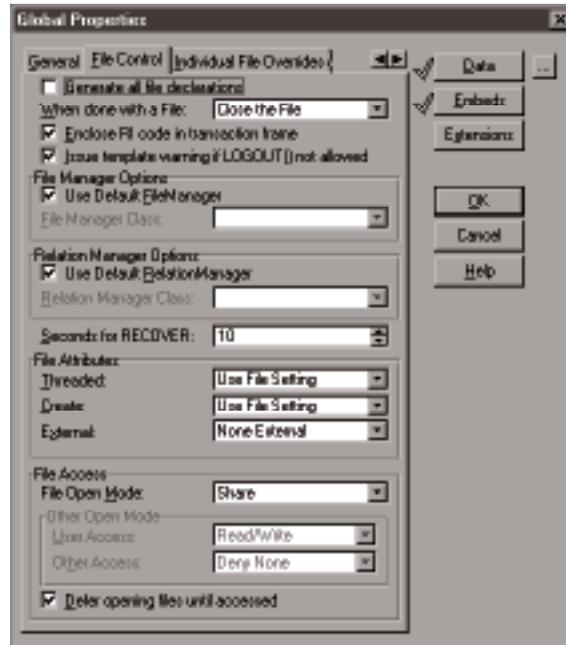
#### Generate all file declarations

Check this box to declare all files in the Data Dictionary, whether or not they are referenced by the application's templates. By declaring all files, you can reference the files in any hand coded source in your application.

#### Enclose RI code in transaction frame

Check this box to ROLLBACK changes if an update fails during a Referential Integrity maintenance operation (transaction). You should clear this box for file systems that do not support transaction frames such as Clipper, dBase, and FoxPro. See *Database Drivers* for information on individual file systems. See *LOGOUT*, *COMMIT*, and *ROLLBACK* in the *Language Reference*.

**Tip:** If all files in a relation chain are using the same file system, and the file system supports transaction framing, and you do not want transaction framing around the RI code, you must clear the check box for each file in Individual File Overrides and in Global Settings.



**Issue Template warning if LOGOUT() not allowed**

When your data dictionary includes a file driver which does not support the LOGOUT statement (used in the Referential Integrity checking routines), checking this box enables a warning at compile time.

You should be sure that this box is *not* checked for drivers such as dBase III. See *Database Drivers* for more information.

**Use default FileManager**

Check this box to generate code that uses the FileManager class named on the **Classes** tab in this dialog. Clear the box to select an alternative class from the **FileManager Class** drop-down list. The ABC Templates instantiate a global FileManager object called *Access:file* for each data dictionary file. The *Access:file* object manages all file access for the ABC Template generated procedures. See *File Manager Class* for more information.

**Use default RelationManager**

Check this box to generate code that uses the RelationManager class named on the **Classes** tab in this dialog. Clear the box to select an alternative class from the **RelationManager Class** drop-down list. The ABC Templates instantiate a global RelationManager object called *Relate:file* for each data dictionary file. The *Relate:file* object manages file relationships, including looking up related records in other files and preserving the integrity of linking fields between related files. See *Relation Manager Class* for more information.

**Seconds for RECOVER**

Specifies the number of seconds to wait before invoking the recovery process. This is applicable only to Clarion files. See *Database Drivers—Clarion Files* for more information.

**Threaded**

Specifies whether the application generator adds the THREAD attribute to FILE structures. THREAD is needed for MDI browse and form procedures to prevent record buffer conflicts when the end user changes focus from one thread to another.

**Use File Setting** Sets the THREAD attribute according to the setting in the data dictionary. See the *User's Guide—Dictionary Editor—File Properties*.

**All Threaded** Adds the THREAD attribute to each FILE.

**None Threaded** Omits the THREAD attribute for each FILE.

**Create**

Specifies whether your application should allow the creation of a data file that does not exist. Adds the CREATE attribute to the FILE structure.

**Use File Setting** Sets the CREATE attribute according to the setting in the data dictionary. See the *User's Guide—The Dictionary Editor—File Properties*.

**Create All** Adds the CREATE attribute to each FILE.

**Create None** Omits the CREATE attribute for each FILE.

### External

Specifies whether the application generator adds the EXTERNAL attribute to FILE structures. EXTERNAL specifies the memory for the FILE's record buffer is allocated by an external library. See the *Language Reference* for more information.

**Note:** When using EXTERNAL to declare a FILE shared by multiple libraries (.LIBs, or .DLLs and .EXE), only one library should define the FILE without the EXTERNAL attribute. This ensures that there is only one record buffer allocated for the FILE and all the libraries and the .EXE will reference the same memory when referring to data elements from that FILE.

**None External** Omits the EXTERNAL attribute from all file declarations and enables the **Export All File Declarations** prompt.

### Export All File Declarations

Check this box to export file declarations (see *Module Definition Files* in the *Programmer's Guide*). This prompt is only available when you specify *Dynamic Link Library (.DLL)* as the **Destination Type** in the **Application Properties** dialog.

**All External** Adds the EXTERNAL attribute to all file declarations *and* lets you specify the **Declaring Module** and whether **All files are declared in another .APP**.

### Declaring Module

The filename (without extension) of the MEMBER module containing the FILE definition without the EXTERNAL attribute. If the FILE is defined in a PROGRAM module, leave this field blank.

### All files are declared in another .APP

Check this box to ensure that files are opened and closed at the right time, thereby preserving

the integrity of the file buffers, when the files are declared in another application (rather than hand code).

#### File Open Mode

Specifies how your application shares files among concurrent users. See the *Language Reference* for more information.

<b>Open</b>	Opens files as: Read/Write (primary user) + Deny Write (all other users).
<b>Share</b>	Opens files as: Read/Write (primary user) + Deny None (all other users).
<b>Other</b>	Specify a custom combination of primary user + other user access.
<b>User Access</b>	Choose from <i>Read Only</i> , <i>Write Only</i> , or <i>Read and Write</i> .
<b>Other Access</b>	Choose from <i>Deny None</i> , <i>Deny All</i> , <i>Deny Read</i> , <i>Deny Write</i> , or <i>Any Access</i> (FCB compatibility mode).

#### Defer opening files until accessed

Specifies when your application opens related files. Check the box to delay opening the file until it is actually accessed.

Delaying the open can improve performance when accessing only one of a series of related files. Clear the box to open the file immediately whenever a related file is opened. See *File Manager Class—LazyOpen* and *UseFile* for more information.

## Individual File Overrides Tab Options

---

Select the **Individual File Overrides** tab to override data dictionary settings or **File Control** tab settings for individual files. Select the file whose attributes you want to change, then press the **Properties** button.

The prompts on this tab mirror those on the **File Control** tab, and they behave exactly the same way, with two exceptions.

- ◆ The settings here apply only to the single file selected.
- ◆ Each drop-down list provides an additional choice: *Use Default*. *Use Default* sets the attribute according to the **File Control** tab.

## External Module Options Tab

---

Select the **External Module Options** tab to set options associated with your application's external modules. This tab is only available when your application contains an external module (LIB or DLL). Select the external module whose attributes you want to change, then press the **Properties** button.

### **Standard ABC LIB/DLL**

Check this box if the LIB or DLL is produced by the ABC Templates or a similar coding scheme. Checking the box generates code to initialize and shut down global objects used by the LIB or DLL. If it is a hand-coded LIB or DLL you should probably clear this box.

## Classes Tab Options—Global

---

By default, the ABC Templates rely heavily on the Application Builder Classes. However, the Templates are highly configurable and are designed to let you substitute your own classes or third party classes if you wish. The Classes tab names *and* configures the classes the ABC Templates use throughout the application. These global settings may be overridden for individual Procedure, Control, and Extension templates. See *Classes Tab Options—Local* for more information on specifying classes for individual templates. We strongly recommend using only ABC Compliant Classes with the ABC Templates. See *ABC Compliant Classes* for more information.

### **Global Properties Classes Tab**

This tab lets you specify the default classes the ABC Templates use to accomplish various tasks. This tab lets you use as much of the ABC Library as you want and as much of your own, or third party classes, as you want. You may override these default class selections with the Classes tab for individual templates.



### Refresh Application Builder Class Information

Press this button if you have changed the contents of or added an include file (.INC) to the \LIBSRC directory. Typically, this is needed when you install third party products that use ABC Compliant Classes, although you may create your own ABC Compliant Classes too. See *ABC Compliant Classes* for more information. The ABC Templates use information gleaned from the header files for generating embed points, loading the Application Builder Class Viewer, application conversion, etc.

### Application Builder Class Viewer

Press this button to display classes, properties, and methods used by the ABC Templates, and the relationships between parent and derived (child) classes. This utility can help you analyze and understand the classes that the ABC Templates use. Once started, the Class Viewer remains open and accessible until you close it, or until you close the Clarion Environment.

### task grouping buttons

Each task grouping button identifies tasks or types of tasks the ABC Templates accomplish. Each button lets you specify the class or classes the ABC Templates use to accomplish the tasks named by the button's text. Following are the ABC Template tasks and their associated default classes.

General	WindowManager
	ErrorClass
	PopupClass
	SelectFileClass

	WindowResizeClass INIClass TranslatorClass
File Management	FileManager ViewManager RelationManager
Browser	BrowseClass StepClass StepLongClass StepRealClass StepStringClass StepCustomClass StepLocator EntryLocator IncrementalLocator FilterLocator FileDropClass FileDropComboClass QueryFormClass QueryFormVisual
Process & Reports	ProcessClass PrintPreviewClass ReportManager
Ascii Viewer	AsciiViewerClass AsciiSearchClass AsciiPrintClass AsciiFileClass
Toolbar Managers	ToolbarClass ToolbarListboxClass ToolbarReltreeClass ToolbarUpdateClass

You may specify alternate classes by typing the class name in the corresponding entry field. The class you name must be an ABC Compliant Class (see *ABC Compliant Classes* for more information).

### **Default Classes Configuration**

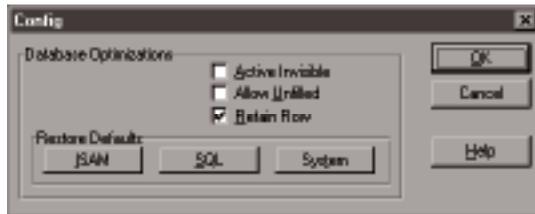
Some of the **General** task classes offer configuration options. This is indicated by the presence of a **Configure** button immediately below the class label. Press the **Configure** button to set default runtime behavior for all objects of the class throughout the application. To override these global configuration settings, you can embed the SetABCProperty code template to set corresponding properties within individual procedures.

## **BrowseClass Configuration**

The ABC Application Template provides the following configuration prompts for the BrowseClass:

### **Active Invisible**

Check this box to fill the browse queue even when the browse LIST is “invisible” because it is on a non-selected TAB or is otherwise hidden. Clear the box to suppress the refill when the listbox is hidden. Clearing the box improves performance for procedures with invisible browse lists; however, buffer contents for the invisible browse list should not be relied upon. See *BrowseClass Properties—ActiveInvisible*.



### **Allow Unfilled**

Check this box to allow a partially filled LIST when the result set “ends” in mid-list. This improves (SQL) performance by suppressing additional reads needed to fill the list. Clear the box to always display a “full” list. See *BrowseClass Properties—AllowUnfilled*.

### **Retain Row**

Check this box to maintain the highlight bar in the same list row following a change in sort order, an update, or other browse refresh action. This can cause a performance penalty in applications using TopSpeed’s pre-Accelerator ODBC driver.. Clear the box to allow the highlight bar to move. See *BrowseClass Properties—RetainRow*.

### **ISAM**

Press this button to optimize the configuration check boxes for ISAM file systems.

### **SQL**

Press this button to optimize the configuration check boxes for SQL database systems.

### **System**

Press this button to set the configuration check boxes to let the BrowseClass object choose the best action.

## WindowManager Configuration

The ABC Application Template provides the following configuration prompts for the WindowManager Class:

### **Reset on gain focus**

Check this box to make the WindowManager unconditionally reset whenever the window receives focus. Clear the box to allow 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). See *WindowManagerClass Properties—ResetOnGainFocus*.

### **Auto Tool Bar**

Check this box to make the WindowManager try to set the appropriate ToolbarTarget whenever the end user selects a new TAB control. Clear the box to manually set the ToolbarTarget or use the current ToolbarTarget. See *WindowManagerClass Properties—AutoToolbar, Toolbar Classes* and *FrameBrowseControl* for more information.

## WindowResizeClass Configuration

The ABC Application Template provides the following configuration prompts for the WindowResizeClass:

### **Automatically find parent controls**

Check this box to make each Resizer object set parent/child relationships among window controls. Clearing the box makes the WINDOW the parent of all its controls. Setting parent/child relationships lets any special scaling cascade from parent to child. See *WindowResizeClass Methods—SetParentDefaults* for more information.

### **Optimize Moves**

Check this box to move all controls at once during the resize operation, producing a snappier resize and avoiding bugs on some windows. See *WindowResizeClass Properties—DeferMoves* for more information.

### **Optimize Redraws**

Check this box to make controls transparent (TRN attribute) during the resize operation, producing a smoother redraw and avoiding bugs on some windows. See *WindowResizeClass Properties—AutoTransparent* for more information.

## TranslatorClass Configuration

The ABC Application Template provides the following configuration prompts for the TranslatorClass:

### **Extract Filename**

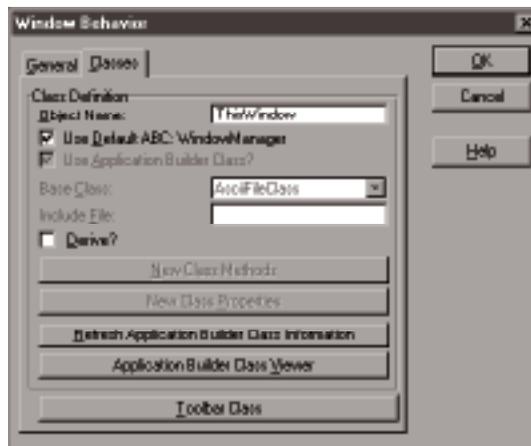
Specify a filename to receive a list of all runtime text that may require translation for multi-language applications. See *TranslatorClass Properties—ExtractText* for more information.

**Tip:** You must check the **Enable Run-Time Translation** box on the **General** tab to enable this option.

## Classes Tab Options—Local

Many of the ABC Procedure, Control and Extension templates provide a Classes tab or dialog. These local Classes tabs let you control the classes (and objects) your procedure uses to accomplish the template's task—that is, they override the global class settings specified in the **Global Properties** dialog (see *Classes Tab Options—Global*).

You may accept the default Application Builder Class specified in the **Global Properties** dialog (recommended), or you may specify your own or a third party class to override the default setting. Deriving your own class can give you very fine control over the procedure when the standard Application Builder Class is not precisely what you need. We strongly recommend using only ABC Compliant Classes with the ABC Templates. See *ABC Compliant Classes* for more information.



### Object Name

Set the object's label for the template generated code. By fine tuning the object names, you can make your generated code easier to read.

### Use Default Application Builder Class?

Check this box to use the default Application Builder Class specified in the **Global Properties** dialog (see *Template Overview—Classes Tab Options* for more information). Clear this box to use a class other than the default, and to enable the following prompts.

### Use Application Builder Class?

Check this box to select a class from the **Base Class** drop-down list. The list includes all ABC Compliant Classes (see *ABC Compliant Classes* for more information). Clear this box to specify a non-compliant class (not recommended).

**Base Class** If you checked the **Use Application Builder Class?** box, select a class from the drop-down list. If you cleared the **Use Application Builder Class?** box, type the class label here, and type the name of the source file that contains the class declaration in the **Include File** entry box.

**Include File** If you cleared the **Use Application Builder Class?** box, type the class label in the **Base Class** entry box, and type the name of the source file that contains the class declaration here.

### Derive?

Check this box to derive a class based on the parent class specified above and to enable the **New Class Methods** and **New Class Properties** buttons to define any *new* properties and methods for the derived class.

This prompt is primarily to allow you to define *new* properties and methods in a derived class. To override *existing* methods, simply embed code in the corresponding method embed points.

Using **Derive?**, **New Class Methods** and **New Class Properties** makes the template generate code similar to the following:

```
MyProcess CLASS(Process)      !derive a class from the parent class
NewMethod PROCEDURE          !prototype new class method
NewProperty BYTE              !declare new class property
END
```

**Tip:** The template automatically derives from the parent class if you embed code into any of the derived method embed points, regardless of the status of this check box. See *Using ABC Templates to Derive Classes* for more information.

### New Class Methods

Press this button to specify the *new* method prototypes to generate into the derived CLASS structure. This opens the **New Class Methods** dialog (see *New Class Methods*).

### New Class Properties

Press this button to specify the new property declarations to generate into the derived CLASS structure. This opens the **New Class Properties** dialog (see *New Class Properties*).

### Application Builder Class Viewer

Press this button to display classes, properties, and methods used by the ABC Templates, and the relationships between parent and derived (child) classes. This utility can help you analyze and understand the classes that the ABC Templates use.

### Refresh Application Builder Class Information

Press this button if you have changed the contents of an include file (.INC) or added an include file to the \LIBSRC directory. Typically, this is needed when you install third party products that use ABC Compliant Classes, although you may create your own ABC Compliant Classes too. See *ABC Compliant Classes* for more information. The ABC Templates use information gleaned from the header files for generating embed points, loading the Application Builder Class Viewer, application conversion, etc.

### composite Class

Press these buttons to open a Classes dialog for each class used by the parent class specified above. For example, the WindowManager uses a Toolbar class, so the WindowManager's Classes dialog contains a Toolbar Class button to open a Classes dialog for its Toolbar Class.

## New Class Methods

### New Class Methods

Press this button to specify the *new* method prototypes to generate into the derived CLASS structure. This opens the **New Class Methods** dialog. Press the **Insert** button to add the new method prototype and the method's associated embed points.

#### New Method Name

Type the method label.

#### New Method Prototype

Type the method parameter list and return data type. If the method takes no parameters but has a return value, type parentheses and a comma before the return data type. Do not type "PROCEDURE" or "FUNCTION" because the template generates the PROCEDURE statement for you.



**Data Embed** Press this button to use the Text Editor to implement the method's data section.

**Code Embed** Press this button to use the Text Editor to implement the method's code section.

## New Class Properties

### New Class Properties

Press this button to specify the new property declarations to generate into the derived CLASS structure. This opens the **New Class Properties** dialog. Press the **Insert** button to add a new property declaration.

**Property Name** Type the property label.

**Property Type** Select a simple data type from the list or select *Other* to enable the **Other Data Type** field.

### Other Data Type

Type the label of a user defined complex data type (such as the label of a GROUP, QUEUE or CLASS), or type a valid entity data type (such as FILE, VIEW, or WINDOW).

**Is a Reference** Check this box to declare a reference variable. You must use a reference variable for entity data types and for any complex data type not valid within a GROUP. You may use a reference variable for any other data type. See *GROUP* and *Reference Variables* in the *Language Reference*.



**Size** Specify the length of the field in bytes.

**Dimensions** To declare the field as an array, and to specify the array dimensions, specify a size for up to four dimensions. Total array size may not exceed 65,520 bytes. See the *Language Reference* for more information on dimensioned variables and arrays.

## ABC Compliant Classes

The classes you use with the ABC Templates must be ABC Compliant Classes. That is, the classes must conform to the ABC Library specification as documented in *Part II—Application Builder Class Library*.

The ABC Templates generate code that refers to the properties, methods, and method parameters documented in *Part II* of this book. If those properties, methods, and parameters are not defined within the classes you specify, the template generated code will not compile. Further, if the classes do not perform as documented, the template generated code probably won't work. The easiest way to create ABC Compliant Classes is to derive classes from the ABC Library. See CLASS in the *Language Reference* for more information on deriving classes.

**Tip:** Copy an AB\*.INC/AB\*.CLW file pair from the Clarion \LIBSRC folder to use as the starting point for you ABC Compliant Classes.

### Requirements for ABC Compliant Classes

- Classes must conform to the ABC Library specification as documented in *Part II—Application Builder Class Library*
- The header file containing the CLASS declarations must have the .INC file extension
- The header file (.INC) containing the CLASS declarations must be in Clarion's \LIBSRC directory
- The header file (.INC) containing the CLASS declaration must contain the following comment before compilable code begins:  
!ABCIncludeFile
- The CLASS declarations must have the LINK attribute naming the corresponding implementation (.CLW) files.

Meeting these requirements ensures that your ABC Compliant Classes appear in the Application Builder Class Viewer, the Embeditor, the **Embedded Source** dialog, and that the development environment has full information about your classes. With this information, the development environment can correctly manage embed points and code generation for the compliant classes.

**Tip:** Clarion's Application Generator automatically provides embed points for each compliant class method.

## Global ABC Embed Points

The ABC Application Template provides global embed points to allow customization of

- the application's global MAP
- the application's global data
- program initialization and termination
- file handling methods (open, close, field validation, record priming, etc.) for all data dictionary files
- the export file (.EXP—also known as the Module Definition file—see the *Programmers Guide* for more information)
- the application ship list

To access these embed points, press the **Embeds** button in the **Global Properties** dialog or from the Application Tree, select the **Module** tab, RIGHT-CLICK on the Default Program module, then choose **Embeds** from the popup menu.

As with any embed point, you can write your own custom code, call a procedure, or use a code template. The Application Generator, when generating code, places your code or calls your procedure at the next source code line following the point you pick from the **Embedded Source** dialog. See *Application Generator—Embedded Source Code* in the *User's Guide* for more information on adding embedded source code to your application.

### File Specific Embed Points

The ABC Template global embed points include embed points for the FileManager and RelationManager for *each* data dictionary file. Embedding code into these embed points generates code to derive FileManager and RelationManager methods that override the parent class methods for the individual data dictionary files. See *File Manager Methods* and *Relation Manager Methods* for more information on these methods.

The ABC Application Template generates the derived FileManager and RelationManager methods into the *appnaBC0.CLW* through *appnaBC9.CLW* modules (where *appna* is the first five characters of the application filename). The number of modules actually generated depends on the number of files in your dictionary. By default, the ABC Templates generate code for twenty (20) data dictionary files in each *appnaBCn.CLW* module. You can change this default by changing the value of the %FilesPerBCModule template symbol.

## Field Specific Embed Points—Field Priming and Validation

The ABC Templates include global embed points for individual field priming and individual field validation (before validation and upon validation failure).

Embedding code into these embed points generates code to derive FileManager and RelationManager methods that override the parent class methods for individual data dictionary files. See *File Manager Methods* and *Relation Manager Methods* for more information on these methods.

These embed points provide a single application-wide place to prime and validate fields. The ABC Template generated code automatically calls the field priming and field validation methods whenever your application adds or changes data dictionary files.

**Tip:** These global ABC Template embed points provide a single place where you can handle NULL settings for SQL based applications.

## Global Data and Variables

Global data must be declared before the CODE statement in your PROGRAM module (see the *Language Reference* for more information). There are several ways to accomplish this with the Clarion environment. You can declare global data in the data dictionary (see *Dictionary Editor—File Properties*); you can declare global data with the **Data** button in the **Global Properties** dialog; and you can declare global data with the **Embeds** button in the **Global Properties** dialog (embed data declarations in a data section embed point—see *Embedded Source Code*).

data dictionary global data

declares global data that can be shared by several applications. Because it is declared with the **Field Properties** dialog, you can specify controls and properties to apply to the data each time you populate them on your application's windows and reports.

**Global Properties** dialog **Data** button

declares global data for a single application. Because it is declared with the **Field Properties** dialog, you can specify controls and properties to apply to the data each time you populate them on your application's windows and reports. You can automatically save and restore these values between sessions with the **Preserve** button in the **Global Properties** dialog.

**Global Properties** dialog **Embeds** button

declares global data for a single application with free form source code.

## Using ABC Templates to Derive Classes

For the purposes of this discussion, deriving classes means generating a CLASS structure containing data declarations (properties) and method prototypes, plus generating the corresponding method implementation code. For example:

```

Whatever PROCEDURE

BRW1 CLASS(BrowseClass)           !derive BRW1 CLASS from BrowseClass
MySwitch BYTE                     !declare a new BRW1 property
ResetSort PROCEDURE,VIRTUAL      !prototype a BRW1 override method
MyMethod PROCEDURE               !prototype a new BRW1 method
                                END
                                CODE
                                !procedure code

BRW1.ResetSort PROCEDURE          !method definition/implementation
                                CODE
                                !some embedded code
                                PARENT.ResetSort           !preserve documented functionality
                                !some embedded code

BRW1.MyMethod PROCEDURE          !method definition/implementation
                                CODE
                                !some embedded code

```

## Why the Templates Derive Classes

---

The ABC Templates derive classes *so they can use virtual methods* to customize the derived class object's (Local Object) behavior for a specific procedure's (or program's) requirements. 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. Then, 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 method 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.

The ABC Templates provide two different mechanisms to derive classes.

- Embed points
- Classes tabs

**Tip:** To derive from the FileManager, you can place code into FileManager global embed points to override existing FileManager methods, or you can create an ABC Compliant FileManager (see *ABC Compliant Classes*) to add new methods.

## Deriving with Embed Points

---

Embedding code into an ABC “Method” (Local Object) embed point automatically generates a CLASS statement if necessary, plus the derived method’s prototype, plus the derived method’s implementation code. The generated implementation code includes your embedded code plus a call to the corresponding parent class method. This guarantees the derived method retains the parent method’s documented functionality, plus your embedded code’s additional functionality. You can remove the parent method functionality by embedding a RETURN before the call to the parent class method.



If the derived method is VIRTUAL, the template generated code need not explicitly call the method, because the parent class object calls the derived VIRTUAL method. However, if the derived method is not VIRTUAL, the template generated code must call the derived method or it won’t execute. The parent class object calls VIRTUAL methods in the derived class; it does *not* call non-virtual methods in the derived class.

**Tip:** To see which methods are virtual and which are not, right-click on a procedure in the Application Tree, then choose Source from the popup menu. In the Embeditor, search for “VIRTUAL.”

See *Application Generator—Embedded Source Code* in the *User’s Guide* for more information.

## Deriving with Classes Tab

---

Each ABC Template that generates code to instantiate an ABC object provides a Classes tab or dialog to help you derive new methods and properties for its object. Checking the **Derive?** box generates the CLASS statement if necessary. Pressing the **New Class Methods** button let’s you specify the new method prototypes *and* implementation code. Pressing the **New Class Properties** button let’s you specify the new data declarations (properties) to generate within the CLASS structure. See *Classes Tab Options—Local* for more information.

To override *existing* methods, use their corresponding embed points. See *Deriving with Embed Points* for more information.

## 2 - WIZARDS AND UTILITY TEMPLATES

### Code Generation Wizards

Clarion provides *WIZARDS*—powerful utility templates that let you create a Browse, Form, or Report procedure by answering a few quick questions. You can even use a wizard to create an entire application from an existing data dictionary!

#### Browse-Form Paradigm

Clarion's code generation wizards follow the Browse-Form application paradigm. Clarion's Browse-Form paradigm uses Browsers (windows with sortable, scrollable, searchable, selectable lists of data), Forms (windows with a single updatable database record), and Reports to organize and present database information to end users. See *Template Overview—Browse-Form Application Paradigm* for more information.

#### Fine Tuning the Wizards

Options you specify in advance in the Clarion data dictionary provide additional control over the procedures the wizards create. See *Optimizing the Wizards* for more information.

#### Starting the Wizards

To start the code generation wizards, choose **Application ► Template Utility**, then select the wizard from the list. Alternatively, create a new procedure (choose **Procedure ► New**), and check the **Use Procedure Wizard** box in the **Select Procedure Type** dialog.

## Application Wizards

Clarion's Application Wizards generate an entire application program. The program includes a main menu and subordinate procedures for viewing, searching, updating, and printing data from one or more files. The Quick Start Wizard generates a one-file data dictionary and program. The Application Wizard generates a full program based on an existing data dictionary with one *or more* related or unrelated files.

### Quick Start Wizard

Using the Quick Start Wizard, you can create a data dictionary and a working application with no coding required. See *Getting Started* for a step-by-step example of using the Quick Start Wizard.

Simply define a data file, then the Quick Start Wizard creates a complete Windows application. The entire process takes less than five minutes! Your application has a form procedure for updating the file, a multi-keyed browse procedure, and as many reports as the data file has keys.



Just define the fields for a single file. For each field, you provide a name, display format picture, and key information. This creates a data dictionary. The Quick Start Wizard creates the application based on this dictionary. Once you've specified all options, the OK button generates the .APP file, and loads the procedures into the Application Tree dialog.

*To use the Quick Start Wizard:*

1. In Clarion, choose **File ► New ► Application**.  
This opens the **New** file dialog.
2. Type a name for the .APP file in the **Filename** field.  
Clarion automatically adds the .APP extension.

3. Check the **Use Quick Start** box below the file list, then press the **Save** button.

This starts the **Quick Start Wizard**. This dialog lets you describe the file on which the application and data dictionary are based. Fill in the prompts as described below.

#### **Application Name**

The file name for the .APP file. The Quick Start Wizard uses the same file name (with the .DCT extension) for the data dictionary file.

Optionally press the ellipsis button ( ... ) to change the directory, then type a file name in the Open File dialog box. The working directory, in which all source code files are generated, depends on where the .APP file resides.

#### **Data File Name**

Type the file name (no extension necessary) for the data file.

#### **Prefix**

This box automatically fills in with the first three letters of the name of the data file when you TAB from the Data File Name box. Optionally specify up to 14 letters of your choice in this field.

The prefix allows your application to distinguish between identical variable names occurring in different file structures. A field called Invoice may exist in one data file called Orders and another called Sales. By establishing a unique prefix for Orders (ORD) and Sales (SAL), the application may distinguish the two fields as *ORD:Invoice* and *SAL:Invoice*.

#### **File Driver**

Specify the data file type. When using the Application Generator, Clarion automatically links in the correct database driver library. See *Database Drivers* for a discussion of the relative advantages of each driver.

The individual file drivers may vary in their support of some of the attributes which you add to the FILE structure in this dialog box.

#### **Field Name**

To name each field, type a valid Clarion label in the Name field. Valid field names may vary slightly according to the file driver.

#### **Picture**

Specify a default picture token by typing it in the **Picture** field. The picture token, together with the selected File Driver, determine the data type which the Quick Start Wizard uses for the field. When the Application Generator creates window and report controls for the field, this also serves as the default picture for the control.

**Key**

This specifies whether to create a key using this field as a component, and if so, the type of key. By specifying *Unique*, your application ensures that each record has a distinct value. *Duplicate* specifies a key that allows more than one record with the same value in the key component. *Autonumber* specifies a unique numeric key that your program automatically increments whenever a new record is added. The Wizard generates the code to increment the key value.

The Quick Start Wizard creates a Browse list sortable on every key you specify. It also creates a Report for each key.

4. Press the TAB key to define the next field in your file.

Alternatively, you may use the command buttons to define fields.

**Insert** This button inserts a new field above the selected field.

**Delete** This button deletes the selected field.



This button moves the selected field up one position in the fields list.



This button moves the selected field down one position in the fields list.

5. When you have defined all the fields, press the **OK** button.

The Quick Start Wizard creates your dictionary and its associated application, then displays the Application Tree.

## Application Wizard

---

The Application Wizard creates a complete application from an existing data dictionary. It creates a main procedure containing a menu with options calling all subordinate procedures it creates. It also creates Browse, Report, and Form (update) procedures for each specified file.

### Two Types of Applications for Compatibility

The Application Wizard can create two different types of applications: “Full” applications for compatibility with Clarion 1.5 through 2.001, and “Simple” applications beginning with Clarion Standard Edition 2.002. By default, the Clarion version 2.002 and higher Application Wizard creates Simple applications.

Simple applications are smaller, simpler, and provide all the functionality of full applications. Simple applications eliminate Range Limited Browsers, and instead access child files through secondary Browsers on Form (update) procedures.

Full application Browsers have buttons to access all child files, as well as buttons to access parent files. These related file buttons call Range Limited Browsers.

You can change the default application type, by editing `..\TEMPLATE\ABWIZARD.TPL` to specify a different value for `%ProgramType`. By default, `%ProgramType` is set to 'Simple.' To generate Full applications compatible with earlier versions of Clarion, set the `%ProgramType` to 'Full' as follows (comment out the 'Simple' declaration and uncomment the 'Full' declaration):

```
#DECLARE(%ProgramType)
#!SET(%ProgramType,'Simple')           #!Mutually exclusive options
#SET(%ProgramType,'Full')
```

## Special Application Features for SQL File Systems

For SQL based file systems, the Application Wizard also generates code to capture user login information upon program startup, then reuse the login information for each file accessed.

## Creating the Application—Starting the Application Wizard

*To use the Application Wizard:*

1. In Clarion, choose **File ► New ► Application**.  
This opens the **New** file dialog.
2. Type a name for the .APP file in the **Filename** field, then press the **Save** button.  
  
Clarion automatically adds the .APP extension. Don't use the **Quick Start Wizard**—clear the box below the file list.  
  
This opens the **Application Properties** dialog which lets you define the basic files and properties for the application.
3. Name the .DCT file the application uses in the **Dictionary File** field, or press the ellipsis (...) button to select the file in the **Select Dictionary** dialog.
4. Optionally, rename the **First Procedure** or accept the default—*Main*.  
This is the name of the application's Frame procedure. You may rename it later if you want to.
5. Choose the **Destination Type** from the drop-down list.  
This defines the target file for your application. Choose from *Executable (.EXE)*, *Library (.LIB)*, or *Dynamic Link Library (.DLL)*.
6. Optionally, type a name for the application's Windows help file in the **Help File** field, or use the ellipsis (...) button to select the file with the **Open File** dialog.

If you name a file, it must exist; however, it need not be a true Windows help file. The Application Generator lets you name the help topics in your application even though the topics do not exist in the specified help file. You are responsible for creating a help file that contains the context strings and keywords that you optionally enter as HLP attributes for the various controls and dialogs.

7. Accept the default *ABC* template in the **Application Template** field.

The selected application template controls source code generation. You may choose other Clarion templates or third party templates that you have registered.

8. Accept the default *ToDo(ABC)* template in the **ToDo Template** field.

The selected ToDo template controls source code generation. You may choose other Clarion templates or third party templates that you have registered.

9. Check the **Use Application Wizard** box to use the wizard to create a complete application based on the selected dictionary and a few answers you specify.

10. Press the **OK** button.

This creates the application file then starts the **Application Wizard**.

**Tip:** To write over part of an existing application, open it, then choose **Application ► Template Utility** to start the Application Wizard.

## Using the Application Wizard

1. Answer the questions in each dialog, then press the **Next** button.

The **Application Wizard** presents the following questions:

### **Generate Procedures for all files in my dictionary**

Check the box for all files, or clear the box to select specific files.

### **Which control model should the Application use?**

**Button**            The wizard builds the application with traditional **Insert, Change, Delete, OK, and Cancel** command buttons that appear on each dialog.

**Toolbar**            The wizard builds the application with global toolbar command buttons that appear on the application frame. The toolbar buttons control each dialog. See *Control Templates—FrameBrowseControl* for more information.



**Both** The wizard builds the application with both the traditional dialog command buttons and the global toolbar command buttons.

### Overwrite existing procedures

Check this box to overwrite existing procedures with the same names. Clear the box to preserve existing procedures.

### Generate Reports for each file

Check this box to automatically generate report procedures. Clear the box to omit report procedures.

2. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

You can press the **Back** button to change a prior selection or press the **Cancel** button to abandon the application.

The Application Wizard creates the .APP file based on the dictionary and the answers you provided, then displays the **Application Tree** dialog for your new application.

### Fine Tuning the Wizard

You can control how the wizard builds your application by specifying options for Files, Fields, Keys, and Relationships in the Data Dictionary (see *Optimizing the Wizards*).

## Procedure Wizards

Clarion's Procedure Wizards generate one or more *data* (file) oriented procedures (data browsing, data entry, and reports) based on specific file descriptions in a data dictionary. The generated code accommodates the defined file relationships, including multiple procedures as needed to support related file updates and data validation.

### Browse Wizard

This wizard creates a multi-keyed Browse procedure from an existing dictionary file definition. The BrowseBox is sorted by each key you specify. The sort order is controlled by TABs. It also creates associated Form (Update) procedures, if you specify that updates are allowed.



To use the Browse Procedure Wizard:

1. Choose **Application ► Template Utility**, then select *Browse Wizard* and skip to step 4.

Or:

1. In the **Select Procedure** dialog, check the **Use Procedure Wizard** box.

You can open the **Select Procedure** dialog by selecting a ToDo procedure in the **Application Tree**, then pressing ENTER, or by simply pressing the INSERT key, then typing the procedure name in the **New Procedure** dialog.

2. In the **Select Procedure** dialog, choose **Browse** from the list of Procedure templates.
3. Press the **Select** button.

This starts the **Browse Wizard**.

4. Answer the questions in each dialog, then press the **Next** button.

The Browse Wizard presents the following questions:

**What name should be used as the label of the procedure?**

Type the browse procedure name.

**Which file do you want to browse?**

Press the ellipsis (...) button to select a file from the dictionary.

**Browse using all record keys**

Check this box to make the list sortable on all keys. Clear the box to specify a single sort key.

**Allow the user to update records**

Check this box to generate a subordinate procedure to update the file. Optionally, provide the name of the update procedure. Clear the box to make the list read only.

**Call update using popup menu**

Check this box to provide RIGHT-CLICK popup menus on the Browse list in addition to any command or toolbar buttons.

**Parent Record Selection**

This prompt appears only if you specify a single sort key that is the linking key in a Many:One relationship. The Browse Wizard infers from this that you may want to browse only the child records for a specific parent record. Select one of the following to confirm or deny this inference.

**Do not select by parent record**

Do not limit the browse—in other words, browse all records.

**Select parent record via button**

Browse only the child records for a specific parent record. Provide a button to select the parent record.

**Assume that the parent record is active**

Browse only the child records for a specific parent record. Assume the parent record is already active.

**Provide buttons for child files**

Check this box to provide buttons on the Browse window to access related child files. Alternatively, related files may be accessed from the generated update procedure.

**Provide a “Select” button**

Check this box to provide a “Select” button that displays when the Browse procedure is called to select a record, but is hidden when the Browse is called to update records.

**Which control model should the Application use?****Button**

The wizard builds the browse with traditional **Insert**, **Change**, and **Delete** command buttons that appear on each dialog.

- |                |  |
|----------------|--|
| <b>Toolbar</b> | The wizard builds the browse to use global toolbar command buttons that appear on the application frame. See <i>Control Templates—FrameBrowseControl</i> . |
| <b>Both</b>    | The wizard builds the browse to use both traditional dialog command buttons and global toolbar command buttons.  |

### Overwrite existing procedures

Check this box to overwrite existing procedures with the same names. Clear the box to preserve existing procedures.

5. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Browse Procedure Wizard creates the procedure(s) based on the dictionary file and the answers you provided, then displays the **Procedure Properties** dialog for your new procedure.

### Fine Tuning the Wizard

You can control how the wizard builds your procedures by setting Options for Files, Fields, Keys, and relationships in the Data Dictionary (see *Optimizing the Wizards*).

## Form Wizard

---

This wizard creates an update Form Procedure from an existing dictionary file definition.



To use the Form Procedure Wizard:

1. Choose **Application ► Template Utility**, then select *FormWizard* and skip to step 4.

Or:

1. In the **Select Procedure** dialog, check the **Use Procedure Wizard** box.  
You can open the **Select Procedure** dialog by selecting a ToDo procedure in the **Application Tree**, then pressing ENTER, or by simply pressing the INSERT key, then typing the procedure name in the **New Procedure** dialog.
2. In the **Select Procedure** dialog, choose **Form** from the list of Procedure templates.
3. Press the **Select** button.

This starts the **Form Wizard**.

4. Answer the questions in each dialog, then press the **Next** button.

The Form Wizard presents the following questions:

**What name should be used as the label of the form procedure?**

Type the procedure name.

**Which file do you want the form to update?**

Press the ellipsis (...) button to select a file from the dictionary.

**Allow Records To Be Added**

Check this box to allow new records.

**Allow Records To Be Modified**

Check this box to allow records to be changed.

**Allow Records To Be Deleted**

Check this box to allow records to be deleted.

**Insert Message**

Type the titlebar text to display when adding a record.

**Change Message**

Type the text to display when changing a record.

**Delete Message**

Type the text to display when deleting a record.

**Where do you want this message to be displayed?**

Choose the title bar or the status bar.

**A field can be displayed that identifies the active record.**

Press the ellipsis button to select a field from the dictionary to display on the window titlebar.

**Validate field values whenever field value changes?**

Check this box for immediate validation when the end user “accepts” the field.

**Validate field values when the OK button is pressed?**

Check this box for field validation on the OK button.

**Browsing child files**

Select one of the following choices.

**Place children on tabs**

### Access children with push buttons Do not provide child access

#### Which control model should the Application use?

<b>Button</b>	The wizard builds the dialogs with traditional <b>Insert</b> , <b>Change</b> , and <b>Delete</b> command buttons.
<b>Toolbar</b>	The wizard builds the form to use global toolbar command buttons that appear on the application frame. See <i>Control Templates—FrameBrowseControl</i> .
<b>Both</b>	The wizard builds the form to use both traditional dialog command buttons and global toolbar command buttons.

#### Overwrite existing procedures

Check this box to overwrite existing procedures with the same names. Clear the box to preserve existing procedures.

5. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Form Procedure Wizard creates the procedure(s) based on the dictionary file and the answers you provided, then displays the **Procedure Properties** dialog for your new procedure.

### Fine Tuning the Wizard

You can control how the wizard builds your procedures by setting Options for Files, Fields, Keys, and relationships in the Data Dictionary (see *Optimizing the Wizards*).

## Report Wizard

---

This wizard creates a Report Procedure from an existing dictionary file definition.

*To use the Report Procedure Wizard:*

1. Choose **Application ► Template Utility**, then Select ReportWizard and skip to step 4.

Or:

7. In the **Select Procedure** dialog, check the **Use Procedure Wizard** box.

You can open the **Select Procedure** dialog by selecting a ToDo procedure in the **Application Tree**, then pressing ENTER, or by simply pressing the INSERT key, then typing the procedure name in the **New Procedure** dialog.

2. In the **Select Procedure** dialog, choose **Report** from the list of Procedure templates.

3. Press the **Select** button.

This starts the **Report Wizard**.

4. Answer the questions in each dialog, then press the **Next** button.

The Report Wizard presents the following questions:

**What name should be used as the label of the report procedure?**

Type the procedure name.

**Which file do you want to report?**

Press the ellipsis (...) button to select a file from the dictionary.

**Enter a key below, or leave the field blank to run in record order.**

Press the ellipsis (...) button to select a sort key. Leave the field blank to specify no sort key.

**How many columns do you want the report to use?**

Type the number of columns for your report. The Report Wizard distributes the report fields evenly across the columns.

**Overwrite existing procedures**

Check this box to overwrite existing procedures with the same names. Clear the box to preserve existing procedures.

5. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Report Procedure Wizard creates the procedure based on the dictionary file and the answers you provided, then displays the **Procedure Properties** dialog for your new procedure.

### Fine Tuning the Wizard

You can control how the wizard builds your procedures by setting Options for Files, Fields, Keys, and relationships in the Data Dictionary (see *Optimizing the Wizards*).

## Dictionary Print Wizard

This wizard prints descriptions of data dictionary files at varying levels of detail for files, fields, keys, and relationships. You may print to the printer or to a file.

*To use the Dictionary Print Wizard:*

1. Open an application that uses the dictionary.
2. Choose **Application ► Template Utility** from the menu.  
This opens the **Select Utility** dialog.
3. Highlight **DictionaryPrint**, then press the **Select** button.  
This starts the **Dictionary Print Wizard**.
4. Answer the question(s) in each dialog, then press the **Next** button.  
After the first dialog, the **Finish** button is enabled. Press the **Finish** button now to print all the information available for all the files, fields, keys, and relationships.  
  
Or, step through the wizard's dialogs, to select specific files, plus the level of detail to print (All, Some, or None) for the various dictionary components.

## Optimizing the Wizards

Wizard Options in the Data Dictionary Editor provide more control over the wizards' functionality. Wizards use the Options specified for a file, field, key, or alias when creating procedures. In addition, the Wizards use file, field, key, and alias names and descriptions for the text on menus, title bars, tabs, etc.

### File Options

---

#### **Do Not Auto-Populate This File**

Directs the wizards to skip this file when creating primary Browse procedures or Report procedures.

#### **User Options**

User Options let you provide information to utility templates. User Options are comma delimited, that is, each entry is separated by a comma. Choose from the following:

**EDITINPLACE**     The Browse Wizard provides edit-in-place updates to the browsed file instead of a separate update (form) procedure. We recommend this option for files with one-way lookup relationships, such as a State Code file. Files with complex relationships are better managed with a separate update procedure.

### Alias Options

---

#### **Do Not Auto-Populate This Aliased File**

Directs the wizards to skip the Aliased File when creating primary Browse procedures or Report procedures.

#### **User Options**

User Options let you provide information to utility templates. User Options are comma delimited, that is, each entry is separated by a comma.

### Field Options

---

#### **Do Not Auto-Populate This Field**

Directs the wizards to skip this field when creating Form, Browse or Report procedures.

#### **Population Order**

Specifies the order in which the wizards populate fields. Choose Normal, First, or Last from the drop-down list. Wizards populate in this order: all Fields specified as First, then all Fields specified as Normal, and finally all Fields specified as Last.

**Form Tab**

Specifies the TAB onto which the wizards populate the field. Type the Caption for the TAB or select one you have previously created from the drop-down list. This lets you direct the wizard to group fields in the manner you want.

**Add Extra Vertical Space Before Field Controls on Forms**

Check this box to direct the wizards to add vertical space between this field's control and the one populated above it.

**User Options**

User Options let you provide information to utility templates. User Options are comma delimited, that is, each entry is separated by a comma.

## Key Options

---

**Do Not Auto-Populate This Key**

Directs the wizards to skip this Key when creating primary Browse procedures or Report procedures.

**Population Order**

Specifies the order in which the wizards populate keys. Choose Normal, First, or Last from the drop-down list. Wizards populate in this order: all Keys specified as First, then all Keys specified as Normal, and finally all Keys specified as Last.

**User Options**

User Options let you provide information to utility templates. User Options are comma delimited, that is, each entry is separated by a comma.

## Relation Options

---

**User Options**

User Options let you provide information to utility templates. User Options are comma delimited, that is, each entry is separated by a comma.

## Naming Conventions

---

When creating procedures, the wizards extract information from your Data Dictionary and apply it to the generated procedures. Understanding how wizards use dictionary information can help you set up your dictionary to get the best results.

## **Naming Fields and Keys**

The wizards use the data dictionary field name for the window and report prompts and column headings for the fields. If you use mixed case names, such as `FirstName`, the wizards insert a space before the capital letters to create multi-word prompts and headings—in this case: `First Name`.

The Browse wizard uses the key description as tab text on multi-key browse procedures. If there is no description, the wizard uses the key name.

## **Field Descriptions**

By default, field descriptions are assigned to the field's `MSG` attribute in the dictionary. The wizards automatically apply this `MSG` attribute to each control in your application so that the description displays in the application's status bar. Providing field descriptions in the dictionary (once) eliminates the need to specify (potentially) several `MSG` attributes within your application.

## **Using Default Window Controls**

---

The Dictionary Editor creates a default control for each field, based on its data type. See the Window and Report tabs in the Field Properties dialog. The wizards use this default control when creating procedures.

For example, a `LONG` becomes an `ENTRY` control. In specific cases, you may want a different type of control. For example, in the case of a `LONG` customer number that is automatically incremented, you never want the user to modify it. In that case you can set the default window control to be a `STRING` control.

Another example is a field which has a finite list of choices. In this case, you can create a Drop List as the default window control and specify the valid choices in Validity Checks.



# 3 - PROCEDURE TEMPLATES

## Overview

This chapter describes the Clarion Procedure templates. It also mentions several Control templates, which are described in the *Control Templates* chapter.

## Procedures and Procedure Templates

---

A *procedure* is a series of Clarion language statements (source code) which perform a task. A *Procedure template* is an *interactive tool* that (with the help of Clarion's development environment) requests information from you, the developer, then generates a custom procedure for just the task you specify. A Procedure as stored in a Clarion application (.app) file, is really a specification that the development environment uses to generate the procedure source code. The specification includes the Procedure template and your answers to its prompts, the WINDOW definition, the REPORT definition, other local data declarations, embedded source code, etc.

Clarion provides a rich assortment of task oriented Procedure templates with which you can rapidly develop database applications. In *Getting Started*, the *Quick Start Tutorial* introduces a few procedure templates; the *Application Generator Tutorial* in *Learning Clarion* introduces more. This chapter describes all the procedure templates and their prompts.

### Using Procedure Templates

You use procedure templates by selecting the template based on the general task you want it to perform, such as browsing or searching data (Browse template), changing data (Form template) or reporting data (Report template). You select the template when you create the procedure (see *Application Generator* in the *User's Guide* for more information). Then you refine the template generated code to fit your specific task by using the **Procedure Properties** dialog to answer template prompts and to access other development environment tools such as the Window Formatter and the Report Formatter.

### Procedure Properties Dialog

The **Procedure Properties** dialog contains standard Application Generator command buttons and prompts, plus any additional prompts provided by the Procedure template. This chapter describes the template generated prompts. See *Application Generator* in the *User's Guide* for more information on the Application Generator command buttons and prompts.

## **Browse-Form Paradigm**

Clarion's Procedure templates follow the Browse-Form application paradigm. Clarion's Browse-Form paradigm uses Browsers (windows with sortable, scrollable, searchable, selectable lists of data), Forms (windows with a single updatable database record), and Reports to organize and present database information to end users. See *Template Overview—Browse-Form Application Paradigm* for more information.

## **Procedures as Containers**

---

Procedures can contain data structures such as WINDOW structures, REPORT structures, and the controls within those structures. And Procedure templates can contain other templates—Control and Extension templates that present additional opportunities to customize the procedure.

### **Procedures Contain Controls**

Procedure templates provide standard prompts for any BUTTON, ENTRY, or CHECK controls you add to the procedure's WINDOW. You access these prompts through the Properties dialog for these controls. For each ENTRY control, for example, the procedure template provides prompts to let you use the ENTRY as a lookup field. For a CHECK box, the procedure template provides prompts to let you update variables and hide or unhide controls based on the state of the CHECK box.

Procedure templates provide standard embed points for controls you add to the procedure's WINDOW. Generally, there is an embed point for each event the control generates. Embedding code into these embed points generates code that executes when the control generates the event. See *ABC Template Embed Points* for more information.

### **Procedures Contain Other Templates**

Finally, Procedure templates can contain other templates—Control templates and Extensions templates which provide their own development environment prompts and embed points, and their own runtime functionality.

Thus, a Procedure and its template act as a container which automatically provides support for many layers of functionality and customization. And the Application as stored in the development environment, acts as a container for the Procedures and their templates.

Many of the ABC Procedure templates already contain Control templates. Control templates generate code to define and manage a specific control, including loading data in and out of the control. In fact, the unique set of Control templates within a Procedure template are what determine the template's primary purpose or task. For example, the Browse Procedure

template is a generic Window Procedure template which contains the BrowseBox and BrowseUpdateButtons Control templates.

## Inter-Procedure Communication

---

Clarion's template generated procedures use a simple system of global variables and EQUATEs to communicate with each other.

The procedures use two global variables named **GlobalRequest** and **GlobalResponse**. The calling procedure uses **GlobalRequest** to tell the called procedure what database action to do. The called procedure uses the **GlobalResponse** variable to tell the calling procedure the result of the requested database action.

Whenever a template generated procedure calls another template generated procedure, the calling procedure sets the value of **GlobalRequest** to one of the EQUATEs declared in ABFILE.INC 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
ProcessRecord     EQUATE (5)      !Process the current record
```

The called procedure checks the **GlobalRequest** variable and tries to carry out the requested action. The called procedure indicates success or failure by setting the value of **GlobalResponse** to one of the EQUATEs declared in ABFILE.INC:

```
RequestCompleted EQUATE (1)      !Update Completed
RequestCancelled EQUATE (2)      !Update Aborted
```

## Window Procedure Templates

Most of the ABC Procedure templates (Frame, Browse, Form, etc.) generate procedures with WINDOWS. Even the Process and Report templates have a predefined progress window. **All of these window procedure templates described in this section are derived from the Window Template and inherit its prompts for controlling basic procedure behavior.** You can access these common window procedure prompts in the **Procedure Properties** dialog for each procedure template.

### Window Template

---

This template functions as a blank slate, upon which you can create your own window procedure of any kind. Most of the ABC Procedure templates are derived from this template and therefore inherit its prompts and behavior.

Press the **Window** button in the **Procedure Properties** dialog to select your window type. See the *User's Guide—The Window Formatter—Choosing a Window Type* for more information.

For the controls and Control templates you add to the window, the Window template adds embed points to handle the events they generate. After you place the controls, the **Embeds** button and the **Source** button let you attach custom source code to the events.

The only “predefined” elements of the template, which you can access through the **Procedure Properties** dialog, are local variables used to pass data to and from the calling procedure and to manage the window and procedure by keeping track of whether the window is open, and whether the procedure needs to respond to a global event.

The code generated by this template processes the WINDOW structure that you create with the Window Formatter. It generates code for handling all the field and window events.

**Tip:** To duplicate a window created for another application or procedure, without copying the entire procedure, copy the WINDOW declaration from the other source code document, then press the Window ellipsis (...) button and paste in the declaration. Caution: do NOT do this with windows that contain Control templates!

## Window Template Prompts



In addition to the standard Application Generator command buttons and prompts (see *Application Generator* in the *User's Guide*), the Window Procedure template **Procedure Properties** dialog contains the following prompts which are inherited by all the Window Procedure templates:

### Parameters

Specify the parameter list for your procedure. See *PROCEDURE* and *Procedure Prototypes* in the *Language Reference* and *Prototyping and Parameter Passing* in the *User's Guide* for more information.

The parameter list is an optional list of datatypes and labels that appear on the generated PROCEDURE statement. The entire list is enclosed in parentheses. There must be a parameter in the parameter list for each parameter defined in the procedure prototype. We recommend providing the data type *and* the parameter label in *both* the parameter list and in the procedure prototype. For example:

```
(SHORT Id,STRING Name)
```

You should handle the parameters in the procedure's embedded source code.

### Return Value

Specify the variable name whose value is returned by the procedure. You must first supply a return data type in the **Prototype** field. See *Prototyping and Parameter Passing* in the *User's Guide* for more information. You should assign the appropriate value to the return variable in the procedure's embedded source code.

## Window Behavior

The **Window Behavior** button provides access to a tabbed dialog where you can specify options for the procedure's WINDOW and its WindowManager.

### Window Operation Mode

Lets you override some window properties specified in the Window Formatter. This prompt provides a quick way change these attributes without using the Window Formatter. Choose from:

#### Use Window Setting

Use the attributes as set in the Window Formatter.

**Normal** Removes the MDI and MODAL attributes from the WINDOW.

**MDI** Adds the MDI attribute to the WINDOW.

**Modal** Adds the MODAL attribute to the WINDOW.

### Save and Restore Window Location

Check this box to make this procedure restore its window size and location from the previous session. You must first check the **Use INI file to save and restore program settings** box in the **Global Properties** dialog. See *Template Overview—General Tab Options*.

### Classes

The Classes tab lets you control the WindowManager class (and object) your procedure uses. You may accept the default Application Builder Class and its object (recommended), or you may specify your own or a third party class.

See *Template Overview—Classes Tab Options—Local* for complete information on these options.

## Browse Template

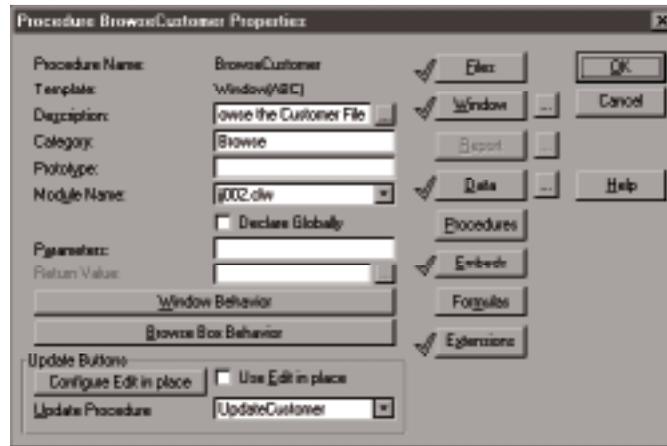
---

The Browse template is derived from the Window Template. It generates a procedure for browsing, scrolling, searching, and navigating through data. The data can come from one or more related files, and the Browse Procedure may update the data or it may call a separate procedure to update the data. The Browse template depends on the BrowseBox Control template for much of its functionality (see *Control Templates* for more information). The **File Schematic Definition** dialog automatically attaches your file choices to the BrowseBox Control template. The generated code implements the lookups of related records.

### Browse-Form Paradigm

The Browse template is an integral part of Clarion's Browse-Form paradigm which uses Browsers (windows with sortable, scrollable, searchable, selectable lists of data), Forms (windows with a single updatable database record), and Reports to organize and present database information to end users. See *Template Overview—Browse-Form Application Paradigm* for more information.

## Browse Template Prompts



In addition to the standard Application Generator command buttons and prompts (see *Application Generator* in the *User's Guide*), the Browse template **Procedure Properties** dialog contains the prompts inherited from the Window Template (**Parameters**, **Return Value**, and **Window Behavior**—see *Window Procedure Templates—Window Template Prompts*), plus the prompts provided by the BrowseBox Control and the BrowseUpdateButtons Control.

### Browse Box Behavior

This button provides access to a tabbed dialog where you can specify options for the BrowseBox Control template. From here you can control the BrowseBox's searching, scrolling, record selection, totalling, colors, icons and more. See *Control Templates—BrowseBox* for a complete description of these prompts.

### Update Buttons

The BrowseUpdateButtons Control template provides additional prompts that determine whether the Browse procedure updates records directly or calls a separate procedure. See *Control Templates—BrowseUpdateButtons* for a complete description of these prompts.

## Form Template

The Form template is derived from the Window Template. It generates code to display and update a single record from a file. It also generates code to display and access related records in other related files.

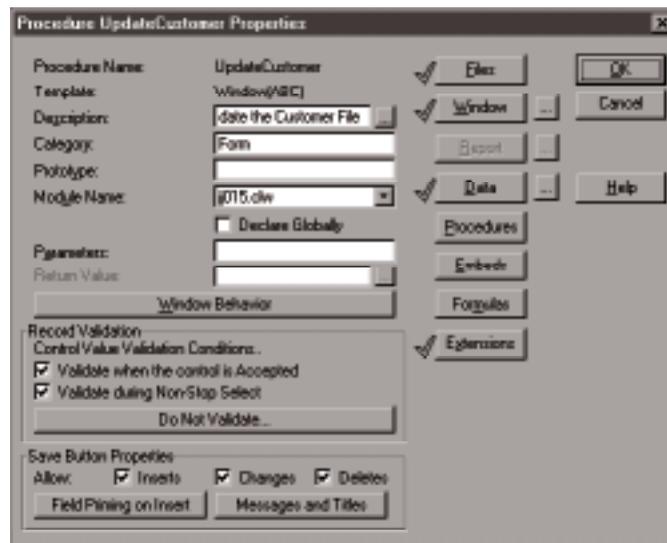
The Form template provides a predefined window, with a SaveButton Control template and a ValidateRecord Extension template. The SaveButton

Control template handles the file I/O and the ValidateRecord template validates incoming data according to data dictionary settings. The **File Schematic Definition** dialog automatically attaches your file choices to the SaveButton Control template. For accessing related records, the Form template optionally provides a BrowseBox Control template. See *Control Templates* and *Code and Extension Templates* for details on these template prompts and functionality.

## Browse-Form Paradigm

The Form template is an integral part of Clarion's Browse-Form paradigm which uses Browsers (windows with sortable, scrollable, searchable, selectable lists of data), Forms (windows with a single updatable database record), and Reports to organize and present database information to end users. See *Template Overview—Browse-Form Application Paradigm* for more information.

## Form Template Prompts



In addition to the standard Application Generator command buttons and prompts (see *Application Generator* in the *User's Guide*), the Form template **Procedure Properties** dialog contains the prompts inherited from the Window Template (**Parameters**, **Return Value**, and **Window Behavior**—see *Window Procedure Templates—Window Template Prompts*), plus the prompts provided by the ValidateRecord Extension and the SaveButton Control.

### Record Validation

The ValidateRecord Extension template adds additional prompts so you can control how and when record validation happens. See *Code and Extension Templates—RecordValidation* for a complete description of these prompts.

### Save Button Properties

The SaveButton Control template provides additional prompts so you can control how and when the record is updated, including the type of updates allowed, whether multiple inserts are allowed, messages shown to the end user, and more. See *Control Templates—SaveButton* for a complete description of these prompts.

## Frame Template

---

This template provides an MDI (Multiple Document Interface) parent frame, containing a predefined Windows standard menu with standard file, editing, window management, and help commands.

### Browse-Form Paradigm

The Frame template is an integral part of Clarion's Browse-Form paradigm which uses Browsers (windows with sortable, scrollable, searchable, selectable lists of data), Forms (windows with a single updatable database record), and Reports to organize and present database information to end users. See *Template Overview—Browse-Form Application Paradigm* for more information.

When creating an MDI application, the Frame should be the main supervisor procedure that controls all the other procedures in your application. You start new execution threads for each MDI child window which you want to appear inside the frame. The **Actions** tab for a control or Menu Item provides a check box to specify the start of a new execution thread (or you can use the InitiateThread Code template).

The predefined window contains a standard Windows menu with the following commands:

**File**—Print Setup, and Exit;

**Edit**—Cut, Copy, and Paste;

**Window**—Tile, Cascade, and Arrange Icons;

**Help**—Contents, How to Use Help, and Search for Help on.

Each of the predefined menu commands implement Standard Windows Behavior. Clarion's run-time libraries provide this standard behavior automatically. You don't have to code anything for these menu commands.

The Frame template includes the standard embed points, plus additional embeds for the menu commands. If you add a TOOLBAR, embed points are added for any TOOLBAR controls.

The FrameBrowseControl template adds standard database navigation and update buttons to the Frame's toolbar (see *Control Templates—FrameBrowseControl*).

## Frame Template Prompts



In addition to the standard Application Generator command buttons (see *Application Generator* in the *User's Guide*), the Frame Procedure template **Procedure Properties** dialog contains the prompts inherited from the Window Template (**Parameters**, **Return Value**, and **Window Behavior**—see *Window Procedure Templates—Window Template Prompts*), plus the following:

### Splash Procedure

Names a procedure to call after the application frame opens, but before any user events are generated. Select from the drop-down list, or type a new procedure name.

By convention, a splash procedure provides a visual or audio (or both) fanfare for your program. A splash screen can provide a recognizable logo or icon whose familiarity may raise the user's comfort level and may serve as an advertisement for your program. Additionally it diverts the user's attention from the sometimes boring task of loading and initializing the program.

See *Splash Template* for more information.

### Date and Time Display

This button provides access to a tabbed dialog where you can specify options for the DateTimeDisplay Extension template. The DateTimeDisplay template lets you display the time, date, or both in the window's status bar, or in a control. See *Code and Extension Templates—DateTimeDisplay* for a complete description of these prompts.

## Menu Template

---

This template provides an SDI (Single Document Interface) window. It is similar to the Frame Template in that it generates a menu that is the starting point for an (SDI) application. In addition to the standard Application Generator command buttons (see *Application Generator* in the *User's Guide*), the Menu Procedure template **Procedure Properties** dialog contains only the prompts inherited from the Window Template (see *Window Procedure Templates—Window Template Prompts*).

## Process Template

---

The Process Procedure template generates code to read through a data file and perform an operation on each record. You can specify a filter or range of records on which to perform the operation. A predefined window contains a progress indicator to show the end user what percentage of the operation is complete.

The PauseButton control template lets the end user suspend and resume process processing (see *Control Templates—PauseButton*).

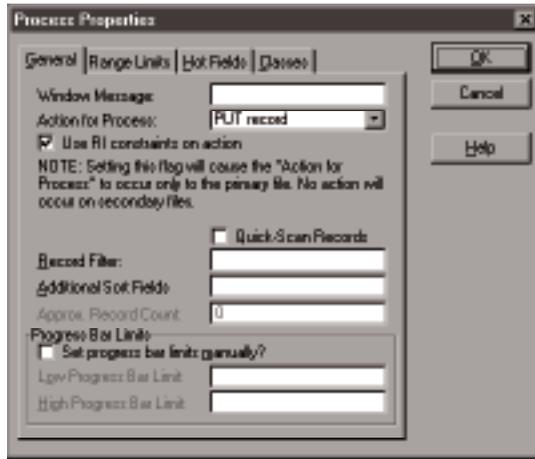
The ExtendProgressWindow template lets you suppress the progress window and lets the process run in two distinct modes: single record mode and all records mode (see *Other Templates—ExtendProgressWindow*).

### Process Template Prompts

In addition to the standard Application Generator buttons and prompts (see *Application Generator* in the *User's Guide*), the ProcessTemplate **Procedure Properties** dialog contains the prompts inherited from the Window Template (**Parameters**, **Return Value**, and **Window Behavior**—see *Window Procedure Templates—Window Template Prompts*), plus the following:

### Process Properties

This button provides access to a tabbed dialog where you can specify a wide variety of functionality for your process procedure. This section describes the **Process Properties** dialog.



**Tip:** By default, the Process procedure template does not create new records nor does it autonumber existing fields or records.

## General

### **Window Message**

Text to display on the progress window.

### **Action for Process**

This prompt lets you specify that the process operation changes (PUTs) or deletes the records that it processes. You can attach code to the **Activity for each Record** embed point to accomplish any custom processing you need.

### **Use RI constraints on action**

Check this box to enforce the RI constraints defined in your data dictionary. Clear this box to generate a simple PUT or DELETE depending on the **Action for Process** chosen.

### **Quick-Scan Records**

Specifies buffered access behavior for file systems that use multi-record buffers (primarily ASCII, BASIC, and DOS). See *Database Drivers* for more information. These file drivers read several records at a time. In a multi-user environment these buffers are not 100% trustworthy, because another user may change a record between accesses. As a safeguard, the driver refills the buffers before each record access.

Quick scanning is the normal way to read records for batch processing. However, rereading the buffer may provide slightly improved data integrity in some multi-user circumstances at the cost of substantially slower processing.

### **Record Filter**

Type an expression to limit the process to only those records

which match the filter expression. You must also specify an approximate record count (see *Approx Record Count*).

This filters all displayable records. When a Record Filter is used in conjunction with a Range Limit, the range limit is applied first. Because range limits use keys, they are much faster than filters.

**Tip:** You must BIND fields used in a filter expression. See Hot Fields below.

### **Additional Sort Fields**

Type a comma delimited list of fields on which to sort. These sort fields are in addition to the key for the process set in the **File Schematic Definition** dialog.

### **Approx Record Count**

When processing in record order (no key), this number is used to calculate what percentage of the operation is complete to provide feedback to the end user. If you don't specify a number, the process "counts" the records before processing begins. This can be relatively fast or slow depending on the file system and the file size. You must supply an approximate record count when you use a Record Filter (or a Range Limit that results in a filter).

### **Set progress bar limits manually?**

Clear this box to make your procedure read the result set and set the progress bar limits automatically. Setting limits automatically may produce poor performance for some SQL data sets, or erratic or inaccurate progress indicator for unevenly distributed result sets. Check this box to manually provide progress bar limits for the process. Setting manual limits can provide faster performance for SQL drivers and more accurate progress indicators for unevenly distributed result sets. This setting is only effective if you specify a Key for the File in the **File Schematic Definition** dialog.

### **Low Progress Bar Limit**

Supply the lowest "free" key element value for the result set. You may type the value or the label of a variable containing the value. Enclose literal string values in single quotes ('value').

### **High Progress Bar Limit**

Supply the highest "free" key element value for the result set. You may type the value or the label of a variable containing the value. Enclose literal string values in single quotes ('value').

## Range Limits

This tab is only available if you specify a Key for the File in the **File Schematic Definition** dialog. Because range limits use keys, they are generally much faster than filters.

### **Range Limit Field**

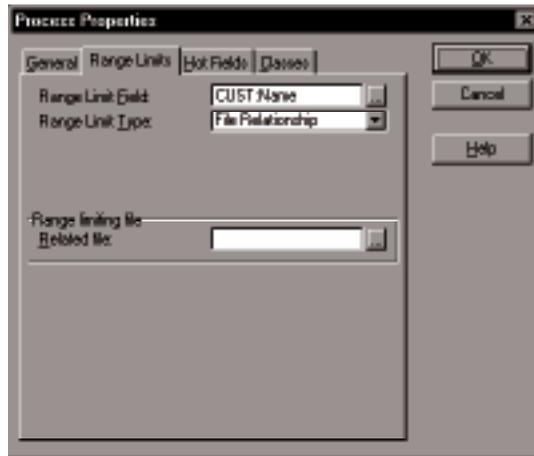
In conjunction with the **Range Limit Type**, specifies a record or group of records for inclusion in the process. Choose a key field on which to limit the records by pressing the ellipsis (...) button.

### **Range Limit Type**

Specifies the type of range limit to apply. Choose one of the following from the drop-down list.

*Current Value* Limits the key field to its current value.

*Single Value* Lets you limit the key field to a single value. Specify the variable containing that value in the **Range Limit Value** box.



*Range of Values* Lets you limit the key field to a range of values. Specify the variables containing the upper and lower limits of the range in the **Low Limit Value** and **High Limit Value** boxes.

### *File Relationship*

Lets you limit the key field to the current value in a related (parent) file. Press the **Related file** ellipsis (...) button to choose the range limiting file. This limits the process to include only those child records matching the current record in the parent file. For example, if your report was a list of Orders, you could limit the process to only those orders for the current Customer.

## Hot Fields

The Hot Fields tab lets you select additional fields to add to the VIEW. When scrolling through the file, the generated source code reads the data from a VIEW, rather than from the disk. This optimizes performance. Elements of the Primary Key and the current key are always included in the VIEW, so they do not need to be added to the Hot Field list. Any field used in a computation or filter must be in the VIEW.

In addition, you can BIND fields through this dialog. You must BIND any field used in a filter.

## Classes

The Classes tab lets you control the class (and object) your Process procedure uses. You may accept the default Application Builder Class and its object (recommended), or you may specify your own or a third party class. Deriving your own class can give you very fine control over the procedure when the standard Application Builder Class is not precisely what you need.

See *Template Overview—Classes Tab Options—Local* for complete information on these options.

## Report Template

---

The Report Procedure template generates code to read through a data file and update the controls in the report DETAIL for each record. You can specify a filter or range of records on which to perform the operation. The predefined window contains a progress indicator to show the end user what percentage of the operation is complete.

The PauseButton control template lets the end user suspend and resume report processing (see *Control Templates—PauseButton*).

The ExtendProgressWindow template lets you suppress the progress window and lets the report run in two distinct modes: single record mode and all records mode (see *Other Templates—ExtendProgressWindow*).

Press the **Report** button to define your REPORT. No report is predefined unless you used the Report Wizard to generate the procedure. See *REPORT* in the *Language Reference* for more information. Use the **String Properties** dialog to specify totals. See *Creating Reports* and *Controls and Their Properties* in the *User's Guide*.

**Tip:** You cannot automatically calculate intermediate group level totals with Clarion's Report Procedure templates and STRINGS. For example, you cannot add together two group level totals to create a third total. This type of calculation requires manual tracking of group breaks.

## Report Template Prompts

In addition to the standard Application Generator buttons and prompts (see *Application Generator* in the *User's Guide*), the Report Template **Procedure Properties** dialog contains the prompts inherited from the Window Template (**Parameters**, **Return Value**, and **Window Behavior**—see *Window Procedure Templates—Window Template Prompts*), plus the following:

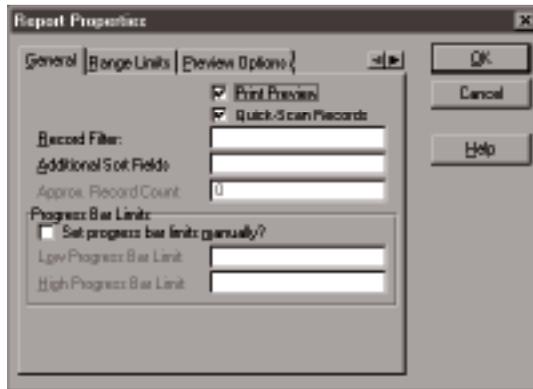
### Report Properties

This button provides access to a tabbed dialog where you can specify a wide variety of functionality for your report procedure. This section describes the **Report Properties** dialog.

#### General

##### **Print Preview**

Check this box to let the end user review the report on-screen before printing it. The end user can then print the report, or cancel it. Checking this box enables the **Preview Options** tab which lets you control the initial appearance of the report preview window.



**Tip:** The ReportManager contains the SkipPreview property that controls whether print preview is invoked. You can use SkipPreview to enable or disable the print preview at runtime: a value of one (1) enables print preview and a value of zero (0) disables it.

##### **Quick-Scan Records**

Specifies buffered access behavior for file systems that use multi-record buffers (primarily ASCII, BASIC, and DOS). See *Database Drivers* for more information. These file drivers read several records at a time. In a multi-user environment these buffers are not 100% trustworthy because another user may change a record between accesses. As a safeguard, the driver refills the buffers before each record access.

Quick scanning is the normal way to read records for batch processing. However, rereading the buffer may provide slightly improved data integrity in some multi-user circumstances at the cost of substantially slower processing.

### **Record Filter**

Type an expression to limit the report to only those records which match the filter expression. You must also specify an approximate record count (see *Approx Record Count*).

This filters all displayable records. When a Record Filter is used in conjunction with a Range Limit, the range limit is applied first. Because range limits use keys, they are much faster than filters.

**Tip:** You must BIND fields used in a filter expression. See Hot Fields below.

### **Additional Sort Fields**

Type a comma delimited list of fields on which to sort. These sort fields are in addition to the key for the report set in the **File Schematic Definition** dialog.

### **Approx Record Count**

When processing in record order (no key), this number is used to calculate what percentage of the operation is complete to provide feedback to the end user. If you don't specify a number, the process "counts" the records before processing begins. This can be relatively fast or slow depending on the file system and the file size. You must supply an approximate record count when you use a Record Filter (or a Range Limit that results in a filter).

### **Set progress bar limits manually?**

Clear this box to make your procedure read the result set and set the progress bar limits automatically. Setting limits automatically may produce poor performance for some SQL data sets, or erratic or inaccurate progress indicator for unevenly distributed result sets. Check this box to manually provide progress bar limits for the procedure. Setting manual limits can provide faster performance for SQL drivers and more accurate progress indicators for unevenly distributed result sets. This setting is only effective if you specify a Key for the File in the **File Schematic Definition** dialog.

### **Low Progress Bar Limit**

Supply the lowest "free" key element value for the result set. You may type the value or the label of a variable containing the value. Enclose literal string values in single quotes ('value').

### High Progress Bar Limit

Supply the highest “free” key element value for the result set. You may type the value or the label of a variable containing the value. Enclose literal string values in single quotes (‘value’).

### Range Limits

This tab is only available if you specify a Key for the File in the **File Schematic Definition** dialog. Because range limits use keys, they are generally much faster than filters.

#### Range Limit Field

In conjunction with the **Range Limit Type**, specifies a record or group of records for inclusion in the process. Choose a key field on which to limit the records by pressing the ellipsis (...) button.

#### Range Limit Type

Specifies the type of range limit to apply. Choose one of the following from the drop-down list.

*Current Value* Limits the key field to its current value.

*Single Value* Lets you limit the key field to a single value. Specify the variable containing that value in the **Range Limit Value** box.

*Range of Values* Lets you limit the key field to a range of values. Specify the variables containing the upper and lower limits of the range in the **Low Limit Value** and **High Limit Value** boxes.

#### *File Relationship*

Lets you limit the key field to the current value in a related (parent) file. Press the **Related file** ellipsis (...) button to choose the range limiting file. This limits the process to include only those child records matching the current record in the parent file. For example, if your report was a list of Orders, you could limit the process to only those orders for the current Customer.

### Preview Options

The Preview Options tab lets you control the initial appearance of the report preview window. This tab is only available if you check the **Print Preview** box on the General tab.

#### Initial Zoom Setting

Sets the initial magnification for the report to one of four discreet magnification choices. The end user may change the initial setting.

### Allow User Variable Zooms?

Check this box to let the end user set custom report magnifications in addition to the preset magnification choices.

### Set Initial Window Position

Check this box to enable the four following prompts to set the initial preview window position and size.

**X Position**      The initial horizontal position of the left edge of the window.

**Y Position**      The initial vertical position of the top edge of the window.

**Width**            The initial width of the window.

**Height**            The initial height of the window.

### Maximize Preview Window

Check this box to initially maximize the preview window. This supersedes the **Set Initial Window Position**, whose coordinates are applied only when the window is restored to its normal unmaximized state.

## Hot Fields

The Hot Fields tab lets you select additional fields to add to the procedure's VIEW. When scrolling through the file, the generated source code reads the data from a VIEW, rather than from the disk. This optimizes performance. Elements of the Primary Key and the current key are always included in the VIEW, so they do not need to be added to the Hot Field list. Any field used in a computation or filter must be in the VIEW.

In addition, you can BIND fields through this dialog. You must BIND any field used in a filter.

## Filters

The Filters tab lets you set an expression which determines whether to print the current item. At runtime, if the expression evaluates to true for the current item, the procedure prints the item. To be an effective print filter, the expression must refer to at least one of the fields in the procedure's VIEW.

This print filter is in addition to the **Record Filter** set on the General tab. The **Record Filter** determines which records are read and processed for the report; the **Detail Filters** determines which of the filtered records are actually printed.



### Properties

Select the **DETAIL** structure to filter, then press the **Properties** button to specify the filter expression. This opens the **Detail Filters** dialog which contains the following prompts.

**Filter** Type a valid Clarion expression. At runtime, if the expression evaluates to true, the procedure prints the **DETAIL** structure.

### Exclude unfiltered

Check this box to apply this filter to any other **DETAIL** structures in the report that do not have a print filter. This lets you set one filter for all the **DETAIL** structures in your report.

### Classes

The **Classes** tab lets you control the class (and object) your Process procedure uses. You may accept the default Application Builder Class and its object (recommended), or you may specify your own or a third party class. Deriving your own class can give you very fine control over the procedure when the standard Application Builder Class is not precisely what you need.

See *Template Overview—Classes Tab Options—Local* for complete information on these options.

## Splash Template

---

The **Splash Template** generates code to display a window with an image and some text. The window closes automatically after a specified amount of time. In addition, you can optionally allow the user to close the window at any time by **CLICKING** on it.

Frame procedures are designed to optionally call Splash procedures. See *Frame Template* for more information. Alternatively, you can call Splash procedures with embedded source code. See *Application Generator—Embedded Source Code*.

By convention, a splash procedure provides a visual or audio fanfare for your program. A splash screen can provide a recognizable logo or icon whose familiarity may raise the user's comfort level and may serve as an advertisement for your program. Additionally it diverts the user's attention from the sometimes boring task of loading and initializing the program.

### **Splash Template Prompts**

In addition to the Application Generator command buttons (see *Application Generator* in the *User's Guide*), the Splash Template **Procedure Properties** dialog contains the prompts inherited from the Window Template (**Parameters**, **Return Value**, and **Window Behavior**—see *Window Procedure Templates—Window Template Prompts*), plus the following:

#### **Display Time (in seconds)**

Specifies the maximum amount of time the splash window remains displayed. The window closes automatically after the time expires.

#### **Close when the user clicks on the splash window**

Checking this box lets the user close the window at any time by CLICKING on it.

## **Viewer Template**

---

The Viewer Template is derived from the Window Template. It provides a predefined window with a list box, an ASCII Search button, an ASCII Print button, and a Close button.

If you wish to use the template to always view the same ASCII file, you can use it as is. To allow viewing of any ASCII file selected from a standard file dialog, you'll need to add an entry box to accept the file name, plus the DOS File Lookup Control template to select the file to view.

## Viewer Template Prompts



In addition to the standard Application Generator command buttons and prompts (see *Application Generator* in the *User's Guide*), the Viewer Template **Procedure Properties** dialog contains the prompts inherited from the Window Template (**Parameters**, **Return Value**, and **Window Behavior**—see *Window Procedure Templates—Window Template Prompts*), plus the prompts provided by the AsciiViewControl.

## AsciiViewControl Properties

The AsciiViewControl Control template provides additional prompts so you can control which file to view, whether the end user can search or print the file, or both. See *Control Templates—AsciiViewControl* for a complete description of these prompts.

## Other Procedure Templates

### External Template

---

The External Procedure template declares a procedure contained in an external library or object file (\*.LIB only). The Application Generator writes no source code for this template, instead, the project system simply links in the named external file as a module. See *Development and Deployment Strategies* for more information.

The External Procedure template requires an associated external Lib or Obj module (see *Application Generator—Application Menu* in the *User's Guide*). If your application has no external modules, the External Procedure template opens the **Select Module Type** dialog so you can create one. Choose OBJ or LIB from the **Select Module Type** dialog; the other choices are not valid for the External Procedure template.

In the **Module Name** field, select the file name of the external library or object file from the drop-down list. Only those external modules already included in the project appear, so if your module does not appear, add the new module first. To add the module in the Application Generator, choose **Application ► Insert Module**. See *Application Generator—Application Menu* in the *User's Guide*).

Optionally type the external procedure's prototype in the **Prototype** field. See *Application Generator—Prototyping and Parameter Passing* in the *User's Guide*.

### Source Template

---

The Source Procedure template provides an elegant and simple way to add hand code to your application. It provides two points at which to embed your code: the data section, and the code section.

The template simply declares the procedure, handles any optional parameters, places the embedded data declarations in the data section, begins the CODE section, then places any embedded executable code in the CODE section:

```
... (local data)
CODE
... (your embedded code)
```

#### **Source Template Prompts**

In addition to the standard Application Generator command buttons and prompts (see *Application Generator* in the *User's Guide*), the Source

template **Procedure Properties** dialog contains the following additional prompts:

#### **Parameters**

Specify the parameter list for your procedure. See *PROCEDURE* and *Procedure Prototypes* in the *Language Reference* and *Prototyping and Parameter Passing* in the *User's Guide* for more information.

The parameter list is an optional list of datatypes and labels that appear on the generated PROCEDURE statement. The entire list is enclosed in parentheses. There must be a parameter in the parameter list for each parameter defined in the procedure prototype. We recommend providing the data type *and* the parameter label in *both* the parameter list and in the procedure prototype. For example:

```
(SHORT Id, STRING Name)
```

You should handle the parameters in the procedure's embedded source code.

# 4 - CONTROL TEMPLATES

## Overview

A control is almost anything you see on a window or a report. For example, a check box, a push button, an entry field, and a list box are all controls.



Control templates generate source code to declare controls *and* manage their associated data. For example, the `BrowseBox` Control template not only generates source code to declare a list box, it also generates code to load data into a `QUEUE`, then display the `QUEUE` in the list box with complete scrolling, searching, sorting, updating, and mouse-click selection capability.

Control templates can also control file I/O; for example, the `SaveButton` Control template can warn that changes were made if the end user tries to close the window without saving the changes to disk.

**Tip:** Generally, it is to your advantage to use a Control template rather than a simple control.

This chapter describes all the Control templates included with Clarion and provides a guide to filling out their prompts.

## Adding Control Templates

---

When starting with a new procedure, to add a Control template:

1. In the Window Formatter or Report Formatter, add a Control template by clicking on the  tool in the Controls tool box.
2. Choose a Control template from the **Select Control template** dialog, then place the control on the window or report by clicking on the desired location.

The formatter places one or more controls (the type of controls depend on the Control template) in the window or report.

3. RIGHT-CLICK on the control, then choose **Actions** from the popup menu to access the Control template prompts.

These prompts define and customize its functionality.

4. Select the other tabs on the **Properties** dialog to set the control's appearance, location, and other functionality.

Once a Control template is added to a procedure, a check box appears next to the **Extensions** button in the **Procedure Properties** dialog. You can access the Control template prompt with this button.

## Read-Only Browse Templates

The read-only file browsing templates include the AsciiViewControl template and its associated print button and search button templates. This section describes these related templates.

The AsciiViewInList Extension template provides the same functionality for an independent LIST control (a LIST not placed by the Extension template). See *Code and Extension Templates—AsciiViewInList* for more information.

### ASCIIVIEWCONTROL

---

The AsciiViewControl template adds a LIST control in which you can display read-only, the contents of a file—including variable length files. It is typically used to display an ASCII text file. The AsciiViewControl template optionally provides search and print capability for the displayed file.

The template lets you select the file to view at design time, or leaves the selection to the end user at runtime if you prefer. Finally, the template optionally allows the LIST control to alternate its display between the selected file and some other data that you specify.

The AsciiViewControl template provides embed points for its LIST control. It also provides the following prompts on the **List Properties** dialog **Actions** tab, the **Procedure Properties** dialog, or the **Extension and Control Templates** dialog:

#### General Options

##### **Initialize Viewer**

Determines when the procedure initializes the Viewer object. Initialization includes selecting the file to view, opening it, and reading it.

##### **On Open Window**

Initializes the Viewer when the window opens so that the Viewer's LIST is full upon initial display.

##### **On Field Selection**

Delays initializing the Viewer until the end user selects the Viewer's LIST control.

##### **Manually**

Does not initialize the Viewer. You must embed a call to the Viewer#.Initialize ROUTINE to initialize the Viewer.

##### **File to Browse**

Specifies the path and name of the file to view, or a variable containing the path and name of the file to view. The variable must be preceded by an exclamation point (!).

If no path is specified, the procedure looks for the file in the current directory.

If omitted (left blank), the Viewer object prompts the end user to select a file.

**Reassign FROM attribute after Kill**

Check this box to reset the Viewer LIST's FROM attribute after the Viewer shuts down. See *FROM* in the *Language Reference*. This lets you use a single LIST control to display both the **File to Browse** and other items as well.

**Value or queue to assign**

Type the label of the QUEUE (or the string constant) to assign to the Viewer LIST's FROM attribute.

**Allow popup menu searching**

Check this box to provide a (RIGHT-CLICK) popup menu choice to search the file.

**Allow popup menu printing**

Check this box to provide a (RIGHT-CLICK) popup menu choice to print some or all of the records in the file.

## Classes Options

See *Procedure Templates—Process Template* for a complete description of these prompts.

## ASCIIPrintButton

---

The ASCIIPrintButton template adds a “Print” button and the underlying code to print some or all records from the associated AsciiViewControl template's file. The ASCIIPrintButton is only available for use with an existing (populated) AsciiViewControl template.

The ASCIIPrintButton template provides no additional prompts. It does add embed points for the BUTTON.

## ASCIISearchButton

---

The ASCIISearchButton template adds a “Search” button and the underlying code to search the associated AsciiViewControl template's file. The ASCIISearchButton is only available for use with an existing (populated) AsciiViewControl template.

The ASCIISearchButton template provides no additional prompts. It does add embed points for the BUTTON.

## Read-Write Browse Templates

The read-write file browsing templates include the BrowseBox template, the RelationTree template and their associated templates (BrowsePrintButton, ReltreeUpdateButtons, etc.). This section describes these templates.

### BrowseBox Overview

---

The BrowseBox Control template places a “page-loaded” or a “file-loaded” LIST control in a window and generates code to fill the list with data, and to scroll, search, sort, and select the listed items. It generates code to select or filter the data, total the data, update the data directly (edit-in-place), or call a separate procedure to update the data. It also generates code to conditionally set the colors and icons associated with each row and column in the LIST. The standard BrowseBox behavior is defined by the ABC Library’s BrowseClass. See *BrowseClass* for more information.

The BrowseBox template is highly configurable and so is its LIST control. That is, each function, from scrolling to colorizing, is customizable with the BrowseBox template prompts described below. The appearance of the BrowseBox’s LIST is fully customizable with the List Box Formatter (see *List Box Formatter* in the *User’s Guide*).

#### Page-loading versus File-loading

The page-loaded BrowseBox LIST loads very quickly, because only a few records, rather than the entire file, are loaded into memory from disk. The primary advantage of page-loading is speed and resource (RAM) savings when browsing large files. The disadvantage is that vertical scroll bars don’t work quite as smoothly as for file-loaded LISTS.

**Tip:** You can use the BrowseBox Control template to manage a *drop-list* by setting the DROP attribute to a value greater than zero (0).

#### Placing a BrowseBox on your window

You can place the *BrowseBox* Control template in a window by clicking on the control template tool , then selecting **BrowseBox - File Browsing List Box** in the **Select Control template** dialog. After you select the BrowseBox template, the Application Generator automatically opens the **List Box Formatter** so you can choose the files, fields and variables to display in the list, and you can set the appearance of the list and its fields.

## Populating and Formatting the List Fields

The **Populate** button lets you add a field or variable to the list box, one field or variable at a time from the **Select Field** dialog. The **Select Field** dialog presents the file schematic. Within the schematic, the BrowseBox control appears, with a <To Do> beneath it. To add a field from a data file defined in the dictionary:

- Select the <To Do> item.
- Press the **Insert** button
- Select the file from the **Insert File** dialog.
- If you want to use a key, press the **Key** button to select the key from the **Key Access** dialog. If you do not select a key, the list is displayed in record order, which also disables the ability to set Range Limits.
- Select a field from the **Fields** list, which appears in the right side of the **Select Field** dialog. After you select the file, key and field (or variable) the formatter opens the **List Field Properties** dialog where you can define the field's appearance within the list.

See *Application Generator—Procedure Files* in the *User's Guide* for more information on the File Schematic. See *List Box Formatter* in the *User's Guide* for more information on formatting the LIST.

When you are finished with the **List Box Formatter**, CLICK in the window to place the BrowseBox's LIST control.

### List Properties

RIGHT-CLICK on the LIST control and choose **Properties** from the popup menu to view the **List Properties** dialog. See *Controls and Their Properties—List* in the *User's Guide* for complete information about the LIST options available in this dialog.

## Scrolling with a Page-loaded BrowseBox

---

Scrolling through a page-loaded, optionally filtered database is very different than scrolling through more typical Windows files such as word-processing documents, spread-sheets, or File Manager/Explorer type directory lists. Primarily, the differences are the large size of the database (requiring page-loading) and the potential variations in size and data distribution due to filtering. These differences can result in vertical scroll bar behavior that is somewhat different than you and your end users might expect.

## Size Considerations

Even with completely accurate calibration, vertical scroll bars can produce less than optimum results on large datasets. For example, for a result set of one hundred thousand items, a perfectly calibrated scroll bar thumb can only provide very gross positioning. This is because the smallest distance the thumb can move (a pixel), represents between 1/200th (typical 640x480 scroll bar) to 1/400th (long 640x480 scroll bar) of the total scroll bar length, and therefore 1/200th (500 items) to 1/400th (250 items) of the records in the result set. For a million record result set, the numbers jump to a whopping 2,500 to 5,000 items per pixel, making the vertical scroll bar a poor choice for navigation.

For large data sets, locators in combination with VCR buttons (no sliding thumb) provide better functionality and happier end users. See *Locator Behavior* below.

## Calibration Considerations

With the performance advantage that comes with page-loaded lists, comes the disadvantage of not having the entire list in memory, and therefore not knowing the total number of records in the result set, nor the relative position of a given record within the result set. To produce “standard” vertical scroll bar behavior for a page-loaded list box, the scrolling procedure must know three things: the number of records in the result set, the relative position of a record in the result set, and the record that resides at a relative position in the result set. Since database engines do not provide this information for keyed or filtered result sets (it would be too slow), the page-loaded scrolling procedure must *estimate* these values when the end user drags the scroll bar thumb or selects or locates an item in the result set.

The BrowseBox template offers a versatile set of options for estimating these values and for calibrating the vertical scroll bar to best fit your database. See *Scroll Bar Behavior* for more information.

## BrowseBox Options

---

The BrowseBox template provides the following prompts (on the **List Properties** dialog **Actions** tab, the **Browse Procedure Properties** dialog, or the **Extension and Control Templates** dialog) as well as embed points for the LIST:

### Default Behavior

This tab contains the prompts that control the default behavior of the BrowseBox.

#### **Loading Method**

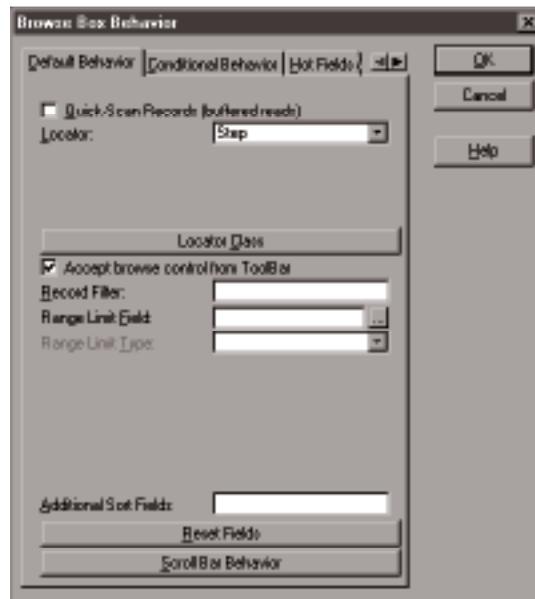
Select the method used to read the BrowseBox data.

- Page** Page-loading provides near-instantaneous displays for unfiltered data, even for very large datasets. Page-loading uses less memory than file-loading, because only a few records are held in memory at a time. We recommend page-loading for larger datasets.
- File** File-loading provides smooth, accurate vertical scroll bar behavior, plus no additional network traffic when scrolling and searching. File-loading is also quite SQL friendly, avoiding problematic backward scrolling. We recommend file-loading for smaller datasets.

#### Quick-Scan Records (buffered reads)

Specifies buffered access behavior for file systems that use multi-record buffers (primarily ASCII, BASIC, and DOS). See *Part III - Database Drivers* for more information. These file drivers read a buffer at a time, allowing for fast access. In a multi-user environment these buffers are not 100% trustworthy, because another user may change a record between accesses. Without quick-scan, the driver refills the buffers before each record access as a safeguard.

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 processing.



Accept browse control from Toolbar

Check this box to accept navigation events and other browse control events generated by the *FrameBrowseControl* control template on the APPLICATION's toolbar. See *FrameBrowseControl* for more information on these toolbar buttons and their operation. Clear this box to disable the *FrameBrowseControl* toolbar buttons for this procedure and use local navigation controls only. See also *SetToolbarTarget*.

## **Locator Behavior**

### **Locator**

A locator lets the user search for specific records in the list box without manually scrolling through the entire list. **Locator** is only available when browsing a file in Key Order (specify a KEY in the File Schematic). The search field must be the first free key element, that is, the first component field of the browse key that is not range limited to a single value. The standard Locator behavior is defined by the ABC Library's *LocatorClass*. See *LocatorClass* for more information.

For multi-key browses (the Wizards create them), you may have multiple locators. Use the **Conditional Behavior** tab to set additional locators for the additional sorts. Choose from the following locator types in the drop-down list:

#### **None**

Specifies no locator.

#### **Step**

Specifies a single-character locator with no locator control required. When the *BrowseBox* has focus and the user types a character, the list box advances to the first occurrence of the key field beginning with that character (or the next higher character if no keys match the locator character). Retyping the same character advances the list to the next occurrence of the key field beginning with that character.

Use a step locator when the first free key element is a *STRING*, *CSTRING*, or *PSTRING* and you want the search to take place immediately upon the user's keystroke. Step locators are not appropriate for numeric keys. If there is no browse key, the Application Generator converts to no locator.

See *StepLocatorClass* for more information.

#### **Entry**

Specifies a multi-character locator that activates when the locator control is *accepted* (not upon each keystroke). The locator control may be an *ENTRY*, *COMBO*, or *SPIN*. Use an Entry locator when you want to search on numeric or alphanumeric keys, and delay the search until the user accepts the locator control (presses *ENTER* or *TAB*). This delayed search reduces network traffic and provides a smoother search in a client-server environment.

The locator control should come *after* the LIST control in the **Set Control Order** dialog.

By default, the locator control is the control whose USE attribute is the first free key element of the browse key. A free component is one that is *not* range limited to a single value. If there is no such control, the Application Generator converts to a Step locator. If there is no browse key, the Application Generator converts to no locator.

When the end user places one or more characters in the locator control, then *accepts* the control by pressing TAB, pressing a locator button (  see *FrameBrowseControl*, or  see *List Properties*), or selecting another control on the screen, the list box advances to the nearest matching record.

See *EntryLocatorClass* for more information.

### Incremental

Specifies a multi-character locator, with no locator control required (but strongly recommended). Use an Incremental locator when you want to search on numeric or alphanumeric keys and you want the search to take place immediately upon the user's keystroke.

The locator control should come *after* the LIST control in the **Set Control Order** dialog.

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, when the list has focus, characters are automatically placed in the locator string for each keystroke, and the list box *immediately* advances to the nearest matching record. The backspace key removes characters from the locator string.

We strongly recommend using a STRING control as the Incremental locator control so the search occurs *immediately* with each keystroke, and so the user can *see* the key value for which the BrowseBox is searching.

By default, the locator control is the control whose USE attribute is the first free key element of the browse key. A free component is one that is *not* range limited to a single value. If there is no such control, the Application Generator converts to a Step locator. If there is no browse key, the Application Generator converts to no locator.

See *IncrementalLocatorClass* for more information.

### Filtered

Specifies a multi-character locator, with no locator control required (but strongly recommended). Use a Filter Locator when

you want to search on alphanumeric keys and you want to *minimize network traffic*.

This locator is like an Incremental Locator with a record filter. It specifies a *range* of values for which to search and returns a *limited* result set—only those records that fall within the specified range. 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’.

The Filtered Locator determines the boundaries for the search based on the user specified search value. The implementation of the boundaries depends on the database driver—for SQL databases, the Filtered Locator uses a LIKE; for ISAM databases it supplies upper and lower bounds.

The locator returns *only* the records that match the search value, providing, in effect, a dynamic range limit or filter for the browse.

**Tip:** The Filtered 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.

See *FilterLocatorClass* for more information.

#### Override default locator control

The *default* locator control is the control whose USE attribute is the first free (unlimited) element of the browse key. To override this default and specify a different locator control, check this box. This option is provided in case you have multiple controls with the same free element as their USE attributes—that is, when you have both ascending and descending keys on the same field.

Select one of the controls to use as the locator control from the **New Locator Control** list.

#### Find Anywhere

For Filtered Locators only, check this box to apply the search value to the entire field (field *contains* search value). Clear the box to apply the search value only to the leftmost field positions (field *begins with* search value). For example a search for “ba” returns different results based on the **Find Anywhere** box:

cleared	checked
Bain	Bain
Barber	Barber
Bayert	Bayert
	Dunbar
	Suba

See *FilterLocatorClass.FloatRight* for more information.

### Locator Class

Press this button to override the global Locator Manager setting. See *Template Overview—Classes Tab Options—Global and Local*.

### Record Filter

Type a valid Clarion expression to limit the contents of the browse list to only those records causing the expression to evaluate to true (nonzero or non-blank). The procedure loops through all displayable records to select only those that meet the filter. Filters are generally much slower than Range Limits.

You must BIND any file field that is used in a filter expression. The **Hot Fields** tab lets you BIND fields. The standard filter behavior is defined by the ABC Library's ViewManager. See *ViewManager* for more information.

### Range Limit Field

In conjunction with the **Range Limit Type**, specifies a record or group of records for inclusion in the list. Choose a field by pressing the ellipsis (...) button. The range limit is key-dependent. Range Limits are generally much faster than filters. The standard range limit behavior is defined by the ABC Library's ViewManager. See *ViewManager.AddRange* for more information.

### Range Limit Type

Specifies the type of range limit to apply. Choose one of the following from the drop-down list.

*Current Value* Limits the key to the current value of the **Range Limit Field**.

*Single Value* Lets you limit the key to a single value. Specify the variable containing that value in the **Range Limit Value** box which appears.

*Range of Values* Lets you specify upper and lower limits. Specify the variables containing the limits in the **Low Limit** and **High Limit** boxes.

### *File Relationship*

Lets you choose a range limiting file from a 1:MANY relationship. This limits the list to display only those child records matching the current record in the parent file. For example, if your list was a list of Orders, you could limit the display to only those orders for the current Customer (in the Customer file).

### Additional Sort Fields

Specify fields to sort on in *addition* to any key specified in the **File Schematic** by typing an ORDER expression list (a comma delimited list of field names). See ORDER in the *Language Reference* for more information. If no key is specified, the list is only ordered by the additional sort fields. See *ViewManager.AppendOrder* for more information.

### Reset Fields

Press this button to add reset fields. If the value of any reset field changes, the procedure resets the BrowseBox list (reapplies sort order, filter, etc.). You don't need a reset field for range limit fields or locators. Use a reset field to reset the BrowseBox based on controls or data that are not directly related to the BrowseBox. See *BrowseClass.AddResetField*, *WindowManager.Reset*, and *WindowManager.AutoRefresh* for more information.

### Scroll Bar Behavior

Pressing this button displays a dialog where you can define vertical scroll bar behavior for page-loaded BrowseBoxes.

**Tip:** For file-loaded lists, you automatically get Standard Windows Behavior (movable thumb) for the scroll bar. However, since this is not possible for page-loaded lists, these options let you choose the behavior that best suits your application. See *Scrolling with a Page-loaded BrowseBox* for more information.

### Scroll Bar Type

Choose from **Fixed Thumb** or **Movable Thumb**.

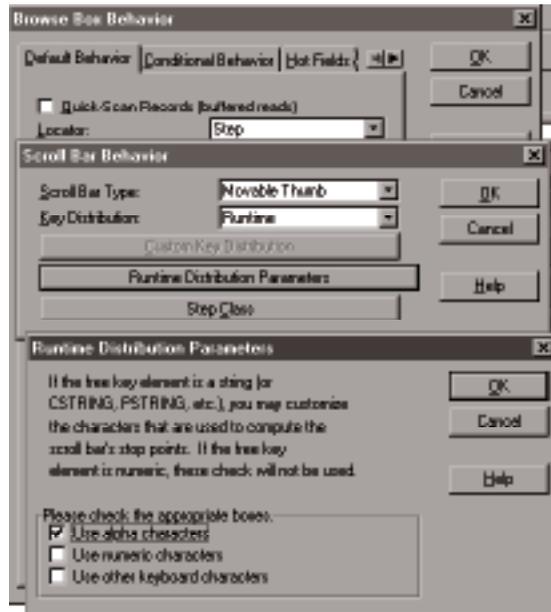
*Fixed Thumb* The thumb (square 3D box in the middle of the scroll bar) remains in the center of the scroll bar. CLICK above the thumb to scroll up one "page." CLICK below the thumb to scroll down one "page." DRAG the thumb to the top or bottom of the scroll bar to scroll the top or bottom of the file.

**Tip:** Choose Fixed Thumb when browsing large SQL tables to get best performance.

*Movable Thumb* CLICK and DRAG the thumb to scroll a proportional distance in the list. The thumb remains where you drag it, and its position on the scroll bar indicates the relative (estimated) position within the browse list.

CLICK above the thumb to scroll up one "page." CLICK below the thumb to scroll down one "page".

When you choose *Movable Thumb*, you can also set the **Key Distribution** to further define how the BrowseBox estimates the thumb's relative position within the browse list.



**Key Distribution** Specifies the distribution of the 100 scroll bar segments. Choose one of the two predefined distributions (Alpha or Last Names), or Custom, or Run-time from the drop-down list.

- Alpha* Defines 100 evenly distributed points based on the English alphabet. If the access key is numeric, you should use a custom or run-time distribution.
- Last Names* Defines 100 points distributed as last names are commonly found in the United States. If the access key is numeric, you should use a custom or run-time distribution.
- Custom* Lets you define your own points.
- Run-time* Reads the first and last record and computes the values for 100 evenly distributed points in between.

#### **Custom Key Distribution**

Sets the break points along the scroll bar (useful when you have data with a skewed distribution). Insert the values for each point in the list. String constants should be in single quotes ( ' ' ).

### Run-time Distribution Parameters

Specify the characters considered when determining the distribution points. This is only appropriate when the free element is a STRING or CSTRING. Choose from **Use alpha characters** (Aa-Zz), **Use numeric characters** (0-9), and **Use other keyboard characters**.

**Step Class** Press this button to override the global Step Manager setting. See *Template Overview—Classes Tab Options—Global* and *Local*.

### Conditional Behavior

This tab contains a list box that lets you define BrowseBox behavior based on conditions or expressions. Add expressions to the list by pressing the **Insert** button. This displays a dialog where you define the expression and the associated behavior when that expression evaluates to true (nonzero or non-blank).

At run-time the expressions are evaluated, and the behavior for the first true condition in the list is used.

In this dialog you can specify:

<b>Condition</b>	Any valid Clarion expression.
<b>Key to Use</b>	Optionally, the key to use to sort the BrowseBox data when the expression is true.

The remaining fields and buttons are the same as the **Default Behavior** tab.

### Hot Fields

When you select the Hot Fields tab, you can specify fields not populated in the list to add to the QUEUE. When scrolling through the file, the generated source code reads the data for these fields from the QUEUE, rather than from the disk. This speeds up list box updates.



Specifying “Hot” fields also lets you place controls outside the BrowseBox that are updated whenever a different record is selected in the list box. Elements of the primary key and the current key are always included in the QUEUE, so they do not need to be inserted in the Hot Field list.

**BIND Field**

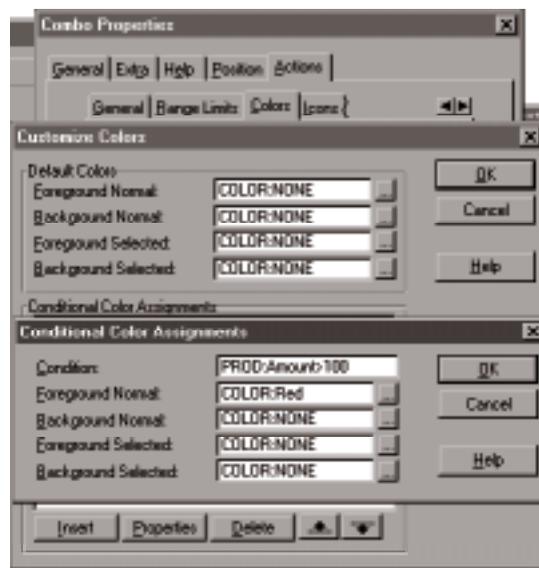
Check this box to BIND the field. You must BIND any field that is used in a filter expression or as a field to total. See *BIND* in the *Language Reference*.

**Not in view**

Check this box to tell the template the selected field is not part of the BrowseBox VIEW—rather it is a global or local variable—and therefore the BrowseBox generated code should not attempt to clear it or otherwise manipulate it.

**Colors**

This tab is only available if you check the **Color** box in the List Box Formatter. It displays a list of the BrowseBox columns which may be colored on a row-by-row basis.



To specify the default colors and any conditional colors, highlight the column's field name, then press the **Properties** button. This opens the **Customize Colors** dialog.

**Customize Colors**

This dialog lets you specify the default and conditional Foreground and Background colors for normal (unselected) rows; and for selected rows.

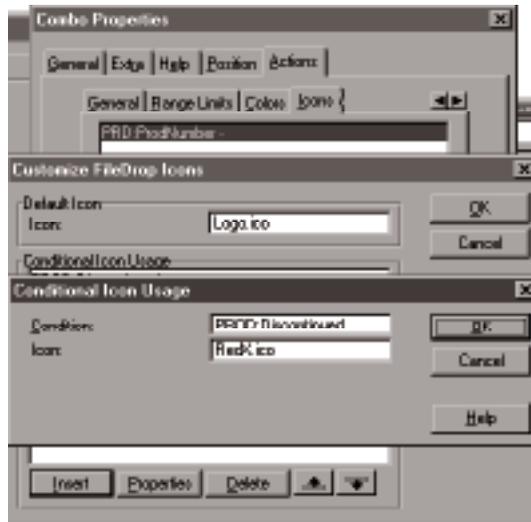
**Conditional Color Assignments**

Below the default colors section is the **Conditional Color Assignments** list. This list lets you set colors to apply when an expression evaluates to true (nonzero or non-blank). To add an expression and its associated colors, press the **Insert** button.

At run-time the expressions are evaluated, and the colors for the first true expression are used.

## Icons

This tab is only available if you check the **Icons** box in the List Box Formatter. It displays a list of the BrowseBox columns which can display icons.



To specify default icons and any conditional icons, highlight the column's field name then press the **Properties** button. This opens the **Customize BrowseBox Icons** dialog.

### **Customize BrowseBox Icons**

This dialog lets you specify the default icon and conditional icons for the BrowseBox column.

#### **Default Icon**

The default icon to display. Type the icon (.ICO) filename.

#### **Conditional Icon Usage**

Below the **Default Icon** section is the **Conditional Icon Usage** list. This list lets you set icons to apply when an expression evaluates to true (nonzero or non-blank). To add an expression and its associated icon, press the **Insert** button.

At run-time the expressions are evaluated, and the colors for the first true expression are used.

## Totaling

This tab contains a list box that lets you define total fields for a BrowseBox. Press the **Insert** button to add total fields. This opens the **Browse Totaling** dialog where you can define total fields for the BrowseBox.

### **Total Target Field**

The variable to store the calculated total. This can be a local, module, or global variable. You may also use a file field; however, you must write the code to update the file.

### **Total Type**

Choose **Count**, **Sum**, or **Average** from the drop-down list. **Count** tallies the number of records. **Sum** adds the values of the Field to Total. **Average** determines the arithmetic mean of the Field to Total.

### **Field to Total**

The field to sum or average. This box is disabled when the Total Type is **Count**.

### **Total Based On**

Choose **Each Record Read** or **Specified Condition** from the drop-down list. This specifies whether to consider every record or only those that meet the Total Condition criteria.

### **Total Condition**

The condition to meet when using a Total based on a specified condition. You can use any valid Clarion expression. You must **BIND** any fieldnames used in this expression. Use the Hot Fields tab to **BIND** fieldnames.

## Classes

The Classes tab lets you control the class (and object) the procedure uses. You may accept the default Application Builder Class and its object (recommended), or you may specify your own or a third party class. Deriving your own class can give you very fine control over the procedure when the standard Application Builder Class is not precisely what you need.

See *Template Overview—Classes Tab Options—Local* for complete information on these options.

## **BrowsePrintButton**

---

The BrowsePrintButton template provides a **Print** button to call a procedure with the ProcessRecord request (see *Procedure Templates—Inter-Procedure Communication* for more information).

## Print All Items

If you use the `BrowsePrintButton` to call a simple Report procedure, the report prints as usual, applying any design-time keys, sort orders, range-limits, and filters.

## Print the Selected Item

If you use the `BrowsePrintButton` to call a Report procedure with the `ExtendProgressWindow` extension template (set to Single record), the report reacts to the `ProcessRecord` request and processes only the selected `BrowseBox` item. See *Other Templates—ExtendProgressWindow* for more information.

**Note:** This option works by using a current-value limit on the report key. Therefore, if you have a non-unique key you can print multiple items—for example, all customers named Smith.

The `BrowsePrintButton` template provides the following prompts:

### **Procedure name**

Type the name of a procedure to call or select a procedure from the drop-down list.

## **BrowsePublishButton**

---

The `BrowsePublishButton` template provides a **Publish** button to generate the Hypertext Markup Language (HTML) to display records from a `BrowseBox` queue. In other words, use this template to publish your `BrowseBox` information to an Internet Web page!

**Note:** The `BrowsePublishButton` template is only available in the Clarion template chain and cannot be used with the ABC Templates.

The resulting Web page displays a headline that you specify, plus the headers from your list box. The Web page formats the list box data using the picture tokens specified in the list box.

At runtime, the user may specify the filename for the generated HTML. Also, the user has the option to publish all the items in the `BrowseBox` queue or just the items currently displayed on the screen.

The `BrowsePublishButton` template provides the following prompts:

### **Use variable for HTML name**

Check this box to specify the HTML file with a variable. This enables the **Variable HTML filename** field to name the variable, and disables the **Default HTML Name** field.

**Default HTML Name**

Specifies the default filename for the HTML code or the variable that contains the HTML filename. Press the ellipsis (...) button to select the file from the standard **Open File** dialog, or to select or define a data dictionary field or memory variable from the **Select Field** dialog.

If you don't specify a full path, your procedure writes the file to the current directory. At runtime, the user may specify a different file and path name.

**HTML Title**

Specifies the title for your HTML document. The title appears in the Web browser's caption when it displays the document.

**Table Heading**

Specifies headline that displays at the top of the Web page.

**Background Graphic**

Specifies a graphic image that displays "behind" the queue items. Note, most Web Browsers support graphic images, however, some older versions do not.

**Use Grid Lines**

Check this box to display the queue items within a rectangular grid. Note, most Web Browsers support grid lines, however, some older versions do not.

**Grid Line Width**

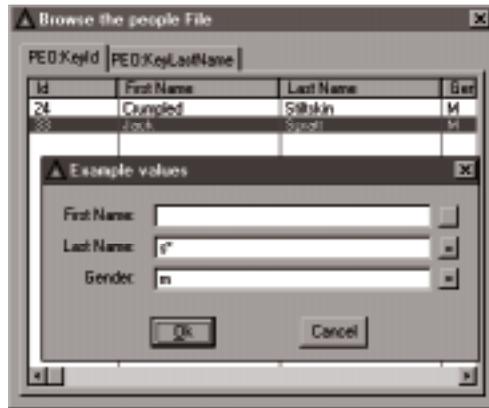
Specifies the thickness of the border defining the grid.

## BrowseQueryButton

---

The BrowseQueryButton template provides a **Query** button to let the end user apply a dynamic (run-time) filter to the BrowseBox result set. In other words, the end user can query the underlying dataset and display the results of the query in the BrowseBox list.

The default query interface is a dialog with an input field and a comparison operator button for each list box column.



The end user may provide filter criteria for zero or more fields. Additional filter criteria result in a more refined search and a smaller result set (the filter conditions are conjunctive—ANDed together).

### Runtime Options

The default comparison operator is ( = ), which searches for an exact match between the BrowseBox field and the corresponding Query input field. By default all matches are case sensitive. Pressing the comparison operator button cycles through all the available operators:

<b>Operator</b>	<b>Filter Effect</b>		
=	<i>browsefield</i>	equal	<i>queryvalue</i>
>=	<i>browsefield</i>	greater than or equal	<i>queryvalue</i>
<=	<i>browsefield</i>	less than or equal	<i>queryvalue</i>
<>	<i>browsefield</i>	not equal	<i>queryvalue</i>
	<i>no filter</i>		

For string fields, you may use the following special characters in the Query input field to refine your search:

<b>Symbol</b>	<b>Position</b>	<b>Filter Effect</b>		
^	prefix	caseless (case insensitive) search		
*	prefix	<i>browsefield</i>	contains	<i>queryvalue</i>
*	suffix	<i>browsefield</i>	begins with	<i>queryvalue</i>

For example:

- d - matches 'd' only
- d\* - matches 'dog', 'david'
- \*d - matches 'dog', 'cod'
- ^\*d - matches 'dog', 'cod', 'coD'

Upon completion of the Query dialog, the current sort order of the BrowseBox is filtered to match the query. If Query is selected again, the

previous query is available by default. This allows sharing of filters between sort orders, as well as successive filter refinements.

The standard Query behavior is defined by the ABC Library's QueryClass. See *QueryClass*, *QueryFormClass*, and *QueryFormVisual* for more information.

The BrowseQueryButton template provides the following prompts:

## General

### Query Interface

Select the query interface from the drop-down list. Choose from

*Form*                      One input field and button per Query field

*List*                        One listbox row per Query field

### Auto Populate

Check this box to provide a query dialog with filter criteria for each field in the BrowseBox. The input fields have the same picture token and prompt as the corresponding BrowseBox field.

Clear this box to enable the **Fields** button and specify custom query input fields.

### Fields

Press this button to populate specific query input fields. You can use this option to restrict the query to some subset of BrowseBox fields, or to expand the query to include fields not in the BrowseBox. You can also implement caseless searches by default.



**Field**                      Type the field name to include in the Query dialog, or press the ellipsis button to select the field from the **Select Field** dialog.

**Title**                        Type the prompt or label associated with the Query field.

**Picture**                    Type a picture token for the Query field, or press the ellipsis button to select a token with the **Edit Picture** dialog.

**Caseless** Check this box to do case insensitive searches on the Query field. Clear the box to do case sensitive searches.

### **QBE Class**

Select this tab to override the global Query Manager setting. See *Template Overview—Classes Tab Options—Global and Local*.

### **QBE Visual Class**

Select this tab to override the global Query Manager setting. See *Template Overview—Classes Tab Options—Global and Local*.

## **BrowseSelectButton**

---

The BrowseSelectButton template provides Select button to choose a record from a list box.

The generated source code gets the currently selected record from the list (makes the selected record the current one in the browsed file's record buffer), and closes down the procedure. For the end user, pressing the Select button is equivalent to double-clicking an item in the list.

The BrowseSelectButton template provides the following prompts:

#### **Hide the Select button when not applicable**

Check this box to hide the Select button when the procedure is not called for selection purposes (GlobalRequest <> SelectRecord).

#### **Allow Select via Popup**

Check this box to allow record selection with a RIGHT-CLICK popup menu. The template adds a popup menu item whose text matches the text on the Select button. The menu item is disabled when the Select button is disabled or hidden.

## **BrowseToolboxButton**

---

The BrowseToolboxButton template provides a **Toolbox** button. Pressing the button starts a floating, dockable toolbox containing buttons that invoke the BrowseBox actions defined by the BrowseBox popup menu (Insert, Change, Delete, Select, Print, etc.).

The BrowseBox template automatically adds the Toolbox choice to its popup menu; therefore you can HIDE the **Toolbox** button but still provide access to the toolbox with the popup menu.

The `BrowseToolboxButton` template provides no configuration prompts.

The standard `Toolbox` behavior is defined by the ABC Library's `PopupClass`. See *PopupClass* for more information.

## BrowseUpdateButtons

---

The `BrowseUpdateButtons` template provides three buttons for managing file I/O for a `BrowseBox`: `Insert`, `Change`, and `Delete`. These three buttons act on the records in a `BrowseBox`. When pressed, the button retrieves the selected record and invokes the respective database action for that record. See *Procedure Templates—Inter-Procedure Communication* for more information on requested database actions.

The `BrowseUpdateButtons` template lets you specify a separate update procedure (recommended for files with two-way relationships) or edit-in-place updates (recommended for lookup files—files with one-way relationships).

The `BrowseUpdateButtons` template provides the following prompts:

### Update Procedure

Type a procedure name or select a procedure name from the drop-down list. If you type a new procedure name, the Application Generator adds the new procedure to the Application Tree.

### Use Edit in place

Check this box to let the end user update the browsed file by typing directly into the `BrowseBox` list. This provides a very direct, intuitive spread-sheet style of update. You may configure the edit-in-place behavior with the **Configure Edit in place** button.

### Configure Edit in place

Press this button to open the **Configure Edit in place** dialog. This dialog provides the following prompts:

### Save

The **Configure Edit in place** dialog offers the **Save** option for four different keyboard actions. These options determine whether changes to an edited record are saved or abandoned upon the following keyboard actions: `TAB` key at end of row, `ENTER` key, up or down arrow key, focus loss (changing focus to another control or window, typically with a mouse-click). Choose from:

<i>Default</i>	Save the record as defined in the <code>BrowseClass.Ask</code> method.
<i>Always</i>	Always save the record.
<i>Never</i>	Never save the record, abandon the changes.

*Prompted*

Ask the end user whether to save, abandon, or continue editing the changes.

**Remain editing**

The **Configure Edit in place** dialog offers the **Remain editing** option for three different keyboard actions. Check these boxes to continue editing upon the following keyboard actions: TAB key at end of row, ENTER key, up or down arrow key. Clear the boxes to stop editing.

**Retain column**

The **Configure Edit in place** dialog offers the **Retain column** option for the up and down arrow keys only. Check this box to continue editing within the same list box column in the new row. Clear to continue editing within the left most editable column in the new row.

**Column Specific**

Press this button, then press the **Insert** button to open the **Column Specific** dialog to specify the CLASS (and object) to use when editing a specific list box field. The **Column Specific** dialog contains the following options.

**Field**

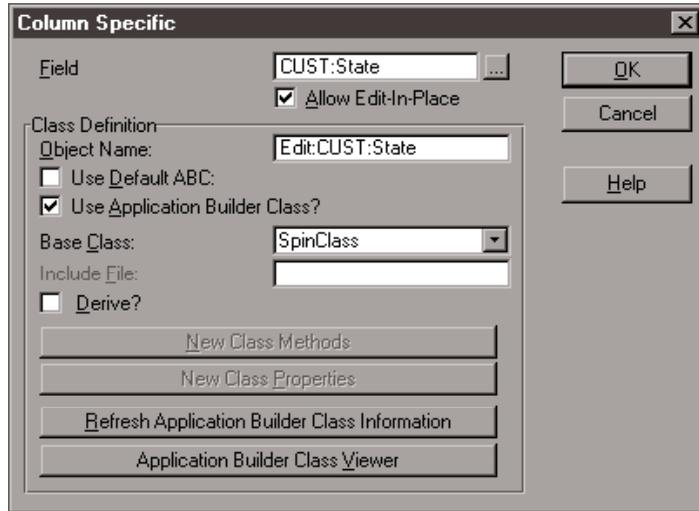
Press the ellipsis (...) button to select the field to edit ( or type the field name in the entry box. This must be one of the fields displayed in the BrowseBox.

**Allow Edit-In-Place**

Check this box to let the end user edit the field. Clear the box to prevent the end user from editing the field. Use this option is to selectively block edit-in-place access to some fields, but not others.

**Class Definition** Specify your own or a third party class. See *Template Overview—Classes Tab Options—Local* for complete information on these options.

By default, the BrowseUpdateButton template generates code to use the EditClass in the ABC Library (see *EditClass* in the *Browse Classes* chapter). You could, however, derive a SpinClass from the EditClass, then use the SpinClass to edit numeric BrowseBox fields.



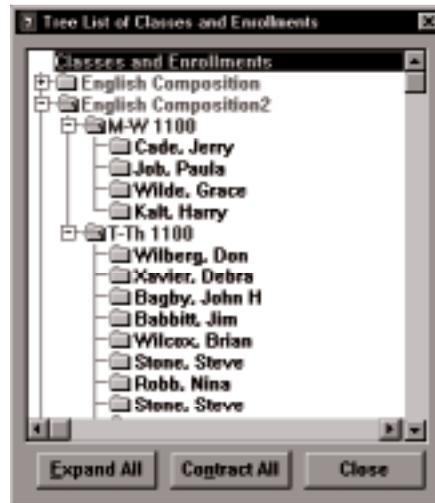
## RelationTree Overview

The tree control is a list box formatted to display as a collapsible hierarchical list. This Control template provides an alternative for the Browse-Form paradigm. A single RelationTree control can replace several Browse-Form pairs.

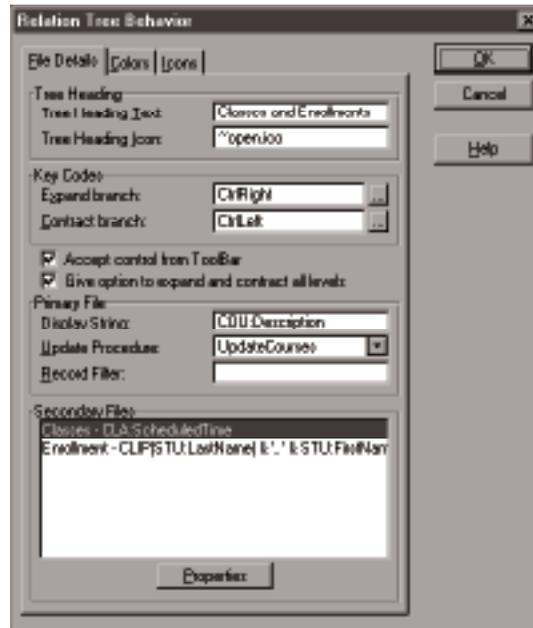
Using the RelationTree Control template, you can specify multiple related files to display on multiple levels (up to 29) of a hierarchical list—*with an associated update procedure for each level*. The related files are declared in the File Schematic—the Primary (Parent) file and a single chain of related secondary Child files (Parent-Child-GrandChild).

The RelationTree template employs a fully-loaded QUEUE for the root level. The child levels are demand-loaded when a branch is expanded.

**Tip:** This template is not appropriate for databases with a very large primary file. For large files you should use the BrowseBox Control template.



The plus (+) sign indicates a collapsed level that expands when the user CLICKS on the plus (+) sign. Conversely, the minus (-) sign indicates an expanded level that collapses when the user CLICKS on the minus (-) sign.



To create a tree using the *RelationTree Control* template:

1. Place a *RelationTree Control* template on a window.

This opens the **List Box Formatter**. Use the **List Box Formatter** to enable colorization, icon display, or horizontal scrolling in your tree control (see *The List Box Formatter*). Do not use the **List Box Formatter** to populate fields in the tree control.

**Tip:** The tree control is a single column list, therefore you must specify a column scroll bar rather than a list scroll bar to accomplish horizontal scrolling.

2. Press the **OK** button on the **List Box Formatter**.
3. RIGHT-CLICK on the *RelationTree Control* template and choose **Actions** from the popup menu.
4. Press the **Files** button to specify the file schematic for the control. Specify the Primary (Parent) file and a single chain of related Secondary Child files (Parent-Child-GrandChild).
5. Complete the *RelationTree* template prompts.

## RelationTree Options

---

The RelationTree template provides the following prompts:

### File Details

#### **Tree Heading Text**

An optional text heading at the top of the tree. Tree Heading Text is required to let the user add a record at the root level.

#### **Tree heading Icon**

An optional icon at the top of the tree. Icons must be enabled in the List Box Formatter for this prompt to be enabled.

#### **Expand Branch**

Specify a keystroke to expand the selected list item—display its children. Press the ellipsis button (...) to select special keys such as ESC, TAB OR ENTER. See *Controls and Their Properties—Common Control Attributes—Setting the KEY Attribute* for more information on this dialog.

#### **Contract Branch**

Specify a keystroke to contract the selected list item—hide its children. Press the ellipsis button (...) to select special keys such as ESC, TAB OR ENTER. See *Controls and Their Properties—Common Control Attributes—Setting the KEY Attribute* for more information on this dialog.

#### **Accept control from Toolbar**

Check this box to accept navigation events and other relation tree control events generated by the *FrameBrowseControl* control template on the APPLICATION's toolbar. See *FrameBrowseControl* for more information on these toolbar buttons and their operation. Clear this box to disable the *FrameBrowseControl* toolbar buttons for this procedure.

#### **Give option to expand and contract all levels**

Specify the RIGHT-CLICK popup menu for the RelationTree includes “Expand All” and “Contract All” commands.

### Primary File Settings

#### **Display String**

The field name or text to display for the primary file level. This may be any valid Clarion expression, for example:

```
CLIP(CUST:LastName)&' '&CUST:FirstName
```

#### **Update Procedure**

The update procedure to call for the primary file. The procedure may be accessed with the RIGHT-CLICK popup menu automatically provided when you specify an update procedure. The default popup menu text is “Insert,” “Change,” and “Delete.”

The procedure may also be accessed with the `RelationTreeUpdateButtons`—see below. If you use the `RelationTreeUpdateButtons` control template, the popup menu inherits the text from the buttons.

#### **Record Filter**

Type a valid Clarion expression to limit the contents of the list to only those records causing the expression to evaluate to true (nonzero or non-blank). The procedure loops through all displayable records to select only those that meet the filter.

You must **BIND** any file field that is used in a filter expression. See **BIND** in the *Language Reference* for more information.

### **Colors (Primary File)**

This tab is only available if you check the **Color Cells** box in the **List Field Properties** in the List Box Formatter.

#### **Default Colors**

To specify the default colors for the primary file display string, type color EQUATES (from `\LIBSRC\EQUATES.CLW`) in the entry fields or press the ellipsis (...) buttons to select colors from the **Select Color** dialog.

#### **Conditional Color Assignments**

To specify conditional colors for the primary file display string, press the **Insert** button. This opens the **Conditional Color Assignments** dialog.

### **Conditional Color Assignments**

This dialog lets you specify the conditional colors for the primary file display string.

#### **Condition**

Type a valid Clarion expression to evaluate at runtime, then type color EQUATES (from `\LIBSRC\EQUATES.CLW`) in the entry fields or press the ellipsis (...) buttons to select colors from the **Select Color** dialog.

At run-time these conditions are evaluated, and the colors for the first true condition in the list are used.

### **Icons (Primary File)**

This tab is only available if you check the **Icons** box in the **List Field Properties** in the List Box Formatter.

#### **Default Icon**

To specify the default icon for the primary file display string, type the icon filename in the entry field.

### **Conditional Icon Usage**

To specify conditional icons for the primary file display string, press the **Insert** button. This opens the **Conditional Icon Usage** dialog.

### **Conditional Icon Usage**

This dialog lets you specify conditional icons for the primary file display string.

#### **Condition**

Type a valid Clarion expression to evaluate at runtime.

#### **Icon**

Type the icon filename in the entry field.

At run-time these conditions are evaluated, and the icon for the first true condition in the list is used.

### **Secondary File Settings**

The secondary file settings are identical to the primary file settings. Highlight the secondary file, then press the **Properties** button below the **Secondary Files** list box. See *RelationTree Overview* for information on how to specify the secondary files with the **Select File** dialog.

### **RelationTree Embed Points**

The RelationTree Control template provides a comprehensive set of embed points to allow full customization of the control's behavior.

## **RelationTreeUpdateButtons**

---

This Control template adds three buttons (**Insert**, **Change**, and **Delete**) which allow the end user to call the associated update procedure for the selected level of a RelationTree. There are no prompts for this control. The Update Procedure is specified for each level of the RelationTree Control template.

The Change and Delete buttons correspond to the currently highlighted record. The **Insert** button adds a child record (the next level down the tree structure).

### **RelationTreeUpdate Embed Points**

The RelationTreeUpdateButtons Control template provides a comprehensive set of embed points to allow full customization of the control's behavior.

## Other Window Control Templates

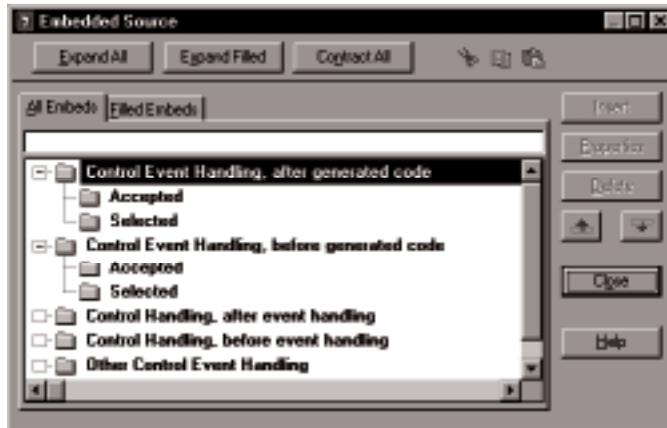
### CancelButton

---

The CancelButton template adds a single button control marked **Cancel**. This button lets the user close a window and it provides a convenient place for the developer to add code to “undo” before closing down the procedure. The generated source code sets a “Request Cancelled” flag and closes down the window procedure.

The CancelButton template provides no configuration options.

You can insert the executable code you need to “clean up” at an embed point.



### CloseButton

---

The CloseButton template adds a single button control marked **Close**. The generated source code sets a “Request Completed” flag and closes down the window procedure.

The CloseButton template provides no configuration options.

### DOSFileLookup

---

The DOSFileLookup template adds an ellipsis (...) button which opens the standard Windows file dialog.



You can specify the file masks, the default directory and filename, and the variable to receive the filename selected by the end user.

In addition, you may optionally allow the selection of multiple files and specify the code to process each selected file. The template generates a LOOP to process all the selected files.

The DOSFileLookup template provides the following prompts:

## General



### File Dialog Header

Type the text for the caption of the Windows file dialog.

### DOS Filename Variable

Press the ellipsis (...) button to choose a variable to receive the end user's choice from the **File Schematic** dialog. You can also type the variable name directly into the entry box.

### Default Directory

Specify the starting directory for the Windows file dialog. If

blank, the file dialog opens to the working directory.

**Default Filename**

Specify the initial filename for the Windows file dialog. If blank, the file dialog opens with no initial filename.

**Return to original directory when done**

Check this box to reset the working directory to its value prior to the file lookup.

**Multi-Select?**

Check this box to allow selection of one *or more* files.

**Action For Each Selection**

Type a valid Clarion language statement to execute for each selected file—typically a procedure call. You may want to pass the **FileName Variable** as a parameter to the procedure.

The template generates a LOOP to execute the code you specify for each selected file. The generated code reloads the **FileName Variable** with the appropriate filename for each loop cycle.

## File Masks

**Use a variable file mask**

Check this box to supply the file mask with a variable. This enables the **Variable Mask Value** field to name the variable, and disables the **Mask Description**, **File Mask**, and **More File Masks** prompts.

**Mask Variable**

Names the variable that contains the file mask. See *FILEDIALOG* in the *Language Reference* for information on the contents of this variable.

**File Mask Description**

Type a file type description. The string appears in the drop-down list in the Windows file dialog. You can add additional masks by pressing the **More File Masks** button.

**File Mask**

Type a file mask specification, such as “\*.TXT” or use multiple patterns for this mask separating each with a semicolon, such as “\*.BMP;\*.GIF”. You can add additional masks by pressing the **More File Masks** button.

**More File Masks**

Press this button to add additional file masks. These masks are available to end the user through the **List files of type** drop-down list in the Windows file dialog.

## Window Update Options

**Update entire window?**

Check this box to refresh the contents of all window controls

after the file selection and processing is complete. Clear the box to select specific fields to refresh.

#### **Update Selected Fields**

Press this button to select specific fields to refresh after the file selection and processing is complete. The template generates a `DISPLAY` statement for each field you specify. See *DISPLAY* in the *Language Reference*.

### **Classes**

The Classes tab lets you control the class (and object) the template uses. You may accept the default Application Builder Class and its object (recommended), or you may specify your own or a third party class. Deriving your own class can give you very fine control over the procedure when the standard Application Builder Class is not precisely what you need.

See *Template Overview—Classes Tab Options—Local* for complete information on these options.

## **FieldLookupButton**

---

The FieldLookupButton template provides an ellipsis (...) button that lets you “look up” the value from a lookup file, such as a state file. `CLICK` next to an input control to place the lookup button.

The FieldLookupButton template provides the following prompts:

#### **Control with Lookup**

Select the associated control for which to perform the lookup by choosing its field equate label from the drop-down list. Typically this is an `ENTRY` control.

The selected control must have an associated lookup procedure. To provide the lookup procedure, `RIGHT-CLICK` on the control, then choose **Actions** to access its prompts.

## **FileDrop**

---

The FileDrop template places a file-loaded scrollable drop-down list on a window. At runtime, the end user can select an item from the list, then assign a value from the selected item’s record to a specified target field. You may display one field (such as a description field) but assign another field (such as a code field) from the selected record (see *How Do I...* in the on-line help).

**Tip:** Set the `DROP` attribute to zero (0) to display a list box rather than a drop-down list.

Immediately before you place the FileDrop Control template on your window, the Application Generator prompts you to specify the file to display in the drop-down list. Specify the file in the **Select Field** dialog. You will also need to select a field to serve as the USE variable for the LIST; however, the field you select is only significant if you are displaying one field but assigning another).

Immediately after you place the FileDrop Control template, the Application Generator opens the List Box Formatter so you can specify the fields to display in your list. You may specify the field containing the lookup value as well as other fields from the same or related files. See *The List Box Formatter* for more information.

After you specify your list fields and return to the window under construction, right-click the control, then choose **Actions** from the popup menu to complete the following FileDrop options:

### **General**

#### **Field to Fill From**

The field in the lookup file whose value is assigned to the Target Field. Press the ellipsis (...) button to select from the **Select Field** dialog.

#### **Target Field**

The field that receives the value from the Field to Fill From. Press the ellipsis (...) button to select from the **Select Field** dialog.

#### **More Field Assignments**

Press this button to specify additional value assignments from the selected item's record.

#### **Record Filter**

Type a valid Clarion expression to limit the contents of the list to only those records causing the expression to evaluate to true (nonzero or non-blank). The procedure loops through all displayable records to select only those that meet the filter. Filters are generally much slower than Range Limits.

You must **BIND** any file field that is used in a filter expression. The **Hot Fields** tab lets you **BIND** fields.

#### **Default to first entry if USE variable empty**

Check this box to provide an initial default selection—the drop-down list is never initially empty (unless the first file record is a blank one).

## Range Limits

This tab is only available if you specify a Key for the File in the **File Schematic Definition** dialog. Because range limits use keys, they are generally much faster than filters.

### **Range Limit Field**

In conjunction with the **Range Limit Type**, specifies a record or group of records for inclusion in the process. Choose a key field on which to limit the records by pressing the ellipsis (...) button.

### **Range Limit Type**

Specifies the type of range limit to apply. Choose one of the following from the drop-down list.

*Current Value* Limits the key field to its current value.

*Single Value* Lets you limit the key field to a single value. Specify the variable containing that value in the **Range Limit Value** box.

### *Range of Values*

Lets you limit the key field to a range of values. Specify the variables containing the upper and lower limits of the range in the **Low Limit Value** and **High Limit Value** boxes.

### *File Relationship*

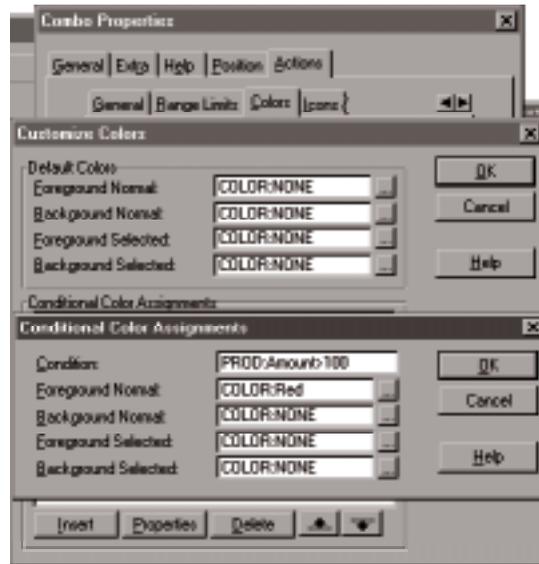
Lets you limit the key field to the current value in a related (parent) file. Press the **Related file** ellipsis (...) button to choose the range limiting file. This limits the process to include only those child records matching the current record in the parent file. For example, if your report was a list of Orders, you could limit the process to only those orders for the current Customer.

## Colors

This tab is only available if you check the **Color Cells** box in the List Box Formatter. It displays a list of the FileDrop columns which may be colored.

To specify the default colors and any conditional colors, highlight the column's field name, then press the **Properties** button. This opens the **Customize Colors** dialog.

### **Customize Colors**



This dialog lets you specify the default and conditional Foreground and Background colors for normal (unselected) rows; and for selected rows.

#### Conditional Color Assignments

Below the default colors section is the **Conditional Color Assignments** list. This list lets you set colors to apply when an expression evaluates to true (nonzero or non-blank). To add an expression and its associated colors, press the **Insert** button.

At run-time the expressions are evaluated, and the colors for the first true expression are used.

### Icons

This tab is only available if you check the **Icons** box in the List Box Formatter. It displays a list of the FileDrop columns which can display icons.

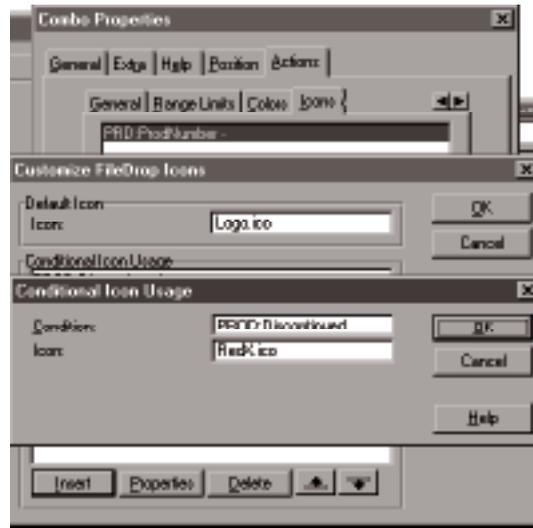
To specify default icons and any conditional icons, highlight the column's field name then press the **Properties** button. This opens the **Customize Icons** dialog.

#### Customize Icons

This dialog lets you specify the default icon and conditional icons for the FileDrop column.

##### Default Icon

The default icon to display. Type the icon (.ICO) filename.



### Conditional Icon Usage

Below the **Default Icon** section is the **Conditional Icon Usage** list. This list lets you set icons to apply when an expression evaluates to true (nonzero or non-blank). To add an expression and its associated icon, press the **Insert** button.

At run-time the expressions are evaluated, and the colors for the first true expression are used.

### Hot Fields

When you select the Hot Fields tab, you can specify fields not populated in the list to add to the QUEUE. When scrolling through the file, the generated source code reads the data for these fields from the QUEUE, rather than from the disk. This speeds up list box updates.



Specifying “Hot” fields also lets you place controls outside the FileDrop that are updated whenever a different record is selected in the list box. Elements of the Primary Key and the current key are always included in the QUEUE, so they do not need to be inserted in the Hot Field list.

This dialog also lets you BIND a field. You must BIND any field that is used in a filter expression or as a field to total.

## Sort Fields

This tab lets you add fields by which the items in the drop-down list are sorted. The sort fields are in addition to any Key specified for the FileDropCombo. Press the **Insert** button to add fields to the list.

## Classes

The Classes tab lets you control the class (and object) the template uses. You may accept the default Application Builder Class and its object (recommended), or you may specify your own or a third party class. Deriving your own class can give you very fine control over the procedure when the standard Application Builder Class is not precisely what you need.

See *Template Overview—Classes Tab Options—Local* for complete information on these options.

## Other Prompts

The List Properties for this control are the same as for a list; however, the following prompts may require some additional explanation:

<b>Use</b>	Takes either a field equate label, or the label of a variable to receive the value from the first field populated in the list. In the FileDrop Control template context, this functionality is replaced by the more flexible Target Field setting.
<b>From</b>	This field defaults to Queue:FileDrop. Queue:FileDrop is the label of the QUEUE the template uses to fill the list. Typically, you should not change this value.
<b>Mark</b>	Takes the label of the Queue:FileDrop:Mark QUEUE field to allow the user to select more than one item from the list. The Queue:FileDrop:Mark field contains 1 for selected items and 0 for unselected items.

## FileDropCombo

---

The FileDropCombo template generates code to display a data file in a scrollable list, select one of the records from the list, then assign a value from the selected record to a specified target field. Note that you may display one field (such as a description field) but assign another field (such as a code field) from the selected record (see How Do I... in the on-line help). Also, because the template is based on a COMBO control, the generated code accepts entry values that may not exist in the displayed list and optionally adds these new values to the lookup file.

Immediately before you place the FileDropCombo Control template on your window, the Application Generator prompts you to specify the file to display in the drop-down list. Specify the file in the **Select Field** dialog. You will also need to select a field from the file to serve as the USE variable for the COMBO. The USE variable is significant when you Allow Updates from the FileDropCombo or when you display one field but assign another. See *Update Behavior* for more information.

Immediately after you place the FileDropCombo Control template, the Application Generator opens the List Box Formatter so you can specify the fields to display in your list. You may specify the field containing the lookup value as well as other fields with associated information. See *The List Box Formatter* for more information.

After you specify your list fields and return to the window under construction, RIGHT-CLICK the control, then choose **Actions** from the popup menu to complete the following FileDropCombo options:

### General

#### **Field to Fill From**

The field in the lookup file whose value is assigned to the Target Field. Press the ellipsis (...) button to select from the **Select Field** dialog.

#### **Target Field**

The field that receives the value from the Field to Fill From. Press the ellipsis (...) button to select from the **Select Field** dialog.

#### **More Field Assignments**

Press this button to specify additional value assignments from the selected item's record.

#### **Record Filter**

Type a valid Clarion expression to limit the contents of the list to only those records causing the expression to evaluate to true (nonzero or non-blank). The procedure loops through all displayable records to select only those that meet the filter.

Filters are generally much slower than Range Limits.

You must **BIND** any file field that is used in a filter expression. The **Hot Fields** tab lets you **BIND** fields.

**Default to first entry if USE variable empty**

Check this box to provide an initial default selection—the drop-down list is never initially empty (unless the first file record is a blank one).

**Remove duplicate entries**

Check this box to remove duplicates from the list.

**Keep View synchronized with Selection?**

Check this box to update the **VIEW**'s record buffers to match the selected item.

**Case Sensitive matches?**

Check this box to consider case when matching entered values with values in the lookup file.

## Range Limits

This tab is only available if you specify a Key for the File in the **File Schematic Definition** dialog. Because range limits use keys, they are generally much faster than filters.

**Range Limit Field**

In conjunction with the **Range Limit Type**, specifies a record or group of records for inclusion in the process. Choose a key field on which to limit the records by pressing the ellipsis (...) button.

**Range Limit Type**

Specifies the type of range limit to apply. Choose one of the following from the drop-down list.

*Current Value* Limits the key field to its current value.

*Single Value* Lets you limit the key field to a single value. Specify the variable containing that value in the **Range Limit Value** box.

*Range of Values* Lets you limit the key field to a range of values. Specify the variables containing the upper and lower limits of the range in the **Low Limit Value** and **High Limit Value** boxes.

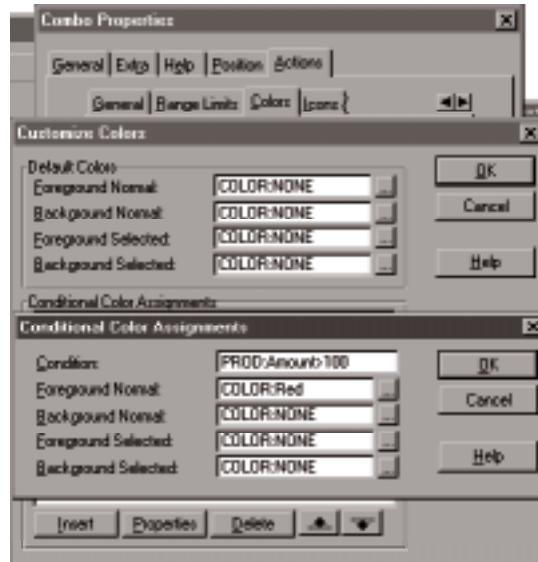
*File Relationship*

Lets you limit the key field to the current value in a related (parent) file. Press the **Related file** ellipsis (...) button to choose the range limiting file. This limits the process to include only those child records matching the current record in the parent file. For

example, if your report was a list of Orders, you could limit the process to only those orders for the current Customer.

## Colors

This tab is only available if you check the **Color Cells** box in the List Box Formatter. It displays a list of the FileDropCombo columns which may be colored.



To specify the default colors and any conditional colors, highlight the column's field name, then press the **Properties** button. This opens the **Customize Colors** dialog.

### Customize Colors

This dialog lets you specify the default and conditional Foreground and Background colors for normal (unselected) rows; and for selected rows.

#### Conditional Color Assignments

Below the default colors section is the **Conditional Color Assignments** list. This list lets you set colors to apply when an expression evaluates to true (nonzero or non-blank). To add an expression and its associated colors, press the **Insert** button.

At run-time the expressions are evaluated, and the colors for the first true expression are used.

## Icons

This tab is only available if you check the **Icons** box in the List Box Formatter. It displays a list of the FileDropCombo columns which can display icons.



To specify default icons and any conditional icons, highlight the column's field name then press the **Properties** button. This opens the **Customize Icons** dialog.

### **Customize Icons**

This dialog lets you specify the default icon and conditional icons for the FileDropCombo column.

#### **Default Icon**

The default icon to display. Type the icon (.ICO) filename.

#### **Conditional Icon Usage**

Below the **Default Icon** section is the **Conditional Icon Usage** list. This list lets you set icons to apply when an expression evaluates to true (nonzero or non-blank). To add an expression and its associated icon, press the **Insert** button.

At run-time the expressions are evaluated, and the colors for the first true expression are used.

## Update Behavior

This tab lets you use the entry portion of the COMBO to initiate adding a new record to the lookup file. If the user types a value in the entry box that is not already in the list, the generated code can add a new record directly, or it can call a separate procedure to add the new entry.

- Allow Updates** Clear this box to accept entries that do not exist in the lookup file. The new (unvalidated) entries are *not* added to the lookup file.
- Check this box to add new entries to the lookup file, and to enable the **Update Procedure** prompt.
- Update Procedure** Name the procedure to call to add the new record, or leave this field blank if no update procedure is needed.
- No update procedure is needed for lookup files with only one required field (the field specified by the COMBO's USE variable). Non-USE fields are CLEARed, unless range limited or auto-incremented.

### Hot Fields

Use the Hot Fields tab to specify fields to add to the QUEUE that are not displayed in the list. When scrolling through the file, the generated source code reads the data for these fields from the QUEUE, rather than from the disk. This speeds up list box updates.

Specifying Hot Fields effectively lets you update other controls whenever a new record is selected in the list box. Elements of the Primary Key and the current key are always included in the QUEUE, so they do not need to be inserted in the Hot Field list.

Press the **Insert** button to add fields to the list.

### Sort Fields

This tab lets you add fields by which the items in the drop-down list are sorted. The sort fields are in addition to any Key specified for the FileDropCombo. Press the **Insert** button to add fields to the list.

### Classes

The Classes tab lets you control the class (and object) the template uses. You may accept the default Application Builder Class and its object (recommended), or you may specify your own or a third party class. Deriving your own class can give you very fine control over the procedure when the standard Application Builder Class is not precisely what you need.

See *Template Overview—Classes Tab Options—Local* for complete information on these options.

## Other Prompts

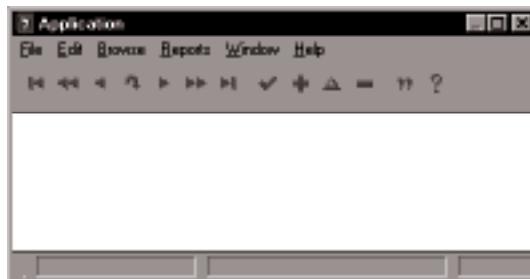
The List Properties for this control are the same as for a list; however, the following prompts may require some additional explanation:

<b>Use</b>	Takes either a field equate label or the label of a variable to receive the value from the first field populated in the list. In the FileDropCombo Control template context, the assignment functionality is replaced by the more flexible Target Field; however, the USE variable is significant when you Allow Updates from the FileDropCombo (see <i>Update Behavior</i> for more information).
<b>From</b>	This field defaults to Queue:FileDropCombo. Queue:FileDropCombo is the field equate label of the QUEUE the template generates to fill the list. Typically, you should not change this value.
<b>Mark</b>	Takes the label of the Queue:FileDropCombo:Mark QUEUE field to allow the user to select more than one item from the list. The Queue:FileDropCombo:Mark field contains 1 for selected items and 0 for unselected items.

## FrameBrowseControl

The FrameBrowseControl template places thirteen (13) standard command buttons on the toolbar of an MDI APPLICATION (Frame procedure). When the user presses these buttons, the template generated code posts appropriate events (scroll up, scroll down, add, change, delete, help, etc.) to the active procedure and control.

**Tip:** You may delete buttons that your application does not use. For example, the ABC Templates by default do not use the  (locate) button.



The buttons are designed to work with the BrowseBox Control template, the RelationTree Control template, and the FormVCRControls Extension template; that is, the buttons remain disabled until the program calls a procedure with a BrowseBox template or a RelationTree template whose **Accept browse control from Toolbar** box is checked, or the BrowseBox procedure calls a Form procedure with a FormVCRControls extension template.

In addition, the called procedure's WINDOW must have the MDI attribute, but don't worry, the standard Browse and Form templates declare MDI windows by default—you don't need to do anything special to accomplish this. The BrowseBox and RelationTree templates also check the **Accept browse control from Toolbar** box by default—so again, you don't need to do anything special to accomplish this.

The FrameBrowseControl toolbar buttons operate as follows:



Scrolls to the first row in a BrowseBox or to the previous parent record in a RelationTree. For Form procedures, saves the current record before scrolling.



Scrolls up one page in a BrowseBox or to the previous record on the same level in a RelationTree. For Form procedures, saves the current record before scrolling.



Scrolls up one row in the BrowseBox or to the previous record on any level in a RelationTree. For Form procedures, saves the current record before scrolling.



Locates a specific record in a BrowseBox. This button is enabled only if you specify an Entry locator or an Incremental locator for the BrowseBox. See *Control Templates—BrowseBox* for information on specifying these locators.



Scrolls down one row in the BrowseBox or to the next record on any level in a RelationTree, expanding the tree branch if necessary. For Form procedures, saves the current record before scrolling.



Scrolls down one page in the BrowseBox or to the next record on the same level in a RelationTree. For Form procedures, saves the current record before scrolling.



Scrolls to the last row in the BrowseBox or to the next parent record in a RelationTree. For Form procedures, saves the current record before scrolling.



Selects the highlighted row in a BrowseBox. This is only appropriate when the procedure is called to select a record. For example, when called as a lookup.



For a BrowseBox, calls a *Form* procedure to add a new record. For a RelationTree, calls a *Form* procedure to add a child record of the currently highlighted record. For a Form procedure, adds another record of the same type.



Calls a *Form* procedure to change the record highlighted in the BrowseBox or RelationTree.



Deletes the record highlighted in the BrowseBox or RelationTree. The BrowseBox delete behavior is determined by the settings on the Update Buttons Control template.



On a *Form* procedure only, pastes into the field with focus, the corresponding value from the previously processed record (the value in the record buffer). In other words, repeat the value from the previous saved record.



Invokes Windows standard help behavior: calls WINHELPEXE with the help topic or keyword specified by the WINDOW's HLP attribute.

The FrameBrowseControl template provides no options.

## PauseButton

---

The PauseButton template places a button on the progress window for a Process or Report procedure. When the user presses the button, the template generated code changes the button text and suspends the procedure until the users presses the button again to restart the procedure.



The PauseButton template provides the following prompts:

**Pause Text**

The text to display on the button face when the procedure resumes.

**Restart Text**

The text to display on the button face when the procedure is suspended.

**Start Paused**

Check this box to initially suspend the procedure so that it only starts when the end user presses the button. Clear the box to initially resume the procedure so that it starts and runs to completion unless the end users presses the button. See *DeferOpenReport* in the *Report Manager* chapter.

**Start Text**

The text to display on the button face when the procedure is initially suspended.

**Allow multiple starts**

Check this box to allow the end user to restart the process or report after it completes. This is useful for rerunning a process or report with user specified filters and sort orders.

## SaveButton

---

The SaveButton template provides an **OK** button for your window, plus the capability to display an action message for the end user. The SaveButton handles most of the file I/O for the procedure.

The SaveButton template provides the following prompts:

**Allow**

Check any combination of the three boxes to specify permitted file I/O operations. Conversely, clear the box to prevent the associated operation.

**Inserts** Generates code to handle record inserts.

**Changes** Generates code to handle record changes.

**Deletes** Generates code to handle record deletes.

**Tip:** The SaveButton template does not detect changes to BLOBs; therefore, if only the BLOB changes, the SaveButton template does not save it. The School example application contains a work around to this problem.

**Field Priming on Insert**

Field Priming lets you provide a default value for fields in a new record. This value supersedes any initial value specified in the data dictionary. You can select a field and set an initial value in the **Field Priming** dialog.

**Messages and Titles**

Press this button to open the **Messages and Titles** dialog to specify update messages and their locations. In addition, this dialog controls some fundamental behavior associated with the procedure, such as whether it confirms before cancelling and whether it allows repetitive adds.



**Insert Message**

Specifies the text for the action message when the procedure is called to add a record.

**Change Message**

Specifies the text for the action message when the procedure is called to change a record.

**Delete Message**

Specifies the text for the action message when the procedure is called to delete a record.

**On Aborted Add/Change**

Specifies the action to take when the user presses the **Cancel** button while adding or modifying a record. Choose from:

*Offer to save changes*

Displays a message box prompting to save changes before cancelling.

*Confirm Cancel*

Displays a message box prompting asking if you really want to cancel.

*Cancel without Confirming*

Displays no message before cancelling.

**Field History Key**

Specify a key that restores the value from the last saved record. When the end user presses the specified key, the generated code retorets the field with focus from the previously processed record.

The default key (734) is CTRL+SINGLE-QUOTE ('). On most US keyboards this is the unshifted double-quote ("). On most UK keyboards this is the unshifted at-sign (@).

Specifvng a key here also enables the FrameBrowseControl's ditto  button. This button also restores the value from the last saved record.

**When called for Delete**

Specify what displays when this procedure is called to delete a record. Choose from:

*Standard Warning*

Displays a message box prompting for confirmation of the delete.

*Show Form*

Displays the form.

*Automatic Delete*

Deletes items without end user confirmation.

**After successful insert**

Select one-at-a-time insert mode or repetitive insert mode. Choose from:

*Return to caller* Generates a RETURN to the calling procedure following a successful insert. This results in a one-at-a-time insert mode.

*Insert another record*  
Does not generate a RETURN to the calling procedure following a successful insert. This results in a repetitive insert mode.

*Ask the user before adding another record*  
Does not automatically generate a RETURN to the calling procedure following a successful insert, but asks the user whether to add another record.

#### **Location of Message**

Specifies where the message displays. Choose from:

*None/Window Control*  
Embed your own code to display the message in a control.

*Title Bar*            Display the message in the window's title bar.

*Status Bar*            Display the message in the window's status bar. Optionally specify which section of the status bar in the **Status Bar Section** box.

#### **Display Record Identifier on the Title Bar**

Check this box to append a string to the caption on the window's titlebar. Specify the string in the **Record Identifier** field.

#### **Record Identifier**

Specifies the string to append to the titlebar caption, which you can use to identify the record. Type a string in the Record Identifier box. To use a variable name, precede it with an exclamation point ( ! ).

## Report Control Templates

The ABC Templates contain a few control templates designed to quickly handle some of the most repetitive report text. These controls include date stamps, time stamps, and page numbers. This section describes the ABC Report Control templates.

### ReportDateStamp

---

The ReportDateStamp template adds two STRING controls to a REPORT: a “Report Date:” text STRING, and a formatted variable STRING to display the date. By default, the ReportDateStamp template displays the system date using the Windows standard long date format (D18). For example, November 18, 1997. However, you may select an alternative format and an alternative date value to display.



The ReportDateStamp template provides the following prompts:

#### Format Picture

Press the ellipsis button to select a date format. See *Picture Tokens* in the *Language Reference*.

#### Use System Clock?

Check this box to display the system date (see *TODAY* in the *Language Reference*). Clear the box to display a variable containing the date value to display.

#### Date Variable

Type the variable name or press the ellipsis button to select the variable from the **Select Fields** dialog.

## ReportTimeStamp

---

The ReportTimeStamp template adds two STRING controls to a REPORT: a “Report Time:” text STRING, and a formatted variable STRING to display the time. By default, the ReportTimeStamp template displays the system time using the Windows standard long time format (T8). For example, 12:90:22 PM. However, you may select an alternative format and an alternative time value to display.

The ReportTimeStamp template provides the following prompts:

**Format Picture**

Press the ellipsis button to select a time format. See *Picture Tokens* in the *Language Reference*.

**Use System Clock?**

Check this box to display the system date (see *CLOCK* in the *Language Reference*). Clear the box to display a variable containing the time value to display.

**Time Variable**

Type the variable name or press the ellipsis button to select the variable from the **Select Fields** dialog.

## ReportPageNumber

---

The ReportPageNumber template adds a variable STRING to display the page number.

The ReportPageNumber template provides no configuration prompts.



# 5 - CODE AND EXTENSION TEMPLATES

## Code Templates

Code templates generate source code into an embed point that you specify, and sometimes into other embed points as well. Their purpose is to make procedure customization quick and easy. Each Code template has one well-defined task. For example, the Initiate Thread Code template simply starts a new execution thread, and no more. Typically, the Code template provides a dialog box with prompts and instructions.

Add Code templates to your procedure with the **Embedded Source** dialog. See *Application Generator—Embedded Source*.

## CallABCMethod

The CallABCMethod template generates code to call an ABC Library object method. See *Part II—ABC Library* for more information on these methods. This template generates code similar to the following:

```
Default:City = INIMgr.Fetch('Preferences','City')
```



### Object Name

Select the label of the object from the list. The list contains all ABC compliant objects in scope for this procedure.

### Method to Call

Select the method to call from the drop-down list. Scroll the list horizontally or press the **Application Builder Class Viewer** button to see all the method parameters and return values. See *Part II—ABC Library* for complete information on these methods, their parameters, and their return values.

### Passed Parameters

Type the parameter list to pass. Enclose the parameters in parentheses and separate them with commas. The parameters may be literal values, expressions, or variable names.

### Return Value Assignment

Type the variable to receive the called method's return value. This field is only available for methods that return a value.

## CallProcedureAsLookup

---

The CallProcedureAsLookup template calls a procedure to select a record. It sets a variable called RequestCompleted to advise whether the lookup was successful or not.

**Lookup Procedure**

Specifies the procedure to call to perform the lookup.

**Code before**

Type in any executable code to execute before performing the lookup. You can use multiple statements by separating them with a semicolon.

**Code After, Completed**

Type in any executable code to execute after completing a lookup. You can use multiple statements by separating them with a semicolon.

**Code After, Canceled**

Type in any executable code to execute if the lookup is canceled. You can use multiple statements by separating them with a semicolon.

## CloseCurrentWindow

---

The CloseCurrentWindow template simply posts an EVENT:CloseWindow, which shuts down the procedure normally. There are no prompts to fill in.

## ControlValueValidation

---

The ControlValueValidation template gets the value of a control and matches it against the value in a key. You can add this Code template at the Accepted or Selected embed point for an ENTRY, SPIN, LIST, or COMBO control. The code generated by this Code template gets the value in the control, then matches it against the value in the key.

It can also call a lookup procedure to let the end user select a value.

**Lookup Key**

Specifies the key to lookup. If the key is a multi-component key you must prime the other (non-lookup field) components before this template's code is executed.

**Lookup Field**

Specifies both the field to validate and the target of a successful lookup. The Lookup Field must be a component of the Lookup Key.

### Lookup Procedure

Specifies the lookup procedure to call.

This template generates code similar to the following.

```

IF CUST:State OR ?CUST:State{Prop:Req} <> False
  ST:StateCode = CUST:State           ! Move value for lookup
  IF Access:State.TryFetch(ST:ByCode) ! IF record not found
    GlobalRequest = SelectRecord      ! Set Action for Lookup
    SelectState                       ! Call Lookup Procedure
    IF GlobalResponse = RequestCompleted! IF Lookup successful
      GlobalResponse = RequestCancelled ! Clear the Action Value
      CUST:State = ST:StateCode        ! Move value to control field
    ELSE                               ! ELSE (IF Lookup NOT...)
      SELECT(?CUST:State)             ! Select the control
      CYCLE                           ! end event processing
    END                               ! END (IF Lookup successful)
  END                                 ! END (IF record not found)
END
END

```

## DisplayPopupMenu

The DisplayPopupMenu template generates code to define and display a popup menu, and optionally, act on the end user's selection. You can set the popup menu items to mimic existing buttons on the window so that the associated menu item text matches the *button* text, is enabled only when the *button* is enabled, and, when selected, invokes the *button* action.

The DisplayPopupMenu template relies on the PopupClass to accomplish its tasks. See *PopupClass* for more information.



### String variable for

Press the ellipsis (...) button to select or define a string variable to receive the end user's popup menu selection. After the popup menu displays, this variable contains the selected item's text minus any special characters. That is, the variable contains only characters 'A-Z', 'a-z', and '0-9'. If the resulting value is not unique for the menu, the PopupClass appends a sequence number to the value to make it unique.

You may interrogate this variable and perform actions depending on its value. If you rely on the `PopupClass` mimic capability to perform appropriate actions, then you can leave this field blank. See *Item Properties* for more information on mimic.

### Build Menu From

Choose how the popup menu and its items are defined:

- |             |   |
|-------------|---|
| Menu String | Use the <b>Menu String</b> field to type the menu definition, then use the <b>Item Properties</b> to define each item's behavior.   |
| Item List   | Use the <b>Menu Items</b> button to define menu items one at a time.  |
| INI File    | Use the <b>Menu Description</b> field to name the INI file section which contains the menu definition. By default, the template code uses the global <code>INIMgr</code> object declared by the ABC Application template. If you have not specified an INI file to use, the <code>INIMgr</code> object uses Windows INI file. See <i>Template Overview—Global Options Tab</i> . |

### Menu Description

Type the INI file section which contains the menu definition. See *PopupClass—Save and Restore* for more information.

### Menu String

Type a menu definition string. The *Language Reference* describes the syntax for the menu definition string under the *selections* parameter for the `POPUP` command.

### Item Properties

Press this button to define the properties for each popup menu item. Only items specified in the Menu String are valid. You can set the popup menu items to mimic existing buttons on the window so that the associated menu item text matches the *button* text, is enabled only when the *button* is enabled, and, when selected, invokes the *button* action. You can also set the popup menu items to post an event to a control.

### Menu Items

Press this button to define the text for each popup menu item. You can set the popup menu items to mimic existing buttons on the window so that the associated menu item text matches the *button* text, is enabled only when the *button* is enabled, and, when selected, invokes the *button* action. You can also set the popup menu items to post an event to a control.

## Classes Tab

Use the Classes tab to override the global Popup Manager setting. See *Template Overview—Classes Tab Options—Global and Local*.

## InitiateThread

---

When opening an MDI window from an Application Frame, you must initiate an execution thread. This Code template provides an easy way to initiate a thread.



When you **START** a procedure on its own thread, the procedure and its window operate independently of other threads in the same program; that is, the end user can switch focus between each execution thread at will. These are “**modeless**” windows.

If you don't initiate a new thread, the program behavior depends on whether the procedure's window has the MDI attribute. A *non-MDI* child window on the same thread as its parent, blocks access to all other threads in the program. This is an “**application modal**” window. When the application modal window closes, the other execution threads are available again. An *MDI* child window on the same thread as its parent, blocks access only to its parent window. When the MDI child window closes, its parent window regains focus.

In the **Prompts for Initiate Thread** dialog, simply name the procedure that opens the MDI window. Optionally, you can modify the size of the stack to allocate to the new execution thread. The default stack is 25,000 bytes.

You can optionally add a line of code to execute if the application was unable to open the thread. Type in the edit box labelled **Error Handling**. For example,

```
MESSAGE('Could not Start Thread', 'Error', ICON:HAND)
```

would display a message box with the halt (hand) icon, if the thread failed to start.

You can add a procedure name to call upon an error by typing the name of the procedure in the **Error Handling** box. You would then add the procedure to the **Application Tree** with the **Insert Procedure** command.

## LookupNonRelatedRecord

---

The LookupNonRelatedRecord template is used to perform a lookup of a value based on a relationship, whether it is or is not defined in the data dictionary (Ad hoc relation). You can add this Code template to the Lookup Up Related Records embed point.

### Lookup Key

Type in the key name or press the ellipsis (...) button to select the key from the File Schematic.

The lookup key is used to perform the lookup into the lookup file. This *must* be a unique key. If the key is a multicomponent key, the other key elements must be primed before executing this Code template.

**Lookup Field**

Type in the field name or press the ellipsis (...) button to select the field from the Component list.

The Lookup Field must be a component of the Lookup Key. This is the unique value within the lookup file.

**Related Field**

Type in the related field or press the ellipsis (...) button to select it from the File Schematic.

The Related Field provides the unique value used to perform the lookup.

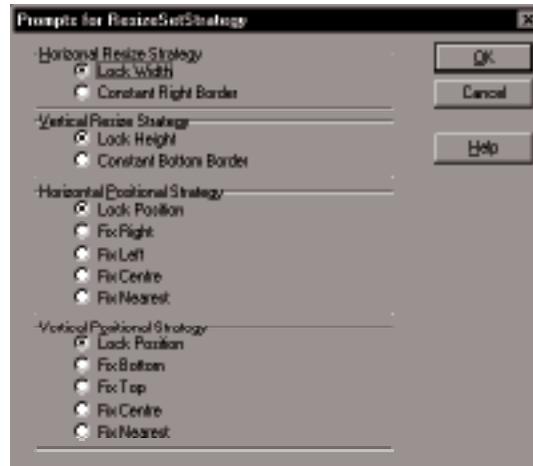
This template generates code similar to the following:

```
ST:StateCode = CUST:State           ! Move value for lookup
Access:State.Fetch(ST:ByCode)      ! Get value from file
```

## ResizeSetStrategy

---

The `ResizeSetStrategy` template lets you override the default resize strategy for a particular control. It is designed exclusively for the **Set resize strategy** embed point for a specific control. See *Extension Templates—WindowResize* for more information on the default resize strategies.



Insert the code template at the **Set resize strategy** embed point for the control for which to set the resize strategy, then complete the following prompts.

**Horizontal Resize Strategy**

Specify how the control's width is determined when the end user resizes the window. Choose from:

**Lock Width**      The control's design time width does not change.

**Constant Right Border**

Locks right edge, moves left.

**Vertical Resize Strategy**

Specify how the control's height is determined when the end user resizes the window. Choose from:

**Lock Height** The control's design time height does not change.

**Constant Bottom Border**

Locks bottom edge, moves top.

**Horizontal Positional Strategy**

Specify how the control's horizontal position is determined when the end user resizes the window. Choose from:

**Lock Position** The control's left edge maintains a fixed distance (the design time distance) from parent's left edge.

**Fix Right** The control's right edge maintains a proportional distance from parent's right edge.

**Fix Left** The control's left edge maintains a proportional distance from parent's left edge.

**Fix Center** The control's center maintains a proportional distance from parent's center.

**Fix Nearest** Applies Fix Right or Fix Left, whichever is appropriate.

**Vertical Positional Strategy**

Specify how the control's vertical position is determined when the end user resizes the window. Choose from:

**Lock Position** The control's top edge maintains a fixed distance (the design time distance) from parent's top edge.

**Fix Bottom** The control's bottom edge maintains a proportional distance from parent's bottom edge.

**Fix Top** The control's top edge maintains a proportional distance from parent's top edge.

**Fix Center** The control's center maintains a proportional distance from parent's center.

**Fix Nearest** Applies Fix Top or Fix Bottom, whichever is appropriate.

## SelectToolbarTarget

---

The `SelectToolbarTarget` template provides an easy way for developers to control which `BrowseBox` in a given procedure is tied to the toolbar navigation buttons (see *FrameBrowseControl* in the *Control Templates* chapter and *SetTarget* in the *Toolbar Classes* chapter).



### Toolbar Navigation Target

Select the `Browsebox` that is controlled by the `FrameBrowseControl` navigation buttons.

## SetABCProperty

The SetABCProperty template generates code to set a public property of an ABC Library object. See *Part II—ABC Library* for more information on these properties. This template generates code similar to the following:

```
BRW2.ActiveInvisible = True
```



### Object Name

Select the label of the object from the list. The list contains all ABC compliant objects in scope for this procedure.

### Property to Set

Select the property to set from the drop-down list. See *Part II—ABC Library* for more information on these properties.

### Value to Set

Type a variable, constant, or valid Clarion expression to assign to the property.

### Assign as Reference?

Check this box to generate a reference assignment (*object.property &= value*). Clear the box to generate a simple assignment (*object.property = value*). See *Reference Assignments* in the *Language Reference* for more information.

## SetProperty

The SetProperty template provides an easy way to set a runtime property of any control on a window.

### Control

Select the field equate label for one of the window controls from the drop down list.

### Property

Select the runtime property to set from the drop down list.

### Value

The label of a variable, a constant, or an expression to assign to the selected runtime property.

This template generates code similar to the following:

```
?MyControl{PROP:Whatever} = value
```

## Extension Templates

Extension templates add functionality to procedures, but are not bound to a control or a single embed point. Each Extension template has one well-defined task. For example, the `DateTimeDisplay` template lets you display the date, time, or both on a WINDOW.

From a **Procedure Properties** dialog, add an Extension template by pressing the **Extensions** button.

**Tip:** Only Extension templates may be added and deleted using the **Extensions** button. Control templates may be modified here, but may *not* be added or deleted. Use the **Window Formatter** to add or delete Control templates.

The ABC Templates include the following Extension templates:

### AsciiViewInListBox

---

The `AsciiViewInListBox` template allows a LIST control to alternate its display between a selected file and some other data that you specify.

The `AsciiViewInListBox` template provides the same functionality and the same prompts as the `AsciiViewControl` template. See *Control Templates—AsciiViewControl* for more information. The `AsciiViewInListBox` template provides one additional prompt. Because it is an Extension template and does not place its own control, the `AsciiViewInListBox` template prompts you for the LIST control to use to display text:

#### General Tab

##### **List box field to use**

Select the LIST control that alternates its display.

##### **Initialize Viewer**

Determines when the procedure initializes the Viewer object. Initialization includes selecting the file to view, opening it, and reading it.

##### **On Open Window**

Initializes the Viewer when the window opens so that the Viewer's LIST is full upon initial display.

##### **On Field Selection**

Delays initializing the Viewer until the end user selects the Viewer's LIST control.

##### **Manually**

Does not initialize the Viewer. You must embed a call to the `Viewer#.Initialize` ROUTINE to initialize the Viewer.

**File to Browse**

Specifies the path and name of the file to view, or a variable containing the path and name of the file to view. The variable must be preceded by an exclamation point (!).

If no path is specified, the procedure looks for the file in the current directory.

If omitted (left blank), the Viewer object prompts the end user to select a file.

**Allow popup menu searching**

Check this box to provide a (RIGHT-CLICK) popup menu choice to search the file.

**Allow popup menu printing**

Check this box to provide a (RIGHT-CLICK) popup menu choice to print some or all of the records in the file.

**Classes Tab**

Use the Classes tab to override the global Ascii Viewer setting. See *Template Overview—Classes Tab Options—Global and Local*.

## **DateTimeDisplay**

---

The DateTimeDisplay template adds to the functionality of a procedure template, allowing you to display the time and/or date in the status bar, or a control.

The options which appear in the Date and Time Display dialog are divided into two group boxes—Date Display and Time Display:

**Display in Window**

Check the box or boxes to add the display to your window.

**Picture**

Choose a date and/or time display picture from the drop-down list. The list displays examples, such as “October 31, 1959,” and “5:30P.M.”

**Other Picture**

Type in a picture of your choice, if the picture type you wish does not appear in the list. See also: *Date Picture Tokens* or *Time Picture Tokens* in the *Language Reference*.

**Day of Week (Date only)**

Optionally displays the day of week.

**Location**

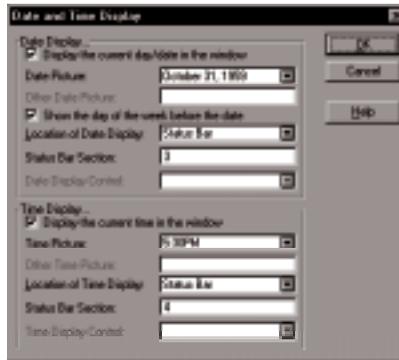
Choose between displaying the date and/or time on the status bar, or in a control.

**Status Bar Section**

When the Date or Time should appear on the status bar, specify the status bar section number.

**Display Control**

When the Date or Time should appear in a control, choose the control from a drop-down list of field equate labels for the window.



## ExtendProgressWindow

---

The ExtendProgressWindow template adds functionality to Process and Report procedures. It is designed to do two things:

- Give you precise control over the visual feedback you provide end users for (small) Process and Report procedures.
- Allow Process and Report procedures to operate in two separate modes—all records mode and single record mode (current value range-limit).

You can use the ExtendProgressWindow template to delay or to completely suppress the progress window for a Process or Report procedure, and you can optionally specify a wait cursor. In single record mode, you can suppress the progress window, the print preview, or both.

The ExtendProgressWindow template provides the following options.

**Delay Showing Window**

Enter the number of seconds to hide the progress window. For example, you may want to hide the progress window for 3 seconds so that processes or reports that finish within 3 seconds limit never show a progress window.

**Wait cursor**

Check this box to display a wait cursor (hour glass cursor) for the duration of the process or report. For small/short processes and reports, your end users may prefer a simple wait cursor over

a progress window. On completion, the procedure restores the cursor to its previous state.

### Single Shot

These options are available only for Processes and Reports that specify a key in the **File Schematic** dialog.

**Single record** Check this box to allow the Report or Process to operate in its normal mode (process all records), or to operate in single record mode (current value range-limit) when GlobalRequest is set to ProcessRecord (see *Procedure Templates—Inter-Procedure Communication* for more information on GlobalRequest).

**Tip:** If your Report or Process procedure uses a non-unique key, you can process all records with the current key value!

The BrowsePrintButton template primes the range-limit field and calls procedures in this single record mode (see *Control Templates—BrowsePrintButton*).

**Use Progress** Check this box to display the progress window in single record mode. Clear the box to suppress the progress window in single record mode.

**Use Preview** Check this box to provide the print preview in single record mode. Clear the box to suppress the print preview in single record mode.

## FormVCRControls

---

The FormVCRControls template adds functionality to a Form procedure by enabling navigation and field history with the *FrameBrowseControl* VCR buttons. See *Control Templates—FrameBrowseControl* for more information on these buttons and their operation.

Essentially, the *FormVCRControls* Extension provides a “scrolling” Form. You can display, add, delete, or edit many records without returning to the calling Browse to select a new record. However, the keys and filters implemented in the calling Browse procedure do control the navigation of the Form. For example, you can only navigate to records that meet the Browse range limit and filter conditions, and when you navigate to the “next” or “previous” record, the Browse key determines the sequence in which the records appear.

For Form procedures generated by the Application Wizard, if the Form procedure also contains a *BrowseBox*, the *FrameBrowseControl* buttons control the Form when the “form” tab is selected, and they control the *BrowseBox* when the “browsebox” tab is selected. See also *Code Templates—SetToolbarTarget*.

## RecordValidation

---

The RecordValidation template adds functionality to a Procedure by enforcing data dictionary-defined control value validation. It also lets you specify controls to exclude from validation.

### **Validate when the control is Accepted**

Specifies that validity checking occurs when the control generates an EVENT:Accepted, which occurs when the end user completes or moves the focus from the field.

### **Validate during NonStop Select**

Specifies that validity checking occurs when any control value changes if the window is in AcceptAll (Non-Stop) mode and has focus.

### **Do Not Validate**

Opens the Do Not Validate dialog, which lets you select fields from a drop-down list. The fields you choose will be excluded from validity checks.

## ReportChildFiles

---

The ReportChildFiles template adds functionality to Process and Report procedures. This extension template provides a simpler, more efficient, more controllable alternative to setting a chain of related files in the File Schematic and having the Report or Process template produce a single multi-tiered VIEW.

The ReportChildFiles template lets you name only the primary file and any lookup files in your procedure’s File Schematic. The template generates code to read (and optionally print a separate DETAIL for) the related child-file records for each primary file record. We recommend the ReportChildFiles template for the typical invoice headers/invoice lines scenario.

### **Multi-tiered View**

Suppose you have an invoice header file and an invoice detail file. You want to print out a header and then a line for each detail. This is somewhat tricky to do with a single view and there are some limitations and inefficiencies with this approach. You must populate each header (parent) file field into a group HEADER and each detail (child) field into a DETAIL. The limitation

is there are no events and no embed points to use when the parent record prints (because it is simply a group break). The inefficiency is that additional GETs are done on parent file lookups for every child record even though the parent record is unchanged. Plus, for SQL you must use a left outer join (inefficient) to force parent headers to print when there are no associated detail lines.

## **ReportChildFiles**

With the ReportChildFiles template you can simply populate the header (parent) as the primary file with its own DETAIL, then populate a second DETAIL for the detail (child) file. The primary view is then read record-by-record (lookups done only once for each parent record) and the child view is range-limited on the parent file linking fields. The Process Manager Method TakeRecord embed point provides an access point for *both* parent and child records. ProcessClass.TakeRecord is called for each record (parent or child), and ProcessClass.ChildLevel indicates which file/record is active. See *ProcessClass* for more information.

## **Using the ReportChildFiles Template**

The ReportChildFiles template provides the following options.

### **Parent File**

Type the label of the parent file, or press the ellipsis button (...) to select the parent file from the **File Schematic** dialog.

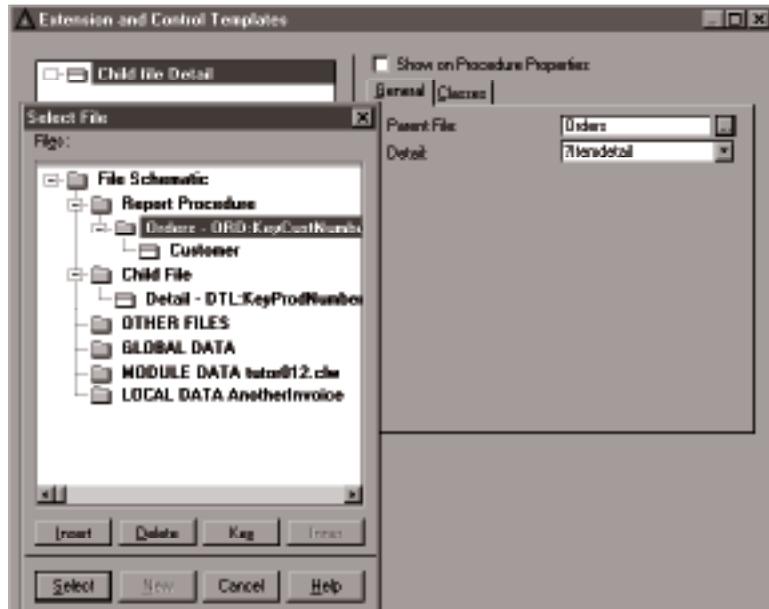
### **Detail**

For Report procedures, select the USE attribute (field equate label) of the REPORT DETAIL structure to print for each child record.

**Tip:** The Detail drop-down list shows DETAIL structures with USE attributes, so populate the DETAIL first, and add a USE attribute.

### **File Schematic <To Do>**

Insert the *child* file to process for each parent file record.



### Classes Tab

Use the Classes tab to override the global ViewManager setting. See *Template Overview—Classes Tab Options—Global and Local*.

## WindowResize

The WindowResize template lets the end user resize windows that have traditionally been fixed in size due to the controls they contain (List boxes, entry controls, buttons, etc.).

**Tip:** The WindowResize code repositions and resizes each control relative to its parent. This approach provides attractive, rational resizing of virtually any window, regardless of the controls it contains.

The template generates code to reposition the controls, resize the controls, or both, when the end user resizes the window.

**Tip:** To allow window resizing you must set the WINDOW's frame type to Resizable. See *Window Formatter—Window Properties Dialog* for more information on this setting.

### Resize Strategy

Specifies the method for resizing and repositioning the controls to fit within the new window size. Chose from:

<i>Resize</i>	Scales all window coordinates by the same amount, thus preserving the relative sizes and positions of all controls. That is, all controls, including buttons and entry fields get taller and wider as the window gets taller and wider. Window fonts are unchanged.
<i>Spread</i>	Maintains the design-time look and feel of the window by applying a strategy specific to each control type. For example, <code>BUTTON</code> sizes are not changed but their positions are tied to the nearest window edge. In contrast, <code>LIST</code> sizes <i>and</i> positions are scaled in proportion to the window.
<i>Surface</i>	Makes the most of the available pixels by positioning other controls to maximize the size of <code>LIST</code> , <code>SHEET</code> , <code>PANEL</code> , and <code>IMAGE</code> controls. We recommend this strategy for Wizard generated windows.

**Tip:** Even though list boxes may be resized, the column widths within the list box are not resized. However, the right-most column does expand or contract depending on the available space.

### Don't Alter Controls

Controls are not resized when the window is resized.

**Tip:** For this strategy, you may add the `SCROLL` attribute to each control plus the `HVSCROLL` attribute to the window to provide a 'moving window' over a larger page.

### Restrict Minimum Window Size

Check this box to specify a minimum window height and width. This lets you enforce a minimum reasonable size of the window based on the size and number of controls on the window. In other words, you can keep your end user from shrinking the window so much that its controls become invisible or unrecognizable.

**Minimum Width** Specify the minimum width of the window in dialog units. Dialog units are based on the window's font and are 1/4 of the average character width.

Zero sets the window minimum to the size at which the window opens (not necessarily the design time size). In other words, it takes into account any `.INI` setting plus any runtime Property syntax. Thus, we allow the developer to open the window, perform any dynamic control production (including resizing the window) before the minimum restriction takes effect.

**Minimum Height** Specify the minimum height of the window in dialog units. Dialog units are based on the window's font and are 1/8 of the character height.

Zero sets the window minimum to the size at which the window opens (not necessarily the design time size). In other words, it takes into account any .INI setting plus any runtime Property syntax. Thus, we allow the developer to open the window, perform any dynamic control production (including resizing the window) before the minimum restriction takes effect.

#### **Restrict Maximum Window Size**

Check this box to specify a maximum window height and width. This lets you enforce a maximum reasonable size of the window.

**Maximum Width** Specify the maximum width of the window in dialog units. Dialog units are based on the window's font and are 1/4 of the average character width.

Zero sets the window maximum to the size at which the window opens (not necessarily the design time size). In other words, it takes into account any .INI setting plus any runtime Property syntax. Thus, we allow the developer to open the window, perform any dynamic control production (including resizing the window) before the maximum restriction takes effect.

#### **Maximum Height**

Specify the maximum height of the window in dialog units. Dialog units are based on the window's font and are 1/8 of the character height.

Zero sets the window maximum to the size at which the window opens (not necessarily the design time size). In other words, it takes into account any .INI setting plus any runtime Property syntax. Thus, we allow the developer to open the window, perform any dynamic control production (including resizing the window) before the maximum restriction takes effect.

#### **Override Control Strategies**

Press this button to override the default resize strategy for individual controls. This opens the **Override Control Strategies** dialog.

## **Override Control Strategies**

The **Override Control Strategies** dialog lets you override the default resize strategy for individual controls. For example, by default, buttons are “fixed” to the nearest window borders and are not repositioned like most other controls. However, if you want your procedure to reposition the button like other controls, you may specify this here. See also *Window Resize Class—SetStrategy*.

Press the **Insert** button to select the control for which to set the resize strategy. Then choose from the following sizing and positioning options:

### **Horizontal Resize Strategy**

Specify how the control’s width is determined when the end user resizes the window. Choose from:

*Lock Width*      The control’s design time width does not change.

*Constant Right Border*  
Locks right edge, moves left.

### **Vertical Resize Strategy**

Specify how the control’s height is determined when the end user resizes the window. Choose from:

*Lock Height*      The control’s design time height does not change.

*Constant Bottom Border*  
Locks bottom edge, moves top.

### **Horizontal Positional Strategy**

Specify how the control’s horizontal position is determined when the end user resizes the window. Choose from:

*Lock Position*      The control’s left edge maintains a fixed distance (the design time distance) from parent’s left edge.

*Fix Right*      The control’s right edge maintains a proportional distance from parent’s right edge.

*Fix Left*      The control’s left edge maintains a proportional distance from parent’s left edge.

*Fix Center*      The control’s center maintains a proportional distance from parent’s center.

*Fix Nearest*      Applies Fix Right or Fix Left, whichever is appropriate.

### **Vertical Positional Strategy**

Specify how the control’s vertical position is determined when

the end user resizes the window. Choose from:

- Lock Position* The control's top edge maintains a fixed distance (the design time distance) from parent's top edge.
- Fix Bottom* The control's bottom edge maintains a proportional distance from parent's bottom edge.
- Fix Top* The control's top edge maintains a proportional distance from parent's top edge.
- Fix Center* The control's center maintains a proportional distance from parent's center.
- Fix Nearest* Applies Fix Top or Fix Bottom, whichever is appropriate.

## **Resizer Configuration Options**

### **Automatically find parent controls**

Check this box to set parent/child relationships among window controls. Clearing the box makes the WINDOW the parent of all its controls. Setting parent/child relationships lets any special scaling cascade from parent to child. See *WindowResizeClass Methods—SetParentDefaults* for more information.

### **Optimize Moves**

Check this box to move all controls at once during the resize operation, producing a snappier resize and avoiding bugs on some windows. See *WindowResizeClass Properties—DeferMoves* for more information.

### **Optimize Redraws**

Check this box to make controls transparent (TRN attribute) during the resize operation, producing a smoother redraw and avoiding bugs on some windows. See *WindowResizeClass Properties—AutoTransparent* for more information.

## **Classes Tab**

Use the Classes tab to override the global Resizer setting. See *Template Overview—Classes Tab Options—Global and Local*.

# PART II

---

## APPLICATION BUILDER CLASS LIBRARY

THE ABCS OF RAPID APPLICATION DEVELOPMENT



# 6 - ABC LIBRARY OVERVIEW

## About This Part

This part of the *Application Handbook* 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.

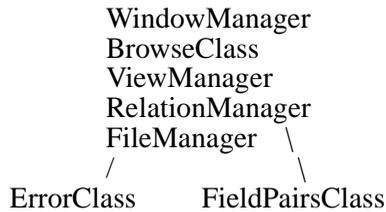
## Core Classes

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>
ABPOPUP.INC	
PopupClass	MODULE('ABPOPUP.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.

# 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)

---

<b>Property</b>	<b>Datatype [, PROTECTED ]</b>
	A complete description of the <b>Property</b> and its uses.
	<b>Datatype</b> 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.
	<pre>ComplexDataType STRUCTURE          !actual structure declaration                            END</pre>
See Also:	Related Methods and Properties

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

---

	<b>Method(   parameter1   [, parameter2 ] ) [, PROTECTED ] [, VIRTUAL ] [, PROC ]</b>   alternate     parameters
	<b>Method</b> A brief statement of <i>what</i> the method does.
<i>parameter1</i>	A complete description of parameter1, along with how it relates to parameter2 and the Method.
<i>parameter2</i>	A complete description of parameter2, along with how it relates to parameter1 and the Method. Brackets [ ] indicate optional parameters.
	A concise description of <i>what</i> the <b>Method</b> does.
Implementation:	A description of <i>how</i> 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:	<pre>FieldOne = FieldTwo + FieldThree          !This is a source code example FieldThree = Method(FieldOne,FieldTwo)    !Comments follow the "!" character</pre>
See Also:	Related Methods and Properties

## PROTECTED, VIRTUAL, 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.

### **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*.



# 7 - ASCIIFILECLASS

## Overview

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.

## 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.

## 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

## 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
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
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

The ASCIIFileClass contains the following 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	USHORT
----------	--------

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 40h (ReadOnly/DenyNone).

The Reset 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, Reset

## ***AsciiFileClass Methods***

The ASCIIFileClass contains the following methods.

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the ASCIIFileClass methods.

#### **Primary Interface Methods**

The primary interface 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 Primary Interface 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)
CODE
line = line[1:5]' '&line[5:55]

```

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)
RVa1        BYTE(True)
SavePath   CSTRING(FILE:MaxFilePath+1),AUTO
CODE
CLOSE(SELF.AsciiFile)
SavePath=PATH()
LOOP
  IF ~FName AND ~SELF.GetDOSFilename(FName)
    RVa1=False
    BREAK
  END
  OPEN(SELF.AsciiFile,ReadOnly+DenyNone)
  IF ERRORCODE()
    MESSAGE('Can't open ' & FName)
    RVa1=False
  ELSE
    BREAK
  END
END
END
IF RVa1
  SELF.FileSize=BYTES(SELF.AsciiFile)
END
SETPATH(SavePath)
RETURN RVa1
```

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')                                !declare ASCIIViewerClass
Viewer      AsciiViewerClass,THREAD                  !declare Viewer object
Filename    STRING(255),THREAD                       !declare filename variable
AsciiFile   FILE,DRIVER('ASCII'),NAME(Filename),PRE(A1),THREAD
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
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
  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)

---

### 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
PctPos=MyASCIIFile.GetPercentile(MyASCIIFile.TopLine+CHOICE(?ASCIIBox)-1)
?ASCIIBox{PROP:VScrollPos}=PctPos
```

!current line is what % thru the file?  
!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)   |
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

```

See Also: **Kill**



## Reset (reset the ASCIIFileClass object)

---

### Reset( *filename* )

<b>Reset</b>	Resets the ASCIIFileClass object.
<i>filename</i>	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**



# 8 - 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.

## 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

## 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

## 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()
ReturnValu     BYTE,AUTO
CODE
ReturnValu = PARENT.Init()
IF ReturnValu THEN RETURN ReturnValu.
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 ReturnValu

ThisWindow.TakeAccepted PROCEDURE()
ReturnValu     BYTE,AUTO
CODE
ReturnValu = 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 ReturnValu
```

## AsciiPrintClass Properties

The AsciiPrintClass contains the following properties.

### FileMgr (AsciiFileClass object)

---

#### 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)

---

#### 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

The AsciiPrintClass contains the following properties.

### 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

<b>Init</b>	Initializes the ASCIIPrintClass object.
<i>ASCIIFileMgr</i>	The label of the ASCIIFileClass object that manages the file to print. The ASCIIPrintClass object uses the <i>ASCIIFileMgr</i> 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

<b>PrintLines</b>	Prints or previews the specified lines.
<i>first</i>	An integer constant, variable, EQUATE, or expression containing the number of the first line of the range of lines to print.
<i>last</i>	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**

# 9 - 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”).

## 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

## 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

## 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

The AsciiSearchClass contains the following 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 datatype 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)

---

#### FileMgr &AsciiFileClass, PROTECTED

The **FileMgr** property is a reference to the AsciiFileClass object that 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)

---

### 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

The AsciiSearchClass contains the following 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                             !call Searcher.Ask method
      StartSearch=CHOOSE(CHOICE(?AsciiBox)>0, | ! passing the currently
        Viewer.TopLine+CHOICE(?AsciiBox)-1,1) ! selected line as the
      Viewer.Searcher.Ask(StartSearch)         ! search's starting point
    END
  END
END
END
END

```

## Init (initialize the ASCIIsearchClass object)

### Init( ASCIIFileMgr ), VIRTUAL

<b>Init</b>	Initializes the ASCIIsearchClass object.
<i>ASCIIFileMgr</i>	The label of the ASCIIFileClass object that manages the file to search. The ASCIIsearchClass object uses the <i>ASCIIFileMgr</i> 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

# 10 - 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.

## 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*.

## 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 Viewer# where # 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

## Conceptual Example

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

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

```

PROGRAM
MAP
END

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

ViewWindow WINDOW('View a text 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
StartSearch LONG                                       !hold selected line number

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

```

## AsciiViewerClass Properties

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

In addition to the inherited properties, the AsciiViewerClass contains the properties listed below.

### 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.

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.

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

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

In addition to (or instead of) the inherited methods, the AsciiViewerClass contains the methods listed below.

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the ASCIIViewerClass methods.

#### Primary Interface Methods

The primary interface 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 <sup>1</sup>	return the filename
GetLastLineNo <sup>1</sup>	return last line number
GetLine <sup>1</sup>	return line of text
GetPercentile <sup>1</sup>	convert file position to percentage
Reset	reset the ASCIIViewerClass object
SetPercentile <sup>1</sup>	convert percentage to file position
SetLine <sup>v</sup>	position to specific line
SetLineRelative	move N lines

<sup>1</sup> These methods are inherited from the ASCIIFileClass.

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

## **Virtual Methods**

Typically you will not call these methods directly—the Primary Interface 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 <sup>1</sup>	format text
SetLine	position to specific line
ValidateLine <sup>1</sup>	implement a filter

<sup>1</sup> 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
    . . .
```

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 (EnablePrint), searching (EnableSearch), 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

ViewerActive=Viewer.Init( AsciiFile,                 !Initialize Viewer with:
                           A1:line,                 ! file label,
                           Filename,                 ! file field to display
                           ?AsciiBox,               ! variable file NAME attribute
                           GlobalErrors,            ! LIST control number
                           EnableSearch+EnablePrint) ! ErrorClass object
! features to implement flag
IF ~ViewerActive THEN RETURN.                       !if init unsuccessful, don't
! call other Viewer methods
!If init succeeded, proceed

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

```

See Also:

**Kill**

## Kill (shut down the ASCIIViewerClass object)

### Kill

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

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

ViewerActive=Viewer.Init( AsciiFile,                 ! Initialize Viewer with:
                           A1:line,                  ! file label,
                           Filename,                 ! file field to display
                           ?AsciiBox,                ! variable file NAME attribute
                           GlobalErrors,             ! LIST control number
                           EnableSearch+EnablePrint) ! ErrorClass object
! features to implement flag
IF ~ViewerActive THEN RETURN.                       ! if init unsuccessful, don't
! call other Viewer methods
! If init succeeded, proceed

ACCEPT
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
```

## 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
```

## Reset (reset the ASCIIViewerClass object)

### Reset( *filename* )

<b>Reset</b>	Resets the ASCIIViewerClass object.
<i>filename</i>	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

**SetLine** Positions the ASCIIViewerClass object to a specific line.  
*line number* 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
```

## SetTranslator (set run-time translator)

### 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,
                           Al: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
Viewer.SetTranslator(Translator)   !enable text translation
!program code
```

## TakeEvent (process ACCEPT loop event)

---

### 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
END
```

# 11 - 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.

## 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*.

## 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

## 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
ReturnValu     BYTE,AUTO
CODE
ReturnValu = PARENT.Init()                      !call base class init
IF ReturnValu THEN RETURN ReturnValu.
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
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
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 ReturnValu

ThisWindow.Kill PROCEDURE()                    !shut down things
ReturnValu     BYTE,AUTO
CODE
ReturnValu = PARENT.Kill()                     !call base class shut down
IF ReturnValu THEN RETURN ReturnValu.
Relate:State.Close                             !close State and related files
Relate:State.Kill                             !shut down Relate:State object
GlobalErrors.Kill                             !shut down GlobalErrors object
RETURN ReturnValu

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**

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
BRW1.ArrowAction = EIPAction:Remain+EIPAction:RetainColumn !default save, keep editing

```

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 (change/edit button)

---

### 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 Init method does not initialize the ChangeControl property. You should initialize the ChangeControl property after the Init method is called. See the *Conceptual Example*. On EVENT:Accepted for the ChangeControl, the TakeEvent method calls the Ask method to do the edit/change.

**Tip:** The ABC BrowseUpdateButton template generates code to update a browse item.

See Also: UpdateToolBarButtons, Ask

## DeleteControl (delete button)

---

### 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 Init method does not initialize the DeleteControl property. You should initialize the DeleteControl property after the Init method is called. See the *Conceptual Example*. On EVENT:Accepted for the DeleteControl, the TakeEvent method calls the Ask method to do the delete.

**Tip:** The ABC BrowseUpdateButton template generates code to delete a browse item.

See Also: UpdateToolBarButtons, Ask

## 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**

## 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**

## 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 QUEUE buffers (and any other data areas managed by the BrowseClass object, such as local or global memory variables).

The AddField method adds field pairs to the Fields property.

Implementation:

The Init method instantiates the FieldPairsClass object. The AskRecord, SetQueueRecord, and UpdateBuffer methods use the Fields property to move and compare data between the field pairs.

See Also:

AddField

## 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

## InsertControl (add/insert button)

---

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 Init method does not initialize the InsertControl property. You should initialize the InsertControl property after the Init method is called. See the *Conceptual Example*. On EVENT:Accepted for the InsertControl, the TakeEvent method calls the Ask method to do the insert.

**Tip:** The ABC BrowseUpdateButton template generates code to insert a browse item.

See Also: UpdateToolBarButtons, Ask

## ListControl (browse LIST control)

---

### ListControl      SIGNED

The **ListControl** property contains the control number of the LIST control that displays the browse data.

The Init method initializes the ListControl property. See the *Conceptual Example*.

See Also:            Init

## ListQueue (browse data queue)

---

### ListQueue      &QUEUE

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

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

See Also:            Init

## Loaded (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 (popup menu manager)

---

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

See Also:            Init

## PrintControl (print button)

---

### 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 Init method does not initialize the PrintControl property. You should initialize the PrintControl property after the Init method is called. See the *Conceptual Example*. On EVENT:Accepted for the PrintControl, the TakeEvent method passes the PrintProcedure value to the WindowManager.Run method to indicate which procedure to execute.

**Tip:** The ABC BrowsePrintButton template generates code to declare and support a Print button.

See Also:

PrintProcedure

## PrintProcedure (print procedure)

---

### PrintProcedure USHORT

The **PrintProcedure** property identifies the procedure to print a browse item. The procedure is registered by number with the BrowseClass object's WindowManager.

Implementation:

Typically, the WindowManager object (Init method) sets the value of the PrintProcedure property. On EVENT:Accepted of the PrintControl, the TakeEvent method passes the PrintProcedure value to the WindowManager.Run method to indicate which procedure to execute.

**Tip:** The ABC BrowsePrintButton and ExtendProgressWindow templates generate code to print a single browse item. The generated code uses the ProcessRecord EQUATE as a switch to indicate whether to process all records or only one record.

See Also:

PrintControl, Window, WindowManager.Run

## Query (ad hoc query manager)

---

### Query &QueryClass

The **Query** property is a reference to the QueryClass class used by this BrowseClass object.

See Also:

Init

## QueryControl (query button)

---

### QueryControl SIGNED

The **QueryControl** property contains the number of the browse's query control. This is typically the value of the Query 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 Query behavior to corresponding popup menu choices, etc.

The QueryShared property determines whether the query applies to the active sort order or to all sort orders.

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.

**Tip:** The ABC BrowseQueryButton template generates code to declare and support a Query button.

See Also:

Query, QueryShared, TakeLocate

## QueryShared (query scope flag)

---

### QueryShared BYTE

The **QueryShared** property determines whether an ad hoc query applies to the active sort order or to all sort orders. A value of zero (0 or False) applies the query to the active sort order; a value of one (1 or True) applies the query to all sort orders.

Implementation:

The Init method does not initialize the QueryShared property, so the default setting is zero—active sort order only. 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 TakeLocate method implements the behavior specified by the QueryShared property.

See Also:

Query, QueryControl, TakeLocate

## 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:

The Fetch method implements the faster reads only during the page-loading process, and only if the QuickScan property is not zero. The 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 (select button)

---

### 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 Init method does not initialize the SelectControl property. You should initialize the SelectControl property after the Init method is called. See the *Conceptual Example*. On EVENT:Accepted for the SelectControl, the TakeEvent method initiates the item selection.

**Tip:** The ABC BrowseSelectButton template generates code to select a browse item.

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

## 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 (toolbox button)

---

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 `Init` method does not initialize the `ToolControl` property. You should initialize the `ToolControl` property after the `Init` method is called. See the *Conceptual Example*. On `EVENT:Accepted` for the `ToolControl`, the `TakeEvent` method calls the `PopupClass.Toolbox` method to display a floating toolbox to collect and apply the end user's selection (insert, change, delete, scroll, select, etc.).

**Tip:** The ABC `BrowseToolButton` template generates code to declare and support a Toolbox button.

See Also:

`Popup`, `PopupClass.Toolbox`

## 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

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

In addition to (or instead of) the inherited methods, the BrowseClass contains the methods listed below.

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the BrowseClass methods.

#### Primary Interface Methods

The primary interface 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
SetAlerts <sup>v</sup>	alert keys for list, locator, and edit controls
UpdateQuery	define default query interface
Kill <sup>v</sup>	shut down the BrowseClass object

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

Next <sup>v</sup>	get the next view record in sequence
Previous <sup>v</sup>	get the previous view record in sequence
Ask	update the selected item
TakeEvent <sup>v</sup>	process the current ACCEPT loop event
TakeNewSelection <sup>v</sup>	process a new browse list item selection

<sup>v</sup> 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 <sup>v</sup>	apply pending scroll, locator, range, etc.

### **Virtual Methods**

Typically you will not call these methods directly—the Primary Interface 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 <sup>l</sup>	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
TakeLocate	collect and apply ad hoc query
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* )

**AddField** Identifies the corresponding FILE and QUEUE fields for a browse list column.

*filefield* The fully qualified label of the FILE field or memory variable. The *filefield* is the original source of the browse LIST's data.

*queuefield* The fully qualified label of the corresponding QUEUE field. The *queuefield* is loaded from the *filefield*, 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)
.
.
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

```

## 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

<b>AddSortOrder</b>	Specifies an additional sort order for the browse list.
<i>thumbstep</i>	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 <i>Control Templates—BrowseBox</i> for more information on thumb behavior.
<i>key</i>	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 AskRecord method supports auto-increment keys, range-limits, and enforces referential integrity constraints for changes. The AskRecord does field validation on a entire record basis, and only updates the BrowseClass primary file.

The AddEditControl method specifies a custom EditClass for a specific browse column.

#### Implementation:

The Ask method calls the AskRecord method to edit the selected row and column. The AskRecord method uses an EditClass object to update the selected row and column. It assigns the EditClass input control to a specific LIST cell (row and column)—See *PROP:Edit* in the *Language Reference* for more information.

The AskRecord method assumes the UpdateViewRecord method has been called to ensure correct record buffer contents. AskRecord should be followed by the ResetFromAsk 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](#), [EditClass](#)

## 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
ELIF 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



## 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 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()
OF Level:Benign
    !get next record
OF Level:Fatal
    !if successful, continue
    !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, ResetThumbLimits

## 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()
OF Level:Benign
OF Level:Fatal
RETURN
OF Level:Notify
MESSAGE('Reached end of file.')
END
!get previous record
!if successful, continue
!if fatal error
! end this procedure
!if end of file reached
! acknowledge EOF

```

See Also: Fetch, ResetThumbLimits

## 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

```

## ResetFromAsk (reset browse after update)

**ResetFromAsk**( *request*, *response* ), VIRTUAL, PROTECTED

<b>ResetFromAsk</b>	Resets the BrowseClass object following an update.
<i>request</i>	An integer constant, variable, EQUATE, or expression that indicates the type of update requested. Valid updates are insert, change, and delete.
<i>response</i>	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

<b>ResetQueue</b>	Fills or refills the browse queue.
<i>resetmode</i>	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                                          !otherwise
  RETURN SELF.SetSort(2,Force)               !apply second sort order
END
```

**See Also:** AddRange, AddResetField, AddSortOrder, SetSort, UpdateWindow

## ScrollEnd (scroll to first or last item)

### ScrollEnd( *scrollevent* ), VIRTUAL, PROTECTED

**ScrollEnd** Scrolls to the first or last browse list item.

*scrollevent* 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

**ScrollOne**                      Scrolls up or down one browse list item.

*scrollevent*                      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`

## SetQueueRecord (copy data from file buffer to queue buffer)

### 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

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)

---

### 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
RVa1 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 RVa1
```

See Also: TakeKey, TakeNewSelection, TakeScroll, WindowManager.TakeEvent

## TakeKey (process an alerted keystroke)

### 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
END

```

See Also: TakeEvent

## TakeLocate (collect and apply ad hoc query)

### TakeLocate, VIRTUAL

The **TakeLocate** method collects and applies ad hoc queries for the BrowseClass object.

Implementation: The TakeEvent method calls the TakeLocate method to collect ad hoc query criteria from the end user and apply it so the browse displays a new result set that matches the query criteria.

The TakeLocate method uses the Query property to solicit the query criteria from the end user and to build the filter expression to apply. The TakeLocate method implements the behavior specified by the QueryShared property.

Example:

```

MyBrowseClass.TakeLocate
CODE
CASE MESSAGE('Warning: new query resets browse totals! Continue?',,, '&Yes|&No')
OF 2                                !user pressed No button
  RETURN                             !don't do a new query
END
PARENT.TakeLocate                   !do a new query

```

See Also: Query, QueryShared, TakeEvent

## TakeNewSelection (process a new selection)

---

### 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
```

## TakeScroll (process a scroll event)

### TakeScroll( [*scrollevent*] ), VIRTUAL

<b>TakeScroll</b>	Processes a scroll event for the browse list.
<i>scrollevent</i>	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
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
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* )

**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.

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 Next and 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
```

## 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 tho 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
  IF EVENT) = EVENT:NewSelection
    IF MyBrowse.TakeNewSelection()
      MyBrowse.SetSort(0,1)
      MyBrowse.UpdateBuffer
      MyBrowse.UpdateWindow
      DISPLAY()
    . . .
    !focus on browse list
    !if new selection
    !BrowseClass object handles it
    !reapply sort order
    !refresh file buffer from selected item
    !update display variables (locator)
    !and redraw the window
```

# 12 - 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.

## 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.

## ABC Template Implementation

---

Various ABC Library objects instantiate BufferedPairsClass objects as needed; therefore, the template generated code does not directly reference the BufferedPairsClass.



## 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

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 the methods listed below.

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the BufferedPairsClass methods.

#### Primary Interface Methods

The primary interface 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
AddPair <sup>v</sup>	add a field pair to the List property
Kill	shut down the BufferedPairsClass object

<sup>v</sup> 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 to the List property

## AddPair (add a field pair)

### AddPair( *left*, *right* ), VIRTUAL

<b>AddPair</b>	Adds a field pair to the RealList 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 RealList 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 RealList 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
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

# 13 - 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).

## **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.

## **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

## 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)
        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

        Const.Init(Term:BYTE)                !process items one-at-a-time:
                                           !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 at a time
          !do something with Text and Repl    ! to AddItem variables
        END
        Const.Kill                            !shut down Const object

        Const.Init(Term:BYTE)                !process all items at a time:
                                           !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

```

## ConstantClass Properties

The ConstantClass contains the following property:

### TerminatorValue (end of data marker)

---

TerminatorValue	CSTRING(33)
-----------------	-------------

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 first 32 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.

See Also:

Init, Next

## ***ConstantClass Methods***

The ConstantClass contains the following methods:

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the ConstantClass methods.

#### **Primary Interface Methods**

The primary interface 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* )

<b>AddItem</b>	Sets the (repeating) constant datatype and its corresponding target variable.
<i>datatype</i>	An integer constant, variable, EQUATE or expression that identifies the datatype of a repeating constant within the constant GROUP structure. Valid <i>datatype</i> values are ConstType:Cstring, ConstType:Pstring, ConstType:Byte, ConstType:Short, ConstType:Ushort, and ConstType:Long.
<i>target</i>	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
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:

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
```

## Next (load all constant items to file or queue)

```
Next( | file | )
      | queue | )
```

<b>Next</b>	Loads all the constant items to a file or queue.
<i>file</i>	The label of the FILE to which to ADD each constant item.
<i>queue</i>	The label of the QUEUE to which to ADD each constant item.

The **Next** method processes all of the constant items and executes an `ADD(file)` or `ADD(queue)` for each item.

Prior calls to the `AddItem` method determine the makeup of the item as well as the target variables that receive the item. The target variables should be within the *file* or *queue* structure to make the corresponding `ADD` meaningful.

The `Init` method determines what constitutes the end of the constant data.

Implementation:

The `Next(FILE)` and `Next(QUEUE)` methods call the `Next()` method for each constant item, then execute an `ADD(file)` or `ADD(queue)` for each item.

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
Text     CSTRING(255),AUTO                    !a variable to receive a constant value
Repl     CSTRING(255),AUTO                    !a variable to receive a constant value
CODE
!process all items at a time
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
```

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)

Const.AddItem(ConstType:PString, Text)
Const.AddItem(ConstType:PString, Repl)
Const.Set(Spanish)
LOOP WHILE Const.Next()=Level:Benign
!do something with Text and Repl
END
Const.Kill                                  !shut down Const object

```

**See Also:** **AddItem, Init**

## 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)

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**

# 14 - 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.

## 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.

## 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	EditCheckClass declarations
ABEIP.CLW	EditCheckClass method definitions

## 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
  _ABCD11Mode_ 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:

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the EditCheckClass methods.

#### Primary Interface Methods

The primary interface 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>VI</sup>	initialize the EditCheckClass object
Kill <sup>VI</sup>	shut down the EditCheckClass object

##### Mainstream Use:

TakeEvent <sup>VI</sup>	handle events for the CHECK control
-------------------------	-------------------------------------

##### Occasional Use:

CreateControl <sup>V</sup>	create the CHECK control
SetAlerts <sup>VI</sup>	alert keystrokes for the CHECK 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 Primary Interface 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 EditCheckClass object
CreateControl	create the CHECK control
SetAlerts <sup>I</sup>	alert keystrokes for the CHECK control
TakeEvent <sup>I</sup>	handle events for the CHECK control
Kill <sup>I</sup>	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



# 15 - 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 `GroupBox`—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

## 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
  _ABCD11Mode_ 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
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:ControlType-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)**

---

<b>FEQ</b>	<b>UNSIGNED</b>
------------	-----------------

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

## ***EditClass Methods***

The EditClass contains the following 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the EditClass methods.

#### **Primary Interface Methods**

The primary interface 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>∇</sup>	initialize the EditClass object
Kill <sup>∇</sup>	shut down the EditClass object

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

TakeEvent <sup>∇</sup>	handle events for the edit control
------------------------	------------------------------------

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

CreateContol <sup>∇</sup>	a virtual to create the edit control
SetAlerts <sup>∇</sup>	alert appropriate keystrokes for the edit control

<sup>∇</sup> These methods are also virtual.

#### **Virtual Methods**

Typically you will not call these methods directly—the Primary Interface 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
TakeEvent <sup>∇</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**

## 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**



# 16 - EDITCOLORCLASS

## Overview

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.

## 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.

## 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

## 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

_ABCD11Mode_ 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***

The EditColorClass 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 EditColorClass contains the following properties:

### **Title (color dialog title text)**

---

<b>Title</b>	<b>CSTRING(256)</b>
--------------	---------------------

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

The EditColorClass 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 EditColorClass contains the following methods:

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the EditColorClass methods.

#### Primary Interface Methods

The primary interface 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>VI</sup>	initialize the EditColorClass object
Kill <sup>VI</sup>	shut down the EditColorClass object

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

TakeEvent <sup>V</sup>	handle events for the edit control
------------------------	------------------------------------

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

CreateControl <sup>V</sup>	create the edit (COMBO) control
SetAlerts <sup>VI</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 Primary Interface 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 EditColorClass object
CreateControl	create the edit (COMBO) control
SetAlerts <sup>I</sup>	alert keystrokes for the edit control
TakeEvent	handle events for the edit control
Kill <sup>I</sup>	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)

### TakeEvent( *event* ), VIRTUAL

**TakeEvent** Processes an event for the EditColorClass 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 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**

# 17 - 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.

## 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.

## 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	EditDropListClass declarations
ABEIP.CLV	EditDropListClass method definitions

## 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
  _ABCD11Mode_ 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
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

Edit:PR:ControlType CLASS(EditDropListClass) !declare Edit:PR:ControlType-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
VCCRRequest          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.VCCRRequest &= VCCRRequest
    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:ControlType,3) !Use Edit:PR:ControlType 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: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***

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

The EditDropListClass 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 EditDropListClass contains the following methods:

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the EditDropListClass methods.

#### Primary Interface Methods

The primary interface 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>VI</sup>	initialize the EditDropListClass object
Kill <sup>VI</sup>	shut down the EditDropListClass object

##### Mainstream Use:

TakeEvent <sup>VI</sup>	handle events for the LIST control
-------------------------	------------------------------------

##### Occasional Use:

CreateContol <sup>V</sup>	create the LIST control
SetAlerts <sup>V</sup>	alert keystrokes for the LIST 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 Primary Interface 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 EditDropListClass object
CreateContol	create the LIST control
SetAlerts	alert keystrokes for the LIST control
TakeEvent <sup>I</sup>	handle events for the LIST control
Kill <sup>I</sup>	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)

### SetAlerts, VIRTUAL

The **SetAlerts** method 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**

# 18 - 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.

## 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).

## 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.CW	EditEntryClass method definitions

## 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
  _ABCD11Mode_ 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)              !edit with 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***

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**

The `EditEntryClass` 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 `EditEntryClass` contains the following methods:

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the `EditEntryClass` methods.

#### **Primary Interface Methods**

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

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

<code>Init</code> <sup>VI</sup>	initialize the <code>EditEntryClass</code> object
<code>Kill</code> <sup>VI</sup>	shut down the <code>EditEntryClass</code> object

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

<code>TakeEvent</code> <sup>VI</sup>	handle events for the ENTRY control
--------------------------------------	-------------------------------------

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

<code>CreateControl</code> <sup>V</sup>	create the ENTRY control
<code>SetAlerts</code> <sup>VI</sup>	alert keystrokes for the ENTRY 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 Primary Interface 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.

<code>Init</code> <sup>I</sup>	initialize the <code>EditEntryClass</code> object
<code>CreateControl</code>	create the ENTRY control
<code>SetAlerts</code> <sup>I</sup>	alert keystrokes for the ENTRY control
<code>TakeEvent</code> <sup>I</sup>	handle events for the ENTRY control
<code>Kill</code> <sup>I</sup>	shut down the <code>EditEntryClass</code> 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



# 19 - 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.

## 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.

## 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

## 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
  _ABCD11Mode_ 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
VCCRRequest     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

The EditFileClass 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 EditFileClass contains the following properties:

### FileMask (file dialog behavior)

FileMask	BYTE
	The <b>FileMask</b> 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 <i>flag</i> parameter to the FILEDIALOG procedure. See <i>FILEDIALOG</i> in the <i>Language Reference</i> for more information.
See Also:	TakeEvent

### FilePattern (file dialog filter)

FilePattern	CSTRING(1024)
	The <b>FilePattern</b> property contains a text string that defines both the file masks and the file mask descriptions that appear in the file dialog's <b>List Files of Type</b> 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 <code>description masks description masks</code> . All elements in the string must be delimited by the vertical bar ( ). For example, <code>'all files *.* *.clw *.inc Clarion source *.clw;*.incl*.clw;*.inc'</code> defines two selections for the File dialog's <b>List Files of Type</b> drop-down list. See the <i>extensions</i> parameter to the FILEDIALOG function in the <i>Language Reference</i> 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**

The `EditFileClass` 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 `EditFileClass` contains the following methods:

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the `EditFileClass` methods.

#### **Primary Interface Methods**

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

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

<code>Init</code> <sup>VI</sup>	initialize the <code>EditFileClass</code> object
<code>Kill</code> <sup>VI</sup>	shut down the <code>EditFileClass</code> object

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

<code>TakeEvent</code> <sup>VI</sup>	handle events for the edit control
--------------------------------------	------------------------------------

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

<code>CreateControl</code> <sup>V</sup>	create the edit (COMBO) control
<code>SetAlerts</code> <sup>VI</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 Primary Interface 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.

<code>Init</code> <sup>I</sup>	initialize the <code>EditFileClass</code> object
<code>CreateControl</code>	create the edit (COMBO) control
<code>SetAlerts</code> <sup>I</sup>	alert keystrokes for the edit control
<code>TakeEvent</code> <sup>I</sup>	handle events for the edit control
<code>Kill</code> <sup>I</sup>	shut down the <code>EditFileClass</code> 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)

### TakeEvent( *event* ), VIRTUAL

**TakeEvent** Processes an event for the EditFileClass 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 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**



# 20 - 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.

## 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.

## 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

## 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

_AB CD11Mode_ EQUATE(0)
_AB CLinkMode_ 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**

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)**

---

<b>Title</b>	<b>CSTRING(256)</b>
--------------	---------------------

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**

The `EditFontClass` 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 `EditFontClass` contains the following methods:

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the `EditFontClass` methods.

#### **Primary Interface Methods**

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

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

<code>Init</code> <sup>VI</sup>	initialize the <code>EditFontClass</code> object
<code>Kill</code> <sup>VI</sup>	shut down the <code>EditFontClass</code> object

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

<code>TakeEvent</code> <sup>V</sup>	handle events for the edit control
-------------------------------------	------------------------------------

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

<code>CreateControl</code> <sup>V</sup>	create the edit (COMBO) control
<code>SetAlerts</code> <sup>VI</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 Primary Interface 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.

<code>Init</code> <sup>I</sup>	initialize the <code>EditFontClass</code> object
<code>CreateControl</code>	create the edit (COMBO) control
<code>SetAlerts</code> <sup>I</sup>	alert keystrokes for the edit control
<code>TakeEvent</code>	handle events for the edit control
<code>Kill</code> <sup>I</sup>	shut down the <code>EditFontClass</code> 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)

### 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**



# 21 - EDITMULTISELECTCLASS

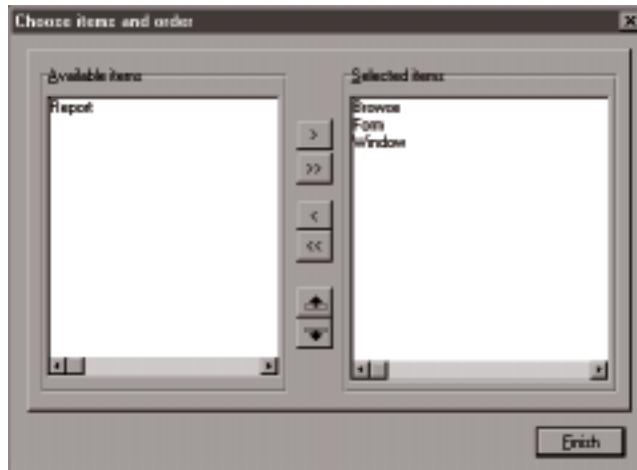
## Overview

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. It looks something like this illustration:



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, cancelling 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.

## 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.

## 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

## 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

_ABCD11Mode_ 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

Edit:PR:ApplyTo CLASS(EditMultiSelectClass)!declare Edit:PR:ApplyTo-EIP multi dialog
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
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(CURSOR: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***

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:

### **Available (multi-select dialog available items queue)**

---

<b>Available</b>	<b>&amp;ItemQueue,PROTECTED</b>
------------------	---------------------------------

The **Available** property is a reference to the QUEUE containing the set of items from which to select in the MultiSelect dialog.

### **Selected (multi-select dialog selected items queue)**

---

<b>Selected</b>	<b>&amp;ItemQueue,PROTECTED</b>
-----------------	---------------------------------

The **Selected** property is a reference to the QUEUE containing the set of selected items in the MultiSelect dialog.

### **FilePattern (multi-select dialog file pattern text)**

---

<b>FilePattern</b>	<b>CSTRING(1024)</b>
--------------------	----------------------

The **FilePattern** property contains a string that sets the pattern of files from which to select in the MultiSelect dialog.

### **Title (multi-select dialog title text)**

---

<b>Title</b>	<b>CSTRING(256)</b>
--------------	---------------------

The **Title** property contains a string that sets the title bar text in the MultiSelect dialog.

## **EditMultiSelectClass Methods**

The EditMultiSelectClass 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 EditMultiSelectClass contains the following methods:

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the EditMultiSelectClass methods.

#### **Primary Interface Methods**

The primary interface 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 EditMultiSelectClass object
AddValue	prime the MultiSelect dialog
Kill <sup>v</sup>	shut down the EditMultiSelectClass object

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

TakeAction <sup>v</sup>	handle user actions for the dialog
TakeEvent <sup>v</sup>	handle events for the edit control

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

CreateControl <sup>v</sup>	create the edit (COMBO) control
Reset	clear the MultiSelect dialog
SetAlerts <sup>vi</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 Primary Interface 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 <sup>1</sup>	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)

### 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 MAction:Add                       !end user selected an Item
  HoldIt=CHOOSE(HoldIt,HoldIt&','&Item,Item)
OF MAction: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 MAction: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)
  HoldIt=CHOOSE(HoldIt,HoldIt&','&ItemQ.Item,ItemQ.Item)
END

```

```
OF MSAction:StartProcess
  SETCURSOR(CURSOR:Wait)
OF MSAction:EndProcess
  SETCURSOR()
END
SELF.UseVar=HoldIt
```

```
!begin AddAll (>>) or DeleteAll (<<)
```

```
!end AddAll (>>) or DeleteAll (<<)
```

See Also: **TakeEvent**

## TakeEvent (process edit-in-place events)

### 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**



## 22 - 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.

### 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*.

### 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

## 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
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 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

```

## ***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.

In addition to (or instead of) the inherited methods, the EntryLocatorClass contains the following methods:

### 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 its 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)

### Set, VIRTUAL

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

## TakeAccepted (process an accepted locator value)

### TakeAccepted, VIRTUAL

The **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)

### TakeKey, VIRTUAL

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.

**Tip:** 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           ;!handle insert
OF DeleteKey          ;!handle delete
OF CtrlEnter          ;!handle enter (change/select)
OF MouseLeft2        ;!handle double-click (change/select)
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, VIRTUAL

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

## 23 - ERROR CLASS

### 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
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.

## 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.

## 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.

## 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) <i>or</i> %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.

## 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
OF Language:German           !which language was selected
    SELF.AddErrors(GermanErrors) !if German
                                !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

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.

### 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

## 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

## ErrorClass Methods

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the ErrorClass methods.

#### Primary Interface Methods

The primary interface 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 SetErrors has been called

#### Virtual Methods

Typically, you will not call these methods directly—the Primary Interface 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**

## Init (initialize the ErrorClass object)

---

### Init

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



## 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

## 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* )

**SetField** Sets the substitution value of the %Field macro.  
*fieldname* 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* )

**SetFile** Sets the substitution value of the %File macro.  
*filename* 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

## 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](#)

## SubString (resolves error message macros)

---

### SubString, PROTECTED

The **SubString** method returns the current error message text with all runtime macros resolved.

Implementation: The **TakeFatal**, **TakeNotify**, **TakeUser**, and **Message** methods call the **SubString** method to resolve macros.

Return Data Type: **STRING**

```
ErrorClass.TakeFatal PROCEDURE
CODE
MESSAGE(Self.SubString() & ' 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**, **Macro Expansion**, **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

**TakeError** Locates the specified error, calls the appropriate method to handle it, then returns the severity level.

*error id* 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



# 24 - 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.

## 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).

## 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 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***

The FieldPairsClass contains the following methods.

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the FieldPairsClass methods.

#### **Primary Interface Methods**

The primary interface 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
        ..
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)

### 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
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, List

## 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, List

## 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, List

## 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



# 25 - 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.

### 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*.

## 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  
ABDROPS.CLW

FileDropClass declarations  
FileDropClass method definitions

## 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()
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)
```

## 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.

### 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

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

In addition to (or instead of) the inherited methods, the FileDropClass contains the methods listed below.

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the FileDropClass methods.

#### Primary Interface Methods

The primary interface 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 <sup>†</sup>	add a range limit to the active sort order
AppendOrder <sup>†</sup>	refine the active sort order
Kill	shut down the FileDropClass object

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

ResetQueue	fill or refill filedrop queue
TakeEvent <sup>∨</sup>	process the current ACCEPT loop event
TakeNewSelection <sup>∨</sup>	processes EVENT:Selected events

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

Open <sup>†</sup>	open the filedrop view
PrimeRecord <sup>†</sup>	prepare an item for adding
SetFilter <sup>†</sup>	specify a filter for the active sort order
ApplyFilter <sup>†</sup>	range limit and filter the result set
ApplyOrder <sup>†</sup>	sort the result set
GetFreeElementName <sup>†</sup>	return the free element field name
SetOrder <sup>†</sup>	replace the active sort order
Close <sup>†</sup>	close the filedrop view

<sup>†</sup> These methods are inherited from the ViewManager Class.

<sup>∨</sup> These methods are also virtual.

## Virtual Methods

Typically you will not call these methods directly—the Primary Interface 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 <sup>1</sup>	reset the view position
TakeEvent <sup>v</sup>	process the current ACCEPT loop event
TakeNewSelection	processes EVENT:Selected events
ValidateRecord	validate the current result set element

<sup>1</sup> 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)
```

## 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

<b>ResetQueue</b>	Fills or refills the filedrop's display queue.
<i>force</i>	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, <i>force</i> 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)

---

### 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*

## TakeEvent (process the current ACCEPT loop event)

---

### 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
RVa1 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 RVa1
```

See Also:               *TakeNewSelection*, *WindowManager.TakeEvent*

## TakeNewSelection (process EVENT:NewSelection events)

---

### 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



# 26 - FILEDROPCOMBOCLASS

## Overview

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.

## 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 (ABDRUPS.INC) in your program's data section. See the *Conceptual Example*.

## 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

## 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

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

### 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

### 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

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

In addition to (or instead of) the inherited methods, the FileDropComboClass contains the methods listed below.

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the FileDropComboClass methods.

#### Primary Interface Methods

The primary interface 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 <sup>I</sup>	specify display fields
AddUpdateField <sup>I</sup>	specify field assignments
AddRange <sup>II</sup>	add a range limit to the active sort order
AppendOrder <sup>II</sup>	refine the active sort order
Kill <sup>I</sup>	shut down the FileDropComboClass object

##### Mainstream Use:

ResetQueue	refresh filedrop queue
GetQueueMatch	locate a list item
Ask <sup>V</sup>	add a record to the lookup file
TakeEvent <sup>V</sup>	process the current ACCEPT loop event
TakeNewSelection <sup>V</sup>	process the EVENT:Selected events

##### Occasional Use:

Open <sup>II</sup>	open the filedrop view
PrimeRecord <sup>II</sup>	prepare an item for adding
SetFilter <sup>II</sup>	specify a filter for the active sort order
ApplyFilter <sup>II</sup>	range limit and filter the result set
ApplyOrder <sup>II</sup>	sort the result set
GetFreeElementName <sup>II</sup>	return the free element field name
SetOrder <sup>II</sup>	replace the active sort order
Close <sup>II</sup>	close the filedrop view

<sup>I</sup> These methods are inherited from the FileDropClass.

<sup>II</sup> These methods are inherited from the ViewManager.

## **Virtual Methods**

Typically you will not call these methods directly—the Primary Interface 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.

## 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):

<code>Level:Benign</code>	<code>EQUATE(0)</code>
<code>Level:User</code>	<code>EQUATE(1)</code>
<code>Level:Program</code>	<code>EQUATE(2)</code>
<code>Level:Fatal</code>	<code>EQUATE(3)</code>
<code>Level:Cancel</code>	<code>EQUATE(4)</code>
<code>Level:Notify</code>	<code>EQUATE(5)</code>

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
```

See Also: **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**

## 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

## TakeEvent (process the current ACCEPT loop event)

### 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
RVa1 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 RVa1
```

See Also:

Ask, GetQueueMatch, ResetQueue, TakeNewSelection,  
WindowManager.TakeEvent

## TakeNewSelection (process EVENT:NewSelection events)

---

### 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



# 27 - FILEMANAGER

## Overview

The FileManager class declares a file manager which 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.

## 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.

## 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.

**Tip:** To derive from the FileManager, you can place code into FileManager global embed points to override existing FileManager methods, or you can create an ABC Compliant FileManager (see *ABC Compliant Classes*) to add new methods.

## 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

## 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

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
    . . . .
  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
IF ID = 3                                !and by this program too
  GlobalErrors.SetField('StateCode')     !validate the statecode (3rd) field
  !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
	<p>The <b>AliasedFile</b> 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.</p> <p><b>Tip:</b> This property should be null (uninitialized) for the actual file and initialized for any aliases.</p>

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
	<p>The <b>Buffer</b> 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.</p>
Implementation:	<p>The SaveBuffer method stores a copy of the current Buffer contents into the Buffers property for subsequent retrieval by the RestoreBuffer method.</p> <p>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 <i>Conceptual Example</i>.</p>
See Also:	<p>Buffers, RestoreBuffer, SaveBuffer</p>

## 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, SaveBuffer, RestoreBuffer

## 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



## 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)
            . .
            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 <b>OpenMode</b> 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 <code>Init</code> method sets the <code>OpenMode</code> property to a hexadecimal value of <code>42h</code> (ReadWrite/DenyNone). The ABC Templates override this default with the appropriate value from the application generator. See <i>Template Overview—File Handling</i> for more information.  The <code>Open</code> method uses the <code>OpenMode</code> property when it <code>OPENS</code> the file for processing. See the <i>Language Reference</i> for more information on <code>OPEN</code> and access modes.
See Also:	<code>Init</code> , <code>Open</code>

## SkipHeldRecords (HELD record switch)

---

SkipHeldRecords	BYTE
	The <b>SkipHeldRecords</b> property contains a value that tells the file manager how to react when it encounters held records. See the <i>Language Reference</i> for more information on <code>HOLD</code> .
	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 <code>Init</code> method sets the <code>SkipHeldRecords</code> property to a value of zero (0).  The <code>Next</code> , <code>TryNext</code> , <code>Previous</code> , and <code>TryPrevious</code> methods implement the action specified by the <code>SkipHeldRecords</code> property when an attempt to read a record fails because the record is held.
See Also:	<code>Init</code> , <code>Next</code> , <code>Previous</code> , <code>TryNext</code> , <code>TryPrevious</code>

# ***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.

## 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the FileManager methods.

### Primary Interface Methods

The primary interface 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 <sup>v</sup>	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 <sup>v</sup>	prepare an autoincremented record for adding
Insert	add a new record
TryInsert	add a new record
CancelAutoInc <sup>v</sup>	restore file to its pre-PrimeAutoInc state
Update	change the current record
TryUpdate	change the current record
Close <sup>v</sup>	close the file

<sup>v</sup> 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 Primary Interface 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

## 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 *ErrorClass* 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

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*.

<i>highvalue</i>	Numeric Fields		String Fields	
	<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')                !declare FileManager class
  MAP                                   !program map
  END
GlobalErrors  ErrorClass              !declare GlobalErrors object
Access:Order  CLASS(FileManager)      !derive Access:Order object
              END

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

```



## 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, 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.

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* ), 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)
  . . .

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
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

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')
  MAP
  LogError      (STRING filename, SHORT error)
  END

GlobalErrors ErrorClass
Access:Client CLASS(FileManager)
END

ErrorLog      FILE(TopSpeed),PRE(LOG),CREATE,THREAD
Record        RECORD
Date          LONG
Time          LONG
File          STRING(20)
ErrorId       SHORT
              . .

CODE
!program code
IF Access:Client.Open()
  LogError(Access:Client.GetName(),Access:Client.GetError())
END
!program code

LogError PROCEDURE(STRING filename, SHORT error)
CODE
LOG:Date      = TODAY()
LOG:Time      = CLOCK()
LOG:File      = filename
LOG:ErrorId   = error
ADD(ErrorLog)

```

## GetField (return a reference to a key component)

### GetField( *key, component* )

<b>GetField</b>	Returns a reference to the specified key component.
<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.

The **GetField** method returns a reference to the specified key component.

Return Data Type:    \*? (untyped variable parameter)

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

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

```

## GetFieldName (return a key component field name)

**GetFieldName**( *key*, *component* )

**GetFieldName** Returns the field name of the specified key component.

*key* The label of the KEY.

*component* A numeric constant, variable, EQUATE, or expression that indicates the component field. A value of one (1) specifies the first component; two (2) specifies the second component, etc.

The **GetFieldName** method returns the field name of the specified key component.

Return Data Type: **STRING**

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

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
```

## 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(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* )

<b>Init</b>	Initializes the FileManager object.
<i>file</i>	The label of the managed file.
<i>error handler</i>	The label of an ErrorClass object. See <i>Error Class</i> 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, VIRTUAL

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
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

## KeyToOrder (return ORDER expression for a key)

**KeyToOrder**( *key*, *component* )

<b>KeyToOrder</b>	Returns an ORDER attribute expression list (for a VIEW) that mimics the specified key components.
<i>key</i>	The label of the KEY.
<i>component</i>	A numeric constant, variable, EQUATE, or expression that indicates the first component field to include in the expression. A value of one (1) specifies the first component; two (2) specifies the second component, etc.

The **KeyToOrder** method returns an ORDER attribute expression list (for a VIEW) that mimics the specified key components. The expression list includes the specified component field plus all the subsequent component fields in the key.

See the *Language Reference* for more information on ORDER.

Implementation: The *component* defaults to one (1). The maximum length of the returned expression is 512 characters.

Return Data Type: **STRING**

Example:

```

PROGRAM
INCLUDE( 'ABFILE.INC' )                                !declare FileManager
MAP                                                     !program map
END

GlobalErrors ErrorClass                               !declare GlobalErrors
Access:Order  CLASS(FileManager)                      !derive Access:Order
END

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

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

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**

## 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
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.

**Tip:** If your autoincremented key has multiple component fields, then you should prime the high-order component fields, either in the global Global Objects, FileManager, PrimeAutoInc embed point, or in the LocalObjects, Browser, PrimeRecord embed point.

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')
  MAP
  END

GlobalErrors ErrorClass
Access:Client CLASS(FileManager)
Init PROCEDURE
PrimeAutoInc PROCEDURE,VIRTUAL
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 !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: **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)

**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

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
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*] )

<b>RestoreBuffer</b>	Restores previously saved record buffer contents.
<i>buffer id</i>	An integer constant, variable, EQUATE, or expression that identifies the buffer contents to restore—this is a value returned by the SaveBuffer method.
<i>restore</i>	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, <i>restore</i> 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
  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* )

<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.

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.WHeId = SELF.File{PROP:HeId}
  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
END

```

See Also:            Throw

## 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: Setame assumes the FileName property is contains 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)
. . .

CODE
GlobalErrors.Init                !initialize the GlobalErrors object
Access:Client.Init               !initialize the Access:Client object
LOOP I# = 1 TO 12
  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
END

```

See Also: **Init, SetError**

## ThrowMessage (pass an error and text to the error handler)

### ThrowMessage( *error id*, *text* ), VIRTUAL, PROC

<b>ThrowMessage</b>	Passes the specified error and text to the error handler object for processing.
<i>error id</i>	A numeric constant, variable, EQUATE, or expression that indicates the error to process.
<i>text</i>	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)
. . .
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')
. . .
RETURN Level:Notify
```

See Also: **Init**

## TryFetch (try to get a specific record by key value)

### TryFetch( *key* ), 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)
                . . .

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
  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
      .
    .
!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:Client.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
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

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
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

```

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

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

## 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

The **UseFile** 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.

Implementation: UseFile return values are declared in ABERROR.INC. See *Error Class* for more information on these severity levels.

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 validates the specified field in the current record buffer and returns a success or failure indicator.

Implementation:

The ValidateField 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 ValidateField method for each file that implements Validity Checks specified in the Clarion data dictionary.

The ValidateFields and ValidateRecord methods each invoke the ValidateField method for each field within their respective scopes.

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, ValidateRecord**

## 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**

## 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: ValidateField

# 28 - 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.

## 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*.

## 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

## 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
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***

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:

#### **FloatRight=False**

Bain  
Barber  
Bayert

#### **FloatRight=True**

Bain  
Barber  
Bayert  
Dunbar  
Suba

See Also:

`UpdateWindow`

## 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.

In addition to (or instead of) the inherited methods, the FilterLocatorClass contains the following methods:

### TakeAccepted (process an accepted locator value)

#### 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 locator is present
IF ~(SELF.Sort.Locator &= NULL)                                    ! redraw locator control
    SELF.Sort.Locator.UpdateWindow
END
```

See Also: FreeElement, Shadow



# 29 - 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.

## 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*.

## 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

## 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
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***

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***

The `IncrementalLocatorClass` inherits all the methods of the `EntryLocatorClass` from which it is derived. See *EntryLocatorClass Methods* and *LocatorClass Methods* for more information.

In addition to (or instead of) the inherited methods, the `IncrementalLocatorClass` contains the following methods:

### **SetAlerts (alert keystrokes for the LIST control)**

---

#### **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)

---

### 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

# 30 - INICLASS

## Overview

The INIClass object centrally handles reads and writes for a given configuration (.INI) file.

## INI Class 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.

## 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.

## 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.

## INI Class 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

## 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',|
            CLIP(Sound)&','&Volume) !store comma delimited values
            !e.g., Sound&Vol=ON,3
        POST(EVENT:CloseWindow)
    END
END
END
INIMgr.Update('Preferences',PWindow) !store window size & pos

```

## INIClass Properties

The INIClass contains the following properties.

### FileName

FileName	CSTRING(File:MaxFilePath)
----------	---------------------------

The **FileName** property contains the name of the managed INI file. The INIClass methods use the FileName property to identify the INI file.

If a full path is specified, 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:

FileName Property	Resulting INI File
'	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

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] | )
          | 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>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</i> 's 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*. 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*,*window*)

Restores several WINDOW attributes saved by a prior corresponding call to Update(*section*,*window*). 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

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
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* )

<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.

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

Implementation: The Init method assigns *filename* to the FileName property.

Example:

```

        INCLUDE('UTILITY.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

```

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
```



## Update (write INI file entries)

```
Update( section, | entry, value | )
          | window |
```

<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>window</i>	The label of a WINDOW or APPLICATION whose position and size parameters the Update method stores.

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*.

`Update(section>window)`  
Writes several WINDOW position and size attributes for retrieval by a subsequent corresponding call to `Fetch(section>window)`. Restoring the values returns the specified WINDOW to its saved position and size.

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',PWindow)         !save PWindow size & position
```

See Also:

Fetch



# 31 - 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.

## 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*.

## 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.CW	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.

### 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



## SetAlerts (alert keystrokes for the LIST control)

---

### 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
```

## TakeAccepted (process an accepted locator value)

---

### 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)

---

### 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`—`IncrementalLocatorClass`, `FilterLocatorClass`, etc.

See Also:             `IncrementalLocatorClass.UpdateWindow`, `FilterLocatorClass.UpdateWindow`

# 32 - POPUPCLASS

## Overview

The PopupClass object defines and manages a full featured popup (context) menu (including multi-level or nested menus with icon support). The PopupClass object optionally presents the popup menu choices in the form of a floating toolbox. 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/icon matches the *button* text/icon, 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. Finally, you can custom code the action associated with a menu item.

The PopupClass supports runtime language translation (see *TranslatorClass*) and runtime reordering of menu items without otherwise changing the code that displays and processes the popup menu or toolbox.

## 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.

## 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:

<code>ABPOPUP.INC</code>	PopupClass declarations
<code>ABPOPUP.CLW</code>	PopupClass method definitions
<code>ABPOPUP.TRN</code>	PopupClass translation strings

## 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),ICON('Save.ico')
            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                    !trap for alerted keys
  IF KEYCODE() = MouseRight          !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.

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the PopupClass methods.

#### **Primary Interface Methods**

The primary interface 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
Toolbox	display and process toolbox

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

DeleteItem	remove menu item
GetLastSelection	return last selected item
SetTranslator	set run-time translator
Save	save a menu for restoration
SetIcon	set menu item icon
SetLevel	set menu item hierarchy level
SetText	set menu item text
SetToolbox	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. 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.

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 menu and items without otherwise changing the code that displays and processes the popup menu/toolbox.

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.

**Tip:** By default, menu items added with this method do not appear on the PopupClass object's toolbox because they have no associated action to execute. Use the AddItemMimic or AddItemEvent methods to include items on the toolbox.

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

<b>AddItemEvent</b>	Associates an event with a menu item.
<i>name</i>	A string constant, variable, EQUATE, or expression containing the name of the menu item associated with the <i>event</i> . If the named item does not exist, AddItemEvent adds it at the bottom of the popup menu.
<i>event</i>	An integer constant, variable, EQUATE, or expression containing the event number to POST when the end user selects the menu item.
<i>control</i>	An integer constant, variable, EQUATE, or expression containing the control number to POST the <i>event</i> to when the end user selects the menu item. To post a field-independent event, use a <i>control</i> value of zero (0). If omitted, <i>control</i> 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.

**Tip:** By default, menu items added with this method appear on the `PopupClass` object's toolbox because they have an associated action to execute. Use the `SetToolbox` method to explicitly include or exclude items from the toolbox.

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, SetToolbox

## AddItemMimic (tie menu item to a button)

**AddItemMimic**( *name*, *button* [, *text* ] ), PROC

<b>AddItemMimic</b>	Associates a menu item with a <b>BUTTON</b> .
<i>name</i>	A string constant, variable, <b>EQUATE</b> , or expression containing the menu item name to associate with the <i>button</i> . If the named item does not exist, <b>AddItemMimic</b> adds it at the bottom of the popup menu. To add a new item, the <i>button</i> must have text, or you must supply the <i>text</i> parameter.
<i>button</i>	A numeric constant, variable, <b>EQUATE</b> , or expression containing the associated <b>BUTTON</b> 's control number. If the button has no text, you should supply the <i>text</i> parameter.
<i>text</i>	A string constant, variable, <b>EQUATE</b> , or expression containing the text of the menu item. This overrides any <i>button</i> text. If omitted, or if the first character is an exclamation point (!), <b>AddItemMimic</b> uses the <i>button</i> 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 and icon matches the *button* text and icon, is enabled only when the *button* is enabled, and, when selected, invokes the *button* action.

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 menu and items without otherwise changing the code that displays and processes the popup menu/toolbox.

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.

**Tip:** By default, menu items added with this method appear on the **PopupClass** object's toolbox because they have an associated action to execute. Use the **SetToolbox** method to explicitly include or exclude items from the toolbox.

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**, **SetToolbox**

## 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.

**Tip:** By default, menu items added with this method do not appear on the PopupClass object's toolbox because they have no associated action to execute. Use the AddItemMimic or AddItemEvent methods to include items on the toolbox.

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* )

<b>AddSubMenu</b>	Adds a submenu to an existing menu.
<i>text</i>	A string constant, variable, EQUATE, or expression containing the submenu text. If omitted, the submenu text must be prepended to the <i>selections</i> parameter.
<i>selections</i>	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 (}).
<i>name to follow</i>	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.

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 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.

**Tip:** By default, menu items added with this method do not appear on the PopupClass object's toolbox because they have no associated action to execute. Use the AddItemMimic or AddItemEvent methods to include items on the toolbox.

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()                                    !display popup menu
    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.
<i>x</i>	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.
<i>y</i>	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**

## DeleteItem (remove menu item)

---

### DeleteItem( *name* )

#### DeleteItem

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 **DeleteItem** 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

## 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                                       !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, 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

## 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*] )

**Init**

Initializes the PopupClass object.

*INIClass*

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* )

**Restore** Restores a menu saved by the `PopupClass.Save` method.  
*menu* 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* )

**Save** Saves a menu for restoration by the PopupClass.Restore method.

*menu* 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

```

See Also: **Init, Restore**

## SetIcon (set menu item icon)

---

**SetIcon**( *name*, *iconpathname* )

<b>SetIcon</b>	Sets the menu item icon.
<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>iconpathname</i>	A string constant, variable, EQUATE, or expression containing the pathname of the icon to display.

The **SetIcon** method sets the icon for a menu item.

Implementation: The *name* parameter accepts up to 1024 characters.

Example:

```
PopupMgr.SetText('Save', '&Save')
PopupMgr.SetIcon('Save', 'save.ico')
```

See Also: AddItem, AddItemMimic, AddMenu

## SetItemCheck (set toggle item status)

**SetItemCheck**( *name*, *status* )

<b>SetItemCheck</b>	Sets the status of a toggle menu item.
<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>status</i>	A Boolean constant, variable, EQUATE, or expression containing the status to which to set the toggle item. A <i>status</i> 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* )

<b>SetItemEnable</b>	Sets the enabled/disabled status of a menu item.
<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>status</i>	A Boolean constant, variable, EQUATE, or expression containing the status to which to set the item. A <i>status</i> 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, 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 (include item on toolbox)

---

### SetToolbox( *name*, *show* )

<b>SetToolbox</b>	Includes or excludes the item from the PopupClass toolbox.
<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>show</i>	An integer constant, variable, EQUATE, or expression indicating whether the item is included in the PopupClass toolbox. A value of one (1 or True) includes the item; a value of zero (0 or False) excludes the item.

The **SetToolbox** method includes or excludes the item from the PopupClass toolbox.

**Tip:** Use the **SetToolbox** method to exclude the “Start Toolbox” choice from the Toolbox.

Implementation:

The *name* parameter accepts up to 1024 characters.

By default, menu items added by the AddItemMimic and AddItemEvent methods are included on the toolbox because they have associated actions the PopupClass can execute. Menu items added with other methods (AddItem, AddMenu, AddSubmenu) are not included on the toolbox.

Example:

```
PopupMgr.SetToolbox('Start Toolbox',False)
```

See Also:

AddItemEvent, AddItemMimic, Toolbox

## SetTranslator (set run-time translator)

### 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
MenuChoices   EQUATE('&Save Menu|&Restore Menu|&Close') !declare menu definition
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 (display the popup toolbox)

---

### Toolbox( *caption* )

**Toolbox**                      Displays the popup menu choices in a floating toolbox.  
*caption*                        A string constant, variable, EQUATE, or expression containing the toolbox caption (titlebar text).

The **Toolbox** method displays the popup menu choices in a floating toolbox and performs any action set by AddItemEvent or AddItemMimic for the selected item.

#### Implementation:

The Toolbox method displays only those items that have an associated action or that are designated as toolbox items by the SetToolbox method. By default, menu items added by the AddItemMimic and AddItemEvent methods are included on the toolbox because they have associated actions the PopupClass can execute. Menu items added with other methods (AddItem, AddMenu, AddSubmenu) are not included on the toolbox.

The Toolbox method indirectly calls the Ask method to invoke the menu item's action.

#### Example:

```
MyBrowseClass.TakeEvent  PROCEDURE
CODE
!browse event processing
CASE ACCEPTED()
OF SELF.ToolControl
    SELF.Popup.Toolbox('Browse Actions')      !start the toolbox
END
```

See Also:                      AddItemEvent, AddItemMimic, Ask, SetToolbox

## 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. See *The Debuggers* in the *User's Guide* for more information.



# 33 - PRINTPREVIEWCLASS

## Overview

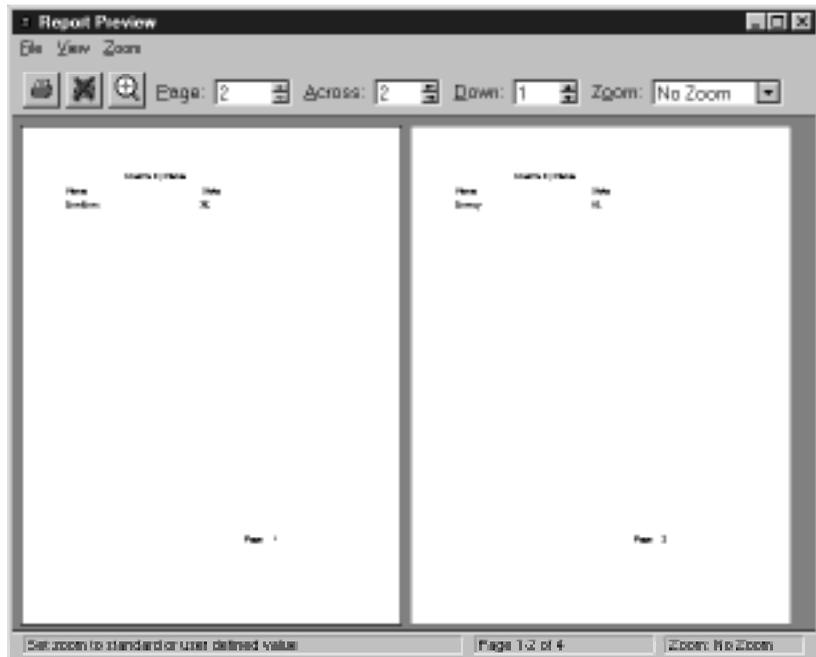
The PrintPreviewClass is a WindowManager that implements a full-featured print preview dialog.

## 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.

## 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.

## 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



```

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),@PAGENO,USE(?PageCount)
            END
            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

### **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.

## 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 or True) indicates a non-default initial position is specified and is applied; a zero (0 or False) 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 or `True`) indicates a non-default initial size is specified and is applied; a zero (0 or `False`) 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 ( <code>PagesAcross</code> and <code>PagesDown</code> 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.

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the PrintPreviewClass methods.

#### **Primary Interface Methods**

The primary interface methods, which you are likely to call fairly routinely from your program, can be further divided into two categories:

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

Init <sup>v</sup>	initialize the PrintPreviewClass object
SetPosition	set initial preview window coordinates
Display <sup>v</sup>	preview the report
Kill <sup>v</sup>	shut down the PrintPreviewClass object

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

SetINIManager	save and restore window coordinates
SetPosition	set print preview position and size
SetZoomPercentile	set user or standard zoom factor

<sup>v</sup> 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.

Init <sup>v</sup>	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
Kill <sup>v</sup>	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
RVa1      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
        RVa1=True                                !SELF.CurrentPage changed
        SELF.CurrentPage=JumpPage
    END
    POST(EVENT:CloseWindow)
OF ?CancelButton
    POST(EVENT:CloseWindow)
. . .
CLOSE(JumpWin)
RETURN RVa1
```

## 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)
```

## 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

**Init** Initializes the PrintPreviewClass object.

*image queue* The label of the QUEUE containing the filenames of the report image metafiles. See *PREVIEW* in the *Language Reference* 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
```

## 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

## TakeAccepted (process EVENT:Accepted events)

### 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
RVa1 BYTE(Level:Benign)
I   USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;  RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

**See Also:** TakeEvent, WindowManager.TakeEvent

## TakeEvent (process all events)

---

### 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      ! Not as dopey at it looks, it is for 'short-stopping' certain events
END
END
```

See Also: WindowManager.Ask

## TakeFieldEvent (a virtual to process field events)

### 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
RVa1 BYTE(Level:Benign)
I   USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;  RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

See Also: [Ask](#)

## TakeWindowEvent (process non-field events)

### 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
RVa1 BYTE(Level:Benign)
I    USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;    RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;   RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

**See Also:** TakeEvent

# 34 - PROCESSCLASS

## Overview

The ProcessClass is a ViewManager with a progress window. The ProcessClass can process multiple levels of related views (parent, child, grandchild, etc.) by reading all the child items for a specific parent item, all the grandchildren of each child item, and so on.

## 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.

## 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.

## ABC Template Implementation

---

### Process and Report Templates

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.

### **ChildFile Template**

The ChildFile Extension template generates code to take advantage of the ProcessClass's multi-level (parent, child, grandchild, etc.) processing. See ChildRead, AddItem, and Next.

## **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	ProcessClass declarations
ABREPORT.CLW	ProcessClass method definitions

## **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
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

```



```
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:

### ChildRead (portion of process completed)

---

ChildRead	BYTE, PROTECTED
-----------	-----------------

The **ChildRead** property determines (keeps track of) the type of item (parent, child, grandchild, etc.) the ProcessClass object reads next. A value of zero indicates a primary view item was read; a value of one (1) indicates a child of the primary view item was read; a value of two (2) indicates a child of the child (grandchild of the primary view) item was read, and so on.

Implementation: The AddItem method registers the types of items to read. The Next method tries to read the type of item specified by the ChildRead property, then sets the ChildRead property to indicate the type of item actually read.

See Also: AddItem, Next

### 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.

In addition to (or instead of) the inherited methods, the ProcessClass contains the methods listed below.

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the ProcessClass methods.

#### Primary Interface Methods

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

Init	initialize the ProcessClass object
AddItem	add a child view to process
AddRange <sup>1</sup>	add a range limit to the active sort order
AddSortOrder <sup>1</sup>	add a sort order
AppendOrder <sup>1</sup>	refine the active sort order
SetProgressLimits	calibrate the StepClass progress monitor
Kill <sup>v</sup>	shut down the ProcessClass object

##### Mainstream Use:

Open <sup>1</sup>	open the view
Next <sup>v</sup>	get the next result set element
Previous <sup>IV</sup>	get the previous result set element
PrimeRecord <sup>1</sup>	prepare a record for adding
ValidateRecord <sup>IV</sup>	validate the current result set element
SetFilter <sup>1</sup>	specify a filter for the active sort order
SetSort <sup>IV</sup>	set the active sort order
ApplyFilter <sup>1</sup>	range limit and filter the result set
ApplyOrder <sup>1</sup>	sort the result set
ApplyRange <sup>1</sup>	conditionally range limit and filter the result set
Close <sup>1</sup>	close the view

##### Occasional Use:

GetFreeElementName <sup>1</sup>	return the free element field name
Reset <sup>v</sup>	reposition to the first result set element
SetOrder <sup>IV</sup>	replace the active sort order

<sup>1</sup> These methods are inherited from the ViewManager class.

<sup>v</sup> These methods are also Virtual.

## **Virtual Methods**

Typically you will not call these methods directly—the Primary Interface 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 <sup>1</sup>	get the previous result set element
Reset	reposition to the first result set element
SetSort <sup>1</sup>	set the active sort order
ValidateRecord <sup>1</sup>	validate the current result set element
TakeRecord	a virtual to process each record
Kill	shut down the ProcessClass object

<sup>1</sup>These methods are inherited from the ViewManager class.

## AddItem (add a child viewmanager)

### AddItem( *viewmanager* )

**AddItem** Adds a “child” ViewManager to the ProcessClass object’s knowledge base.

*viewmanager* The label of the child ViewManager object.

The **AddItem** method adds a “child” ViewManager to the ProcessClass object’s knowledge base, and returns the ordinal (or sequence) number of the added ViewManager. The ProcessClass object uses the child ViewManager to read the child records belonging to a specific primary file record.

Implementation: The Next method reads the child records for each ViewManager added by the AddItem method.

Return Data Type: USHORT

Example:

```
PrintCUST:ByName PROCEDURE

ORD:View          VIEW(Orders).           !declare ORD:View
ORD:Level         BYTE,AUTO              ! (Orders is child of Customer)
ORD:ViewManager   ViewManager           !declare ORD:ViewManager
ThisProcess       CLASS(ProcessClass)
TakeRecord        PROCEDURE(),BYTE,PROC,VIRTUAL
                  END

ThisWindow.Init   PROCEDURE()
ReturnValue       BYTE,AUTO
CODE
ORD:ViewManager.Init(ORD:View,Relate:Orders) !manage the ORD:View (Order file)
ORD:ViewManager.AddSortOrder(ORD:AsEntered) !range limit Orders by Customer
ORD:ViewManager.AddRange(ORD:OrdNo,Relate:Orders,Relate:Customer)
ORD:Level = ThisProcess.AddItem(ORD:ViewManager) !register ORD:ViewManager
RETURN ReturnValue ! with ThisProcess object

ThisProcess.TakeRecord PROCEDURE()
SkipDetails       BYTE
CODE
IF SELF.ChildRead = ORD:Level           !if this is a child record
  SkipDetails = TRUE                    ! set a flag
  PRINT(RPT:ChildDetail)                ! print the child DETAIL
END
IF ~SkipDetails                          !if this is a parent record
  PRINT(RPT:Parentdetail)               ! print the parent DETAIL
END
RETURN PARENT.TakeRecord()
```

See Also: ChildRead, Next

## 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:

```
Process.Init( Process:View,      |      !initialize the ProcessClass object
              Relate:Client,    |      !set the VIEW
              ?PctText,        |      !set the primary file RelationManager
              PctDone,         |      !set the Window control for text messages
              ProgressMgr,     |      !set the PROGRESS USE variable
              CLI:Name)        |      !set StepClass object to monitor progress
                               |      !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:

```
Process.Init( Process:View, | !initialize the ProcessClass object
               Relate:Client, | !set the VIEW
               ?PctText, | !set the primary file RelationManager
               PctDone, | !set the Window control for text messages
               ProgressMgr, | !set the PROGRESS USE variable
               CLI:Name) | !set StepClass object to monitor progress
                       | !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

<b>Next</b>	Gets the next result set element.
<i>process records</i>	A boolean constant, variable, EQUATE, or expression that tells the ProcessClass object whether to update its progress indicators. A zero (0 or False) value does not update the progress indicators; a value of one (1 or True) does update the indicators. If omitted, <i>process records</i> defaults to 1.

The **Next** method gets the next element in the result set and returns a value indicating its success or failure. A return value of Level:Benign indicates a successful read; any other value indicates no new item was read.

The Next method sets the ChildRead property to indicate the type of element actually read (parent, child, grandchild, etc.)

#### Implementation:

The Next method calls the ViewManager.Next method for the ViewManager indicated by the ChildRead property. The ChildRead property indicates whether the next element is a parent, child, grandchild, etc. If there are no more items at the current level, the Next method reverts to a higher level to get the next element.

The 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 next record
    POST(Event:CloseWindow)                   !if no records, shut down
  CYCLE
END
OF Event:Timer
  StartOfCycle=Process.RecordsProcessed        !process records with timer
  LOOP WHILE Process.RecordsProcessed-StartOfCycle<RecordsPerCycle
  CASE Process.Next()                          !get next record
  OF Level:Notify
    MESSAGE('Process Completed')              !tell end user
    POST(EVENT:CloseWindow)                   !and shut down
    BREAK
  OF Level:Fatal
    POST(EVENT:CloseWindow)                   !shut down
    BREAK
  . . . .
```

#### See Also:

AddItem, ChildRead, 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.SetLimits`

## TakeRecord (a virtual to process each record)

---

### TakeRecord, VIRTUAL, PROC

The **TakeRecord** method is a virtual placeholder to process 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.

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**



# 35 - 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.

## 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*.

## 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

## 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

  _ABCD11Mode_ 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)
CUS:State      LIKE(CUS:State)
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('&Zoom In'),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          QueryClass                                        !declare Query 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                                     !initialize the 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 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 no public properties.

## QueryClass Methods

The QueryClass contains the following methods:

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the QueryClass methods.

#### Primary Interface Methods

The primary interface 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
Kill <sup>∇</sup>	shut down the QueryClass object

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

Ask <sup>∇</sup>	a virtual to accept query criteria
GetFilter	return filter expression

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

Reset	reset the QueryClass object
GetLimit	get searchvalues
SetLimit	set search values

<sup>∇</sup> 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* ] )

**AddItem** Adds specific functionality to the QueryClass.  
*name* 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*.

*title* 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.

*picture* A string constant, variable, EQUATE, or expression containing the display picture for the queryable item. If omitted, *picture* defaults to S255 (unformatted string). See *Picture Tokens* in the *Language Reference* 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

**Ask** A virtual to accept query criteria (search values) from the end user.

*uselast* 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 queryable item (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

## 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
END
RETURN Level:Notify

```

See Also: [AddItem](#), [Ask](#), [SetLimit](#)

## Init (initialize the QueryClass object)

---

### Init

The **Init** method initializes the QueryClass 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

## 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>	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)

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



# 36 - 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.

## 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*.

## 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

## 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, 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

_ABcd11Mode_ 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()
ReturnValu     BYTE,AUTO
CODE
ReturnValu = PARENT.Init()
IF ReturnValu THEN RETURN ReturnValu.
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 ReturnValu

ThisWindow.Kill    PROCEDURE()
ReturnValu     BYTE,AUTO
CODE
ReturnValu = PARENT.Kill()
IF ReturnValu THEN RETURN ReturnValu.
Relate:Customer.Close
RETURN ReturnValu

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.

In addition to (or instead of) the inherited methods, the QueryFormClass contains the following methods:

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the QueryFormClass methods.

#### Primary Interface Methods

The primary interface 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 <sup>†</sup>	add a field to query
Kill <sup>∇</sup>	shut down the QueryFormClass object

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

Ask <sup>∇</sup>	accept query criteria
GetFilter <sup>†</sup>	return filter expression

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

Reset <sup>†</sup>	reset the QueryFormClass object
GetLimit <sup>†</sup>	get searchvalues
SetLimit <sup>†</sup>	set search values

<sup>∇</sup> These methods are also Virtual.

<sup>†</sup> 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**

**Ask** Accepts query criteria (search values) from the end user.

*uselast* 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* )

**Init**                      Initializes the QueryFormClass object.

*query window manager*

The label of the QueryFormVisual object that displays the query input dialog and processes its events.

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**



# 37 - QUERYFORMVISUAL

## Overview

The QueryFormVisual class is a WindowManager 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.

## Relationship to Other Application Builder Classes

---

The QueryFormVisual class is derived from the WindowManager.

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*.

## 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

## 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 WindowManager and BrowseClass objects internally handle the normal events surrounding the query.

```
PROGRAM

_ABCD11Mode_ 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
QBEWindow    QueryFormVisual                                  !declare QBEWindow 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 WindowManager from which it is derived. See *WindowManager Properties* for more information.

In addition to the inherited properties, the QueryFormVisual contains the following property:

### **QFC (reference to the QueryFormClass)**

---

<b>QFC</b>	<b>&amp;QueryFormClass</b>
------------	----------------------------

	The <b>QFC</b> 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
-----------	---------------------



## 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

## TakeAccepted (handle query dialog EVENT:Accepted events)

### 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 queryable item (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
RVa1 BYTE(Level:Benign)
I    USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;   RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

**See Also:** QFC

## TakeCompleted (complete the query dialog)

### 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`



# 38 - 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.

## 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.

## 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

## 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} = '% 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
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***

The Relation Manager contains the following properties.

### **Me (the primary file's FileManager object)**

---

<b>Me</b>	<b>&amp;FileManager</b>
-----------	-------------------------

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)**

---

<b>UseLogout</b>	<b>BYTE</b>
------------------	-------------

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

The Relation Manager contains the following methods.

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the RelationManager methods.

#### Primary Interface Methods

The primary interface 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 <sup>v</sup>	open a file and any related files
Save <sup>v</sup>	copy current and designated related records
Update <sup>v</sup>	update current record subject to RI constraints
Delete <sup>v</sup>	delete current record subject to RI constraints
Close <sup>v</sup>	close a file and any related files

<sup>v</sup> 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**

<b>AddRelation</b>	Describes a relationship between this object's primary file (see <i>Me</i> ) and another file.
<i>relationmanager</i>	The label of the related file's RelationManager object.
<i>updatemode</i>	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 <i>deletemode</i> and <i>relatedkey</i> must also be omitted, and the relationship is unconstrained.
<i>deletemode</i>	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 <i>updatemode</i> and <i>relatedkey</i> must also be omitted, and the relationship is unconstrained.
<i>relatedkey</i>	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 <i>updatemode</i> and <i>deletemode</i> 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       END         !delete the linked records in related files

```

## 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: **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
ShipTo         STRING(32)
Shipped        BYTE
. .
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
. .
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

**Close** Closes this object's primary file (see *Me*) and any related files.

*cascading* 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, *cascading* defaults to zero (0). You should *always* 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 *FileManager 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

<b>Delete</b>	Deletes the record from the primary file subject to any specified referential integrity constraints.
<i>confirm</i>	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, <i>confirm</i> 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 *uselogout* 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

## Init (initialize the RelationManager object)

**Init**( *filemanager* [, *uselogout*] )

<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>uselogout</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 *uselogout* 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

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                                     !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
```

See Also:

## Open (open a file and any related files)

---

### Open( *cascading* ), VIRTUAL, PROC

**Open** Opens this object's primary file (see *Me*) and any related files.

*cascading* 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, *cascading* defaults to zero (0). You should *always* 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

<b>SetQuickScan</b>	Enables or disables quick scanning on this object's primary file and on the <i>propagated</i> related files.
<i>on</i>	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.
<i>propagate</i>	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, <i>propagate</i> 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

**Update** Updates this object's primary file (see *Me*) subject to the specified referential integrity constraints.

*fromform* 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**



# 39 - 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.

## 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*.

## ABC Template Implementation

---

### Report Templates

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.

### Other Templates

The ChildFile extension template generates code so your reports can efficiently process related child file items for each primary file item. See *Code and Extension Templates—ChildFile* for more information.

The ExtendProgressWindow extension template generates code to help you control the visual feedback for your report (progress window and wait cursor), and to allow your report to alternatively process all items or a single item. See *Code and Extension Templates—ExtendProgressWindow* for more information.

The PauseButton control template generates code to let the end user suspend, resume, and rerun the report without restarting the report procedure. See *Control Templates—PauseButton* 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

## 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),PAGENO,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:

### 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 display)

---

**DeferWindow**      **USHORT, PROTECTED**

The **DeferWindow** property controls when the ReportManager displays the progress window. A value of zero (0 or False) displays the progress window at the first opportunity (immediately); any other value delays the display for *value* seconds. For example, a value of 5 delays the progress window for 5 seconds.

The DeferWindow property lets you delay or completely suppress the progress window for your reports.

**Tip:**      Use the ExtendProgressWindow extension template to generate references to the DeferWindow property.

Implementation:      The Ask and TakeWindowEvent methods implement the behavior specified by the DeferWindow property.

See Also:              Ask, TakeWindowEvent

## KeepVisible (persistent progress window)

---

**KeepVisible**      **BYTE, PROTECTED**

The **KeepVisible** property controls whether the ReportManager hides or displays the progress window prior to invoking the print preview. A value of one (1 or True) displays the progress window; a value of zero (0 or False) hides the progress window.

The KeepVisible property lets you present a persistent progress window for reports with suspend-resume and multi-start capabilities.

**Tip:**      **Use the PauseButton control template to generate references to the KeepVisible property.**

Implementation:      The TakeCloseEvent method implements the behavior specified by the KeepVisible property.

See Also:              TakeCloseEvent

## 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

## 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

## 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

## 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 (defer progress window display)

---

**WaitCursor**     **BYTE, PROTECTED**

The **WaitCursor** property controls whether the ReportManager displays the standard Windows wait cursor while the report is processing. A value of one (1 or True) displays the wait cursor; a value of zero (0 or False) does not.

The WaitCursor property is useful especially when you use the DeferWindow property to delay or suppress the progress window display.

**Tip:**     **Use the ExtendProgressWindow extension template to generate references to the WaitCursor property.**

Implementation:     The Ask and TakeCloseEvent methods implement the behavior specified by the WaitCursor property.

See Also:     Ask, TakeCloseEvent

## 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.

In addition to (or instead of) the inherited methods, the ReportManager contains the following methods:

### 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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the ReportManager methods.

#### Primary Interface Methods

The primary interface 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
Ask <sup>v</sup>	display window and process its events
Kill <sup>v</sup>	shut down the ReportManager object

<sup>v</sup> These methods are also Virtual.

#### Virtual Methods

Typically you will not call these methods directly—the primary interface 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
Next	get next report record
Open	prepare progress window
OpenReport	prepare report for execution
TakeNoRecords	handle empty report
TakeCloseEvent	process EVENT:CloseWindow events
TakeWindowEvent	process non-field events
Kill	shut down the ReportManager object

## Ask (display window and process its events)

---

### 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

## 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
END
```

**See Also:** **ProcessClass.Next**, **WindowManager.Response**

## Open (a virtual to execute on EVENT:OpenWindow)

### 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
RVa1 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:Sized
  IF BAND(SELF.Inited,2)
    SELF.Reset
  ELSE
    SELF.Inited = BOR(SELF.Inited,2)
  END
OF EVENT:Completed
  RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
  RVa1 = SELF.TakeCloseEvent()
END
RETURN RVa1
```

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
END

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
        . . .
```

**See Also:** DeferOpenReport, Next, Open, TakeWindowEvent, Process.Reset

## 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 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
RVa1 BYTE(Level:Benign)
I   USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;  RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

**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:

```
MyReportr.TakeNoRecords PROCEDURE
CODE
PARENT.TakeNoRecords
CLI:CustomerName = 'No Customers'
PRINT(CustomerDetail)
```

See Also: **OpenReport**

## TakeWindowEvent (a virtual to process non-field events)

### 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
RVa1 BYTE(Level:Benign)
I   USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;  RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

**See Also:** DeferOpenReport, Next, TimeSlice, TakeRecord, WindowManager.TakeEvent, WindowManager.TakeWindowEvent



# 40 - SELECTFILECLASS

## Overview

### 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.

### 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.

### 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

## 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
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

The SelectFileClass contains the following 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 *.*|*. *')      a!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 *.*|*.*')      a!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**



# 41 - 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.

## 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.

## 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	StepClass declarations
ABBROWSE.CLW	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)**

---

<b>Controls</b>	<b>BYTE</b>
-----------------	-------------

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

The StepClass contains the following methods.

### GetPercentile (return a value's percentile)

---

**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)

---

**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)

---

### SetLimit( *lower*, *upper* ), VIRTUAL

**SetLimit** Sets the StepClass object's upper and lower boundaries.

*lower* A constant, variable, EQUATE, or expression that specifies the StepClass object's lower boundary. The value may be numeric or alphanumeric.

*upper* 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)

---

### 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, BrowseClass.ResetThumbLimits

# 42 - 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).

## 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*.

## 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

## 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

The StepCustomClass inherits all the methods of the StepClass from which it is derived. See *StepClass Concepts* and *StepClass Methods* for more information.

In addition to (or instead of) the inherited methods, the StepCustomClass contains the following methods:

### 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: **Markers**

## GetPercentile (return a value's percentile)

---

### 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
END

```

See Also: [AddItem](#)

## GetValue (return a percentile's value)

---

### 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* )

**Init** Initializes the StepCustomClass object.

*controls* 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
```



# 43 - 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).

## 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*.

## ABC Template Implementation

---

The ABC BrowseBox template generates code to instantiate the StepLocatorClass for your BrowseBoxes. The StepLocatorClass objects are called `BRWn::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.CW	StepLocatorClass method definitions

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

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)

#### Set, VIRTUAL

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
END
```

### TakeKey (process an alerted keystroke)

#### 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.

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

# 44 - 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.

## 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*.

## 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

## 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.VCRRRequest &= VCRRRequest
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)

---

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)

---

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)

#### 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)

---

### 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)

---

**SetLimit**( *lower*, *upper* ), VIRTUAL

**SetLimit** Sets the StepLongClass object's evenly distributed steps between *upper* and *lower*.

*lower* An integer constant, variable, EQUATE, or expression that specifies the StepLongClass object's lower boundary.

*upper* 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 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, BrowseClass.ResetThumbLimits, ProcessClass.SetProgressLimits



# 45 - 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.

## 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*.

## 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

## **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
ProgressMgr     StepRealClass                !declare ThisProcess object
                                                !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

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)

---

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)

---

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

## StepRealClass Methods

The StepRealClass 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 StepRealClass contains the following methods:

### GetPercentile (return a value's percentile)

#### GetPercentile( *value* ), VIRTUAL

**GetPercentile** Returns the specified *value*'s percentile relative to the StepRealClass 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 StepRealClass 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 StepRealClass 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 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)

---

**SetLimit**( *lower*, *upper* ), VIRTUAL

<b>SetLimit</b>	Sets the <b>StepRealClass</b> 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 <b>StepRealClass</b> object's lower boundary.
<i>upper</i>	An integer constant, variable, EQUATE, or expression that specifies the <b>StepRealClass</b> 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** 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**, **BrowseClass.ResetThumbLimits**,  
**ProcessClass.SetProgressLimits**



# 46 - 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.

## 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*.

## 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

## 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)
Q          &StateQ
          END

StStep     StepStringClass

          CODE
          ThisWindow.Run()

ThisWindow.Init  PROCEDURE()
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)
          BrowseSt.AddSortOrder(StStep,ST:StateCodeKey)
          BrowseSt.AddField(ST:STATECODE,BrowseSt.Q.ST:STATECODE)
          BrowseSt.AddField(ST:STATENAME,BrowseSt.Q.ST:STATENAME)
          SELF.SetAlerts()
          RETURN ReturnValue

          !declare BrowseSt object

          !declare StStep object

          !run the window procedure

          !initialize things

          !initialize the StStep object
          ! & plug in to the BrowseSt
          ! BrowseSt calls SetLimit to
          ! calculate data distribution
          ! from the poles of the data

ThisWindow.Kill  PROCEDURE()
ReturnValue     BYTE,AUTO
          CODE
          ReturnValue = PARENT.Kill()
          IF ReturnValue THEN RETURN ReturnValue.
          Relate:State.Close
          Relate:State.Kill
          GlobalErrors.Kill
          RETURN ReturnValue

          !shut down things

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 DFDNDE EFENETF FFFNFT' |
                    &'G GFGNGTH HFHNHTI IFINIT' |
                    &'J JFJNJTK KFKNKTL LFLNLT' |
                    &'M MFMNMTN NFNNNTD OFONOT' |
                    &'P PFPNPTQ QNR RFRNRTS SF' |
                    &'SNSTT TFTNTTU UFUNUTV VF' |
                    &'VNVTW WFWNWTX XFXNXTY YF' |
                    &'YNYTZ ZN')

Scroll:Name   STRING(' ALBAMEARNBAKBATBENBIABOBBRA' |
                    &'BROBUACACCARCENCHRCOECONCORCRU' |
                    &'DASDELDIADONDURELDEVFELFISFLO' |
                    &'FREFUTGARGIBGOLGOSGREGUTHAMHEM' |
                    &'HOBHOTINGJASJONKAGKEAKIRKORKYO' |
                    &'LATLEOLIGLOUMACMAQMARMAUMCKMER' |
                    &'MILMONMORNATNOLOKEPAGPAUPETPIN' |
                    &'PORPULRAUREYROBROSUBSALSASCASCH' |
                    &'SCRSHASIGSKISNASOUSTESTISUNTAY' |
                    &'TIRTUCVANWACWASWEIWIEMWOLYOR')
```

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-determinitive 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)

### SetLimit( *lower*, *upper* ), VIRTUAL

**SetLimit** Sets the StepStringClass object's evenly distributed steps between *upper* and *lower*.

*lower* A string constant, variable, EQUATE, or expression that specifies the StepStringClass object's lower boundary. The value may be numeric or alphanumeric.

*upper* 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 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, BrowseClass.ResetThumbLimits, ProcessClass.SetProgressLimits, StepClass.Controls

## SetLimitNeeded (return static/dynamic boundary flag)

### 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, BrowseClass.ResetThumbLimits

# 47 -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`)<sup>1</sup> 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.
```

<sup>1</sup>If the procedure has a `WindowManager` object, the `WindowManager` object takes the event (`WindowManager.TakeEvent`) and forwards it to the `ToolbarClass` object (`WindowManager.TakeAccepted`).

## 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:

ToolbarListboxClass	BrowseClass toolbar target
ToolbarReltreeClass	Reltree control toolbar target
ToolbarUpdateClass	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*.

## 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 FormVCRCControls 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.CLV	ToolbarClass method definitions

## 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()
ReturnValu        BYTE,AUTO
CODE
ReturnValu = 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 ReturnValu

ThisWindow.Kill    PROCEDURE()
ReturnValu        BYTE,AUTO
CODE
ReturnValu = PARENT.Kill()
Relate:Customer.Close
RETURN ReturnValu

ThisWindow.TakeSelected    PROCEDURE()
ReturnValu                BYTE,AUTO
Looped                    BYTE
CODE
LOOP
  IF Looped THEN RETURN Level:Notify ELSE Looped=1.
  ReturnValu = 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 ReturnValu
END

```

## ***ToolBarClass Methods***

The ToolBarClass contains the methods listed below.

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the ToolBarClass methods.

#### **Primary Interface Methods**

The primary interface 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 <sup>∨</sup>	shut down the ToolBarClass object

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

SetTarget	set active target & appropriate toolbar state
TakeEvent <sup>∨</sup>	process toolbar event for active target

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

DisplayButtons <sup>∨</sup>	enable appropriate toolbar buttons
-----------------------------	------------------------------------

<sup>∨</sup> 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* )

<b>AddTarget</b>	Adds a toolbar target to the ToolbarClass object's list of potential toolbar targets.
<i>target</i>	The label of a ToolbarTarget object.
<i>control</i>	An integer constant, variable, EQUATE, or expression containing the <i>target's</i> 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)

---

### 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



## 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,
END                                  ! 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)

**TakeEvent**( [ *vcr* ], *windowmanager* ), **VIRTUAL**

<b>TakeEvent</b>	Processes toolbar events for the active toolbar target.
<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 ToolbarTarget object does no “post processing” navigation.
<i>windowmanager</i>	The label of the ToolbarTarget object’s WindowManager object. See <i>WindowManager</i> 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



# 48 - 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.

## 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.

## 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:

<code>ABTOOLBA.INC</code>	<code>ToolbarListboxClass</code> declarations
<code>ABTOOLBA.CLW</code>	<code>ToolbarListboxClass</code> method definitions

## 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()
ReturnValu        BYTE,AUTO
CODE
ReturnValu = 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 ReturnValu

ThisWindow.Kill    PROCEDURE()
ReturnValu        BYTE,AUTO
CODE
ReturnValu = PARENT.Kill()
Relate:Customer.Close
RETURN ReturnValu

ThisWindow.TakeSelected    PROCEDURE()
ReturnValu                BYTE,AUTO
CODE
ReturnValu = 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 ReturnValu

```

## ***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)

#### 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)

**TakeEvent**( [ *vcr* ], *window manager* ), **VIRTUAL**

<b>TakeEvent</b>	Handles toolbar events for the <code>ToolbarListboxClass</code> object.
<i>vcr</i>	An integer variable to receive the control number of the accepted <i>vcr</i> button. This lets the <code>TakeEvent</code> method specify an appropriate subsequent action. If omitted, the <code>ToolbarListboxClass</code> object does no “post processing” navigation.
<i>windowmanager</i>	The label of the <code>ToolbarListboxClass</code> object’s <code>WindowManager</code> 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
    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
END
```

See Also: `TryTakeToolbar`, `ToolbarClass.SetTarget`

## TryTakeToolbar (return toolbar control indicator)

---

### 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
END
```

**See Also:** **TakeToolbar, ToolbarClass.SetTarget**



# 49 - 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.

## 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.

## 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	<code>ToolbarReltreeClass</code> declarations
ABTOOLBA.CLW	<code>ToolbarReltreeClass</code> method definitions

## 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

```

_ABCD11Mode_ 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
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)**

---

#### **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)

---

### 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`

# 50 - 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)<sup>1</sup> 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.
```

<sup>1</sup>If the procedure has a WindowManager object, the WindowManager object takes the event (WindowManager.TakeEvent) and forwards it to the ToolBarClass object (WindowManager.TakeAccepted).

## Relationship to Other Application Builder Classes

---

At present, the ABC Library has three classes derived from the `ToolbarTarget` class:

<code>ToolbarListboxClass</code>	BrowseClass toolbar target
<code>ToolbarReltreeClass</code>	Reltree control toolbar target
<code>ToolbarUpdateClass</code>	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.

## 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:

<code>ABTOOLBA.INC</code>	ToolbarTarget declarations
<code>ABTOOLBA.CLW</code>	ToolbarTarget method definitions

## ***ToolBarTarget Properties***

The ToolBarTarget contains the following 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.

## 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

The ToolBarTarget class contains the methods listed below.

### 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)

---

#### 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, ToolBarListboxClass.TakeToolBar, ToolBarRelTreeClass.TakeToolBar, ToolBarUpdateClass.TakeToolBar

## TakeEvent (convert toolbar events)

**TakeEvent**( [ *vcr* ], *window manager* ), **VIRTUAL**

<b>TakeEvent</b>	Process toolbar events for this toolbar target.
<i>vcr</i>	An integer variable to receive the control number of the accepted VCR navigation button. If omitted, the <i>ToolbarTarget</i> object does no “post processing” navigation.
<i>windowmanager</i>	The label of the <i>ToolbarTarget</i> object’s <i>WindowManager</i> object. See <i>Window Manager</i> 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)

---

### 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)

---

### 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`

# 51 - 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.

## 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.

## 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

## 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
  _ABCD11Mode_ EQUATE(0)
  _ABCLinkMode_ 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)**

#### **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)

**TakeEvent**( [ *vcr* ], *window manager* ), **VIRTUAL**

<b>TakeEvent</b>	Handles toolbar events for the <code>ToolbarUpdateClass</code> object.
<i>vcr</i>	An integer variable to receive the control number of the accepted VCR navigation button. This lets the <code>TakeEvent</code> method specify an appropriate subsequent action. If omitted, the <code>ToolbarUpdateClass</code> object does no “post processing” navigation.
<i>windowmanager</i>	The label of the <code>ToolbarUpdateClass</code> object’s <code>WindowManager</code> 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)

---

### 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)

### 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
END
```

**See Also:** TakeToolbar, ToolbarClass.SetTarget

# 52 - 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.

## 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*.

## 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 Properties***

The TranslatorClass contains the following properties:

### **ExtractText (identify text to translate)**

---

<b>ExtractText</b>	<b>CSTRING(File:MaxFilePath)</b>
--------------------	----------------------------------

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
                                                  !add local translation pairs
OPEN(MyWindow)
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 calls the TranslateProperty method to translate 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, TranslateProperty, TranslateString

## TranslateControls (translate text for range of controls)

**TranslateControls**( *first control*, *last control* [, *window*] ), **VIRTUAL**

**TranslateControls** Translates text for a range of controls.

<i>first control</i>	An integer constant, variable, EQUATE, or expression containing the control number of the first control to translate.
<i>last control</i>	An integer constant, variable, EQUATE, or expression containing the control number of the last control to translate.
<i>window</i>	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

## TranslateProperty (translate textual control property)

**TranslateProperty**( *property*, *control* [, *window*] ), **VIRTUAL**

**TranslateProperty** Translates textual property for a control.

<i>property</i>	An integer constant, variable, EQUATE, or expression containing the property EQUATE of the property to translate.
<i>control</i>	An integer constant, variable, EQUATE, or expression containing the control number of the control to translate.
<i>window</i>	The label of the APPLICATION, WINDOW, or REPORT to translate. If omitted, TranslateControl operates on the active target.

The **TranslateProperty** method translates the text for the specified *control property*, such as PROP:TIP or PROP:MSG. The AddTranslation method sets the translation values for the text.

Implementation: The TranslateControl method calls the TranslateProperty method for the specified control property. The TranslateProperty method calls the TranslateString method to translate MSG attribute text, TIP attribute text, and FORMAT attribute text.

Example:

```
MyTranslator.TranslateControl PROCEDURE(SHORT CtlID,<WINDOW Win>)
CtrlType USHORT,AUTO
CODE
CtrlType=Win$CtlId{PROP:Type}
SELF.TypeMapping.ControlType=CtrlType
GET(SELF.TypeMapping,SELF.TypeMapping.ControlType)
LOOP WHILE ~ERRORCODE() AND SELF.TypeMapping.ControlType=CtrlType
    SELF.TranslateProperty(SELF.TypeMapping.Property,CtlId,Win)
    GET(SELF.TypeMapping,POINTER(SELF.TypeMapping)+1)
END
```

See Also: AddTranslation, TranslateControl, TranslateString

## 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



# 53 - 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.

## 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.

## **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, the FileDropClass 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 these other classes. The BrowseBox control template instantiates the BrowseClass, the FileDrop control template instantiates the FileDropClass, and the Process and Report procedure templates instantiate the ProcessClass.

## **ViewManager Source Files**

---

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

ABFILE.INC	ViewManager declarations
ABFILE.CLW	ViewManager method definitions

## 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.Init            !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()
  OF Level:Notify
    POST(EVENT:CloseWindow)
    BREAK
  OF Level:Fatal
    POST(EVENT:CloseWindow)
    BREAK
  END
  CUS:ZIP=33065                          !process the record
  IF Relate:CUSTOMER.Update()
    BREAK                                !update customer & related files
  ELSE
    ProgressMsg = CLIP(CUS:Name)&' zip changed to '&CUS:ZIP
    DISPLAY(ProgressMsg)
  END
END
IF FIELD() = ?Cancel
  IF EVENT() = Event:Accepted
    POST(Event:CloseWindow)
  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

## 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.

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the ViewManager methods.

#### **Primary Interface Methods**

The primary interface 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 <sup>v</sup>	shut down the ViewManager object

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

Open <sup>v</sup>	open the VIEW
Next <sup>v</sup>	get the next element
Previous <sup>v</sup>	get the previous element
PrimeRecord	prepare a record for adding
ValidateRecord <sup>v</sup>	validate the current element
SetFilter <sup>v</sup>	specify a filter for the active sort order
SetSort <sup>v</sup>	set the active sort order
Close <sup>v</sup>	close the VIEW

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

SetOrder <sup>v</sup>	replace the active sort order
UseView	use LazyOpen files

<sup>v</sup> These methods are also Virtual.

## **Virtual Methods**

Typically, you will not call these methods directly—the Primary Interface 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* )

**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:

```
MyView.AddSortOrder(ORD:ByCustomer)           !sort by customer no
!program code
MyView.ApplyOrder                             !apply the order
MyView.Next()                                 !get next in specified order
```

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: AddRange, ApplyFilter, SetSort

## Close (close the view)

---

### Close, VIRTUAL

The **Close** method closes the managed VIEW.

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

```

## 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()
OF Level:Benign
  !process the record
  !& check for success
OF Level:Notify
  !write error log
  !& check for failure
OF Level:Fatal
  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)

### PrimeRecord( [*suppress clear* ] ), VIRTUAL

<b>PrimeRecord</b>	Prepares the VIEW's primary file record buffer to add a new record.
<i>suppress clear</i>	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, <i>suppress clear</i> 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 the VIEW'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**

## 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 with subsequent calls to SetFilter. If omitted, the filter gets a default id 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

<b>SetOrder</b>	Replaces the active sort order.
<i>expression list</i>	A string constant, variable, EQUATE, or expression that contains an ORDER attribute expression list. See the <i>Language Reference—ORDER</i> 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**

## SetSort (set the active sort order)

### SetSort( *sortnumber* ), VIRTUAL

**SetSort** Set the view's active sort order.

*sortnumber* An integer constant, variable, EQUATE, or expression that specifies the sort order to use. Sort orders are numbered in the sequence they are added by the AddSortOrder method.

The **SetSort** method sets the view's active sort order and returns a value indicating whether the active sort (*sortnumber*) changed.

Implementation: SetSort returns one (1) if the *sortnumber* changed; otherwise it returns zero (0).

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
    MESSAGE('New Sort Order')                      !acknowledge new order
END

```

See Also: AddSortOrder

## 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 (and is included in the result set); any other value indicates the item is invalid (and is filtered out of the result set).

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
END

```

See Also: **Next, Previous**

# 54 - 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—procedures that support record inserts, changes, and deletes. The WindowManager carries out the specified options for these update procedures (forms). For example see the CancelAction, ChangeAction, and DeleteAction properties.

### 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 all non-field events)
TakeAccepted	(handle all EVENT:Accepted events)
TakeRejected	(handle all EVENT:Rejected events)
TakeSelected	(handle all EVENT:Selected events)
TakeNewSelection	(handle all EVENT:NewSelection events)
TakeCompleted	(handle all EVENT:Completed events)
TakeCloseEvent	(handle all EVENT:Close events)
TakeFieldEvent	(handle all field specific events)

For example, if you want to intercept and process an event *before* any other ABC Library objects, then do so at the beginning of the TakeEvent method. If you want to process a field-specific event *after* the ABC Library is done with the event, then do so at the end of the TakeFieldEvent method.

Each “Take” method returns a value indicating how the ACCEPT loop processing should continue. 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. This concept is readily visible in this implementation of the WindowManager.Ask and the WindowManager.TakeEvent methods:

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

```
WindowManager.TakeEvent PROCEDURE
RVa1 BYTE(Level:Benign)
CODE
    IF ~FIELD()
        RVa1 = SELF.TakeWindowEvent()
        IF RVa1 THEN RETURN RVa1.
    END
    CASE EVENT()
    OF EVENT:Accepted
        RVa1 = SELF.TakeAccepted()
    OF EVENT:Rejected
        RVa1 = SELF.TakeRejected()
    OF EVENT:Selected
        RVa1 = SELF.TakeSelected()
    OF EVENT:NewSelection
        RVa1 = SELF.TakeNewSelection()
    END
    IF RVa1 THEN RETURN RVa1.
    IF FIELD()
        RVa1 = SELF.TakeFieldEvent()
    END
    RETURN RVa1
```

## 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.

## 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

## 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) ! Customer ID
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:
!insert records only
SELF.Request = InsertRecord
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
  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*.

## 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, Init, TakeCompleted, Request, Response

## OKControl (window acceptance control—OK button)

---

<b>OKControl</b>	<b>SIGNED</b>
------------------	---------------

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)

---

<b>Opened</b>	<b>BYTE</b>
---------------	-------------

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  
 ProcessRecord  
 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 the current record
ProcessRecord   EQUATE (5)  ! Process the current 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
	<p>The <b>Request</b> 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:</p> <p>InsertRecord            ChangeRecord            DeleteRecord            ProcessRecord            SelectRecord</p>
Implementation:	<p>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 <i>Conceptual Example</i>.</p> <p>EQUATEs for the OriginalRequest and Request properties are declared in \LIBSRC\TPLEQU.CLW as follows:</p> <pre>           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           ProcessRecord   EQUATE (5)  ! Process the current record         </pre> <p>See Also:           Init, OriginalRequest</p>

## ResetOnGainFocus (gain focus reset flag)

---

ResetOnGainFocus	BYTE
	<p>The <b>ResetOnGainFocus</b> 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).</p>
Implementation:	<p>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.</p>
See Also:	ForcedReset

## 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)

---

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)
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

```

## ***WindowManager Methods***

The WindowManager contains the following methods.

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the WindowManager methods.

#### **Primary Interface Methods**

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

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

Run <sup>∨</sup>	run this procedure
Init <sup>∨</sup>	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 <sup>∨</sup>	shut down the WindowManager object

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

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

Run <sup>∨</sup>	run another procedure
SaveHistory	save history fields for later restoration
PostCompleted	a virtual to prime fields

<sup>∨</sup> These methods are also Virtual.

#### **Virtual Methods**

Typically you will not call these methods directly—the Primary Interface 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* )

<b>AddHistoryField</b>	Adds a history field to the WindowManager object.
<i>control</i>	An integer constant, variable, EQUATE, or expression containing the control number of the control whose contents to restore from the <i>field</i> . This is the field equate number of the control.
<i>field</i>	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 <i>WHAT</i> and <i>WHERE</i> in the <i>Language Reference</i> 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* )

<b>AddHistoryFile</b>	Adds a history file to the WindowManager object.
<i>record buffer</i>	The label of the history file's RECORD.
<i>save buffer</i>	The label of a STATIC variable declared LIKE( <i>record buffer</i> ). 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:

AddHistoryField, HistoryKey, RestoreField, SaveHistory

## AddItem (program the WindowManager object)

```

AddItem( BrowseClass      | )
           | FileDropClass   |
           | ToolBarClass    |
           | ToolBarUpdateClass |
           | TranslatorClass  |
           | WindowResizeClass |
           | control, response |

```

<b>AddItem</b>	Adds specific functionality to the WindowManager.
<i>BrowseClass</i>	The label of a BrowseClass object.
<i>FileDropClass</i>	The label of a DropListClass object.
<i>ToolBarClass</i>	The label of a ToolBarClass object.
<i>ToolBarUpdateClass</i>	The label of a ToolBarUpdateClass object.
<i>TranslatorClass</i>	The label of a TranslatorClass object.
<i>WindowResizeClass</i>	The label of a WindowResizeClass object.
<i>control</i>	An integer constant, variable, EQUATE, or expression containing the control number of the control whose acceptance invokes the <i>response</i> —typically OK and Cancel buttons.
<i>response</i>	An integer constant, variable, EQUATE, or expression indicating the action to register when the <i>control</i> is accepted.

The **AddItem** method registers another ABC Library object with the WindowManager object to add the object's specific functionality to the WindowManager.

### 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

```

Example:

```
ThisWindow.Init PROCEDURE()                                !program the WindowManager
CODE
!procedure code
SELF.AddItem(ToolBar)                                     !add toolbar functionality
SELF.AddItem(ToolBarForm)                                !must follow AddItem(ToolBar)
SELF.AddItem(?Cancel,RequestCancelled)                   !add cancel button functionality
SELF.AddItem(Resizer)                                    !add window resize functionality
SELF.AddItem(Translator)                                 !add language translation functionality
MyBrowse.Init(?CusList,Cus:Q.Position,Cus:View,Cus:Q,Relate:Cus,ThisWindow)
!procedure code

MyBrowse.Init PROCEDURE|
(SIGNED ListBox,*STRING Posit,VIEW V,QUEUE Q,RelationManager F,WindowManager WM)
CODE
!procedurecode
WM.AddItem(SELF)                                         !add Browse functionality
```

See Also: **Response, 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)

### 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, or until an EVENT:CloseWindow occurs.

**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
END
END

```

See Also: Dead, Init, Kill, Run, TakeEvent

## 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**

## 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)

---

### 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
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
```

## 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
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

<b>RestoreField</b>	Restores the contents of the specified control.
<i>control</i>	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:

**AddHistoryFile, AddHistoryField, HistoryKey, RestoreField**

## 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 **ALRTs** 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 *ALRT* 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)

### 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
RVa1 BYTE(Level:Benign)
I    USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;  RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

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
RVa1 BYTE(Level:Benign)
I   USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;  RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

**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
RVa1 BYTE(Level:Benign)
I    USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;    RVa1 = SELF.TakeRejected()
OF EVENT:Selected;    RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;    RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

See Also: [InsertAction](#), [Request](#), [Response](#), [TakeEvent](#), [VCRRequest](#)

## TakeEvent (a virtual to process all events)

---

### 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)

### 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
RVa1 BYTE(Level:Benign)
I   USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;  RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

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
RVa1 BYTE(Level:Benign)
I    USHORT,AUTO
CODE
IF ~FIELD()
    RVa1 = SELF.TakeWindowEvent()
    IF RVa1 THEN RETURN RVa1.
END
CASE EVENT()
OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
OF EVENT:Completed;  RVa1 = SELF.TakeCompleted()
OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVa1 = SELF.TakeCloseEvent()
END
IF RVa1 THEN RETURN RVa1.
IF FIELD()
    RVa1 = SELF.TakeFieldEvent()
END
RETURN RVa1
```

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
RVa1 BYTE(Level:Benign)
I    USHORT,AUTO
    CODE
    IF ~FIELD()
        RVa1 = SELF.TakeWindowEvent()
        IF RVa1 THEN RETURN RVa1.
    END
    CASE EVENT()
    OF EVENT:Accepted;    RVa1 = SELF.TakeAccepted()
    OF EVENT:Rejected;   RVa1 = SELF.TakeRejected()
    OF EVENT:Selected;   RVa1 = SELF.TakeSelected()
    OF EVENT:NewSelection;RVa1 = SELF.TakeNewSelection()
    OF EVENT:Completed;  RVa1 = SELF.TakeCompleted()
    OF EVENT:CloseWindow OR OF EVENT:CloseDown
        RVa1 = SELF.TakeCloseEvent()
    END
    IF RVa1 THEN RETURN RVa1.
    IF FIELD()
        RVa1 = SELF.TakeFieldEvent()
    END
    RETURN RVa1
```

See Also: **TakeEvent**

## TakeWindowEvent (a virtual to process non-field events)

### 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
```

# 55 - 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.

## 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.

## 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

## 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('TopSpeed.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)**

---

<b>AutoTransparent</b>	<b>BYTE</b>
------------------------	-------------

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)**

---

<b>DeferMoves</b>	<b>BYTE</b>
-------------------	-------------

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.

### **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 primary interface and the virtual methods. This organization reflects what we believe is typical use of the WindowResizeClass methods.

#### **Primary Interface Methods**

The primary interface 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 <sup>v</sup>	resize and reposition all controls
---------------------	------------------------------------

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

SetParentControl	set control's parent
SetStrategy	set control's resize strategy

<sup>v</sup> These methods are also Virtual.

#### **Virtual Methods**

Typically you will not call these methods directly—the Primary Interface 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 AppStrategy:Resize, 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
    Resizer.Kill                             ! shut down Resizer object
OF EVENT:Sized
    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)
    . . .
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)
. . .
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**

## SetPosition (calculate control coordinates)

**SetPosition**( *control*, *parentpos*, *parentnewpos*, *pos*, *newpos*)

<b>SetPosition</b>	Calculates the control's new coordinates.
<i>control</i>	An integer constant, variable, EQUATE, or expression containing the control number (field equate) of the control whose coordinates are calculated.
<i>parentpos</i>	The label of the structure that contains the original (before resizing) coordinates of the <i>control's</i> parent.
<i>parentnewpos</i>	The label of the structure that contains the new (after resizing) coordinates of the <i>control's</i> parent.
<i>pos</i>	The label of the structure that contains the <i>control's</i> original (before resizing) coordinates.
<i>newpos</i>	The label of the structure that receives the <i>control's</i> newly calculated coordinates.

The **SetPosition** method calculates a control's new coordinates based on the control's resizing strategy and on the coordinates of the control's parent.

The *parentpos*, *parentnewpos*, *pos*, and *newpos* parameters must name a structure that begins like the PositionGroup structure declared in ABRESIZE.INC.

Implementation: The `Resize` method (indirectly) calls the `SetPosition` method for each control in the window. The `SetPosition` method sets the new control coordinates based on the resize strategy for each control.

The *parentpos*, *parentnewpos*, *pos*, and *newpos* parameters must name a structure that begins like the PositionGroup structure declared in ABRESIZE.INC as follows:

```

PositionGroup GROUP,TYPE      !Control coordinates
XPos          SIGNED          ! Horizontal coordinate
YPos          SIGNED          ! Vertical coordinate
Width         UNSIGNED        ! Width
Height        UNSIGNED        ! Height
END

```

Example:

```

MyWindowResizeClass.SetPosition PROCEDURE(SIGNED ControlID,PositionGroup ParentOrigPos,|
    PositionGroup ParentCurrentPos, PositionGroup OrigPos, PositionGroup NewPos)
CODE
IF NoResize THEN RETURN.                !conditionally disable the resize
IF ControlID{PROP:Type}=CREATE:Entry    ! For ENTRY controls
    ParentCurrentPos.Width=0{PROP:Width} ! tweak the coordinate calculation
END
PARENT.SetPosition(ControlID,ParentOrigPos,ParentCurrentPos,OrigPos,NewPos)

```

See Also: [Resize](#)

## 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**

# INDEX

## Symbols

- !, PopupClass.AddItemMimic ..... 595
  - %Error ..... 419
  - %ErrorCode ..... 419
  - %ErrorText ..... 419
  - %Field ..... 419
  - %File ..... 419
  - %FileError ..... 419
  - %FileErrorCode ..... 419
  - %Message ..... 419
  - %Previous ..... 419
  - %ProgramType ..... 73
  - .DLL ..... 52
  - .EXP ..... 65
  - .INI File Support ..... 48
  - .LIB ..... 109
  - 801 variable not bound ..... 121, 125, 148
- ## A
- ABC Coding Conventions ..... 193
  - ABC Compliant Classes ..... 64
  - ABC Library ..... 187
  - ABC Templates ..... 42
    - embed points ..... 65
  - ABC Templates and SQL ..... 46
  - Aborted Add/Change ..... 160
  - ACCEPT ..... 902
  - Accept browse control from Toolbar ..... 117, 156
  - Accept control from Toolbar ..... 138
  - Access:file ..... 192. *See* FileManager
  - Access:filename ..... 492
  - Action For Each Selection ..... 143
  - Action for Process ..... 98
  - Active Invisible ..... 57
  - ActiveInvisible
    - BrowseClass ..... 251
  - Add Extra Vertical Space ..... 84
  - AddEditControl
    - BrowseClass ..... 268
  - AddErrors
    - ErrorClass ..... 425
  - AddField
    - BrowseClass ..... 269
    - FileDropClass ..... 466
  - AddHistoryField
    - WindowManagerClass ..... 897
  - AddHistoryFile
    - WindowManagerClass ..... 898
  - AddItem
    - AsciiViewerClass ..... 236
    - ConstantClass ..... 322
    - FieldPairsClass ..... 450
    - PopupClass ..... 592
    - ProcessClass ..... 645
    - QueryClass ..... 659
    - StepCustomClass ..... 750
    - WindowManagerClass ..... 899
  - AddItemEvent
    - PopupClass ..... 594
  - AddItemMimic
    - PopupClass ..... 595
  - Additional Sort Fields ..... 99, 103
  - AddKey
    - FileManagerClass ..... 505
  - AddLocator
    - BrowseClass ..... 270
  - AddMask
    - SelectFileClass ..... 734
  - AddMenu
    - PopupClass ..... 596
  - AddPair
    - BufferedPairsClass ..... 312
    - FieldPairsClass ..... 451
  - AddRange
    - ViewManagerClass ..... 860
  - AddRelation
    - RelationManagerClass ..... 697
  - AddRelationLink
    - RelationManagerClass ..... 699
  - AddResetField
    - BrowseClass ..... 271
  - AddSortOrder
    - BrowseClass ..... 272
    - ViewManagerClass ..... 861
  - AddSubMenu
    - PopupClass ..... 598
  - AddTarget
    - ToolbarClass ..... 796
  - AddToolbarTarget
    - BrowseClass ..... 273
  - AddTranslation
    - TranslatorClass ..... 842

AddUpdateField		
FileDropClass .....	467	
AddUpdateFile		
WindowManagerClass .....	901	
AddValue		
EditMultiSelectClass .....	402	
After successful insert .....	160	
Alias Options .....	83	
AliasedFile		
FileManagerClass .....	496	
Allow Unfilled .....	57	
Allow User Variable Zooms? .....	105	
AllowUnfilled		
BrowseClass .....	251	
AllowUserZoom		
PrintPreviewClass .....	620	
AppendOrder		
ViewManagerClass .....	862	
Application		
deployment .....	65	
Application Builder Class (ABC) Library .....	187	
Application Builder Class Viewer .....	55, 61	
application defaults .....	47	
application modal windows .....	169	
Application Wizard .....	72	
control model .....	74	
Full vs Simple .....	72	
Generate Reports for each file .....	75	
Overwrite existing procedures .....	75	
SQL Applications .....	73	
Toolbar .....	74	
ApplyFilter		
ViewManagerClass .....	862	
ApplyOrder		
ViewManagerClass .....	863	
ApplyRange		
BrowseClass .....	274	
ViewManagerClass .....	863	
Approx Record Count .....	99, 103	
Array		
Definition .....	63	
ArrowAction		
BrowseClass .....	252	
ASCII viewer .....	107	
ASCIIFile		
ASCIIFileClass .....	202	
ASCIIFileClass .....	229	
ASCIIFile .....	202	
ErrorMgr .....	202	
FormatLine .....	204	
GetDOSFilename .....	205	
GetFilename .....	206	
GetLastLineNo .....	206	
GetLine .....	207	
GetPercentile .....	208	
Init .....	209	
Kill .....	210	
methods .....	203	
OpenMode .....	202	
properties .....	202	
Reset .....	211	
SetLine .....	212	
SetPercentile .....	213	
ValidateLine .....	214	
ASCIIPrintButton .....	113	
ASCIIPrintClass .....	229	
Ask .....	219	
Init .....	219	
methods .....	219	
PrintLines .....	220	
properties .....	218	
AsciiPrintClass		
FileMgr .....	218	
PrintPreview .....	218	
Translator .....	218	
ASCIISearchButton .....	113	
ASCIISearchClass .....	230	
Ask .....	226	
FileMgr .....	224	
Find .....	224	
Init .....	227	
LineCounter .....	224	
methods .....	226	
Next .....	227	
properties .....	224	
Setup .....	228	
Translator .....	225	
ASCIIViewControl .....	112	
AsciiViewControl		
classes .....	113	
ASCIIViewerClass .....	229	
AskGotoLine .....	237	
DisplayPage .....	237	
Init .....	238	
Kill .....	240	
methods .....	234	
PageDown .....	241	
PageUp .....	241	
Popup .....	232	
Printer .....	232	
properties .....	232	
Reset .....	242	
Searcher .....	232	
SetLine .....	243	

- SetLineRelative ..... 244
- SetTranslator ..... 245
- TakeEvent ..... 246
- TopLine ..... 233
- AsciiViewerClass
  - AddItem ..... 236
- AsciiViewInListBox ..... 174
- Ask
  - ASCIIPrintClass ..... 219
  - ASCIISearchClass ..... 226
  - BrowseClass ..... 275
  - FileDropComboClass ..... 483
  - PopupClass ..... 600
  - QueryClass ..... 660
  - QueryFormClass ..... 675
  - ReportManagerClass ..... 721
  - SelectFileClass ..... 735
  - WindowManagerClass ..... 902
- Ask the user before adding another record ..... 161
- AskGotoLine
  - ASCIIViewerClass ..... 237
- AskPage
  - PrintPreviewClass ..... 624
- AskPreview
  - ReportManagerClass ..... 721
- AskProcedure
  - BrowseClass ..... 252
- AskRecord
  - BrowseClass ..... 276
- AskThumbnails
  - PrintPreviewClass ..... 625
- Assign as Reference? ..... 173
- AssignBufferToLeft
  - BufferedPairsClass ..... 313
- AssignBufferToRight
  - BufferedPairsClass ..... 313
- AssignLeftToBuffer
  - BufferedPairsClass ..... 314
- AssignLeftToRight
  - FieldPairsClass ..... 452
- AssignRightToBuffer
  - BufferedPairsClass ..... 314
- AssignRightToLeft
  - FieldPairsClass ..... 453
- Auto Tool Bar ..... 58
- autoincrement ..... 507, 528, 531
- Automatically find parent controls ..... 58, 184
- AutoRefresh
  - WindowManagerClass ..... 885
- AutoToolbar
  - WindowManagerClass ..... 885
- AutoTransparent
  - WindowResizeClass ..... 930
- Available
  - EditMultiSelectClass ..... 399
- B**
  - background processes
    - ReportManagerClass ..... 718
  - Base Class ..... 61
  - base class configuration ..... 60
  - Batch Process ..... 97
  - begins with, Filter Locator ..... 557
  - BIND ..... 101, 105, 121, 125, 127, 139, 145, 148, 151
  - BindFields
    - FileManagerClass ..... 506
  - Browse
    - ToolbarListBoxClass ..... 805
  - Browse Box Behavior ..... 93
  - Browse, initial position ..... 263
  - Browse Optimization, ISAM ..... 57
  - Browse Optimization, SQL ..... 57
  - Browse Template ..... 92
  - Browse Wizard ..... 76
  - Browse-Form Application Paradigm ..... 45
  - BrowseBox ..... 114
    - Accept browse control from Toolbar ..... 117
    - classes ..... 127, 134
    - colors ..... 125
    - conditional behavior ..... 124
    - Entry Locator ..... 118
    - Filter Locator ..... 119
    - filtering ..... 130
    - filtering and record selection ..... 121
    - Find Anywhere ..... 120
    - Icons ..... 126
    - Incremental Locator ..... 119
    - Loading Method ..... 116
    - Locator Behavior ..... 118
    - print single record ..... 127
    - QBE ..... 129
    - refreshing ..... 122
    - scroll bars ..... 115
    - scrolling ..... 122
    - selecting items ..... 132
    - sorting ..... 122
    - Step Locator ..... 118
    - totals ..... 127
    - updating records ..... 133
  - browsebox navigation ..... 118
  - BrowseBox Options ..... 116
  - BrowseBox, reset ..... 267
  - BrowseClass ..... 247
    - ActiveInvisible ..... 251

AddEditControl .....	268	SetAlerts .....	294
AddField .....	269	SetQueueRecord .....	294
AddLocator .....	270	SetSort .....	
AddResetField .....	271	Sort .....	262
AddSortOrder .....	272	StartAtCurrent .....	263
AddToolBarTarget .....	273	TabAction .....	263
AllowUnfilled .....	251	TakeAcceptedLocator .....	296
ApplyRange .....	274	TakeEvent .....	297
ArrowAction .....	252	TakeKey .....	298
Ask .....	275, 276	TakeLocate .....	298
AskProcedure .....	252	TakeNewSelection .....	299
ChangeControl .....	253	TakeScroll .....	300
DeleteControl .....	253	TakeVCRScroll .....	301
EditList .....	254	ToolBar .....	264
EnterAction .....	254	ToolBarItem .....	264
Fetch .....	278	ToolControl .....	265
Fields .....	255	UpdateBuffer .....	302
FocusLossAction .....	255	UpdateQuery .....	303
HasThumb .....	256	UpdateResets .....	304
HideSelect .....	256	UpdateThumb .....	304
Init .....	279	UpdateThumbFixed .....	305
InsertControl .....	256	UpdateViewRecord .....	305
Kill .....	280	UpdateWindow .....	306
ListControl .....	257	Window .....	265
ListQueue .....	257	BrowseClass Configuration .....	57
Loaded .....	257	BrowsePrintButton .....	127
methods .....	266	BrowsePublishButton .....	128
Next .....	281	BrowseQueryButton .....	129
Popup .....	257	BrowseSelectButton .....	132
PostNewSelection .....	281	BrowseToolboxButton .....	132
Previous .....	282	BrowseUpdateButtons .....	133
PrintControl .....	258	BRWn .....	192. See BrowseClass
PrintProcedure .....	258	BRWn::Sortn:Locator .....	192. See LocatorClass
properties .....	251	BRWn::Sortn:StepClass .....	192. See StepClass
Query .....	258	Buffer .....	
QueryControl .....	259	FileManagerClass .....	496
QueryShared .....	259	buffer management .....	445
QuickScan .....	260	BufferedPairsClass .....	
Records .....	282	AddPair .....	312
ResetFromAsk .....	283	AssignBufferToLeft .....	313
ResetFromBuffer .....	285	AssignBufferToRight .....	313
ResetFromFile .....	286	AssignLeftToBuffer .....	314
ResetFromView .....	287	AssignRightToBuffer .....	314
ResetQueue .....	288	EqualLeftBuffer .....	315
ResetResets .....	289	EqualRightBuffer .....	315
ResetSort .....	290	Init .....	316
RetainRow .....	260	Kill .....	316
ScrollEnd .....	291	methods .....	310
ScrollOne .....	292	properties .....	309
ScrollPage .....	293	RealList .....	309
SelectControl .....	261	Buffers .....	
Selecting .....	261	FileManagerClass .....	497

- Build Menu From ..... 168
- C**
- CallABCMethod ..... 165
- CallProcedureAsLookup ..... 166
- Cancel without Confirming ..... 160
- CancelAction
  - WindowManagerClass ..... 886
- CancelAutoInc
  - FileManagerClass ..... 507
  - RelationManagerClass ..... 700
- CancelButton ..... 141, 162, 163
- Case Sensitive matches? ..... 151
- Change Message ..... 159
- ChangeAction
  - WindowManagerClass ..... 886
- ChangeButton
  - ToolBarTargetClass ..... 819
- ChangeControl
  - BrowseClass ..... 253
- ChangeRecord ..... 89
- Changes ..... 159
- ChildRead
  - ProcessClass ..... 641
- choose a record from a list box ..... 132
- class configuration ..... 60
- class library ..... 187
- Class Viewer ..... 55, 61
- Classes tab ..... 60
  - WindowManager ..... 92
- ClearKey
  - FileManagerClass ..... 509
- ClearKeycode
  - PopupClass ..... 590
- ClearLeft
  - FieldPairsClass ..... 454
- ClearRight
  - FieldPairsClass ..... 455
- Close
  - FileManagerClass ..... 511
  - RelationManagerClass ..... 701
  - ViewManagerClass ..... 864
- Close Button control template ..... 141
- Close when the user clicks on the splash window ..... 107
- CloseCurrentWindow ..... 166
- CODE ..... 66
- Code After, Canceled ..... 166
- Code After, Completed ..... 166
- Code before ..... 166
- Code Embed ..... 62
- Code Generation ..... 42
- Code Generation Wizards ..... 69
- Code template ..... 165
- Colorization, Browse Box ..... 125
- Colorization, FileDrop ..... 146, 152
- Colorization, Tree control ..... 137
- Colors
  - relation tree ..... 139
- Conditional Behavior
  - BrowseBox ..... 124
- configuration files ..... 48
- Confirm Cancel ..... 160
- confirmation of delete ..... 160
- ConstantClass
  - AddItem ..... 322
  - Init ..... 323
  - Kill ..... 324
  - Load ..... 325
  - Next ..... 326
  - Reset ..... 327
  - Set ..... 328
- ConstantClass Methods ..... 321
- ConstantClass Properties ..... 320
- context menu ..... 167
- Contract Branch ..... 138
- Contracting Locator ..... 119
- Control
  - LocatorClass ..... 581
  - ToolBarTargetClass ..... 819
- control file ..... 49
- control model
  - Application Wizard ..... 74
- Control Template ..... 111
- Control Templates
  - adding ..... 111
- Control templates ..... 88
- Controls
  - StepClass ..... 741
- ControlValueValidation ..... 166
- Conventions
  - documentation ..... 38, 196
- Core Classes ..... 188
- CREATE
  - attribute ..... 51
- Create
  - FileManagerClass ..... 497
- CreateControl
  - EditCheckClass ..... 335, 363, 371
  - EditClass ..... 344
  - EditColorClass ..... 355, 380
  - EditFontClass ..... 390
  - EditMultiSelectClass ..... 403
- CurrentPage

PrintPreviewClass .....	620	deploying application files .....	65
<b>D</b>		Derive Classes .....	67
data		Derive? .....	61
external .....	48	derived class configuration .....	60
Data Dictionary		Deriving with Classes Tab .....	68
Printing .....	82	Deriving with Embed Points .....	68
Data Embed .....	62	Dictionary Print Wizard .....	82
Data Integrity .....	50	DIM .....	63
Database Operations .....	491	dimensions .....	63
FileManagerClass .....	502	Display	
Date		PrintPreviewClass .....	626
display template .....	175	Display Control .....	176
DateTimeDisplay .....	175	Display in Window .....	175
Day of Week .....	175	Display Record Identifier on the Title Bar .....	161
Dead		Display Time (in seconds) .....	107
WindowManagerClass .....	887	DisplayButtons	
declaring files .....	50	ToolBarClass .....	796
Default Classes .....	56	ToolBarListBoxClass .....	806
Default Directory .....	142	ToolBarReltreeClass .....	815
Default Filename .....	143	ToolBarTargetClass .....	822
Default Icon .....	147, 153	ToolBarUpdateClass .....	833
Default message .....	48	DisplayPage	
Default to first entry if USE variable empty .....	145, 151	ASCIIViewerClass .....	237
Default Window Controls .....	85	DisplayPopupMenu .....	167
DefaultDirectory		Ditto Button .....	157
SelectFileClass .....	733	Ditto Key .....	160
DefaultFile		DLL .....	52
SelectFileClass .....	733	DLL attribute .....	48
DefaultFill		Do Not Auto-Populate This Aliased File .....	83
FileDropClass .....	464	Do Not Auto-Populate This Field .....	83
defaults		Do Not Auto-Populate This File .....	83
application wide .....	47	Do Not Auto-Populate This Key .....	84
Defer opening files until accessed .....	53	Do Not Validate .....	178
DeferMoves		dockable toolbars .....	132
WindowResizeClass .....	930	Documentation Conventions .....	38, 196
DeferOpenReport		Don't Alter Controls .....	181
ReportManagerClass .....	715	DOS File Lookup Control template .....	107
DeferWindow		DOS Filename Variable .....	142
ReportManagerClass .....	715	DOSFileLookup .....	141
Delete		Drop down list .....	144, 150
RelationManagerClass .....	702	<b>E</b>	
Delete Message .....	160	Edit in place .....	133
DeleteAction		EditCheckClass .....	329
WindowManagerClass .....	887	CreateControl .....	335, 363, 371
DeleteControl		methods .....	334
BrowseClass .....	253	properties .....	333
DeleteItem		EditClass .....	337
PopupClass .....	601	CreateControl .....	344
DeleteRecord .....	89	FEQ .....	342
Deletes .....	159	Init .....	345

- Kill ..... 345
- methods ..... 345
- SetAlerts ..... 346, 364
- TakeEvent ..... 347
- EditColorClass ..... 349
- CreateControl ..... 355, 380
- methods ..... 354
- properties ..... 353
- TakeEvent ..... 356, 381
- Title ..... 353
- EditDropListClass ..... 357
- methods ..... 362
- properties ..... 361
- EditEntryClass ..... 365
- methods ..... 370
- properties ..... 369
- EditFileClass ..... 373
- FileMask ..... 377
- FilePattern ..... 377
- methods ..... 379
- properties ..... 377
- Title ..... 378
- EditFontClass ..... 383
- CreateControl ..... 390
- methods ..... 389
- properties ..... 388
- TakeEvent ..... 391
- Title ..... 388
- EDITINPLACE ..... 83
- EditInPlace::field ..... 191, 192. See EditClass
- EditList
- BrowseClass ..... 254
- EditMultiSelectClass ..... 393
- AddValue ..... 402
- Available ..... 399
- CreateControl ..... 403
- FilePattern ..... 399
- methods ..... 400
- properties ..... 399
- Reset ..... 403
- Selected ..... 399
- TakeAction ..... 404
- TakeEvent ..... 407
- Title ..... 399
- Embed Points ..... 41
- Embedded source
- comments ..... 48
- finding ..... 48
- Embedded Source dialog ..... 65
- Enable Run-Time Translation ..... 48
- Enclose RI code in transaction frame ..... 50
- EnterAction
- BrowseClass ..... 254
- Entries
- StepCustomClass ..... 749
- Entry locator ..... 118
- EntryCompletion
- FileDropComboClass ..... 480
- EntryLocatorClass
- Init ..... 413
- methods ..... 413
- properties ..... 412
- Set ..... 414
- Shadow ..... 412
- TakeAccepted ..... 414
- TakeKey ..... 415
- Update ..... 416
- UpdateWindow ..... 416
- Equal
- FieldPairsClass ..... 456
- EqualBuffer
- FileManagerClass ..... 512
- EqualLeftBuffer
- BufferedPairsClass ..... 315
- EqualLeftRight
- FieldPairsClass ..... 456
- EqualRightBuffer
- BufferedPairsClass ..... 315
- error severity ..... 417
- Error Treatment ..... 417
- ErrorClass ..... 417
- AddErrors ..... 425
- Errors ..... 422
- FieldName ..... 423
- FileName ..... 423
- Init ..... 426
- Kill ..... 426
- macros ..... 419
- Message ..... 427
- MessageText ..... 423
- methods ..... 424
- properties ..... 422
- RemoveErrors ..... 428
- SetErrors ..... 429
- SetFatality ..... 430
- SetField ..... 431
- SetFile ..... 431
- SetId ..... 432
- SubsString ..... 433
- TakeBenign ..... 434
- TakeError ..... 435
- TakeFatal ..... 436
- TakeNotify ..... 437
- TakeOther ..... 438

TakeProgram .....	439	AssignLeftToRight .....	452
TakeUser .....	440	AssignRightToLeft .....	453
Throw .....	441	ClearLeft .....	454
ThrowFile .....	442	ClearRight .....	455
ThrowMessage .....	443	Equal .....	456
ErrorMgr		EqualLeftRight .....	456
ASCIIFileClass .....	202	Init .....	457
Errors		Kill .....	457
ErrorClass .....	422	List .....	448
WindowManagerClass .....	888	methods .....	449
evaluating expressions .....	125, 148	properties .....	448
Event Processing, ABC .....	878	Fields	
Exclude unfiltered .....	106	BrowseClass .....	255
Expand Branch .....	138	printing .....	82
expanding list .....	136	File	
ExtendProgressWindow .....	176	FileManagerClass .....	498
Extension templates .....	174	File Control .....	49
EXTERNAL .....	52	File Dialog Header .....	142
EXTERNAL attribute .....	48	file handling .....	51
External Globals and ABC's Source Module .....	48	File Manager and Threaded Files .....	492
External library .....	109	File Manager Class .....	491
External Module Options .....	54	File Mask .....	143
External Template .....	109	File Mask Description .....	143
Extract Filename .....	59	File Masks .....	143
ExtractText		File Open Mode .....	53
TranslatorClass .....	841	File Options .....	83
<b>F</b>		File Overrides .....	53
FDBn .....	192. See FileDropClass	file processing	
FDCBn .....	192. See FileDropComboClass	multiple files .....	141
FEQ .....	345	File selection .....	141
EditClass .....	342	file sharing .....	51, 53
Fetch		FileDrop	
BrowseClass .....	278	colors .....	146, 152
FileManagerClass .....	513	icons .....	147, 153
INIClass .....	570	range limit .....	146, 151
FetchField		FileDrop control template .....	144, 150
INIClass .....	572	FileDropClass	
FetchQueue		AddField .....	466
INIClass .....	573	AddUpdateField .....	467
Field History Key .....	160	DefaultFill .....	464
Field Options .....	83	Init .....	468
Field Pairs Classes .....	445	InitSyncPair .....	464
Field Priming .....	159	Kill .....	469
Field to Fill From .....	145, 150	methods .....	465
FieldLookupButton .....	144	properties .....	464
FieldName		ResetQueue .....	470
ErrorClass .....	423	SetQueueRecord .....	471
FieldPairsClass .....	445	TakeEvent .....	471
AddItem .....	450	TakeNewSelection .....	472
AddPair .....	451	ValidateRecord .....	473
		FileDropCombo	
		Classes .....	154

- Hot Fields ..... 154
- Sort Fields ..... 149, 154
- Update Behavior ..... 153
- FileDropComboClass
  - Ask ..... 483
  - EntryCompletion ..... 480
  - GetQueueMatch ..... 484
  - Init ..... 485
  - methods ..... 481
  - properties ..... 480
  - ResetQueue ..... 487
  - TakeEvent ..... 488
  - TakeNewSelection ..... 489
  - UseField ..... 480
- FileLookupN ..... 192. See SelectFileClass
- FileManager
  - methods ..... 502
  - properties ..... 496
- FileManagerClass ..... 51
  - AddKey ..... 505
  - AliasedFile ..... 496
  - BindFields ..... 506
  - Buffer ..... 496
  - Buffers ..... 497
  - CancelAutoInc ..... 507
  - ClearKey ..... 509
  - Close ..... 511
  - Create ..... 497
  - Database Operations ..... 502
  - EqualBuffer ..... 512
  - Fetch ..... 513
  - File ..... 498
  - FileName ..... 498
  - FileNameValue ..... 499
  - GetComponents ..... 514
  - GetEOF ..... 515
  - GetError ..... 516
  - GetField ..... 517
  - GetFieldName ..... 518
  - GetName ..... 519
  - Init ..... 520
  - Insert ..... 521
  - Interactive Database Operations ..... 502
  - KeyToOrder ..... 522
  - Kill ..... 523
  - LazyOpen ..... 500
  - LockRecover ..... 500
  - Next ..... 524
  - Open ..... 525
  - OpenMode ..... 501
  - Position ..... 526
  - Previous ..... 527
  - PrimeAutoInc ..... 528
  - PrimeFields ..... 530
  - PrimeRecord ..... 531
  - RestoreBuffer ..... 533
  - RestoreFile ..... 534
  - SaveBuffer ..... 535
  - SaveFile ..... 536
  - SetError ..... 537
  - SetKey ..... 537
  - SetName ..... 538
  - Silent Database Operations ..... 502
  - SkipHeldRecords ..... 501
  - Throw ..... 539
  - ThrowMessage ..... 540
  - TryFetch ..... 541
  - TryInsert ..... 542
  - TryNext ..... 543
  - TryOpen ..... 544
  - TryPrevious ..... 545
  - TryPrimeAutoInc ..... 546
  - TryReget ..... 548
  - TryUpdate ..... 548
  - Update ..... 549
  - UseFile ..... 549
  - ValidateField ..... 550
  - ValidateFields ..... 551
  - ValidateRecord ..... 552
- FileMask
  - EditFileClass ..... 377
- FileMgr
  - AsciiPrintClass ..... 218
  - ASCIISearchClass ..... 224
- FileName
  - ErrorClass ..... 423
  - FileManagerClass ..... 498
  - INIClass ..... 569
- Filename Variable ..... 142
- FileNameValue
  - FileManagerClass ..... 499
- FilePattern
  - EditFileClass ..... 377
  - EditMultiSelectClass ..... 399
- Files
  - Creating ..... 51
  - printing ..... 82
- files
  - declaring ..... 50
  - sharing ..... 49
- Filter ..... 98, 103, 121
  - FileDrop ..... 145
  - FileDropCombo ..... 150
  - relation tree ..... 139

Filter locator .....	119	GetError	
filtering browses and reports .....	876	FileManagerClass .....	516
FilterLocatorClass		GetField	
FloatRight .....	557	FileManagerClass .....	517
methods .....	558	GetFieldName	
properties .....	557	FileManagerClass .....	518
Reset .....	584	GetFilename	
TakeAccepted .....	558	ASCIIFileClass .....	206
UpdateWindow .....	559	GetFilter	
Filters .....	105	QueryClass .....	661
Find .....	118	GetFreeElementName	
ASCIISearchClass .....	224	ViewManagerClass .....	864
find .....	113	GetFreeElementPosition	
Find Anywhere .....	120	ViewManagerClass .....	865
find record .....	156	GETINI .....	49
FirstField		GetItemChecked	
WindowManagerClass .....	888	PopupClass .....	602
Fixed Thumb .....	122	GetItemEnabled	
Flags		PopupClass .....	603
SelectFileClass .....	733	GetLastLineNo	
flat file applications .....	70	ASCIIFileClass .....	206
FloatRight		GetLastSelection	
FilterLocatorClass .....	557	PopupClass .....	603
FocusLossAction		GetLimit	
BrowseClass .....	255	QueryClass .....	663
ForcedReset		GetLine	
WindowManagerClass .....	888	ASCIIFileClass .....	207
Form Tab .....	84	GetName	
Form Template .....	93	FileManagerClass .....	519
Form Wizard .....	78	GetParentControl	
FormatLine		WindowResizeClass .....	932
ASCIIFileClass .....	204	GetPercentile	
FormVCRControls .....	177	ASCIIFileClass .....	208
Frame Template .....	95	StepClass .....	742
FrameBrowseControl .....	118, 138, 155, 177	StepCustomClass .....	751
free element .....	124	StepLongClass .....	765
FreeElement		StepRealClass .....	773
LocatorClass .....	581	StepStringClass .....	784
<b>G</b>		GetPositionStrategy	
Generate EMBED Comments .....	48	WindowResizeClass .....	933
Generate Reports for each file		GetQueueMatch	
Application Wizard .....	75	FileDropComboClass .....	484
Generate Template global data and ABC's as EXTERNA .....	48	GetResizeStrategy	
generating source code .....	42	WindowResizeClass .....	934
GetComponents		GetValue	
FileManagerClass .....	514	StepClass .....	742
GetDOSFilename		StepCustomClass .....	752
ASCIIFileClass .....	205	StepLongClass .....	766
GetEOF		StepRealClass .....	774
FileManagerClass .....	515	StepStringClass .....	785
		Give option to expand and contract all levels .....	138
		Global Data .....	65
		Global Data, saving and restoring .....	49

- global INCLUDEs ..... 65
  - global MAP ..... 65
  - Global Properties dialog ..... 47
  - GlobalErrors ..... 192. *See* ErrorClass
  - GlobalRequest ..... 89
  - GlobalResponse ..... 89
- ## H
- hand code ..... 109
  - HasThumb
    - BrowseClass ..... 256
  - Help File ..... 73
  - HelpButton
    - ToolBarTargetClass ..... 820
  - Hide:Access:filename ..... 492
  - Hide:Relate:filename ..... 493, 690
  - HideSelect
    - BrowseClass ..... 256
  - hierarchical list ..... 136
  - High
    - StepLongClass ..... 764
    - StepRealClass ..... 772
  - History
    - ToolBarUpdateClass ..... 832
  - HistoryKey
    - WindowManagerClass ..... 889
  - Horizontal Positional Strategy ..... 171, 183
  - Horizontal Resize Strategy ..... 170, 183
  - Hot Fields ..... 101, 104, 105, 124, 145, 148, 151
    - FileDrop Control Template ..... 154
  - HTML code generation
    - publish BrowseBox ..... 128
- ## I
- Icon
    - Browse Box ..... 126
    - FileDrop ..... 147, 153
  - Icons
    - relation tree ..... 139
  - Include File ..... 61
  - Incremental Locator ..... 119
  - IncrementalLocatorClass
    - methods ..... 565
    - properties ..... 565
    - SetAlerts ..... 565
    - TakeKey ..... 566
  - Individual File Overrides ..... 53
  - INI file
    - location ..... 49
  - INI File Settings ..... 92
  - INI File Support ..... 48
  - INIClass ..... 567
    - Fetch ..... 570
    - FetchField ..... 572
    - FetchQueue ..... 573
    - FileName ..... 569
    - Init ..... 574
    - methods ..... 570
    - properties ..... 569
    - TryFetch ..... 575
    - TryFetchField ..... 576
    - Update ..... 577
  - INIMgr ..... 192. *See* INIClass
  - Init
    - ASCIIFileClass ..... 209
    - ASCIIPrintClass ..... 219
    - ASCIISeachClass ..... 227
    - ASCIIViewerClass ..... 238
    - BrowseClass ..... 279
    - BufferedPairsClass ..... 316
    - ConstantClass ..... 323
    - EditClass ..... 345
    - EntryLocatorClass ..... 413
    - ErrorClass ..... 426
    - FieldPairsClass ..... 457
    - FileDropClass ..... 468
    - FileDropComboClass ..... 485
    - FileManagerClass ..... 520
    - INIClass ..... 574
    - LocatorClass ..... 583
    - PopupClass ..... 604
    - PrintPreviewClass ..... 628
    - ProcessClass ..... 646
    - QueryClass ..... 664
    - QueryFormClass ..... 676
    - QueryFormVisual ..... 685
    - RelationManagerClass ..... 703
    - ReportManagerClass ..... 722
    - SelectFileClass ..... 736
    - StepClass ..... 743
    - StepCustomClass ..... 753
    - StepStringClass ..... 786
    - ToolBarClass ..... 797
    - TranslatorClass ..... 844
    - ViewManagerClass ..... 866
    - WindowManagerClass ..... 903
    - WindowResizeClass ..... 935
  - Initial Window Position ..... 105
  - Initial Zoom ..... 104
  - Initializing Fields ..... 159
  - Initiate Thread ..... 169
  - InitSyncPair
    - FileDropClass ..... 464

Insert		StepCustomClass .....	753
FileManagerClass .....	521	StepStringClass .....	787
Insert Message .....	159	ToolBarClass .....	797
InsertAction		TranslatorClass .....	844
WindowManagerClass .....	889	ViewManagerClass .....	867
InsertButton		WindowManagerClass .....	905
ToolBarTargetClass .....	820	WindowResizeClass .....	937
InsertControl			
BrowseClass .....	256	<b>L</b>	
InsertRecord .....	89	language, multiple language user interface .....	48
Inserts .....	159	large files	
Inter-Procedure Communication .....	89	browsing .....	114
Interactive Database Operations .....	491	scrolling .....	115
FileManagerClass .....	502	LazyOpen .....	53
internationalization .....	48	FileManagerClass .....	500
Internet		Level:Benign .....	418
publish BrowseBox .....	128	Level:Cancel .....	418
invoice reports .....	178	Level:Fatal .....	194, 418
Is a Reference .....	63	Level:Notify .....	194, 418
Issue Template warning if LOGOUT() not allowed .....	51	Level:Program .....	418
Item Properties .....	168	Level:User .....	194, 418
<b>K</b>		LineCounter	
Keep View synchronized with Selection? .....	151	ASCIISearchClass .....	224
KeepVisible		LINK .....	64
ReportManagerClass .....	716	LIST .....	114
Key Distribution .....	123	List	
Key Options .....	84	FieldPairsClass .....	448
Keys		list box navigation .....	118, 138
printing .....	82	List to use .....	174
KeyToOrder		ListControl	
FileManagerClass .....	522	BrowseClass .....	257
Kill		ListLinkingFields	
ASCIIFileClass .....	210	RelationManagerClass .....	705
ASCIIViewerClass .....	240	ListQueue	
BrowseClass .....	280	BrowseClass .....	257
BufferedPairsClass .....	316	Load	
ConstantClass .....	324	ConstantClass .....	325
EditClass .....	345	Loaded	
ErrorClass .....	426	BrowseClass .....	257
FieldPairsClass .....	457	Loading Method, BrowseBox .....	116
FileDropClass .....	469	Locate record .....	156
FileManagerClass .....	523	Location .....	175
PopupClass .....	604	Location of Message .....	161
PrintPreviewClass .....	628	Locator .....	118
ProcessClass .....	648	Contracting .....	119
QueryClass .....	664	Entry .....	118
QueryFormClass .....	677	Filter .....	119
RelationManagerClass .....	704	Incremental .....	119
ReportManagerClass .....	723	Step .....	118
StepClass .....	743	Locator Behavior .....	118
		Locator Class .....	121

- LocatorClass ..... 553, 579
  - Control ..... 581
  - FreeElement ..... 581
  - Init ..... 583
  - methods ..... 583
  - NoCase ..... 581
  - properties ..... 581
  - Set ..... 584
  - SetAlerts ..... 585
  - SetEnabled ..... 585
  - TakeAccepted ..... 586
  - TakeKey ..... 586
  - UpdateWindow ..... 586
  - ViewManager ..... 582
- LockRecover
  - FileManagerClass ..... 500
- Logo Screen ..... 106
- LOGOUT ..... 51
- look up ..... 144
- Lookup Field ..... 170
- Lookup Key ..... 169
- Lookup Procedure ..... 166, 167
- LookupMode
  - StepStringClass ..... 781
- LookupUpNon-RelatedRecord ..... 169
- Low
  - StepLongClass ..... 764
  - StepRealClass ..... 772
- M**
- Macro Substitution
  - TranslatorClass ..... 838
- main procedure ..... 95
- Marking list items ..... 149, 155
- Mask Variable ..... 143
- Maximize
  - PrintPreviewClass ..... 620
- Maximize Preview Window ..... 105
- Maximum Height ..... 182
- Maximum Width ..... 182
- MDI ..... 169
- MDI application ..... 95
- Me
  - RelationManagerClass ..... 695
- MEMBER ..... 52
- menu
  - popup menu ..... 167
- Menu Description ..... 168
- Menu Items ..... 168
- Menu String ..... 168
- Menu Template ..... 97
- Message
  - ErrorClass ..... 427
- Messages and Titles ..... 159
- MessageText
  - ErrorClass ..... 423
- Method to Call ..... 165
- Minimum Height ..... 182
- Minimum Width ..... 181
- Minimum Window Size ..... 181, 182
- Modeless windows ..... 169
- modeless windows ..... 169
- Module Definition file ..... 65
- Module Name ..... 109
- More Field Assignments ..... 145, 150
- More File Masks ..... 143
- Movable Thumb ..... 122, 123
- MSG attribute ..... 85
- Multi-Select? ..... 143
- Multi-threading ..... 45
- multiuser files ..... 49
- N**
- navigation in browse box ..... 118
- navigation in relation tree ..... 138
- New Class Methods ..... 61, 62
- New Class Properties ..... 61, 63
- New Method Name ..... 62
- New Method Prototype ..... 62
- Next
  - ASCIISearchClass ..... 227
  - BrowseClass ..... 281
  - ConstantClass ..... 326
  - FileManagerClass ..... 524
  - ProcessClass ..... 649
  - ReportManagerClass ..... 724
  - ViewManagerClass ..... 867
- NoCase
  - LocatorClass ..... 581
- Normalized Data ..... 45
- O**
- Object Name ..... 60, 165, 173
- Offer to save changes ..... 160
- OK button ..... 158
- OKControl
  - WindowManagerClass ..... 890
- OPEN ..... 53
- Open
  - FileManagerClass ..... 525
  - PrintPreviewClass ..... 629
  - RelationManagerClass ..... 706

ReportManagerClass .....	725	BrowseClass .....	257
ViewManagerClass .....	868	popup menu	
WindowManagerClass .....	906	template .....	167
Opened		PopupClass .....	587
WindowManagerClass .....	890	AddItem .....	592
OpenMode		AddItemEvent .....	594
ASCIIFileClass .....	202	AddItemMimic .....	595
FileManagerClass .....	501	AddMenu .....	596
OpenReport		AddSubMenu .....	598
ReportManagerClass .....	726	Ask .....	600
Optimize Moves .....	58, 184	ClearKeycode .....	590
Optimize Redraws .....	58, 184	DeleteItem .....	601
Optimizing a Dictionary for Wizards .....	84	GetItemChecked .....	602
Order		GetItemEnabled .....	603
ViewManagerClass .....	855	GetLastSelection .....	603
OriginalRequest		Init .....	604
WindowManagerClass .....	891	Kill .....	604
Other Data Type .....	63	methods .....	591
Other Picture .....	175	properties .....	590
outline list .....	136	Restore .....	605
Override Control Strategies .....	182	Save .....	606
Override default locator control .....	120	SetIcon .....	607
Overwrite existing procedures		SetItemCheck .....	608
Application Wizard .....	75	SetItemEnable .....	609
		SetLevel .....	609
		SetText .....	610, 611
		SetTranslator .....	612
		Toolbox .....	613
		ViewMenu .....	613
<b>P</b>		Position	
PageDown		FileManagerClass .....	526
ASCIIViewerClass .....	241	PostCompleted	
PagesAcross		WindowManagerClass .....	907
PrintPreviewClass .....	621	PostNewSelection	
PagesAhead		BrowseClass .....	281
ViewManagerClass .....	856	Preview	
PagesBehind		ReportManagerClass .....	716
ViewManagerClass .....	856	Preview Options .....	104
PagesDown		Previewer .....	192. See PrintPreviewClass
PrintPreviewClass .....	621	PreviewQueue	
PageSize		ReportManagerClass .....	717
ViewManagerClass .....	856	Previous	
PageUp		BrowseClass .....	282
ASCIIViewerClass .....	241	FileManagerClass .....	527
Parameters		ViewManagerClass .....	868
Procedure Properties .....	91, 93	Primary	
Source Template .....	110	ViewManagerClass .....	857
Passed Parameters .....	165	WindowManagerClass .....	891
PauseButton .....	157	PrimeAutoInc	
Percentile		FileManagerClass .....	528
ProcessClass .....	641	PrimeFields	
Picture .....	175	FileManagerClass .....	530
Population Order .....	83, 84		
Popup .....	192. See PopupClass		
ASCIIViewerClass .....	232		

- WindowManagerClass ..... 908
  - PrimeRecord
    - FileManagerClass ..... 531
    - ViewManagerClass ..... 869
  - PrimeUpdate
    - WindowManagerClass ..... 908
  - Priming Fields ..... 159
  - Print Preview ..... 102, 104
  - print preview, suppress ..... 176
  - PrintControl
    - BrowseClass ..... 258
  - Printer
    - ASCIIViewerClass ..... 232
  - Printing
    - Data Dictionary ..... 82
  - printing text ..... 113
  - PrintLines
    - ASCIIPrintClass ..... 220
  - PrintPreview
    - AsciiPrintClass ..... 218
  - PrintPreviewClass ..... 615, 711
    - AllowUserZoom ..... 620
    - AskPage ..... 624
    - AskThumbnails ..... 625
    - CurrentPage ..... 620
    - Display ..... 626
    - Init ..... 628
    - Kill ..... 628
    - Maximize ..... 620
    - methods ..... 623
    - Open ..... 629
    - PagesAcross ..... 621
    - PagesDown ..... 621
    - properties ..... 620
    - SetINIManager ..... 630
    - SetPosition ..... 631
    - SetZoom ..... 632
    - TakeAccepted ..... 633
    - TakeEvent ..... 634
    - TakeFieldEvent ..... 635
    - TakeWindowEvent ..... 636
    - UserPercentile ..... 621
    - WindowPosSet ..... 621
    - WindowSizeSet ..... 622
    - ZoomIndex ..... 622
  - PrintProcedure
    - BrowseClass ..... 258
  - PROC ..... 197
  - Procedure Communication ..... 89
  - Procedure Properties
    - Return Value ..... 91, 93, 94, 96, 97, 102, 107, 108
    - Window Behavior ..... 91
  - Procedure properties
    - Parameters ..... 91, 93
  - Procedure Properties dialog ..... 87
  - Procedure Templates ..... 87
  - Procedure Wizard ..... 76, 78, 80
  - Process
    - ReportManagerClass ..... 717
  - Process Template
    - classes ..... 101
  - Process template ..... 97
    - RI constraints ..... 98
    - single record mode ..... 177
  - ProcessClass ..... 637
    - AddItem ..... 645
    - ChildRead ..... 641
    - Init ..... 646
    - Kill ..... 648
    - methods ..... 643
    - Next ..... 649
    - Percentile ..... 641
    - properties ..... 641
    - PText ..... 642
    - RecordsProcessed ..... 642
    - RecordsToProcess ..... 642
    - Reset ..... 650
    - SetProgressLimits ..... 650
    - TakeRecord ..... 651
  - ProcessRecord ..... 89
  - Program Author ..... 47
  - progress bar ..... 103
  - progress window, suppress ..... 176
  - ProgressMgr ..... 192. *See* StepClass
  - Property Name ..... 63
  - Property to Set ..... 173
  - Property Type ..... 63
  - PROTECTED ..... 197
  - PText
    - ProcessClass ..... 642
  - public data ..... 48
  - PUTINI ..... 49
- ## Q
- QBE ..... 129
  - QFC
    - QueryFromVisual ..... 683
  - Query
    - BrowseClass ..... 258
  - Query button ..... 129
  - Query-by-example ..... 129
  - QueryClass ..... 303, 653
    - AddItem ..... 659
    - Ask ..... 660

GetFilter .....	661	RecordValidation .....	178
GetLimit .....	663	RECOVER .....	51
Init .....	664	Reference .....	63
Kill .....	664	Referential Integrity .....	50
methods .....	658	referential integrity	
properties .....	657	enforcement of .....	689
Reset .....	665	Refresh Application Builder Class Information .....	55, 62
SetLimit .....	666	refresh/redisplay ABC BrowseBoxes .....	267
QueryControl		Relate:file .....	192. See RelationManager
BrowseClass .....	259	Relate:filename .....	493, 690
QueryFormClass .....	669	Related Field .....	170
Ask .....	675	Relation Tree	
Init .....	676	filtering and record selection .....	139
Kill .....	677	relation tree navigation .....	138
methods .....	674	RelationManager .....	689
properties .....	673	methods .....	696
QueryFormVisual .....	679	properties .....	695
Init .....	685	RelationManagerClass	
QFC .....	683	AddRelation .....	697
TakeAccepted .....	686	AddRelationLink .....	699
TakeCompleted .....	687	CancelAutoInc .....	700
QueryFormVisual methods .....	684	Close .....	701
QueryFormVisual properties .....	683	Delete .....	702
QueryShared		Init .....	703
BrowseClass .....	259	Kill .....	704
Quick Start Wizard .....	70	ListLinkingFields .....	705
Quick-Scan Records .....	98, 102, 117	Me .....	695
QuickScan		Open .....	706
BrowseClass .....	260	Save .....	706
		SetAlias .....	707
		SetQuickScan .....	708
		Update .....	709
<b>R</b>		Relationship	
Range Limit .....	100, 104, 121	printing .....	82
FileDrop .....	146, 151	RelationTree	
Range Limit Field .....	100, 104, 146, 151	colors .....	139
Range Limit Type .....	100, 104, 121, 146, 151	icons .....	139
Read and write .....	53	Primary File Settings .....	138
Read only .....	53	Secondary File Settings .....	140
RealList		RelationTree Control template .....	136
BufferedPairsClass .....	309	RelationTreeUpdateButtons .....	140
Reassign FROM attribute after Kill .....	113	RELn::Toolbar .....	192. See ToolbarReltreeClass
Record Identifier .....	161	Remove Duplicates .....	151
Record order .....	115	RemoveErrors	
record selection .....	98, 100, 103, 104	ErrorClass .....	428
FileDrop .....	146, 151	Repeat previous record .....	157, 160
Record Validation .....	95, 108	repetitive insert/add .....	160
Records		Report	
BrowseClass .....	282	ReportManagerClass .....	717
RecordsProcessed		Report Properties .....	102
ProcessClass .....	642	Report Template .....	101
RecordsToProcess		classes .....	106
ProcessClass .....	642		

- Report template
  - single record mode ..... 177
- Report Wizard ..... 80
- ReportChildFiles ..... 178
  - Child File ..... 179
  - Detail ..... 179
- ReportDateStamp ..... 162
- ReportManagerClass
  - Ask ..... 721
  - AskPreview ..... 721
  - DeferOpenReport ..... 715
  - DeferWindow ..... 715
  - Init ..... 722
  - KeepVisible ..... 716
  - Kill ..... 723
  - methods ..... 720
  - Next ..... 724
  - Open ..... 725
  - OpenReport ..... 726
  - Preview ..... 716
  - PreviewQueue ..... 717
  - Process ..... 717
  - properties ..... 715
  - Report ..... 717
  - SkipPreview ..... 718
  - TakeCloseEvent ..... 727
  - TakeNoRecords ..... 728
  - TakeWindowEvent ..... 729
  - TimeSlice ..... 718
  - WaitCursor ..... 719
  - Zoom ..... 719
- ReportPageNumber ..... 163
- ReportTimeStamp ..... 163
- Request
  - ToolBarUpdateClass ..... 832
  - WindowManagerClass ..... 892
- RequestCancelled ..... 89
- RequestCompleted ..... 89
- Reset
  - ASCIIFileClass ..... 211
  - ASCIIViewerClass ..... 242
  - ConstantClass ..... 327
  - EditMultiSelectClass ..... 403
  - FilterLocatorClass ..... 584
  - ProcessClass ..... 650
  - QueryClass ..... 665
  - ViewManagerClass ..... 870
  - WindowManagerClass ..... 909
  - WindowResizeClass ..... 937
- reset ABC BrowseBoxes ..... 267
- Reset Fields ..... 122
- Reset on gain focus ..... 58
  - ResetFromAsk
    - BrowseClass ..... 283
  - ResetFromBuffer
    - BrowseClass ..... 285
  - ResetFromFile
    - BrowseClass ..... 286
  - ResetFromView
    - BrowseClass ..... 287
  - ResetOnGainFocus
    - WindowManagerClass ..... 892
  - ResetQueue
    - BrowseClass ..... 288
    - FileDropClass ..... 470
    - FileDropComboClass ..... 487
  - ResetResets
    - BrowseClass ..... 289
  - ResetSort
    - BrowseClass ..... 290
  - Resize ..... 181
    - WindowResizeClass ..... 938
  - Resize Strategy ..... 180
  - resize strategy
    - for a single control ..... 182, 183
  - resize windows ..... 180
  - Resizer ..... 192. See WindowResizeClass
  - Resize Configuration Options ..... 184
  - ResizeSetStrategy ..... 170
  - Response
    - WindowManagerClass ..... 893
  - Restore
    - PopupClass ..... 605
  - RestoreBuffer
    - FileManagerClass ..... 533
  - RestoreField
    - WindowManagerClass ..... 910
  - RestoreFile
    - FileManagerClass ..... 534
  - RestoreWindow
    - WindowResizeClass ..... 939
- Retain Row ..... 57
- RetainRow
  - BrowseClass ..... 260
- Return to original directory when done ..... 143
- Return Value
  - Procedure Properties 91, 93, 94, 96, 97, 102, 107, 108
  - Return Value Assignment ..... 165
- Reusability ..... 187
- Reusable code ..... 41
- RI constraints
  - Process template ..... 98
- Root
  - StepStringClass ..... 782

Run		Ask .....	735
WindowManagerClass .....	911	DefaultDirectory .....	733
<b>S</b>		DefaultFile .....	733
Save		Flags .....	733
PopupClass .....	606	Init .....	736
RelationManagerClass .....	706	methods .....	734
Save and Restore Window Location .....	92	properties .....	733
Save Button control template .....	158	SetMask .....	737
SaveBuffer		WindowTitle .....	733
FileManagerClass .....	535	Selecting	
Saved		BrowseClass .....	261
WindowManagerClass .....	893	selecting records .....	121
SaveFile		FileDrop .....	145
FileManagerClass .....	536	FileDropCombo .....	150
SaveHistory		Relation tree .....	139
WindowManagerClass .....	913	SelectRecord .....	89
Saving Global Data Between Sessions .....	49	SelectToolBarTarget .....	172
Scroll Bar Behavior .....	122	Set	
Scroll bottom .....	157	ConstantClass .....	328
Scroll down one page .....	156	EntryLocatorClass .....	414
Scroll down one row .....	156	LocatorClass .....	584
Scroll top .....	156	StepLocatorClass	
Scroll up one page .....	156	Set Initial Window Position .....	105
Scroll up one row .....	156	Set progress bar limits manually? .....	99, 103
ScrollEnd		SetABCProperty .....	173
BrowseClass .....	291	SetAlerts	
scrolling		BrowseClass .....	294
large files .....	115	EditClass .....	346, 364
Scrolling Form .....	177	IncrementalLocatorClass .....	565
ScrollOne		LocatorClass .....	585
BrowseClass .....	292	WindowManagerClass .....	914
ScrollPage		SetAlias	
BrowseClass .....	293	RelationManagerClass .....	707
SDI .....	97	SetEnabled	
SDI (Single Document Interface) .....	97	LocatorClass .....	585
Search .....	118	SetError	
search .....	113	FileManagerClass .....	537
search record .....	156	SetErrors	
Searcher		ErrorClass .....	429
ASCIIViewerClass .....	232	SetFatality	
Seconds for RECOVER .....	51	ErrorClass .....	430
Select File dialog .....	141	SetField	
SelectButton		ErrorClass .....	431
ToolBarTargetClass .....	821	SetFile	
SelectControl		ErrorClass .....	431
BrowseClass .....	261	SetFilter	
Selected		ViewManagerClass .....	871
EditMultiSelectClass .....	399	SetIcon	
SelectFileClass .....	731	PopupClass .....	607
AddMask .....	734	SetId	
		ErrorClass .....	432
		SetNIManager	

- PrintPreviewClass ..... 630
- SetItemCheck
  - PopupClass ..... 608
- SetItemEnable
  - PopupClass ..... 609
- SetKey
  - FileManagerClass ..... 537
- SetLevel
  - PopupClass ..... 609
- SetLimit
  - QueryClass ..... 666
  - StepClass ..... 744
  - StepLongClass ..... 767
  - StepRealClass ..... 775
  - StepStringClass ..... 787
- SetLimitNeeded
  - StepClass ..... 744
  - StepStringClass ..... 788
- SetLine
  - ASCIIFileClass ..... 212
  - ASCIIViewerClass ..... 243
- SetLineRelative
  - ASCIIViewerClass ..... 244
- SetMask
  - SelectFileClass ..... 737
- SetName
  - FileManagerClass ..... 538
- SetOrder
  - ViewManagerClass ..... 873
- SetParentControl
  - WindowResizeClass ..... 940
- SetParentDefaults
  - WindowResizeClass ..... 941
- SetPercentile
  - ASCIIFileClass ..... 213
- SetPosition
  - PrintPreviewClass ..... 631
  - WindowResizeClass ..... 942
- SetProgressLimits
  - ProcessClass ..... 650
- SetProperty ..... 173
- SetQueueRecord
  - BrowseClass ..... 294
  - FileDropClass ..... 471
- SetQuickScan
  - RelationManagerClass ..... 708
- SetResponse
  - WindowManagerClass ..... 915
- SetSort
  - BrowseClass ..... 295
  - ViewManagerClass ..... 874
- SetStrategy
  - WindowResizeClass ..... 943
- SetTarget
  - ToolBarClass ..... 798
- SetText
  - PopupClass ..... 610, 611
- SetTranslator
  - ASCIIViewerClass ..... 245
  - PopupClass ..... 612
- Setup
  - ASCIISearchClass ..... 228
- SetZoomPercentile
  - PrintPreviewClass ..... 632
- Shadow
  - EntryLocatorClass ..... 412
- SHARE ..... 53
- sharing files ..... 49, 53
- Ship List ..... 65
- Silent Database Operations ..... 491
  - FileManagerClass ..... 502
- SkipHeldRecords
  - FileManagerClass ..... 501
- SkipPreview
  - ReportManagerClass ..... 718
- Sort
  - BrowseClass ..... 262
- Sort Fields ..... 99, 103
- SortChars
  - StepStringClass ..... 782
- sorting
  - browse lists ..... 122
- Source Template ..... 109
  - Parameters ..... 110
- Splash Screen ..... 96
- Splash Template ..... 106
- Spread ..... 181
  - WindowResizeClass ..... 927
- SQL
  - Application Wizard ..... 73
- SQL and ABC Templates ..... 46
- SQL Browse Optimization ..... 57
- Standard Windows Behavior ..... 95
- START ..... 169
- StartAtCurrent
  - BrowseClass ..... 263
- Status Bar Section ..... 176
- Step Locator ..... 118
- StepClass ..... 739
  - Controls ..... 741
  - GetPercentile ..... 742
  - GetValue ..... 742
  - Init ..... 743
  - Kill ..... 743

methods .....	742
properties .....	741
SetLimit .....	744
SetLimitNeeded .....	744
StepCustomClass .....	745
AddItem .....	750
Entries .....	749
GetPercentile .....	751, 784
GetValue .....	752, 785
Init .....	753, 786
Kill .....	753
methods .....	750
properties .....	749
StepLocatorClass .....	
methods .....	760
properties .....	759
Set .....	760
TakeKey .....	760
StepLongClass .....	761
GetPercentile .....	765
GetValue .....	766
High .....	764
Low .....	764
methods .....	765
properties .....	764
SetLimit .....	767
StepRealClass .....	769
GetPercentile .....	773
GetValue .....	774
High .....	772
Low .....	772
methods .....	773
properties .....	772
SetLimit .....	775
StepStringClass .....	777
Kill .....	787
LookupMode .....	781
methods .....	784
properties .....	781
Root .....	782
SetLimit .....	787
SetLimitNeeded .....	788
SortChars .....	782
TestLen .....	783
String variable for .....	167
sub-class configuration .....	60
SubsString .....	
ErrorClass .....	433
Surface .....	181
WindowResizeClass .....	927
Syntax Diagram .....	196

**T**

TabAction .....	
BrowseClass .....	263
tagging records in a list .....	149, 155
TakeAccepted .....	
EntryLocatorClass .....	414
FilterLocatorClass .....	558
LocatorClass .....	586
PrintPreviewClass .....	633
QueryFormVisual .....	686
WindowManagerClass .....	916
TakeAcceptedLocator .....	
BrowseClass .....	296
TakeAction .....	
EditMultiSelectClass .....	404
TakeBenign .....	
ErrorClass .....	434
TakeCloseEvent .....	
ReportManagerClass .....	727
WindowManagerClass .....	917
TakeCompleted .....	
QueryFormVisual .....	687
WindowManagerClass .....	918
TakeError .....	
ErrorClass .....	435
TakeEvent .....	
ASCIIViewerClass .....	246
BrowseClass .....	297
EditClass .....	347
EditColorClass .....	356, 381
EditFontClass .....	391
EditMultiSelectClass .....	407
FileDropClass .....	471
FileDropComboClass .....	488
PrintPreviewClass .....	634
ToolBarClass .....	799
ToolBarListBoxClass .....	807
ToolBarTargetClass .....	823
ToolBarUpdateClass .....	834
WindowManagerClass .....	920
TakeFatal .....	
ErrorClass .....	436
TakeFieldEvent .....	
PrintPreviewClass .....	635
WindowManagerClass .....	921
TakeKey .....	
BrowseClass .....	298
EntryLocatorClass .....	415
IncrementalLocatorClass .....	566
LocatorClass .....	586
StepLocatorClass .....	760

- TakeLocate
  - BrowseClass ..... 298
- TakeNewSelection
  - BrowseClass ..... 299
  - FileDropClass ..... 472
  - FileDropComboClass ..... 489
  - WindowManagerClass ..... 922
- TakeNoRecords
  - ReportManagerClass ..... 728
- TakeNotify
  - ErrorClass ..... 437
- TakeOther
  - ErrorClass ..... 438
- TakeProgram
  - ErrorClass ..... 439
- TakeRecord
  - ProcessClass ..... 651
- TakeRejected
  - WindowManagerClass ..... 923
- TakeScroll
  - BrowseClass ..... 300
- TakeSelected
  - WindowManagerClass ..... 924
- TakeToolbar
  - ToolbarListBoxClass ..... 808
  - ToolbarReltreeClass ..... 816
  - ToolbarTargetClass ..... 824
  - ToolbarUpdateClass ..... 835
- TakeUser
  - ErrorClass ..... 440
- TakeVCRScroll
  - BrowseClass ..... 301
- TakeWindowEvent
  - PrintPreviewClass ..... 636
  - ReportManagerClass ..... 729
  - WindowManagerClass ..... 925
- Target Field ..... 145, 150
- Template
  - Ascii viewer ..... 107
  - AsciiPrintButton ..... 113
  - AsciiSearchButton ..... 113
  - ASCIIViewControl ..... 112
  - browse procedure ..... 92
  - External procedure ..... 109
  - file viewer ..... 107
  - form procedure ..... 93
  - Frame ..... 95
  - overview ..... 41
  - process ..... 97
  - report procedure ..... 101
  - Source ..... 109
  - splash ..... 106
- Template Embed Points ..... 41
- Template Prompts ..... 41
- Templates
  - BrowseBox ..... 114
  - CancelButton ..... 141, 162, 163
  - class configuration ..... 60
  - control ..... 88, 111
  - DOSFileLookup ..... 141
  - procedure ..... 87
- TestLen
  - StepStringClass ..... 783
- ThisProcess ..... 192. *See* ProcessClass
- ThisReport ..... 192
- ThisWindow ..... 192. *See* ReportManager; WindowManager
- THREAD ..... 51, 492
- Throw
  - ErrorClass ..... 441
  - FileManagerClass ..... 539
- ThrowFile
  - ErrorClass ..... 442
- ThrowMessage
  - ErrorClass ..... 443
  - FileManagerClass ..... 540
- Time display template ..... 175
- TimeOut
  - ViewManagerClass ..... 857
- TimeSlice
  - ReportManagerClass ..... 718
- Title
  - EditColorClass ..... 353
  - EditFileClass ..... 378
  - EditFontClass ..... 388
  - EditMultiSelectClass ..... 399
- Toolbar ..... 117, 192
  - Application Wizard ..... 74
  - BrowseClass ..... 264
- Toolbar Control Buttons ..... 177
- Toolbar Navigation Target ..... 172
- ToolbarClass ..... 789
  - AddTarget ..... 796
  - DisplayButtons ..... 796
  - Init ..... 797
  - Kill ..... 797
  - methods ..... 795
  - properties ..... 795
  - SetTarget ..... 798
  - TakeEvent ..... 799
- ToolbarForm ..... 192. *See* ToolbarUpdateClass
- ToolbarItem
  - BrowseClass ..... 264
- ToolbarListBoxClass
  - Browse ..... 805



- FileManagerClass ..... 549
  - INIClass ..... 577
  - RelationManagerClass ..... 709
  - WindowManagerClass ..... 926
  - update a single record from a file ..... 93
  - Update entire window? ..... 143
  - Update Procedure ..... 133, 138
  - Update Selected Fields ..... 144
  - Update(FileManager) ..... 897
  - UpdateBuffer
    - BrowseClass ..... 302
  - UpdateQuery
    - BrowseClass ..... 303
  - UpdateResets
    - BrowseClass ..... 304
  - UpdateThumb
    - BrowseClass ..... 304
  - UpdateThumbFixed
    - BrowseClass ..... 305
  - UpdateViewRecord
    - BrowseClass ..... 305
  - UpdateWindow
    - BrowseClass ..... 306
    - EntryLocatorClass ..... 416
    - FilterLocatorClass ..... 559
    - LocatorClass ..... 586
  - Use a variable file mask ..... 143
  - Use Application Builder Class ..... 60
  - Use Application Wizard ..... 74
  - Use Default Application Builder Class ..... 60
  - Use default FileManager ..... 51
  - Use default RelationManager ..... 51
  - Use field description as MSG() when MSG() is blank ..... 47
  - Use RI constraints on action ..... 98
  - Use Window Setting ..... 92
  - UseField
    - FileDropComboClass ..... 480
  - UseFile
    - FileManagerClass ..... 549
  - User Options ..... 83, 84
  - UserPercentile
    - PrintPreviewClass ..... 621
  - UseView
    - ViewManagerClass ..... 875
  - Utility Templates ..... 69
- V**
- Validate during NonStop Select ..... 178
  - Validate when the control is Accepted ..... 178
  - ValidateField
    - FileManagerClass ..... 550
  - ValidateFields
    - FileManagerClass ..... 551
  - ValidateLine
    - ASCIIFileClass ..... 214
  - ValidateRecord
    - FileDropClass ..... 473
    - FileManagerClass ..... 552
    - ViewManagerClass ..... 876
  - Value or queue to assign ..... 113
  - Value to Set ..... 173
  - VCR buttons ..... 177
  - VCCRRequest
    - WindowManagerClass ..... 894
  - Vertical Positional Strategy ..... 171, 183
  - Vertical Resize Strategy ..... 171, 183
  - View
    - ViewManagerClass ..... 857
  - Viewer Template ..... 107
  - ViewerN ..... 192. *See* AsciiViewerClass
  - ViewManager ..... 851
    - LocatorClass ..... 582
    - methods ..... 858
    - properties ..... 855
  - ViewManagerClass
    - AddRange ..... 860
    - AddSortOrder ..... 861
    - AppendOrder ..... 862
    - ApplyFilter ..... 862
    - ApplyOrder ..... 863
    - ApplyRange ..... 863
    - Close ..... 864
    - GetFreeElementName ..... 864
    - GetFreeElementPosition ..... 865
    - Init ..... 866
    - Kill ..... 867
    - Next ..... 867
    - Open ..... 868
    - Order ..... 855
    - PagesAhead ..... 856
    - PagesBehind ..... 856
    - PageSize ..... 856
    - Previous ..... 868
    - Primary ..... 857
    - PrimeRecord ..... 869
    - Reset ..... 870
    - SetFilter ..... 871
    - SetOrder ..... 873
    - SetSort ..... 874
    - TimeOut ..... 857
    - UseView ..... 875
    - ValidateRecord ..... 876
    - View ..... 857

ViewMenu		
PopupClass .....	613	
VIRTUAL .....	197	
<b>W</b>		
WaitCursor		
ReportManagerClass .....	719	
Web page		
publish BrowseBox .....	128	
When called for Delete .....	160	
Window		
BrowseClass .....	265	
generic .....	90	
individual control resizing .....	170	
Window Behavior		
Procedure Properties .....	91	
Window Controls		
default control settings for a field .....	85	
Window Message .....	98	
Window Operation Mode .....	92	
Window Template .....	90	
Window Update Options .....	143	
WindowManager		
methods .....	877	
properties .....	895	
template configuration .....	885	
template configuration .....	92	
WindowManager Configuration .....	58	
WindowManagerClass		
AddHistoryField .....	897	
AddHistoryFile .....	898	
AddItem .....	899	
AddUpdateFile .....	901	
Ask .....	902	
AutoRefresh .....	885	
AutoToolbar .....	885	
CancelAction .....	886	
ChangeAction .....	886	
Dead .....	887	
DeleteAction .....	887	
Errors .....	888	
FirstField .....	888	
ForceRefresh .....	888	
HistoryKey .....	889	
Init .....	903	
InsertAction .....	889	
Kill .....	905	
OKControl .....	890	
Open .....	906	
Opened .....	890	
OriginalRequest .....	891	
PostCompleted .....	907	
Primary .....	891	
PrimeFields .....	908	
PrimeUpdate .....	908	
Request .....	892	
Reset .....	909	
ResetOnGainFocus .....	892	
Response .....	893	
RestoreField .....	910	
Run .....	911	
Saved .....	893	
SaveHistory .....	913	
SetAlerts .....	914	
SetResponse .....	915	
TakeAccepted .....	916	
TakeCloseEvent .....	917	
TakeCompleted .....	918	
TakeEvent .....	920	
TakeFieldEvent .....	921	
TakeRejected .....	923	
TakeSelected .....	924	
TakeWindowEvent .....	925	
Translator .....	894	
Update .....	926	
VCRRequest .....	894	
WindowPosSet		
PrintPreviewClass .....	621	
WindowResize .....	180	
WindowResizeClass .....	927	
AutoTransparent .....	930	
DeferMoves .....	930	
GetParentControl .....	932	
GetPositionStrategy .....	933	
GetResizeStrategy .....	934	
Init .....	935	
Kill .....	937	
methods .....	931	
properties .....	930	
Reset .....	937	
Resize .....	938	
RestoreWindow .....	939	
SetParentControl .....	940	
SetParentDefaults .....	941	
SetPosition .....	942	
SetStrategy .....	943	
WindowResizeClass Configuration .....	58	
windows		
application modal .....	169	
modeless .....	169	
Windows file dialog .....	141	
Windows help file .....	73	
WindowSizeSet		
PrintPreviewClass .....	622	

WindowTitle	
SelectFileClass .....	733
Wizard	
browse procedure .....	76
report procedure .....	80
update form procedure .....	78
Wizards .....	42, 69
build entire application from dictionary .....	72
customizing .....	83
starting .....	69
Write only .....	53

## Z

Zoom	
ReportManagerClass .....	719
Zoom Setting .....	104
ZoomIndex	
PrintPreviewClass .....	622













