SUPPLEMENT

Purpose

The addition of five new classes and several properties and methods into the C5a Application Builder Classes has created an expanded Application Handbook. As a service to our customers, TopSpeed is providing this PDF of the new chapters and the revised PrintPreviewClass chapter, for easy viewing and printing of the majority of this new material.

This document is a duplication of the new material found in the Application Handbook (c5-ah.pdf), and is provided solely for the convenience of our customer's and does not supercede the complete Application Handbook (c5-ah.pdf). This document is not a complete compilation of all new properties and methods; see the ***readme file*** for a complete list of changes.

Contents

Note: The chapter and page numbers in this document reflect the chapter and page numbers in the new c5-ah.pdf distributed with C5a.

12- BROWSEEIPMANAGERCLASS

Overview

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

BrowseEIPManagerClass Concepts

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

Relationship to Other Application Builder Classes

EIPManagerClass

The BrowseEIPManager class is derived from the EIPManager class.

BrowseClass

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

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

EditClass

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

ABC Template Implementation

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

See *Control Templates—BrowseBox,* and *BrowseUpdateButton*s for more information.

BrowseEIPManagerClass Source Files

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

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

Conceptual Example

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

```
   PROGRAM

_ABCDllMode_  EQUATE(0)
_ABCLinkMode_ EQUATE(1)

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

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

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

```
ActionsView   VIEW(Actions)
              END

Queue:Browse  QUEUE
ACT:Action      LIKE(ACT:Action)
ACT:Priority    LIKE(ACT:Priority)
ViewPosition    STRING(1024)
              END
BrowseWindow WINDOW('Browse Records'),AT(0,0,247,140),SYSTEM,GRAY
              LIST,AT(5,5,235,100),USE(?List),IMM,HVSCROLL,MSG('Browsing Records'),|
              FORMAT('80L~Action~@S20@8R~Priority~L@N2@'),FROM(Queue:Browse)
              BUTTON('&Insert'),AT(5,110,40,12),USE(?Insert),KEY(InsertKey)
              BUTTON('&Change'),AT(50,110,40,12),USE(?Change),KEY(CtrlEnter),DEFAULT
              BUTTON('&Delete'),AT(95,110,40,12),USE(?Delete),KEY(DeleteKey)
             END

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

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

BRW1::EIPManager     BrowseEIPManager       ! Browse EIP Manager for Browse using ?List

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


ThisWindow.Init PROCEDURE

ReturnValue          BYTE,AUTO
  CODE
  SELF.Request = GlobalRequest
  ReturnValue =PARENT.Init()
  IF ReturnValue THEN RETURN ReturnValue.
  SELF.FirstField = ?List
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
  SELF.AddItem(Toolbar)
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  Relate:Actions.Open
  FilesOpened = True
  BRW1.Init(?List,Queue:Browse.ViewPosition,BRW1::View:Browse,Queue:Browse,Relate:Actions,SELF)
  OPEN(BrowseWindow)
  SELF.Opened=True
  BRW1.Q &= Queue:Browse
  BRW1.AddSortOrder(,ACT:KeyAction)
  BRW1.AddLocator(BRW1::Sort0:Locator)
  BRW1::Sort0:Locator.Init(,ACT:Action,1,BRW1)
```

```
  BRW1.AddField(ACT:Action,BRW1.Q.ACT:Action)
  BRW1.AddField(ACT:Priority,BRW1.Q.ACT:Priority)
  BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
  BRW1.InsertControl=?Insert
  BRW1.ChangeControl=?Change
  BRW1.DeleteControl=?Delete
  BRW1.AddToolbarTarget(Toolbar)
  SELF.SetAlerts()
  RETURN ReturnValue


ThisWindow.Kill PROCEDURE

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


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

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

BrowseEIPManagerClass Properties

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

BC (browse class)

BC **&BrowseClass, PROTECTED**

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

BrowseEIPManagerClass Methods

The BrowseEIPManagerClass contains the following methods, and inherits all the methods of the EIPManagerClass.

Functional Organization—Expected Use

As an aid to understanding the EIPManagerClass, it is useful to organize its methods into two large categories according to their expected use—the primary interface and the virtual methods. This organization reflects what we believe is typical use of the EIPManagerClass 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[D]	initialize the BrowseEditClass object
Kill[D]	shut down the BrowseEditClass object

Mainstream Use:

TakeNewSelection[D]	handle Event:NewSelections

Occasional Use:

ClearColumn[D]	reset column property values
TakeCompleted[D]	process completion of edit

[D] These methods are also Derived

Derived 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 derived, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

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

ClearColumn (reset column property values)

ClearColumn, DERIVED

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

The TakeCompleted method calls the ClearColumn method.

Example:

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

See Also: Column

Init (initialize the BrowseEIPManagerClass object)

Init, DERIVED, PROC

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

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

Return Data Type: BYTE

Example:

```
WindowManager.Run PROCEDURE
  CODE
  IF ~SELF.Init()
    SELF.Ask
  END
  SELF.Kill
  RETURN CHOOSE(SELF.Response=0,RequestCancelled,SELF.Response)
```

See Also: BrowseClass.ResetFromAsk

Kill (shut down the BrowseEIPManagerClass object)

Kill, DERIVED, PROC

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

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

Return Data Type: BYTE

Example:

```
WindowManager.Run PROCEDURE
  CODE
  IF ~SELF.Init()
    SELF.Ask
  END
  SELF.Kill
  RETURN CHOOSE(SELF.Response=0,RequestCancelled,SELF.Response)
```

See Also: BrowseClass.ResetFromAsk

TakeCompleted (process completion of edit)

TakeCompleted(*force*), DERIVED

TakeCompleted	Determines the edit-in-place dialog's action after either a loss of focus or an end user action.
force	An integer constant, variable, EQUATE, or expression that indicates the record being edited should be saved without prompting the end user.

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

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

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

Example:

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

See Also: EIPManager.TakeFocusLoss, EIPManager.TakeAction

TakeNewSelection (reset edit-in-place column)

TakeNewSelection, DERIVED, PROC

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

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

Return Data Type: BYTE

Example:

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

See Also: ResetColumn

23 - EDITSPINCLASS

Overview

The EditSpinClass is an EditClass that supports a SPIN control. The EditSpinClass lets you implement a dynamic edit-in-place SPIN control for a column in a LIST.

EditSpinClass Concepts

The EditSpinClass creates a SPIN control, accepts input from the end user, then returns the input to the variable specified by the Init method, typically the variable associated with a specific LIST cell—a field in the LIST control's data source QUEUE. The EditSpinClass also signals the calling procedure whenever significant edit-in-place events occur, such as tabbing to a new column, cancelling the edit, or completing the edit (moving to a new record or row). The EditSpinClass provides a virtual TakeEvent method to let you take control of significant edit-in-place events.

Relationship to Other Application Builder Classes

EditClass

The EditSpinClass is derived from the EditClass. The EditClass serves as the foundation and framework for its derived classes. These derived classes each provide a different type of input control or input user interface. You can control the values returned by these derived EditClass objects by using their virtual methods. See the *Conceptual Example*.

BrowseEIPManagerClass

The EditClass is managed by the BrowseEIPManagerClass. The BrowseEIPManagerClass depends on the EditClass operating according to it's 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 EditSpinClass object called EditInPlace::*fieldname* and register the object with the BrowseClass object. The BrowseClass object then calls the registered EditSpinClass object's methods as needed. See *Control Templates—BrowseUpdateButton*s for more information.

EditSpinClass Source Files

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

ABEIP.INC	EditSpinClass declarations
ABEIP.CLW	EditSpinClass method definitions

Conceptual Example

The following example shows a sequence of statements to declare, instantiate, initialize, use, and terminate an EditSpinClass object and a related BrowseClass object. The example page-loads a LIST of actions and associated attributes (priority and completed), then edits the "Priority" items with an EditSpinClass object. Note that the BrowseClass object calls the "registered" EditSpinClass object's methods as needed.

Note: The EditSpinClass requires values for PROP:RangeLow, PROP:RangeHigh, and PROP:Step to function correctly. The EditSpinClass.Init method is the proper place to set these properties. See *SPIN* in the *Language Reference* for more information.

```
   PROGRAM
_ABCDllMode_  EQUATE(0)
_ABCLinkMode_ EQUATE(1)
   INCLUDE('ABWINDOW.INC'),ONCE
   INCLUDE('ABEIP.CLW'),ONCE
   INCLUDE('ABBROWSE.CLW'),ONCE
   MAP
   END
Actions        FILE,DRIVER('TOPSPEED'),PRE(ACT),CREATE,BINDABLE,THREAD
KeyAction        KEY(ACT:Action),NOCASE,OPT
Record           RECORD,PRE()
Action           STRING(20)
Priority         DECIMAL(2)
Completed        DECIMAL(1)
                 END
               END
ViewActions    VIEW(Actions)
               END
ActQ           QUEUE
ACT:Action       LIKE(ACT:Action)
ACT:Priority     LIKE(ACT:Priority)
ACT:Completed    LIKE(ACT:Completed)
ViewPosition     STRING(1024)
               END

ActionWindow  WINDOW('Actions File'),AT(,,164,144),IMM,HLP('BrowseActions'),SYSTEM,GRAY
                LIST,AT(8,6,148,115),USE(?List),IMM,HVSCROLL,FORMAT('80L(2)|~Action~'&|
                '@S20@31C|~Priority~@N2@40L(2)|~Done~L(0)@N1@'),FROM(ActQ)
                BUTTON('&Insert'),AT(10,126,45,14),USE(?Insert:2)
```

```
                BUTTON('&Change'),AT(59,126,45,14),USE(?Change:2),DEFAULT
                BUTTON('&Delete'),AT(108,126,45,14),USE(?Delete:2)
              END
ThisWindow     CLASS(WindowManager)
Init            PROCEDURE(),BYTE,PROC,DERIVED
Kill            PROCEDURE(),BYTE,PROC,DERIVED
              END
BRW1          CLASS(BrowseClass)
Q               &ActQ
              END
Edit:ACT:Priority CLASS(EditSpinClass)    ! Edit-in-place class for field ACT:Priority
Init              PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar),DERIVED
                END
  CODE
  GlobalResponse = ThisWindow.Run()

ThisWindow.Init PROCEDURE
ReturnValue          BYTE,AUTO
  CODE
  SELF.Request = GlobalRequest
  ReturnValue =PARENT.Init()
  IF ReturnValue THEN RETURN ReturnValue.
  SELF.FirstField = ?List
  SELF.Errors &= GlobalErrors
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  Relate:Actions.Open
  FilesOpened = True
  BRW1.Init(?List,ActQ.ViewActions,BRW1::ViewActions,ActQ,Relate:Actions,SELF)
  OPEN(ActionWindow)
  SELF.Opened=True
  BRW1.Q &= ActQ
  BRW1.AddSortOrder(ACT:KeyAction)
  BRW1.AddField(ACT:Action,BRW1.Q.ACT:Action)
  BRW1.AddField(ACT:Priority,BRW1.Q.ACT:Priority)
  BRW1.AddField(ACT:Completed,BRW1.Q.ACT:Completed)
  BRW1.AddEditControl(EditInPlace::ACT:Priority,2)    !Add cutom edit-inplace class
  BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
  BRW1.InsertControl=?Insert:2
  BRW1.ChangeControl=?Change:2
  BRW1.DeleteControl=?Delete:2
  SELF.SetAlerts()
  RETURN ReturnValue

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

Edit:ACT:Priority.Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)
  CODE
  PARENT.Init(FieldNumber,ListBox,UseVar)
  SELF.FEQ{PROP:RANGE,1} = 1                  !Set the Low Range for the Spinbox
  SELF.FEQ{PROP:RANGE,2} = 10                 !Set the High Range for the Spinbox
  SELF.FEQ{PROP:Step} = 1                     !Set the incremental steps of the Spinbox
```

EditSpinClass Properties

The EditSpinClass inherits all the properties of the EditClass from which it is derived. See *EditClass Properties* and *EditClass Concepts* for more information.

EditSpinClass Methods

The EditSpinClass 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 EditSpinClass contains the following methods:

Functional Organization—Expected Use

As an aid to understanding the EditSpinClass it is useful to organize its methods into two large categories according to their expected use—the primary interface and the virtual methods. This organization reflects what we believe is typical use of the EditSpinClass 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[VI]	initialize the EditSpinClass object
Kill[VI]	shut down the EditSpinClass object

Mainstream Use:

TakeEvent[VI]	handle events for the SPIN control

Occasional Use:

CreateContol[V]	create the SPIN control
SetAlerts[VI]	alert keystrokes for the SPIN control

[V] These methods are also virtual.
[I] 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[I]	initialize the EditSpinClass object
CreateContol	create the SPIN control
SetAlerts[I]	alert keystrokes for the SPIN control
TakeEvent[I]	handle events for the SPIN control
Kill[I]	shut down the EditSpinClass object

CreateControl (create the edit-in-place SPIN control)

CreateControl, VIRTUAL, PROTECTED

The **CreateControl** method creates the edit-in-place SPIN control and sets the FEQ property.

Implementation: The Init method calls the CreateControl method. The CreateControl method sets the value of the FEQ property. Use the Init method or the CreateControl method to set any required properties of the SPIN control.

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
  CODE
  SELF.ListBoxFeq = ListBox
  SELF.CreateControl()
  ASSERT(SELF.Feq)
  SELF.UseVar &= UseVar
  SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
  SELF.Feq{PROP:Use} = UseVar
  SELF.SetAlerts
```

See Also: FEQ, EditClass.CreateControl

24 - EIPMANAGERCLASS

Overview

The EIPManagerClass is a WindowManager that displays an edit-in-place dialog, and handles events for that dialog . The EIPManagerClass is an abstract class—it is not useful by itself, but serves as the foundation and framework for the BrowseEIPManagerClass. See *BrowseEIPManagerClass*.

EIPManagerClass Concepts

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

Relationship to Other Application Builder Classes

WindowClass

The EIPManager class is derived from the WindowManager class.

BrowseClass

Each BrowseClass utilizing edit-in-place requires an BrowseEIPManager to manage the events and processes that are used by each edit-in-place field in the browse.

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

EditClasses

The EIPManager provides the basic or “under the hood” interface between the Edit classes and the Browse class. The EIPManager uses the EditQueue to keep track of the fields in the browse utilizing edit-in-place.

ABC Template Implementation

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

See *Control Templates—BrowseBox* and *BrowseUpdateButton*s for more information.

EIPManagerClass Source Files

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

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

Conceptual Example

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

```
   PROGRAM

_ABCDllMode_  EQUATE(0)
_ABCLinkMode_ EQUATE(1)

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

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

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

```
ActionsView   VIEW(Actions)
              END

Queue:Browse  QUEUE
ACT:Action      LIKE(ACT:Action)
ACT:Priority    LIKE(ACT:Priority)
ViewPosition    STRING(1024)
              END
BrowseWindow WINDOW('Browse Records'),AT(0,0,247,140),SYSTEM,GRAY
              LIST,AT(5,5,235,100),USE(?List),IMM,HVSCROLL,MSG('Browsing Records'),|
              FORMAT('80L~Action~@S20@8R~Priority~L@N2@'),FROM(Queue:Browse)
              BUTTON('&Insert'),AT(5,110,40,12),USE(?Insert),KEY(InsertKey)
              BUTTON('&Change'),AT(50,110,40,12),USE(?Change),KEY(CtrlEnter),DEFAULT
              BUTTON('&Delete'),AT(95,110,40,12),USE(?Delete),KEY(DeleteKey)
             END

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

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

BRW1::EIPManager     BrowseEIPManager       ! EIPManager for Browse using ?List

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


ThisWindow.Init PROCEDURE

ReturnValue          BYTE,AUTO
  CODE
  SELF.Request = GlobalRequest
  ReturnValue =PARENT.Init()
  IF ReturnValue THEN RETURN ReturnValue.
  SELF.FirstField = ?List
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
  SELF.AddItem(Toolbar)
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  Relate:Actions.Open
  FilesOpened = True
  BRW1.Init(?List,Queue:Browse.ViewPosition,BRW1::View:Browse,Queue:Browse,Relate:Actions,SELF)
  OPEN(BrowseWindow)
  SELF.Opened=True
  BRW1.Q &= Queue:Browse
  BRW1.AddSortOrder(,ACT:KeyAction)
  BRW1.AddLocator(BRW1::Sort0:Locator)
  BRW1::Sort0:Locator.Init(,ACT:Action,1,BRW1)
```

```
  BRW1.AddField(ACT:Action,BRW1.Q.ACT:Action)
  BRW1.AddField(ACT:Priority,BRW1.Q.ACT:Priority)
  BRW1.ArrowAction = EIPAction:Default+EIPAction:Remain+EIPAction:RetainColumn
  BRW1.InsertControl=?Insert
  BRW1.ChangeControl=?Change
  BRW1.DeleteControl=?Delete
  BRW1.AddToolbarTarget(Toolbar)
  SELF.SetAlerts()
  RETURN ReturnValue


ThisWindow.Kill PROCEDURE

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


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

  CODE
  PARENT.Init(ListBox,Posit,V,Q,RM,WM)
  SELF.EIP &= BRW1::EIPManager      ! Browse object's reference to the BrowseEIPManager
```

EIPManagerClass Properties

The EIPManagerClass contains the following properties.

Again (column usage flag)

Again BYTE, PROTECTED

The **Again** property contains a value that indicates whether or not the current edit-in-place column has been selected by the user during an edit-in-place process.

The TakeEvent method is where the Again property receives a value.

Arrow (edit-in-place action on arrow key)

Arrow &BYTE

The **Arrow** property is a reference to a BYTE which indicates the action to take when the end user presses the up or down arrow key during an edit-in-place process.

Note: The Arrow property should be treated as a PROTECTED property except during initialization.

Implementation: When the EIPManager is instantiated from a browse the Arrow property will point to the BrowseClass.ArrowAction.

See Also: BrowseClass.ArrowAction

Column (listbox column)

Column UNSIGNED

The **Column** property contains a value that indicates the column number of the listbox field which currently has focus in an edit-in-place process.

Enter (edit-in-place action on enter key)

Enter &BYTE

The **Enter** property is a reference to the BrowseClass.EnterAction property, and indicates the action to take when the end user presses the ENTER key during an edit-in-place process.

Note: The Enter property should be treated as a PROTECTED property except during initialization.

See Also: BrowseClass.EnterAction

EQ (list of edit-in-place controls)

EQ &EditQueue

The **EQ** property is a reference to a structure containing a list of browse list columns that will not utilize the default edit-in-place control. This list includes columns that will not utilize edit-in-place.

Implementation: The AddControl method adds browse list columns to the EQ property. An entry without an associated control indicates a column that has been specified as non-edit-in-place.

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

The EQ property is a reference to a QUEUE declared in ABEdit.INC as follows:

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

Note: The EQ property should be treated as a PROTECTED property except during initialization.

See Also: AddControl

Fields (managed fields)

Fields &FieldPairsClass, PROTECTED

The **Fields** property is a reference to the FieldPairsClass object that moves and compares data between the BrowseClass object's FILE and the EditClasses.

> **Note: The Fields property should be treated as a PROTECTED property except during initialization.**

See Also: BrowseClass.TabAction

FocusLoss (action on loss of focus)

FocusLoss &BYTE

The **FocusLoss** property is a reference to the BrowseClass.FocusLossAction property, and indicates the action to take with regard to pending changes when the edit control loses focus during an edit-in-place process.

> **Note: The FocusLoss property should be treated as a PROTECTED property except during initialization.**

See Also: BrowseClass.TabAction, BrowseClass.FocusLossAction

Insert (placement of new record)

Insert	BYTE

The **Insert** property indicates where in the list a new record will be added when the end user inserts a new record. The default placement is below the selected record.

Implementation: There are three places a new record can be placed in a list when using edit-in-place: above the selected record; below the selected record (the default); or appended to the bottom of the list.

> **Note: This does not change the sort order. After insertion, the list is resorted and the new record appears in the proper position within the sort sequence.**

The specified placements are implemented by the BrowseEIPManager.Init method. Set the record insertion point by assigning, adding, or subtracting the following EQUATEd values to Insert. The following EQUATEs are in ABEdit.INC:

```
   ITEMIZE,PRE(EIPAction)
Default  EQUATE(0)
Always   EQUATE(1)
Never    EQUATE(2)
Prompted EQUATE(4)
Save     EQUATE(7)
Remain   EQUATE(8)
Before   EQUATE(9)      ! insert before/above selected record
Append   EQUATE(10)     ! insert at the bottom of the list
RetainColumn   EQUATE(16)
   END
```

See Also: BrowseEIPManager.Init

ListControl (listbox control number)

ListControl	SIGNED

The **ListControl** property contains the control number of the LIST control that is utilizing edit-in-place.

> **Note: The ListControl property should be treated as a PROTECTED property except during initialization.**

See Also: BrowseClass.TabAction

LastColumn (previous edit-in-place column)

LastColumn	BYTE, PROTECTED

The **LastColumn** property contains the column number of the previously used edit-in-place control to facilitate the appropriate processing of a NewSelection.

Implementation: The LastColumn method is assigned the value of the Column property in the ResetColumn method.

Repost (event synchronization)

Repost	UNSIGNED, PROTECTED

The **Repost** property indicates the appropriate event to post to the edit-in-place control based on events posted from the browse procedure window.

Implementation: The TakeEvent and TakeFieldEvent methods assign the appropriate value to the Repost property. The Kill method posts the specified event to the appropriate edit-in-place control based on the value contained in the RepostField property.

See Also: RepostField

RepostField (event synchronization field)

RepostField	UNSIGNED, PROTECTED

The **RepostField** property contains the field control number of the listbox field that is being edited.

Implementation: The TakeFieldEvent method assigns the appropriate value to the RepostField property. The Kill method posts the specified event to the appropriate edit-in-place control based on the value contained in the RepostField property.

See Also: Repost

Req (database request)

Req	BYTE, PROTECTED

The **Req** property indicates the database action the procedure is handling. The EIPManager uses this property to make appropriate processing decisions with regard to priming records, saving or abandoning changes, etc.

Implementation: The Run method is passed a parameter which contains the value assigned to the Req property.

See Also: WindowManager.Request

SeekForward (get next field flag)

SeekForward	BYTE, PROTECTED

The **SeekForward** property indicates that the end user has pressed the TAB key during an edit-in-place process.

Implementation: The TakeAction method conditionally assigns a value of one (1) to the SeekForward property based on the actions of the end user.

See Also: Next

Tab (action on a tab key)

Tab	&BYTE

The **Tab** property is a reference to the BrowseClass.TabAction property that indicates the action to take when the end user presses the TAB key during an edit-in-place process.

> **Note: The Tab property should be treated as a PROTECTED property except during initialization.**

See Also: BrowseClass.TabAction

EIPManagerClass Methods

The EIPManagerClass contains the following methods.

Functional Organization—Expected Use

As an aid to understanding the EIPManagerClass, it is useful to organize its methods into two large categories according to their expected use—the primary interface and the virtual methods. This organization reflects what we believe is typical use of the EIPManagerClass 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	run this procedure
Init[D]	initialize the EditClass object
InitControls	initialize edit-in-place controls
Kill[D]	shut down the EditClass object

Mainstream Use:

TakeEvent[D]	handle events for the edit control
TakeNewSelection[D]	handle Event:NewSelection

Occasional Use:

AddControl	register edit-in-place controls
ClearColumn[V]	reset column property values
CreateContol[V]	a virtual to create the edit control
GetEdit[V]	identify edit-in-place field
Next	get the next edit-in-place field
ResetColumn[V]	reset edit-in-place object to selected field
SetAlerts[V]	alert appropriate keystrokes for the edit control
TakeAction[V]	process end user actions
TakeCompleted[V]	process completion of edit
TakeFocusLoss[V]	process loss of focus
TakeFieldEvent[D]	handle field specific events

[D] These methods are also derived.
[V] These methods are also virtual.

Virtual and Derived 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 either derived or virtual, they are very easy to override. These methods do provide reasonable default behavior in case you do not want to override them.

Init[D]	initialize the EditClass object
Kill[D]	shut down the EditClass object
TakeEvent[D]	handle events for the edit control
TakeNewSelection[D]	handle Event:NewSelection
ClearColumn[V]	reset column property values
CreateContol[V]	a virtual to create the edit control
GetEdit[V]	identify edit-in-place field
ResetColumn[V]	reset edit-in-place object to selected field
SetAlerts[V]	alert appropriate keystrokes for the edit control
TakeAction[V]	process end user actions
TakeCompleted[V]	process completion of edit
TakeFocusLoss[V]	process loss of focus
TakeFieldEvent[D]	handle field specific events

AddControl (register edit-in-place controls)

AddControl([*EditClass*], *Column*, *AutoFree*)

AddControl	Specifies an edit-in-place control.
EditClass	The label of the EditClass. If omitted, the specified *column* is not editable.
Column	An integer constant, variable, EQUATE, or expression that indicates the browse list column to edit with the specified *editclass* object.
AutoFree	A numeric constant, variable, EQUATE, or expression that indicates whether the BrowseClass.Kill method DISPOSEs of the *editclass* object. A zero (0) value leaves the object intact. A non-zero value DISPOSEs the object.

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

The AddControl method also registers fields which will not be editable via edit-in-place. In this instance the EditClass parameter is omitted.

Implementation:

The InitControls and BrowseClass.AddEditControl methods call the AddControl method. The BrowseClass.AddEditControl method defines the *editclass* for a column not utilizing the default *editclass.*

The AddControl method ADDs a record containing the values of *EditClass*, *Column*, and *AutoFree*, to the EditQueue which is declared in ABEdit.INC as follows:

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

Example:

```
BrowseClass.AddEditControl PROCEDURE(EditClass EC,UNSIGNED Id,BYTE Free)
  CODE
    SELF.CheckEIP
    SELF.EIP.AddControl(EC,Id,Free)
```

See Also: EQ, InitControls, BrowseClass.AddEditControl

ClearColumn (reset column property values)

ClearColumn, VIRTUAL

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

The TakeAction and TakeNewSelection methods call the ClearColumn method.

Example:

```
EIPManager.TakeNewSelection PROCEDURE   ! Must be overridden to handle out-of-row clicks
  CODE
  IF FIELD() = SELF.ListControl AND KEYCODE() = MouseLeft       ! An in-row mouse click
    SELF.ClearColumn
    SELF.Column = SELF.ListControl{PROPLIST:MouseUpField}
    SELF.ResetColumn
  END
  RETURN Level:Benign
```

See Also: Column, TakeAction, TakeNewSelection

GetEdit (identify edit-in-place field)

GetEdit, VIRTUAL, PROTECTED

The **GetEdit** method checks for a value in the Control field of the EditQueue.

Implementation: GetEdit is called by the Next method, and returns one (1) if any value is in the Control field of the EditQueue, otherwise it returns zero (0).

Return Data Type: BYTE

Example:

```
EIPManager.Next PROCEDURE
  CODE

  GET(SELF.EQ,RECORDS(SELF.EQ))
? ASSERT(~ERRORCODE())
  LastCol=SELF.EQ.Field

  LOOP
    CLEAR(SELF.EQ)
    SELF.EQ.Field = SELF.Column
    GET(SELF.EQ,SELF.EQ.Field)
    IF ~ERRORCODE() AND SELF.GetEdit()
      BREAK
    END
    !executable code
```

See Also: EQ, Next

Init (initialize the EIPManagerClass object)

Init, DERIVED, PROC

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

Implementation: The BrowseEIPManager.Init method calls the Init method. The Init method primes variables and calls the InitControls method which then initializes the appropriate edit-in-place controls.

Return Data Type: BYTE

Example:

```
BrowseEIPManager.Init       ! initialize BrowseEIPManagerClass object
!program code
RETURN PARENT.Init()        ! call to the EIPManager.Init
```

See Also: BrowseEIPManager.Init, InitControls

InitControls (initialize edit-in-place controls)

InitControls, VIRTUAL

The **InitControls** method registers the default edit-in-place controls with the EIPManager by calling the AddControl method, and initializes each added control.

Implementation: The Init method calls the InitControls method. The InitControls method checks for custom edit-in-place controls in the EditQueue before adding a default edit-in-place control.

Example:

```
EIPManager.Init PROCEDURE
  CODE
  IF SELF.Column = 0 THEN SELF.Column = 1.
  SELF.LastColumn = 0
  SELF.Repost = 0
  SELF.RepostField = 0
  ASSERT(~SELF.EQ &= NULL)
  SELF.EQ.Field = 1

  SELF.InitControls
  SELF.ResetColumn
  RETURN Level:Benign
```

See Also: Init, EQ, AddControl

Kill (shut down the EIPManagerClass object)

Kill, DERIVED, PROC

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

Implementation: The BrowseEIPManager.Kill method calls the Kill method with a PARENT call. The Kill method destroys the edit-in-place controls created by the InitControls method.

Return Data Type: BYTE

Example:

```
BrowseEIPManager.Kill PROCEDURE
  CODE
  SELF.BC.ResetFromAsk(SELF.Req,SELF.Response)
  RETURN PARENT.Kill()
```

See Also: BrowseEIPManager.Kill

Next (get the next edit-in-place field)

Next, PROTECTED

The **Next** method gets the next edit-in-place control in the direction specified (forward or backward) by the end user.

Implementation: The Next method loops through the EditQueue and gets the next edit-in-place control based on the RETURN value of the GetEdit method.

Example:

```
EIPManager.ResetColumn PROCEDURE
  CODE
  SETKEYCODE(0)
  SELF.Next
  IF SELF.Column <> SELF.LastColumn
    SELF.ListControl{PROP:Edit,SELF.EQ.Field} = SELF.EQ.Control.Feq
    SELECT(SELF.EQ.Control.Feq)
    SELF.LastColumn = SELF.Column
  END
```

See Also: GetEdit, SeekForward, Column, EQ

ResetColumn (reset edit-in-place object to selected field)

ResetColumn, VIRTUAL, PROTECTED

The **ResetColumn** method selects the appropriate edit-in-place control based on the selected listbox field.

Implementation: The ResetColumn method resets the FEQ to the selected ListControl field.

Example:

```
EIPManager.TakeCompleted PROCEDURE(BYTE Force)
  CODE
  SELF.Column = 1
  IF SELF.Again
     SELF.ResetColumn
  END
```

See Also: EditClass.FEQ, Init, ListControl, TakeAction, TakeCompleted, TakeNewSelection

Run (run the EIPManager)

Run(*request*)

Run Run the EIPManager.

request An integer constant, variable, EQUATE, or expression identifying the database action (insert, change, delete) requested.

The **Run** method assigns the passed value to the Req property and executes the EIPManager.

Implementation: Return value EQUATEs are declared in \LIBSRC\TPLEQU.CLW as follows:

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

Return Data Type: BYTE

Example:

```
BrowseClass.AskRecord PROCEDURE(BYTE Req)
  CODE
  SELF.CheckEIP
  RETURN SELF.EIP.Run(Req)
```

See Also: BrowseEIPManager.Run, Req

TakeAction (process edit-in-place action)

TakeAction(*action*), VIRTUAL

TakeAction	Processes edit-in-place action.
action	An integer constant, variable, EQUATE, or expression that contains the action to process. Valid EQUATEs are forward, backward, next, previous, complete, and cancel.

The **TakeAction** method processes an EIPManager dialog action. The TakeAction method is your opportunity to interpret and implement the meaning of the end user's selection.

Implementation: The TakeFieldEvent conditionally calls the TakeAction method.

Corresponding EQUATEs are declared in ABEIP.INC as follows:

```
EditAction ITEMIZE(0),PRE
None     EQUATE
Forward  EQUATE     ! Next field
Backward EQUATE     ! Previous field
Complete EQUATE     ! OK
Cancel   EQUATE     ! Cancel
Next     EQUATE     ! Focus moving to Next record
Previous EQUATE     ! Focus moving to Previous record
Ignore   EQUATE
  END
```

Example:

```
EIPManager.TakeFieldEvent       PROCEDURE
I UNSIGNED(1)
  CODE
  IF FIELD() = SELF.ListControl THEN RETURN Level:Benign .
  LOOP I = 1 TO RECORDS(SELF.EQ)+1
   ! Optimised to pick up subsequent events from same field
    IF ~SELF.EQ.Control &= NULL AND SELF.EQ.Control.Feq = FIELD()
      SELF.TakeAction(SELF.EQ.Control.TakeEvent(EVENT()))
      RETURN Level:Benign
    END
    GET(SELF.EQ,I)
  END
  ! Code to handle an unknown field
```

See Also: TakeFieldEvent

TakeCompleted (process completion of edit)

TakeCompleted(*force*), VIRTUAL

TakeCompleted	Determines the edit-in-place dialog's action after either a loss of focus or an end user action.
action	An integer constant, variable, EQUATE, or expression that indicates an end user requested action.

The **TakeCompleted** method conditionally calls the ResetColumn method. The BrowseEIPManager.TakeCompleted provides the bulk of the process completion functionality, and is derived from the TakeCompleted method.

Implementation: The BrowseEIPManager.TakeCompleted method calls the TakeCompleted method via PARENT syntax. TakeFocusLoss and TakeAction also call the TakeCompleted method.

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

Example:

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

See Also: BrowseEIPManager.TakeCompleted, TakeFocusLoss, TakeAction

TakeEvent (process window specific events)

TakeEvent, DERIVED, PROC

The **TakeEvent** method processes window specific events and returns Level:Notify for an EVENT:Size, EVENT:Iconize, or EVENT:Maximize; it returns a Level:Fatal for an EVENT:CloseDown, EVENT:CloseWindow, or EVENT:Sized; all other window events return a Level:Benign.

Implementation: The TakeFieldEvent method calls the TakeEvent method. The TakeEvent method calls the TakeFocusLoss method subsequent to returning a Level:Fatal.

Return Data Type: BYTE

Example:

```
EIPManager.TakeFieldEvent       PROCEDURE
I UNSIGNED(1)
  CODE
  IF FIELD() = SELF.ListControl THEN RETURN Level:Benign .
  LOOP I = 1 TO RECORDS(SELF.EQ)+1
   ! Optimised to pick up subsequent events from same field
    IF ~SELF.EQ.Control &= NULL AND SELF.EQ.Control.Feq = FIELD()
      SELF.TakeAction(SELF.EQ.Control.TakeEvent(EVENT()))
      RETURN Level:Benign
    END
    GET(SELF.EQ,I)
  END
  ! Code to handle an unknown field
```

See Also: TakeFieldEvent, TakeFocusLoss

TakeFieldEvent (process field specific events)

TakeFieldEvent, DERIVED, PROC

The **TakeFieldEvent** method processes all field-specific/control-specific events for the window. It returns a value indicating whether edit-in-place process is complete and should stop.

TakeFieldEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The WindowManager.TakeEvent method calls the TakeFieldEvent method.

Return value EQUATEs are declared in ABERROR.INC.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
  CODE
  IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
  END
  CASE EVENT()
  OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
  OF EVENT:Rejected;    RVal = SELF.TakeRejected()
  OF EVENT:Selected;    RVal = SELF.TakeSelected()
  OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
  OF EVENT:Completed;     RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVal = SELF.TakeCloseEvent()
  END
  IF RVal THEN RETURN RVal.
  IF FIELD()
    RVal = SELF.TakeFieldEvent()
  END
  RETURN RVal
```

TakeFocusLoss (a virtual to process loss of focus)

TakeFocusLoss, VIRTUAL

The **TakeFocusLoss** method determines the appropriate action to take when the EIPManager window loses focus, and calls the TakeCompleted method with the appropriate parameter.

Implementation: TakeEvent and TakeFieldEvent methods conditionally call the TakeFocusLoss method.

Example:

```
EIPManager.TakeFieldEvent       PROCEDURE
I UNSIGNED(1)
  CODE
  IF FIELD() = SELF.ListControl THEN RETURN Level:Benign .
  LOOP I = 1 TO RECORDS(SELF.EQ)+1
   ! Optimised to pick up subsequent events from same field
    IF ~SELF.EQ.Control &= NULL AND SELF.EQ.Control.Feq = FIELD()
      SELF.TakeAction(SELF.EQ.Control.TakeEvent(EVENT()))
      RETURN Level:Benign
    END
    GET(SELF.EQ,I)
  END
  ! Code to handle an unknown field
```

See Also: TakeCompleted

TakeNewSelection (reset edit-in-place column)

TakeNewSelection, DERIVED, PROC

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

Implementation: TakeNewSelection is called by the BrowseEIPManager.TakeNewSelection method.

TakeNewSelection calls ResetColumn, and returns a Level:Benign.

Return Data Type: BYTE

Example:

```
BrowseEIPManager.TakeNewSelection PROCEDURE
  CODE
  IF FIELD() = SELF.ListControl
    IF CHOICE(SELF.ListControl) = SELF.BC.CurrentChoice
      RETURN PARENT.TakeNewSelection()
    ELSE
      ! Code to handle Focus change to different record
    END
  END
```

See Also: ResetColumn

36 - PRINTPREVIEWCLASS

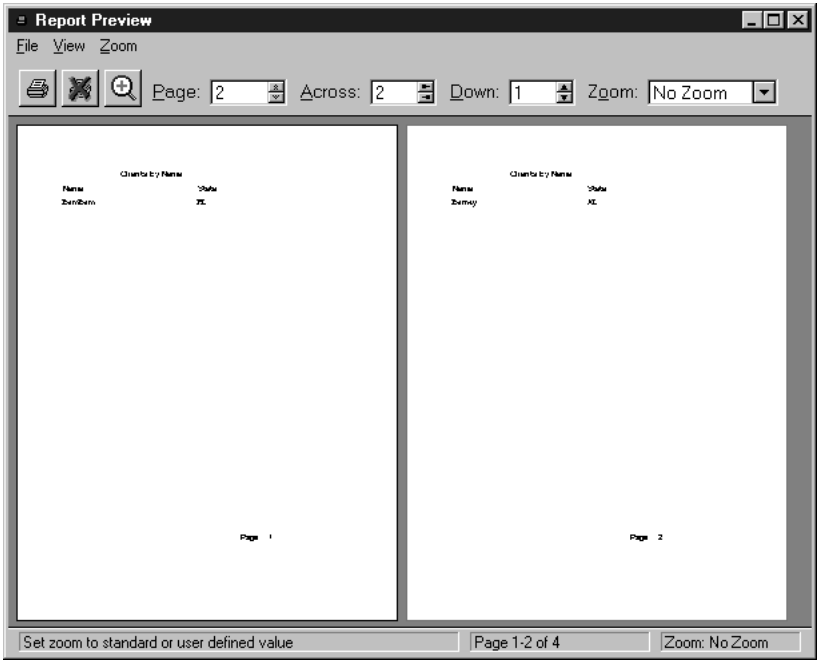
Overview

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 te 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

Zoom Configuration

The user interface text and the standard zoom choices the PrintPreviewClass displays at runtime are defined in the ABREPORT.TRN file. To modify or customize this text or the standard zoom choices, simply back up the ABREPORT.TRN file then edit it to suit your needs. See *ZoomIndex* for more information.

Conceptual Example

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

This example uses the PrintPreviewClass object to preview a very simple report before printing it. The program specifies an initial position and size for the print preview window and allows custom zoom factors.

```
   PROGRAM
_ABCDllMode_  EQUATE(0)
_ABCLinkMode_ EQUATE(1)
   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 procedure

ThisProcedure.Init    PROCEDURE()                 !initialize ThisProcedure
ReturnValue           BYTE,AUTO
  CODE
  GlobalErrors.Init
  Relate:Customer.Init
  ReturnValue = PARENT.Init()
  SELF.FirstField = ?PctDone
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
  Relate:Customer.Open
  OPEN(ProgressWindow)
  SELF.Opened=True
  CusReport.Init(CusView,Relate:Customer,?TxtDone,PctDone,RECORDS(Customer))
  CusReport.AddSortOrder(CUS:BYNUMBER)
  SELF.AddItem(?Cancel,RequestCancelled)
  SELF.Init(CusReport,report,Previewer)           !register Previewer with ThisProcedure
  SELF.Zoom = PageWidth
  Previewer.AllowUserZoom=True                    !allow custom zoom factors
  Previewer.Maximize=True                         !initially maximize preview window
  SELF.SetAlerts()
  RETURN ReturnValue
```

```
ThisProcedure.Kill    PROCEDURE()
ReturnValue           BYTE,AUTO
  CODE
  ReturnValue = PARENT.Kill()
  Relate:Customer.Close
  Relate:Customer.Kill
  GlobalErrors.Kill
  RETURN ReturnValue

CusReport.TakeRecord  PROCEDURE()
ReturnValue           BYTE,AUTO
SkipDetails BYTE
  CODE
  ReturnValue = PARENT.TakeRecord()
  PRINT(RPT:detail)
  RETURN ReturnValue

Access:Customer.Init  PROCEDURE
  CODE
  PARENT.Init(Customer,GlobalErrors)
  SELF.FileNameValue = 'Customer'
  SELF.Buffer &= CUS:Record
  SELF.Create = 0
  SELF.LazyOpen = False
  SELF.AddKey(CUS:BYNUMBER,'CUS:BYNUMBER',0)

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

PrintPreviewClass Properties

The PrintPreviewClass contains properties that primarily allow configuration of the print preview window and its features. The PrintPreviewClass properties are described below.

AllowUserZoom (allow any zoom factor)

AllowUserZoom BYTE

The **AllowUserZoom** property indicates whether the PrintPreviewClass object provides user zoom capability for the end user. The user zoom lets the end user apply any zoom factor. Without user zoom, the end user may only apply the standard zoom choices.

The ZoomIndex property indicates whether a user zoom factor or a standard zoom factor is applied.

Implementation: A value of one (1) enables user zoom capability; a value of zero (0) disables user zoom. The UserPercentile property contains the user zoom factor.

See Also: UserPercentile, ZoomIndex

ConfirmPages (force 'pages to print' confirmation)

ConfirmPages BYTE

The **ConfirmPages** property indicates whether or not the AskPrintPages method should be called before printing.

Implementation: Zero (0) is the default; a value of one (1) forces the enduser to choose the pages to print before the print job is sent to the printer.

See Also: AskPrintPages

CurrentPage (the selected report page)

CurrentPage LONG

The **CurrentPage** property contains the number of the selected report page. The PrintPreviewClass object uses this property to highlight the selected report page when more than one page is displayed, to navigate pages, and to dislay the current page number for the end user.

ImageQueue (page list)

ImageQueue &PreviewQueue, PROTECTED

The **ImageQueue** property is a reference to the ReportManager.PreviewQueue property which contains a list of the full pathnames for the page images generated by the report.

Maximize (number of pages displayed horizontally)

Maximize BYTE

The **Maximize** property indicates whether to open the preview window mazimized. A value of one (1 or True) maximizes the window; a value of zero (0 or False) opens the window according to the WindowSizeSet property.

See Also: WindowSizeSet

PagesAcross (number of pages displayed horizontally)

PagesAcross USHORT

The **PagesAcross** property contains the number of thumbnail pages the PrintPreviewClass object displays *horizontally* within the preview window. The PrintPreviewClass object uses this property to calculate appropriate positions and sizes when displaying several pages at a time.

The PrintPreviewClass object displays the PagesAcross value at runtime and lets the end user set the value as well.

PagesDown (number of vertical thumbnails)

PagesDown USHORT

The **PagesDown** property contains the number of thumbnail pages the PrintPreviewClass object displays *vertically* within the preview window. The PrintPreviewClass object uses this property to calculate appropriate positions and sizes when displaying several pages at a time.

The PrintPreviewClass object displays the PagesDown value at runtime and lets the end user set the value as well.

PagesToPrint (the pages to print)

PagesToPrint CSTRING(256), PROTECTED

The **PagesToPrint** property contains the page range to print.

The default value is 1-*n*, where *n* is equal to the total number of pages in the report. Individual pages can be printed by seperating page numbers by commas. A range of pages to print can be specified by seperating the first page number to print and the last page number to print by a dash (-). Combinations of individual pages and ranges of pages are allowed.

Popup (popup menu)

Popup &PopupClass, PROTECTED

The **Popup** property is a reference to the PopupClass object PrintPreview uses to provide alternate zoom factors.

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 theprint 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 of 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 (PagesAcross and PagesDown properties) in a tiled arrangement in the preview window.
Page Width	Displays a single page whose width is the same as the width of the preview window.
50%	Displays a single page at 50% of actual print size.
75%	Displays a single page at 75% of actual print size.
100%	Displays a single page at 100% of actual print size.
200%	Displays a single page at 200% of actual print size.
300%	Displays a single page at 300% of actual print size.

A ZoomIndex value of zero (0) indicates a nonstandard zoom factor is specified. Nonstandard zoom factors may be specified when the AllowUserZoom property is True. The UserPercentile property contains the nonstandard zoom factor.

See Also: AllowUserZoom, PagesAcross, PagesDown, UserPercentile, SetZoomPercentile

PrintPreviewClass Methods

The PrintPreviewClass contains the methods listed below.

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[V]	initialize the PrintPreviewClass object
SetPosition	set initial preview window coordinates
Display[V]	preview the report
Kill[V]	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

[V] These methods are also Virtual.

Virtual Methods

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

Init[V]	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[V]	shut down the PrintPreviewClass object

AskPage (prompt for new report page)

AskPage, PROC, VIRTUAL, PROTECTED

The **AskPage** method prompts the end user for a specific report page to display and returns a value indicating whether a new page is selected. A return value of one (1) indicates a new page is selected and a screen redraw is required; a return value of zero (0) indicates a new page is not selected and a screen redraw is not required.

Implementation: The PrintPreviewClass.Display method calls the AskPage method. The AskPage method displays a dialog that prompts for a specific report page.

Return Data Type: BYTE

Example:

```
!Virtual implementation of AskPage: a simplified version with no translator...
PrintPreviewClass.AskPage FUNCTION
JumpPage LONG,AUTO
RVal     BOOL(False)

JumpWin WINDOW('Jump to Page'),AT(,,181,26),CENTER,GRAY,DOUBLE
          PROMPT('&Page:'),AT(5,8),USE(?JumpPrompt)
          SPIN(@n5),AT(30,7),USE(JumpPage),RANGE(1,10),STEP(1)
          BUTTON('OK'),AT(89,7),USE(?OKButton),DEFAULT
          BUTTON('Cancel'),AT(134,7),USE(?CancelButton)
        END
  CODE
  JumpPage=SELF.CurrentPage
  OPEN(JumpWin)
  ACCEPT
    CASE EVENT()
    OF EVENT:OpenWindow
      ?JumpPage{PROP:RangeHigh}=RECORDS(SELF.ImageQueue)
    OF EVENT:Accepted
      CASE ACCEPTED()
      OF ?OKButton
        IF JumpPage NOT=SELF.CurrentPage
          RVal=True                                    !SELF.CurrentPage changed
          SELF.CurrentPage=JumpPage
        END
        POST(EVENT:CloseWindow)
      OF ?CancelButton
        POST(EVENT:CloseWindow)
  . . .
   CLOSE(JumpWin)
   RETURN RVal
```

AskPrintPages (prompt for pages to print)

AskPrintPages, VIRTUAL, PROTECTED, PROC

The **AskPrintPages** method prompts the end user for the number(s) of the pages to print from the previewed report.

Implementation: The PrintPreviewClass.TakeAccepted method calls the AskPrintPages method and returns TRUE (1) when completed or FALSE (0) if the user presses the cancel button. The AskPrintPages method displays a dialog that prompts for the page numbers to print.

Return Data Type: BYTE

Example:

```
!Virtual implementation of AskThumbnails
PrintPreviewClass.AskPrintPages PROCEDURE
Preserve  LIKE(PrintPreviewClass.PagesToPrint),AUTO
Window WINDOW('Pages to Print'),AT(,,260,37),CENTER,SYSTEM,GRAY
       PROMPT('&Pages to Print:'),AT(4,8),USE(?Prompt)
       ENTRY(@s255),AT(56,4,200,11),USE(SELF.PagesToPrint, , ?PagesToPrint)
       BUTTON('&Reset'),AT(116,20,45,14),USE(?Reset)
       BUTTON('&Ok'),AT(164,20,45,14),USE(?Ok),DEFAULT
       BUTTON('&Cancel'),AT(212,20,45,14),USE(?Cancel),STD(STD:Close)
     END
RVal  BYTE(False)
  CODE
  Preserve = SELF.PagesToPrint
  OPEN(Window)
  ACCEPT
    CASE EVENT()
    OF EVENT:Accepted
      CASE ACCEPTED()
      OF ?Cancel
        SELF.PagesToPrint = Preserve
        POST(EVENT:CloseWindow)
      OF ?Ok
        RVal = True
        POST(EVENT:CloseWindow)
      OF ?Reset
        SELF.SetDefaultPages
        SELECT(?PagesToPrint)
      END
    OF EVENT:OpenWindow
      ! INIMgr code for FETCHing window settings
    OF EVENT:CloseWindow
      ! INIMgr code for UPDATEing window settings
    END
  END
  CLOSE(Window)
  RETURN RVal
```

AskThumbnails (prompt for new thumbnail configuration)

AskThumbnails, VIRTUAL, PROTECTED

The **AskThumbnails** method prompts the end user for the number of pages to tile across and down the preview window.

Implementation: The PrintPreviewClass.Display method calls the AskThumbnails method. The AskThumbnails method displays a dialog that prompts for the number of thumbnails to display horizontally, and the number of thumbnails to display vertically.

Example:

```
!Virtual implementation of AskThumbnails
! a slightly simplified version with no translator...
PrintPreviewClass.AskThumbnails PROCEDURE

SelectWindow WINDOW('Pages Displayed'),AT(,,141,64),GRAY,DOUBLE
         GROUP('Across'),AT(7,10,62,32),BOXED
           SPIN(@N2),AT(13,22,15),USE(SELF.PagesAcross,,?PagesAcross),RANGE(1,10)
         END
         GROUP('Down'),AT(72,10,62,32),BOXED
           SPIN(@N2),AT(79,22,15),USE(SELF.PagesDown,,?PagesDown),RANGE(1,10)
         END
       BUTTON('OK'),AT(98,47,40,14),KEY(EnterKey),USE(?OK)
     END

   CODE
   OPEN(SelectWindow)
   ACCEPT
     CASE EVENT()
     OF EVENT:Accepted
       CASE FIELD()
       OF ?OK
         IF SELF.PagesAcross*SELF.PagesDown>RECORDS(SELF.ImageQueue)
           SELECT(?PagesAcross)
         ELSE
           POST(EVENT:CloseWindow)
         END
       END
     END
   END
   CLOSE(SelectWindow)
```

DeleteImageQueue (remove non-selected pages)

DeleteImageQueue(*page*), VIRTUAL, PROC

DeleteImageQueue Removes a page number from the ImageQueue.

page An integer constant, variable, EQUATE, or expression containing the page number to delete.

The **DeleteImageQueue** method removes records from the ImageQueue, and the associated image file, which have not been selected for printing.

Implementation: The SyncImageQueue method calls the DeleteImageQueue method. The value contained in the PagesToPrint property determines which records and images are deleted.

Return Data Type: BYTE

Example:

```
PrintPreviewClass.SyncImageQueue PROCEDURE
i LONG,AUTO

  CODE
  LOOP i = RECORDS(SELF.ImageQueue) TO 1 BY -1
    IF ~SELF.InPageList(i)
      SELF.DeleteImageQueue(i)
    END
  END
```

See Also: PagesToPrint,ImageQueue

Display (preview the report)

Display([*zoom*] [*, page*] [*, across*] [*, down*] **), VIRTUAL, PROC**

Display	Displays the report image metafiles.
zoom	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.
page	An integer constant, variable, EQUATE, or expression containing the initial page number to display. If omitted, *page* defaults to one (1).
across	An integer constant, variable, EQUATE, or expression containing the number of horizontal thumbnails for the initial print preview display. If omitted, *across* defaults to one (1).
down	An integer constant, variable, EQUATE, or expression containing the number of vertical thumbnails for the initial print preview display. If omitted, *down* 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
```

InPageList (check page number)

InPageList(*page*)

InPageList Evaluates page against value(s) in PagesToPrint.

page An integer constant, variable, EQUATE, or expression containing the page number to check.

The **InPageList** method evaluates a page number against the value(s) contained in the PagesToPrint property, and returns TRUE (1) if the page is in PagesToPrint or FALSE (0) if it is not.

Implementation: The PageManagerClass.Draw and SyncImageQueue methods call the InPageList method to verify report pages for inclusion in the preview window and the printed report respectively.

Return Data Type: BYTE

Example:

```
PrintPreviewClass.SyncImageQueue PROCEDURE
i LONG,AUTO
  CODE
  LOOP i = RECORDS(SELF.ImageQueue) TO 1 BY -1
    IF ~SELF.InPageList(i)
      SELF.DeleteImageQueue(i)
    . .
```

See Also: PagesToPrint, PageManagerClass.Draw

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

SetDefaultPages (set the default pages to print)

SetDefaultPages, VIRTUAL

The **SetDefaultPages** method sets the initial value of the PagesToPrint property. The initial value is 1-*n*, where *n* is equal to the total number of pages in the report.

Implementation: The Display and AskPrintPreview methods call the SetDefaultPages method.

Example:

```
!Virtual implementation of SetDefaultPages method
PrintPreviewClass.SetDefaultPages PROCEDURE
  CODE
  SELF.PagesToPrint = '1-' & RECORDS(SELF.ImageQueue)
```

See Also: PagesToPrint

SetPosition (set initial preview window coordinates)

SetPosition([*x*] [,*y*] [,*width*] [,*height*] **)**

SetPosition	Sets the initial position and size of the print preview window.
x	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.
y	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.
width	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.
height	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 contant, 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

SyncImageQueue (sync image queue with PagesToPrint)

SyncImageQueue, VIRTUAL

The **SyncImageQueue** method synconizes the image queue with the contents of PagesToPrint to ensure that only the specified pages are sent to the printer.

Implementation: The Display method calls the SyncImageQueue method. The value contained in the PagesToPrint property determines which pages are printed.

Example:

```
PrintPreviewClass.Display PROCEDURE
! Window declaration
! executable Display code
  IF SELF.PrintOk
    SELF.SyncImageQueue
  END
  RETURN SELF.PrintOK
```

See Also: PagesToPrint,ImageQueue

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
RVal BYTE(Level:Benign)
I    USHORT,AUTO
  CODE
  IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
  END
  CASE EVENT()
  OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
  OF EVENT:Rejected;    RVal = SELF.TakeRejected()
  OF EVENT:Selected;    RVal = SELF.TakeSelected()
  OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
  OF EVENT:Completed;     RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVal = SELF.TakeCloseEvent()
  END
  IF RVal THEN RETURN RVal.
  IF FIELD()
    RVal = SELF.TakeFieldEvent()
  END
  RETURN RVal
```

See Also: TakeEvent, WindowManager.TakeEvent

TakeEvent (process all events)

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
RVal BYTE(Level:Benign)
I    USHORT,AUTO
  CODE
  IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
  END
  CASE EVENT()
  OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
  OF EVENT:Rejected;    RVal = SELF.TakeRejected()
  OF EVENT:Selected;    RVal = SELF.TakeSelected()
  OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
  OF EVENT:Completed;     RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVal = SELF.TakeCloseEvent()
  END
  IF RVal THEN RETURN RVal.
  IF FIELD()
    RVal = SELF.TakeFieldEvent()
  END
  RETURN RVal
```

See Also: Ask

TakeWindowEvent (process non-field events)

TakeWindowEvent, VIRTUAL, PROC

The **TakeWindowEvent** method processes all non-field events for the preview window and returns a value indicating whether window ACCEPT loop processing is complete and should stop. TakeWindowEvent returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeEvent method calls the TakeWindowEvent method. The TakeWindowEvent method calls the WindowManager.TakeWindowEvent method for all events except EVENT:GainFocus.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
  CODE
  IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
  END
  CASE EVENT()
  OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
  OF EVENT:Rejected;    RVal = SELF.TakeRejected()
  OF EVENT:Selected;    RVal = SELF.TakeSelected()
  OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
  OF EVENT:Completed;     RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVal = SELF.TakeCloseEvent()
  END
  IF RVal THEN RETURN RVal.
  IF FIELD()
    RVal = SELF.TakeFieldEvent()
  END
  RETURN RVal
```

See Also: TakeEvent

41 - QUERYLISTCLASS

Overview

The QueryListClass is a QueryClass with a “list” user interface. The QueryListClass provides support for ad hoc queries against Clarion VIEWs. The list interface includes is an edit-in-place, 3-column listbox with a field column, an equivalence operator (contains, begins, equal, not equal, greater than, less than) column, and a value (to query for) column.

QueryListClass Concepts

Use the AddItem method to define a user input dialog at runtime. Or create a custom dialog to plug into your QueryClass object. Use the Ask method to solicit end user query criteria (search values) or use the SetLimit method to programmatically set query search values. Finally, use the GetFilter method to build the filter expression to apply to your VIEW. Use the ViewManager.SetFilter method or the PROP:Filter property to apply the resulting filter.

Relationship to Other Application Builder Classes

The QueryListClass is derived from the QueryClass, plus it relies on the QueryListVisual class to display its input dialog and handle the dialog events.

The BrowseClass optionally uses the QueryListClass to filter its result set. If your BrowseClass object uses a QueryListClass object, it must instantiate a QueryListClass object and a QueryListVisual object.

The BrowseClass automatically provides a default query dialog that solicits end user search values for each field displayed in the browse list. See the *Conceptual Example*.

ABC Template Implementation

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

The templates optionally derive a QueryListClass object for *each* BrowseQBEButton control in the application. The derived class is called

QBE# where # is the instance number of the BrowseQBEButton template. The templates provide the derived class so you can use the BrowseQBEButton template **Classes** tab to easily modify the query's behavior on an instance-by-instance basis.

> **Tip:** **Use the BrowseQBEButton control template to add a QueryListClass object to your template generated BrowseBoxes.**

QueryListClass Source Files

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

ABQUERY.INC	QueryListClass declarations
ABQUERY.CLW	QueryListClass method definitions

Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a QueryListClass object and related objects. The example plugs a QueryListClass into a BrowseClass object. The QueryListClass object solicits query criteria (search values) with a "list" dialog, then generates a filter expression based on the end user input.

```
  PROGRAM

_ABCDllMode_  EQUATE(0)
_ABCLinkMode_ EQUATE(1)

  INCLUDE('ABWINDOW.INC')
  INCLUDE('ABBROWSE.INC')
  INCLUDE('ABQUERY.INC')

   MAP
   END

GlobalErrors    ErrorClass
Access:Customer CLASS(FileManager)
Init              PROCEDURE
                END

Relate:Customer CLASS(RelationManager)
Init              PROCEDURE
Kill              PROCEDURE,VIRTUAL
                END

GlobalRequest   BYTE(0),THREAD
GlobalResponse  BYTE(0),THREAD
VCRRequest      LONG(0),THREAD
```

```
Customer        FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,THREAD
CustomerIDKey     KEY(CUS:ID),NOCASE,OPT,PRIMARY
NameKey           KEY(CUS:LastName),NOCASE,OPT
Record            RECORD,PRE()
ID                  LONG
LastName            STRING(20)
FirstName           STRING(15)
City                STRING(20)
State               STRING(2)
ZIP                 STRING(10)
                  END
                END

CustView        VIEW(Customer)
                END
CustQ           QUEUE
CUS:LastName      LIKE(CUS:LastName)
CUS:FirstName     LIKE(CUS:FirstName)
CUS:ZIP           LIKE(CUS:ZIP)
ViewPosition      STRING(1024)
              END

CusWindow     WINDOW('Browse Customers'),AT(,,210,105),IMM,SYSTEM,GRAY
                LIST,AT(5,5,200,80),USE(?CusList),IMM,HVSCROLL,FROM(CustQ),|
                FORMAT('80L(2)|M~Last~@s20@64L(2)|M~First~@s15@44L(2)|M~ZIP~@s10@')
                BUTTON('&Query'),AT(50,88),USE(?Query)
                BUTTON('Close'),AT(90,88),USE(?Close)
              END

ThisWindow    CLASS(WindowManager)                    !declare ThisWindow object
Init            PROCEDURE(),BYTE,PROC,VIRTUAL
Kill            PROCEDURE(),BYTE,PROC,VIRTUAL
              END

Query         QueryListmClass                         !declare Query object
QBEWindow     QueryListVisual                         !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
```

QueryListClass Properties

The QueryListClass inherits all the properties of the QueryClass from which it is derived.

QueryListClass Methods

The QueryListClass 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 QueryListClass contains the following methods:

Functional Organization—Expected Use

As an aid to understanding the QueryListClass, it is useful to organize its various methods into two large categories according to their expected use—the primary interface and the virtual methods. This organization reflects what we believe is typical use of the QueryListClass 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 QueryListClass object
AddItem[I]	add a field to query
Kill[V]	shut down the QueryListListClass object

Mainstream Use:

Ask[V]	accept query criteria
GetFilter[I]	return filter expression

Occasional Use:

Reset[I]	reset the QueryListClass object
GetLimit[I]	get search values
SetLimit[I]	set search values

[V] These methods are also Virtual.
[I] These methods are inherited from the QueryClass.

Virtual Methods

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

Ask	accept query criteria
Kill	shut down the QueryListClass object

Ask (solicit query criteria)

Ask([*uselast*] **), DERIVED, PROC**

Ask	Accepts query criteria (search values) from the end user.
uselast	An integer constant, variable, EQUATE, or expression that determines whether the QueryListClass object carries forward previous query criteria. A value of one (1) carries forward input from the previous query; a value of zero (0) discards previous input.

The **Ask** method displays a query dialog, processes its events, and returns a value indicating whether to apply the query or abandon it. A return value of Level:Notify indicates the QueryListClass object should apply the query criteria; a return value of Level:Benign indicates the end user cancelled the query input dialog and the QueryListClass object should not apply the query criteria.

Implementation: The Ask method declares a generic (empty) dialog to accept query criteria. The Ask method calls the QueryListClass object's WindowManager to define the dialog and process it's events.

The GetFilter method generates filter expressions using the search values set by the Ask method.

The Init method sets the value of the QueryListClass object's WindowManager.

Return Data Type: BYTE

Example:

```
MyBrowseClass.TakeLocate PROCEDURE
CurSort USHORT,AUTO
I USHORT,AUTO
  CODE
  IF ~SELF.Query&=NULL AND SELF.Query.Ask()
    CurSort = POINTER(SELF.Sort)
    LOOP I = 1 TO RECORDS(SELF.Sort)
      PARENT.SetSort(I)
      SELF.SetFilter(SELF.Query.GetFilter(),'9 - QBE')
    END
    PARENT.SetSort(CurSort)
    SELF.ResetSort(1)
  END
```

See Also: GetFilter, Init, QueryListVisual

Init (initialize the QueryListClass object)

Init(*querywindowmanager* **)**

Init Initializes the QueryListClass object.

querywindowmanager
The label of the QueryListVisual object that displays the query input dialog list and processes it's events.

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

Implementation: The Init method sets the QFC property for the *querywindowmanager*.

Example:

```
ThisWindow.Init PROCEDURE()
ReturnValue          BYTE,AUTO
  CODE
  !other initialization code
  Query.Init(QueryWindow)
  Query.AddItem('UPPER(CLI:LastName)','Name','s20')
  Query.AddItem('CLI:ZIP+1','ZIP+1','')
  RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue          BYTE,AUTO
  CODE
  !other termination code
  Query.Kill
  RETURN ReturnValue
```

See Also: Kill, QueryListVisual, QueryListVisual.QFC

Kill (shut down the QueryListClass object)

Kill, DERIVED

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

Example:

```
ThisWindow.Init PROCEDURE()
ReturnValue          BYTE,AUTO
  CODE
  !other initialization code
  Query.Init(QueryWindow)
  Query.AddItem('UPPER(CLI:LastName)','Name','s20')
  Query.AddItem('CLI:ZIP+1','ZIP+1','')
  RETURN ReturnValue

ThisWindow.Kill PROCEDURE()
ReturnValue          BYTE,AUTO
  CODE
  !other termination code
  Query.Kill
  RETURN ReturnValue
```

See Also: Init

42 - QUERYLISTVISUAL

Overview

The QueryListVisual class is a WindowManager that displays a query input dialog and handles the dialog events. The query dialog includes an edit-in-place, 3-column listbox which allows the end user to choose the fields to query, the equivalence operator, and the value to query for.

QueryListVisual Concepts

The QueryListVisual provides the query window for a QueryListClass object. The Init method defines and "programs" the query input dialog at runtime. The query interface includes an edit-in-place, 3-column listbox with a field column, an equivalence operator (contains, begins, equal, not equal, greater than, less than)column, and a value (to query for) column.

Relationship to Other Application Builder Classes

The QueryListVisual class is derived from the WindowManager.

The BrowseClass optionally uses the QueryListVisual class to provide the user an edit-in-place list interface to it's query facility.

The QueryListClass requires the QueryListVisual class as a window manager.

ABC Template Implementation

The ABC Templates declare a local QueryListVisual 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 QueryListVisual 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 QueryListClass object to your template generated BrowseBoxes.

QueryListVisual Source Files

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

ABQUERY.INC	QueryListVisual declarations
ABQUERY.CLW	QueryListVisual method definitions

Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a QueryListVisual object and related objects. The example plugs a QueryListClass into a BrowseClass object. The QueryListClass object uses the QueryListVisual to solicit query criteria (search values) from the end user.

Note that the WindowManager and BrowseClass objects internally handle the normal events surrounding the query.

```
  PROGRAM

_ABCDllMode_  EQUATE(0)
_ABCLinkMode_ EQUATE(1)

  INCLUDE('ABWINDOW.INC')
  INCLUDE('ABBROWSE.INC')
  INCLUDE('ABQUERY.INC')

   MAP
   END

GlobalErrors    ErrorClass
Access:Customer CLASS(FileManager)
Init              PROCEDURE
                END

Relate:Customer CLASS(RelationManager)
Init              PROCEDURE
Kill              PROCEDURE,VIRTUAL
                END

GlobalRequest   BYTE(0),THREAD
GlobalResponse  BYTE(0),THREAD
VCRRequest      LONG(0),THREAD

Customer        FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,THREAD
CustomerIDKey     KEY(CUS:ID),NOCASE,OPT,PRIMARY
NameKey           KEY(CUS:LastName),NOCASE,OPT
Record            RECORD,PRE()
ID                  LONG
LastName            STRING(20)
FirstName           STRING(15)
```

```
City                  STRING(20)
State                 STRING(2)
ZIP                   STRING(10)
                    END
                  END

CustView          VIEW(Customer)
                  END
CustQ             QUEUE
CUS:LastName        LIKE(CUS:LastName)
CUS:FirstName       LIKE(CUS:FirstName)
CUS:ZIP             LIKE(CUS:ZIP)
ViewPosition        STRING(1024)
                END

CusWindow       WINDOW('Browse Customers'),AT(,,210,105),IMM,SYSTEM,GRAY
                  LIST,AT(5,5,200,80),USE(?CusList),IMM,HVSCROLL,FROM(CustQ),|
                  FORMAT('80L(2)|M~Last~@s20@64L(2)|M~First~@s15@44L(2)|M~ZIP~@s10@')
                  BUTTON('&Query'),AT(50,88),USE(?Query)
                  BUTTON('Close'),AT(90,88),USE(?Close)
                END

ThisWindow      CLASS(WindowManager)                    !declare ThisWindow object
Init              PROCEDURE(),BYTE,PROC,VIRTUAL
Kill              PROCEDURE(),BYTE,PROC,VIRTUAL
                END

Query           QueryListClass                          !declare Query object
QBEWindow       QueryListVisual                         !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
```

QueryListVisual Properties

The QueryListVisual 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 QueryListVisual contains the following property:

QFC (reference to the QueryListClass)

QFC &QueryListClass

The **QFC** property is a reference to the QueryListClass that uses this QueryListVisual object to solicit query criteria (search values) from the end user.

Implementation: The QueryListClass.Init method sets the QFC property.

See Also: QueryListClass.Init

OpsEIP (reference to the EditDropListClass)

OpsEIP &EditDropListClass,PROTECTED

The **OpsEIP** property is a reference to the EditDropListClass that displays the available operators in the QueryList dialog.

FldsEIP (reference to the EditDropListClass)

FldsEIP &EditDropListClass,PROTECTED

The **FldsEIP** property is a reference to the EditDropListClass that displays the available fields to query in the QueryList dialog.

QueryListVisual Methods

The QueryListVisual inherits all the methods of the WindowManagerClass from which it is derived. See *WindowManager Methods* for more information.

In addition to (or instead of) the inherited methods, the QueryListVisual contains the following methods:

Functional Organization—Expected Use

As an aid to understanding the QueryListVisual class, it is useful to organize its various methods into two large categories according to their expected use—the primary interface and the virtual methods. This organization reflects what we believe is typical use of the QueryListVisual 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[D]	program the QueryListVisual object

MainStream Use:

none

Occasional Use:

none

[D] These methods are Derived.

Derived Methods

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

Init	program the QueryListVisual object
SetAlerts	prepare the query dialog for EIP
TakeEvent	Handle events for the query dialog
TakeCompleted	wrap up the query dialog
TakeAccepted	handle EVENT:Accepted events

Init (initialize the QueryListVisual object)

Init, DERIVED PROC

The **Init** method initializes the QueryListVisual object. Init returns Level:Benign to indicate normal initialization.

The Init method “programs” the QueryListVisual object.

Implementation: The QueryListClass.Ask method (indirectly) calls the Init method to configure the QueryListClass WINDOW.

The Init method reads each queryable item (defined by the QFC property) from a queue, then creates an edit-in-place, 3-column listbox with a field column, an equivalence operator (equal, not equal, greater than, etc.) column, and a value (to query for) column.

The Init method sets the coordinates for the QueryListClass WINDOW and for the individual controls.

Return Data Type: BYTE

Example:

```
QueryListClass.Ask       PROCEDURE(BYTE UseLast)
W WINDOW('Query'),AT(,,300,200),FONT('MS SansSerif',8,,FONT:regular),SYSTEM,GRAY,DOUBLE
    LIST,AT(5,5,290,174),USE(?List,FEQ:ListBox),|
    FORMAT('91L|M~Field~@s20@44C|M~Operator~L@s10@120C|M~Value~L@s30@')
    BUTTON('Insert'),AT(5,183,45,14),USE(?Insert,FEQ:Insert)
    BUTTON('Change'),AT(52,183,45,14),USE(?Change,FEQ:Change)
    BUTTON('Delete'),AT(99,183,45,14),USE(?Delete,FEQ:Delete)
    BUTTON('&OK'),AT(203,183,45,14),USE(?Ok,FEQ:OK),DEFAULT
    BUTTON('Cancel'),AT(250,183,45,14),USE(?Cancel,FEQ:Cancel)
  END
  CODE
  OPEN(W)
  IF ~UseLast THEN SELF.Reset().
  RETURN CHOOSE(SELF.Win.Run()=RequestCancelled,Level:Benign,Level:Notify)
```

See Also: QFC

SetAlerts (alert keystrokes for the edit control)

SetAlerts, DERIVED

The **SetAlerts** method method alerts appropriate keystrokes for the edit-in-place control.

Implementation: The Init method calls the CreateControl method to create the input control and set the FEQ property. The Init method then calls the SetAlerts method to alert specific keystrokes for the query dialog. Alerted keys are:

```
MouseLeft2        !edit selected record
InsertKey         !add a query field
CtrlEnter         !edit selected record
DeleteKey         !delete query field
```

Example:

```
EditClass.Init PROCEDURE(UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
  CODE
  SELF.ListBoxFeq = ListBox
  SELF.CreateControl()
  ASSERT(SELF.Feq)
  SELF.UseVar &= UseVar
  SELF.Feq{PROP:Text} = ListBox{PROPLIST:Picture,FieldNo}
  SELF.Feq{PROP:Use} = UseVar
  SELF.SetAlerts
```

See Also: Init

TakeAccepted (handle query dialog EVENT:Accepted events)

TakeAccepted, DERIVED, PROC

The **TakeAccepted** method processes EVENT:Accepted events for the query dialog's controls, and returns a value indicating whether ACCEPT loop processing is complete and should stop. TakeAccepted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: The TakeAccepted method handles the processing of the update butons (Insert, Change, Delete) on the Query list dialog.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
  CODE
  IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
  END
  CASE EVENT()
  OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
  OF EVENT:Rejected;    RVal = SELF.TakeRejected()
  OF EVENT:Selected;    RVal = SELF.TakeSelected()
  OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
  OF EVENT:Completed;     RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVal = SELF.TakeCloseEvent()
  END
  IF RVal THEN RETURN RVal.
  IF FIELD()
    RVal = SELF.TakeFieldEvent()
  END
  RETURN RVal
```

See Also: QFC

TakeCompleted (complete the query dialog)

TakeCompleted, DERIVED, PROC

The **TakeCompleted** method processes the EVENT:Completed event for the query dialog and returns a value indicating whether window ACCEPT loop processing is complete and should stop.

TakeCompleted returns Level:Benign to indicate processing of this event should continue normally; it returns Level:Notify to indicate processing is completed for this event and the ACCEPT loop should CYCLE; it returns Level:Fatal to indicate the event could not be processed and the ACCEPT loop should BREAK.

Implementation: Based on the current state of the querydialog, the TakeCompleted method sets the search values in the QFC property. The QFC property may use these search values to create a filter expresssion.

Return Data Type: BYTE

Example:

```
MyWindowManager.TakeEvent PROCEDURE
RVal BYTE(Level:Benign)
I    USHORT,AUTO
  CODE
  IF ~FIELD()
    RVal = SELF.TakeWindowEvent()
    IF RVal THEN RETURN RVal.
  END
  CASE EVENT()
  OF EVENT:Accepted;    RVal = SELF.TakeAccepted()
  OF EVENT:Rejected;    RVal = SELF.TakeRejected()
  OF EVENT:Selected;    RVal = SELF.TakeSelected()
  OF EVENT:NewSelection;RVal = SELF.TakeNewSelection()
  OF EVENT:Completed;     RVal = SELF.TakeCompleted()
  OF EVENT:CloseWindow OROF EVENT:CloseDown
    RVal = SELF.TakeCloseEvent()
  END
  IF RVal THEN RETURN RVal.
  IF FIELD()
    RVal = SELF.TakeFieldEvent()
  END
  RETURN RVal
```

See Also: QFC

TakeEvent (process edit-in-place events)

TakeEvent(*event*), VIRTUAL

TakeEvent	Processes an event for the QueryListVisualClass object.
event	An integer constant, variable, EQUATE, or expression that contains the event number (see EVENT in the *Language Reference*).

The **TakeEvent** method processes an event for the QueryListVisualClass object and returns a value indicating the user requested action. Valid actions are none, insert (InsertKey), change (MouseLeft2 or CtrlEnter), or delete (DeleteKey).

Implementation: The EIPManager.TakeFieldEvent method calls the TakeEvent method. The TakeEvent method process an EVENT:AlertKey for the edit-in-place control and returns a value indicating the user requested action.

Return Data Type: BYTE

Example:

```
EIPManager.TakeFieldEvent       PROCEDURE
I UNSIGNED(1)
  CODE
  IF FIELD() = SELF.ListControl THEN RETURN Level:Benign .
  LOOP I = 1 TO RECORDS(SELF.EQ)+1
   ! Optimised to pick up subsequent events from same field
    IF ~SELF.EQ.Control &= NULL AND SELF.EQ.Control.Feq = FIELD()
      SELF.TakeAction(SELF.EQ.Control.TakeEvent(EVENT()))
      RETURN Level:Benign
    END
    GET(SELF.EQ,I)
  END
  ! Not a known field
  IF ?{PROP:Type} <> CREATE:Button OR EVENT() <> EVENT:Selected ! Wait to post accepted
for button
    SELF.Repost = EVENT()
    SELF.RepostField = FIELD()
    SELF.TakeFocusLoss
  END
  RETURN Level:Benign
```

See Also: EIPManager.TakeFieldEvent, SetAlerts