MacApp 2.0b5 View Architecture Release Notes

Curt Bianchi

Introduction

This document describes the architecture for MacApp 2.0's view classes, and describes all methods and fields of the view classes defined in UMacApp. This represents the major change between MacApp 1.x and MacApp 2.0. The motivations for introducing the new architecture are

- To simplify the architecture
- To provide a class that is suitable as the basis for all displayed objects, from windows to controls
- To have a single nesting and layering mechanism for display objects
- To support large coordinate systems
- To simplify the implementation of display objects within views, including controls and fields

The MacApp 1.x Display Achitecture

If you're already familiar with the MacApp 1.x display architecture, you should skip this section. If you're not, here it is in a nutshell.

The old display architecture consists basically of three classes: `TWindow`, `TFrame`, and `TView`.

`TWindow` A `TWindow` object represents a Window Manager window. As such, it handles opening, closing, moving, resizing, activating, and deactivating a window.

`TFrame` A `TFrame` object is a rectangular area within a window, handling scrolling and coordinate transformations. `TFrame` objects are similar to QuickDraw grafports in that each has its own coordinate system. Frames typically tile windows. In the simplest case, windows have a single frame whose size is that of the window's content rectangle. However, windows can be subdivided into sections by using multiple frames. Furthermore, frames can be nested inside one another. Within frames are displayed Control Manager controls (for example, scroll bars) and a portion of a view, or a collection of subframes, or both.

`TView` A `TView` object renders the display image of a document's data and responds to mouse clicks and keystrokes in the display image. It is a rectangular area of any size up to 30,000 pixels in each dimension. A view is always displayed in a frame. The frame's scroll bars determine what part of the view is actually visible, since large views cannot be displayed in their entirety.

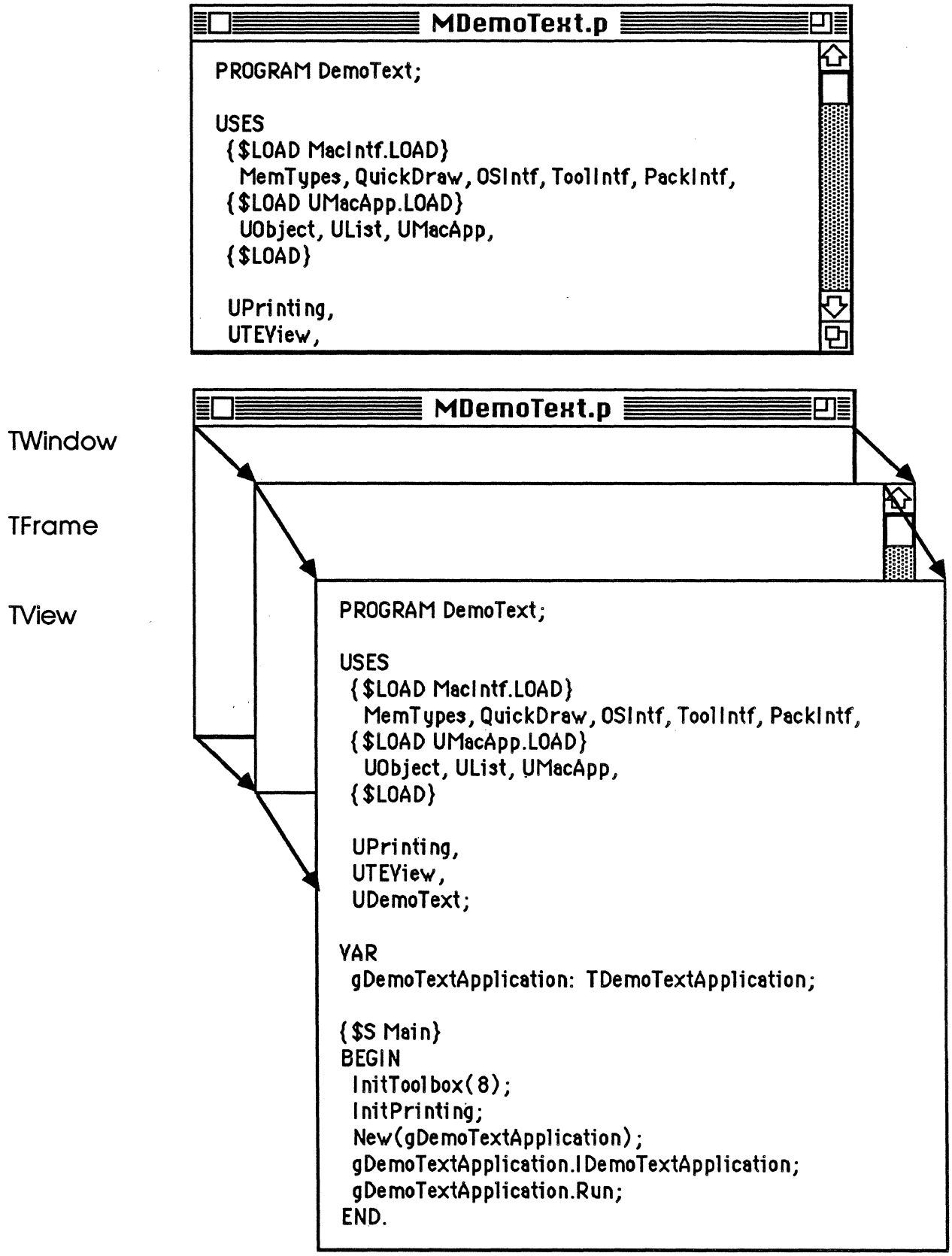

Figure 1: Old MacApp Display Architecture

Figure 1 shows a typical text edit window and the MacApp objects used to build it. Here, the `TWindow` object displays the grow icon (the rest of the window frame is drawn by the Window Manager) and handles mouse clicks outside the window's content region. The `TFrame` object draws the scroll bar, handles mouse clicks in the scroll bar, and focuses the window so that the framed portion of the view is drawn. Focusing consists of setting the port, its origin, and its clipping region. Setting the port origin determines what part of the view is visible in the frame. The frame responds to clicks in the scroll bar by changing the port origin, thereby changing the visible part of the view, and causing the view to be redrawn. The view draws the text, and handles keystrokes and mouse clicks in the text.

As part of MacApp's dialog box support, the display architecture is augmented to incorporate nested views. `TCatView`, a descendant of `TView`, is a type of view that contains a collection of subviews. Thus you can have frames within frames, each of which can contain a single view, which in turn can contain views within views.

This brings us to the shortcomings of the old architecture. First, having two separate nesting models complicates and duplicates much functionality. Neither nesting scheme is totally satisfactory, and it isn't clear to the user which scheme to use. Second, given that each frame has its own set of controls, MacApp has three different ways of layering the contents of a window (frames, views, and controls). Third, it is difficult to deal with Control Manager controls as separate entities (for example, buttons in a dialog box). Fourth, it is cumbersome to use views as the basis for display objects such as icons or form fields, since each must be placed in its own frame or must be used with cat views. Either way adds an unneeded layer of abstraction. Furthermore, `TCatViews` have been regarded as temporary citizens in the MacApp object library, and users are warned not to use them as they may be done away with "any time now."

Another source of trouble with the old MacApp architecture is dialog boxes. They are implemented with a descendant of the `TWindow` class, and with `TCatViews`. In many cases, it is more difficult to deal with MacApp's dialog box support than to implement dialog boxes from scratch.

The MacApp 2.0 Display Architecture

So how does the new architecture differ from the old? Primarily by providing a robust display object class, called `TView`, from which all other display classes descend. The fundamental difference between the new architecture and the old is this: *All display objects descend from the same class.* (All subsequent references to `TView` are to the new `TView`.) The role of the `TView` class is to define a set of properties common to all display objects. These properties include nesting, focusing, drawing, and event handling. MacApp also defines a number of standard views that are subclasses of `TView`. These include `TWindow`, `TControl`, TCtlMgr, `TScrollBar`, `TSScrollBar` and `TScroller`. These classes will be described in more detail later.

Nesting

A view is a rectangular display object of a given size and position with respect to its background view. Any view can serve as the background for any other view. A set of views related in this way forms a tree, in which all but the leaf views serve as backgrounds for other views. This relationship is called the *superview/subview relationship*. Each view can be contained in another view, called its *superview*. Each view can contain any number of views, called *subviews*. A view's subviews form an ordered list, allowing overlapping views to have a front-to-back order.

The benefit of this arrangement is that views can be tiled or layered. Tiling separates logically distinct portions of a window or view into individual sections. Layering organizes groups of display objects in layers from "front" to "back," much like the way the Window Manager layers windows.

Figure 2 at the end of this section shows the same window as Figure 1 did, but this time decomposed into MacApp 2.0 objects. The window consists of four views: the window itself, a standard "scroller" and scroll bar, and a standard text editing view. There are three layers: the bottom layer consists of the window view; the middle layer consists of the scroller and scroll bar; and the top layer consists of the text.

Focusing

Focusing is the act of setting up the graphics environment to draw the visible part of a view. This entails setting the QuickDraw grafport, setting the port's coordinate system, and clipping a view's drawing to the margins of its superview. Each view has a local coordinate system whose top-left corner is coordinate (0, 0).

Drawing

The `TView` class implements a simple algorithm for drawing a collection of views. Beginning with the view representing the window, each view draws itself and then tells its subviews to draw themselves. Each subview in turn draws itself and tells its subviews to draw themselves. This continues until there are no subviews to draw. Effectively, views are drawn from the bottom to the top. Changing the drawing algorithm is a matter of overriding a single `TView` method.

Event Handling

The `TView` class is a subclass of `TEvtHandler`, so it inherits the ability to handle keystrokes and menu commands. In addition, `TView` has the ability to handle mouse clicks. A simple algorithm is used to determine which view receives the mouse click. Basically, mouse clicks are handled from the top layer of views down. This is done by finding the topmost view containing the mouse point, and then working back to the bottom until a view has handled the click.

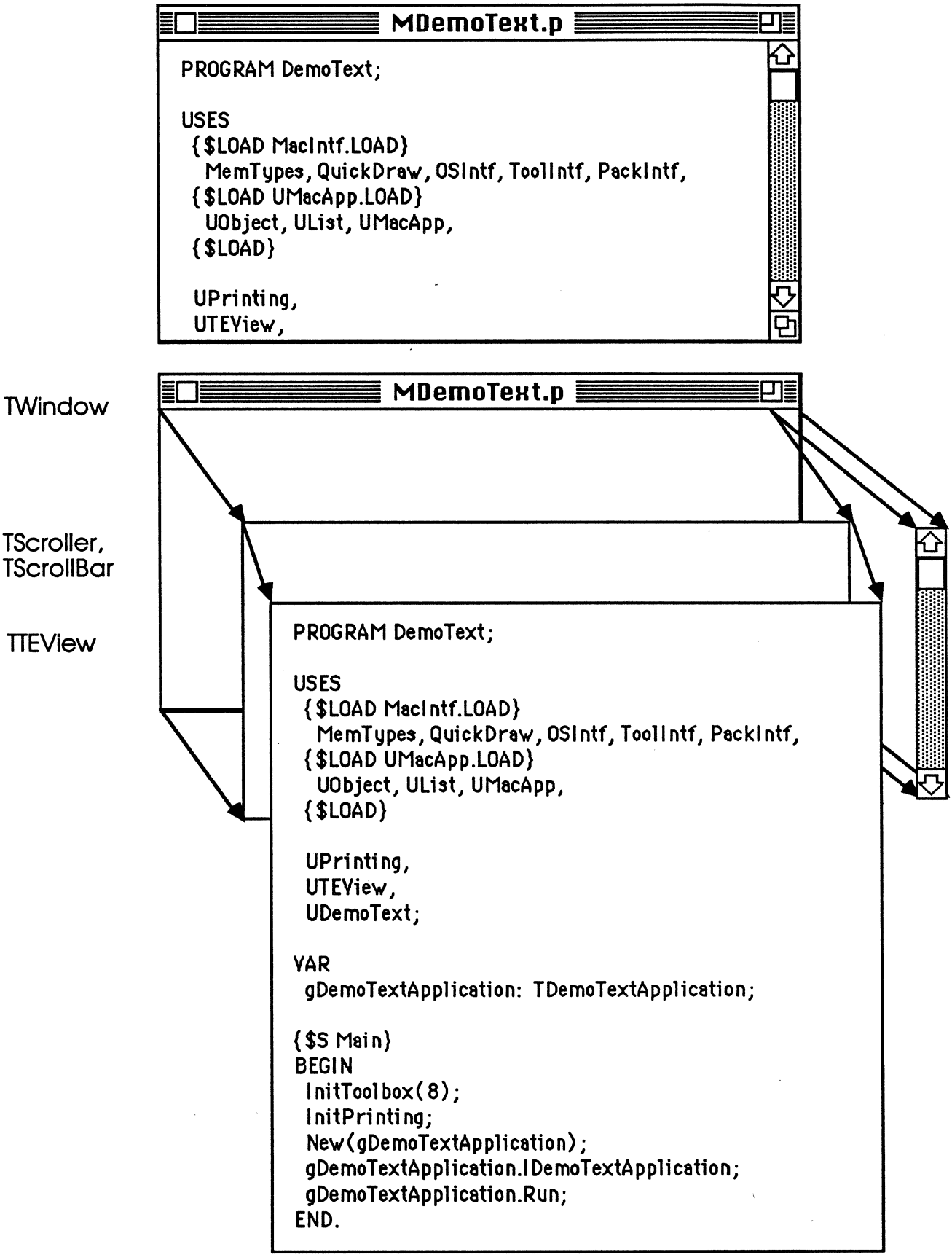

Figure 2: New MacApp Display Architecture

Large Coordinate Systems

One of the limitations of MacApp 1.x is that the maximum size of a view is 30,000 pixels horizontally and vertically, or about 40 8-1/2 by 11 pages. This makes it difficult to use MacApp for applications like word processors and spreadsheets. For example, consider a spreadsheet with 8,000 rows. If each row is 16 pixels high, then its view height is 8,000 * 16, or 128,000 pixels.

To overcome this, we've introduced a large coordinate system where coordinates are represented by 32 bits rather than 16 bits. A large coordinate is defined by the `VCoordinate` type. `VPoint` and `VRect` define large points and rectangles:

```
VCoordinate = LONGINT;

VPoint =    RECORD
               CASE INTEGER OF
                  0: (v, h:  VCoordinate);
                  1: (vh:    ARRAY [VHSelect] OF VCoordinate);
               END;

VRect =     RECORD
               CASE INTEGER OF
                  0: (top, left, bottom, right: VCoordinate);
                  1: (topLeft, botRight:        VPoint);
               END;
```

(V stands for view. We would have used `LPoint`, `LRect`, and so on, had the List Manager not defined `LRect`.)

Implementing a large coordinate system on top of QuickDraw requires mapping view coordinates to QuickDraw coordinates. This is different from scaling, which condenses the entire view into QuickDraw's coordinate system, transforming *many* view coordinates to one QuickDraw coordinate. Mapping as discussed here overlays QuickDraw's coordinates space on some part of the view, transforming *one* view coordinate to one QuickDraw coordinate.

Two possibilities exist for handling coordinate mapping. MacApp can either implicitly map views to QuickDraw coordinates, or the programmer must explicitly map them. The former requires implementing an entire set of drawing and calculation routines for large coordinates, as was done for the Lisa Toolkit. This approach has the advantage of being clean and simple, but it has the disadvantages of slowing all drawing, forcing the developer to adopt a new set of drawing routines, and, of course, forcing us to implement all of those routines.

Operating under the assumption that most views do not require a large coordinate system, we chose the latter approach. Implementing views smaller than QuickDraw's coodinate system imposes little to no burden on the programmer, since coordinate mapping is not necessary. For views larger than QuickDraw's coordinate system, the burden is on the programmer to map between view and QuickDraw coordinates as necessary, using view methods for this purpose.

MacApp provides a unit, UViewCoords, for manipulating `VPoints` and `VRects`. It includes routines to convert between `Points`, `Rects`, `VPoints` and `VRects` and for performing various operations on `VPoints` and `VRects` similar to those found in QuickDraw.

The New Classes

Figure 3 shows a hierarchy of the primary MacApp 2.0 display object classes.

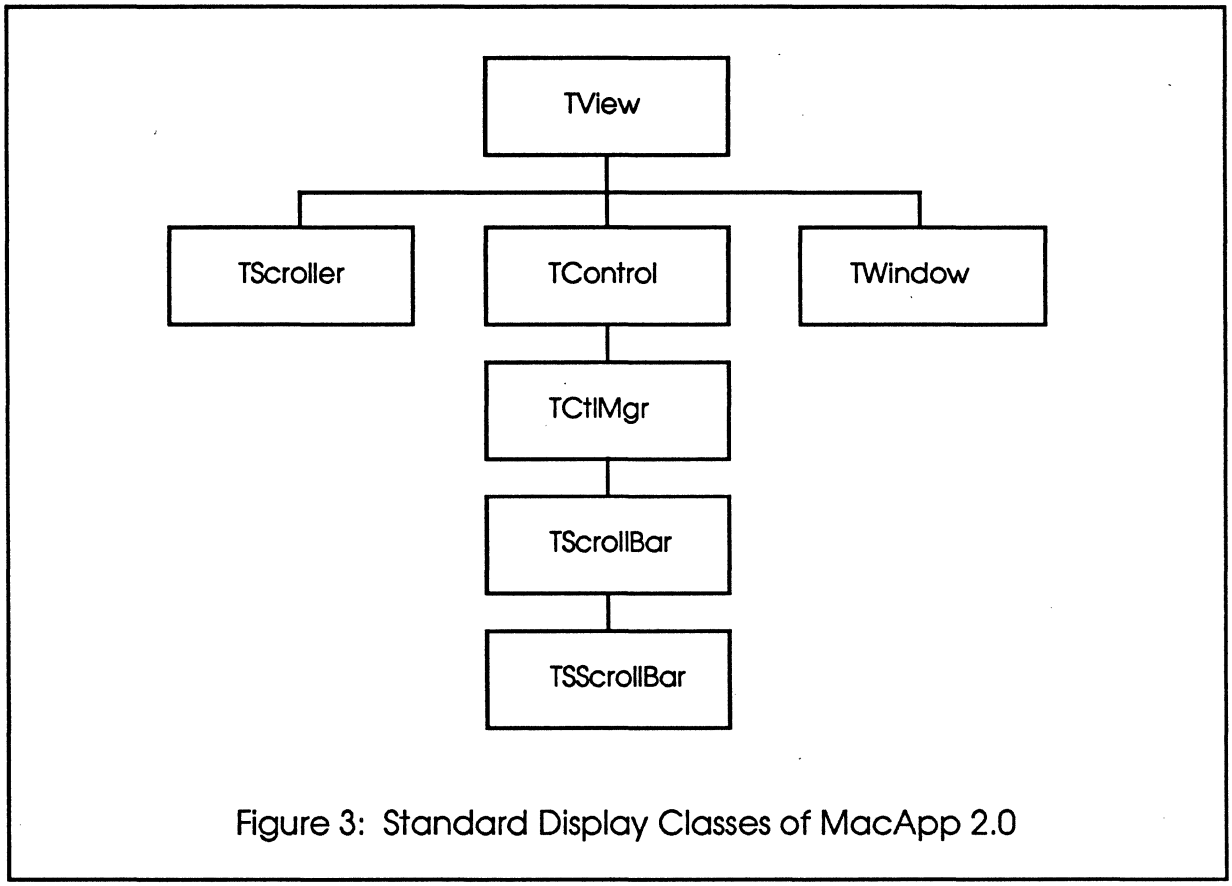

Figure 3: Standard Display Classes of MacApp 2.0

Additional display classes are included in the building blocks UTEView, UDialog and UGridView. A brief description of each class follows:

```
TView = OBJECT(TEvtHandler)
```

As described above, `TView` is the basic display object class of MacApp, from which all other display classes descend. As a descendant of `TEvtHandler`, it inherits the ability to handle keystroke events and menu commands. In addition, it provides these general capabilities: nesting, drawing, focusing, and mouse handling. This class forms the basis for all other MacApp display objects.

```
TScroller = OBJECT(TView)
```

TScroller is a subclass of TView that applies a coordinate translation to its contents. Changing the translation effectively scrolls the scroller's contents. A scroller may refer to a vertical and/or horizontal scroll bar, which the scroller keeps in sync. It is also possible to scroll without scroll bars.

```
TWindow = OBJECT(TView)
```

TWindow is a class that represents a Window Manager window. It responds to mouse clicks outside the window's content region, draws the window's grow box, and overrides other view methods where appropriate. Since TWindow objects represent windows, they never have superviews. They must have at least one subview or nothing will be drawn in the window's content region.

```
TControl = OBJECT(TView)
```

TControl is an abstract class (one that must be subclassed to obtain useful instances) that represents a MacApp control, of which Control Manager controls are a subset. Properties of MacApp controls are as follows: they can be tracked with the mouse without explicitly creating a command object; they have an optional adornment (frame); they are limited in size to QuickDraw's coordinate space; and they can pass messages (known as choices) to their superviews.

```
TCtlMgr = OBJECT(TControl)
```

TCtlMgr is an abstract class that represents Control Manager controls.

```
TScrollBar = OBJECT(TCtlMgr)
```

TScrollBar is a class that represents generic scroll bars. It contains references to one or more scrollers, which it sends messages to when the user clicks in the scroll bar.

```
TSScrollBar = OBJECT(TScrollBar)
```

TSScrollBar is a class that represents scroll bars used in conjunction with TScroller views. It contains references to one or more scrollers, which it sends messages to when the user clicks in the scroll bar.

The TView Class

The TView class is the ancestor of all other display classes. As mentioned above, the TView class is used to define a set properties common to all display objects. These properties include nesting, focusing, drawing and event handling.

Fields

fDocument	The document whose data this view represents, or NIL if the view is not associated with a document. Usually this is NIL if the view does not directly represent a document's data. Thus, views such as scrollers and controls typically have a NIL fDocument.
fSuperView	The view in which this view is contained. The fSuperView field of a window is

NIL.

`fSubViews` — A list of views contained in this view, implemented as a `TList`. It is created on demand and is `NIL` until the first subview is added to the view. Thus the `fSubViews` field of the frontmost views are `NIL`. Note that the list of subviews has an implied ordering which determines which subview is in front of another subview in cases where subviews overlap (that is, occupy the same space in their superview). The last subview in the list is in front of all other subviews while the first subview is behind the other subviews. This has ramifications for cursor and mouse handling in that the subview list is searched from last to first to determine which view contains the mouse.

`fLocation` — The location of this view in its superview's coordinate space.

`fSize` — The horizontal and vertical size of the view.

`fSizeDeterminer` — Indicates the techniques used to compute the view's width and height. (The technique use to compute the width may be different than for the height.) The choices are

`sizeFixed`	The view's width or height is constant.
`sizeVariable`	The view determines its width or height by overriding the `CalcMinSize` method.
`sizePage`	The view's width or height is exactly one page. (The page size is determined by the view's print handler.)
`sizeFillPages`	The view overrides `CalcMinSize` to determine its minimum size, which is then rounded up to fill the last page in the given direction.
`sizeSuperView`	The view's width or height is the same as its superview. When its superview changes size, the view's size changes as well.
`sizeRelSuperView`	The view's width or height changes relative to its superview's size. When the superview's size changes, the view's size changes an equal amount.

`fHLDesired` — The type of highlighting used to draw the view's selection. This field is set to `hlOn` when the view is activated, and to `hlOff` when the view is deactivated.

`fIdentifier` — A four-character identifier of type `IDType (PACKED ARRAY [1..4] OF CHAR)`. Given a view's identifier, you can obtain a reference to the view by using the `TView.FindSubView` method.

`fShown` — True indicates the view is shown, false indicates the view is hidden. This field is not usually accessed directly. Instead use the `Show` and `IsShown` methods.

`fViewEnabled` — True indicates the view responds to mouse clicks, false indicates the view does not respond to mouse clicks. This field is not usually accessed directly. Instead use the `ViewEnable` and `IsViewEnabled` methods.

`fPrintHandler` — A reference to the print handler associated with this view.

Coordinate Conversion Methods

These methods convert between QuickDraw and view coordinates. For views whose width and height is smaller than kMaxCoord (30,000), this amounts to nothing more than converting coordinates between short and long data structures. (Note: It would have been much more natural and convenient for routines of this type to be functions. Unfortunately, the code generated for function calls is less efficient than that generated for procedures with var parameters. Therefore, we felt there the better choice was to use procedures and var parameters.)

```
PROCEDURE TView.QDToViewPt (qdPoint: Point; VAR viewPt: VPoint);
```

Returns in viewPt the view point corresponding to the given QuickDraw point. For views whose height and width is less than kMaxCoord (30, 000), qdPoint and viewPt will have the same value. For views whose height or width is greater than kMaxCoord, qdPoint and viewPt will have different values. QDToViewPt assumes the view is focused.

```
FUNCTION  TView.ViewToQDPt (viewPt: VPoint): Point;
```

Returns the QuickDraw point corresponding to the given view point. For views whose height or width is greater than kMaxCoord, the point returned will have a different value than viewPt. ViewToQDPt assumes the view is focused.

```
PROCEDURE TView.QDToViewRect (qdRect: Rect; VAR viewRect: VRect);
```

Returns in viewRect the view rectangle corresponding to the given QuickDraw rectangle. For views whose height or width is greater than kMaxCoord, qdPoint and viewPt will have different values, and QDToViewRect assumes the view is focused.

```
PROCEDURE TView.ViewToQDRect (viewRect: VRect; VAR qdRect: Rect);
```

Returns in qdRect the QuickDraw rect corresponding to the given view rectangle. For views whose height or width is greater than kMaxCoord, qdPoint and viewPt will have different values, and ViewToQDRect assumes the view is focused.

```
PROCEDURE TView.SuperToLocal (VAR thePoint: VPoint);
```

Analogous to the Toolbox routine GlobalToLocal, SuperToLocal converts the given point from superview coordinates to view coordinates, without changing the focused view.

```
PROCEDURE TView.LocalToSuper (VAR thePoint: VPoint);
```

Analogous to the Toolbox routine LocalToGlobal, LocalToSuper converts the given point from local view coordinates to superview coordinates, without changing the focused view.

```
PROCEDURE TView.WindowToLocal (VAR thePoint: VPoint);
```

WindowToLocal converts the given point from window coordinates to view coordinates, without changing the focused view.

```
PROCEDURE TView.LocalToWindow (VAR thePoint: VPoint);
```

LocalToWindow converts the given point from local view coordinates to window coordinates, without changing the focused view.

Subview Management Methods

Note that the subviews are maintained in an ordered list in which the order determines which subview is in front of another subview when they overlap (that is occupy the same space in their superview). The last subview in the list is in front of all other subviews in the list while the first subview is behind the other subviews. This has ramifications for cursor and mouse handling in that the list is searched from last to first to determine which view contains the mouse.

```
PROCEDURE TView.AddSubView (theSubView: TView);
```

This method adds theSubView to the end of a view's list of subviews and sets theSubView's fSuperView field to its new superview. If the subview list is NIL, then it is created. If the view is in a grafport, then the subview's BeInGrafPort method is called to notify the subview that it too is in a grafport. If theSubView's fNextHandler is NIL then it is set to its new superview.

```
PROCEDURE TView.RemoveSubView (theSubView: TView);
```

This method removes a subview from a view's list of subviews and sets theSubView's fSuperView field to NIL. If theSubView's fNextHandler is the superview then it is set to NIL.

```
PROCEDURE TView.EachSubView (PROCEDURE DoToSubView (aSubView: TView));
```

This method calls the procedure DoToSubView for each subview, in order of their occurrence in fSubViews. (Analogous to TList.Each.)

```
FUNCTION TView.FirstSubViewThat (FUNCTION TestSubView (theSubView: TView):
      BOOLEAN): TView;
```

This method returns the first subview for which TestSubView returns true, or NIL if TestSubView never returns true. The subviews are tested in order of occurance in fSubViews, from first to last. (Analogous to TList.FirstThat.)

```
FUNCTION TView.LastSubViewThat (FUNCTION TestSubView (theSubView: TView):
      BOOLEAN): TView;
```

This method returns the last subview for which TestSubView returns true, or NIL if TestSubView never returns true. The subviews are tested in reverse order of occurrence in fSubViews, from last to first.

```
PROCEDURE TView.MakeFirstSubView (theSubView: TView);
```

This method makes the given subview the first subview in fSubViews, if it isn't already the first. For debugging purposes, a ProgramBreak is generated if the given subview is not in fSubViews. The purpose of this method is to allow ordering of overlapping views.

```
PROCEDURE TView.MakeLastSubView (theSubView: TView);
```

This method makes the given subview the last subview in `fSubViews`, if it isn't already the last. For debugging purposes, a `ProgramBreak` is generated if the given subview is not in `fSubViews`. The purpose of this method is to allow ordering of overlapping views.

```
FUNCTION TView.CountSubViews: INTEGER;
```

This method returns the number of subviews for a view. Using this method avoids depending on the implementation of the subview lists.

```
FUNCTION TView.FindSubView (itsIdentifier: IDType): TView;
```

This method returns the view whose `fIdentifier` field is equal to `itsIdentifier`, and that is directly or indirectly a subview of `SELF`.

Open/Close/Activate Methods

```
PROCEDURE TView.Open;
```

This method is called when the window containing this view is opened. It simply calls the `Open` method for each of its subviews. It is intended to be overridden by views that need special processing when they are opened, in which case the override should call `INHERITED Open`.

```
PROCEDURE TView.Close;
```

This method is called when the window containing this view is closed. It simply calls the `Close` method for each of its subviews. It is intended to be overridden by views that need special processing when they are closed, in which case the override should call `INHERITED Close`.

```
PROCEDURE TView.Activate (entering: BOOLEAN);
```

This method is called when the window containing this view is activated (`entering` is true) or deactivated (`entering` is false). It focuses the view, calls the view's `DoHighlightSelection` method to change the selection from inactive to active or vice versa, and sets `fHLDesired` to `hlOn` when activating or to `hlDim` when deactivating. Finally, the `Activate` method is called for each subview. This method is overridden by views that require special processing when their windows are activated or deactivated, in which case the override normally calls `INHERITED Activate`.

```
PROCEDURE TView.ShowReverted;
```

This method is called when a document containing this view is reverted. It calls the view's `AdjustSize` and `ForceRedraw` methods. It is overridden by views that need to take more action when a document is reverted, in which case the override normally calls `INHERITED ShowReverted`.

```
PROCEDURE TView.BeInPort (itsPort: GrafPtr);
```

This method is called when the view is placed into, or removed from a grafport. `itsPort` is either the grafport the view is being placed in (as a result of adding it as a subview of another view), or `NIL` if the

view is being removed from a grafport. `BeInPort` simply calls `BeInPort` for each of its subviews. This method is intended to be overridden by views that must be notified when they are in a port, in which case the override should call `INHERITED BeInPort`. (The reason a view may require this notification is that it may not be in a port when it is initialized.)

```
PROCEDURE TView.BeInScroller (itsScroller: TScroller);
```

This method is called when the view becomes a subview of a scroller, or when a view is no longer a subview of a scroller (`itsScroller` is `NIL`). If `itsScroller` is not `NIL` then the scroller's `SetScrollLimits` method is called, passing the view's `fSize` as the scroll limits. You may need to override this if you have a scroller containing more than one view and you don't want the scroller's scroll limits to be set by each view.

Size Management Methods

This section describes the methods related to setting or changing a view's size. In many cases the size of a view is dictated by the data it represents. For example, the size of a text-editing view depends on the amount of text it displays. In such a case the size determiners of the view are usually `sizeVariable`, and the `CalcMinSize` method is overridden to compute the view's current size. On the other hand, many views are not dependent upon the data they display, and in those cases overridding `CalcMinSize` is not required. An example is a scroller whose size is usually `sizeFixed` or `sizeRelSuperView`, both of which are handled by MacApp.

For views that vary in size, you must notify the view when its size *may* have changed. This is done by calling the view's `AdjustSize` method. This in turn calls `ComputeSize`, which calls `CalcMinSize` to find out what the view's size should be. If that is different from the view's current size, then `AdjustSize` calls `Resize` to set the new size. Thus, for views whose size varies, the key to using these methods is to override `CalcMinSize` and to call `AdjustSize` whenever some action takes place that may change the view's size. (In the case of a text editing view, `AdjustSize` would have to be called after every keystroke, for example.)

```
PROCEDURE TView.Resize (width, height: VCoordinate; invalidate: BOOLEAN);
```

This method sets the view's size to `width` and `height`, only if the new size is different than the current size. If `invalidate` is true, then the difference between the view's old and new size is invalidated. (This strategy will not work for a view that needs to be entirely redrawn when its size changes. In this case you should override `Resize` to call `ForceRedraw` to invalidate the old size, call `INHERITED Resize` with `invalidate` false, then call `ForceRedraw` to invalidate the new size. Or you can override `Resize` completely.) If the view has a superview, the superview's `SubViewChangeSize` method is called. Similarly, the `SuperViewChangeSize` method is called for each subview. Typically, you don't call `Resize` directly. Instead, you call `AdjustSize`. (If you do call `Resize` directly, be aware that it may change the focused view.)

```
PROCEDURE TView.AdjustSize;
```

You call this method whenever a change is made that may affect the size of the view, typically when the data represented by the view changes. `AdjustSize` calls `ComputeSize` to find out what the view's size should be. If this is different from the view's current size, then `Resize` is called with the `invalidate` parameter set to true; and if the view has a print handler, then the view's `DoPaginiation` method is called. Note that `AdjustSize` may change the focused view.

```
PROCEDURE TView.ComputeSize (VAR newSize: VPoint);
```

This method computes the view's size by calling its `CalcMinSize` method and then applying the view's size determiners and returning that size in `newSize`. The action taken is determined by the view's size determiners:

`sizePage`	The new size is set to the page size.
`sizeFillPages`	The view's "minimum" is computed by calling `CalcMinSize`, and then rounded up to fill the last page.
`sizeVariable`	The new size is the same as that returned by `CalcMinSize`.
`sizeFixed`	The new size is the same as the view's existing size.
`sizeSuperView`	The new size is set to the superview's size.
`sizeRelSuperView`	The new size is the same as that passed to `ComputeSize` in the `newSize` parameter..

`ComputeSize` is called from `AdjustSize` and `SuperViewChangedSize`. It is rarely overridden.

```
PROCEDURE TView.CalcMinSize (VAR minSize: VPoint);
```

This method returns in `minSize` the view's minimum size. This method is frequently overridden to compute the view's size. Unless overridden, the view's current size is returned. `CalcMinSize` is called from `ComputeSize`.

```
PROCEDURE TView.SuperViewChangedSize (delta: VPoint; invalidate: BOOLEAN);
```

This method informs a view that its superview has changed size, where `delta` is the amount of change. If either the horizontal or vertical size determiner of the view is `sizeSuperView` or `sizeRelSuperView`, then the view's size is adjusted accordingly by calling `ComputeSize` and `Resize`. This method may be overridden to enhance or change this behavior. The `invalidate` parameter indicates whether the view should be invalidated (and redrawn) if it needs resizing. Because this method is called whenever the view's superview is resized, you don't ordinarily call it directly.

```
PROCEDURE TView.SubViewChangedSize (theSubView: TView; delta: VPoint);
```

This method informs a view that one of its subviews has changed size, where `delta` is the amount of change. It does nothing, but may be overridden by views that need to take action when a subview changes size. Because this method is called whenever one of the view's subviews is resized, you don't ordinarily call this directly.

```
PROCEDURE TView.GetExtent (VAR itsExtent: VRect);
```

This method returns the view's current extent, which is a rectangle defining the view's coordinate system. The top-left is (0, 0) and the bottom-right is the view's size. This method is rarely overridden.

```
PROCEDURE TView.GetQDExtent (VAR qdExtent: Rect);
```

This method returns the view's extent, converted to QuickDraw coordinates. For views whose extent is larger than `kMaxCoord` (30,000) the rectangle returned will be smaller than the view's actual extent. This method is simply for convenience so that you don't have to call `GetExtent` and `ViewToQDRect` everytime you want to get a view's extent and convert it to QuickDraw coordinates. This method assumes that the view is focused.

Location Management Methods

```
PROCEDURE TView.Locate (h, v: VCoordinate; invalidate: BOOLEAN);
```

This method changes the location of a view within its superview, where `h` and `v` are the view's new horizontal and vertical location expressed in the superview's coordinate system. If `invalidate` is true, then the view is invalidated in both its old and new locations. The superview's `SubViewMoved` method is called. Similarly, `SuperViewMoved` is called for each of the view's subviews. If `invalidate` is true, then the view is invalidated in both its old and new positions.

```
PROCEDURE TView.SuperViewMoved (invalidate: BOOLEAN);
```

This method informs a view that its superview has moved. It does nothing, but it may be overridden for views that need to take action when their superview moves. `invalidate` indicates whether the view should be invalidated if it moves. Because this method is called whenever the view's superview moves, you don't ordinarily call this directly.

```
PROCEDURE TView.SubViewMoved (theSubView: TView);
```

This method informs a view that one of its subviews has moved. It does nothing, but it may be overridden by views that need to take action when a subview changes size. Because this method is called whenever one of the view's subviews move, you don't ordinarily call this directly.

```
PROCEDURE TView.GetFrame (VAR itsFrame: VRect);
```

This method returns the view's frame, which is a rectangle defining the view's coordinate system *in superview coordinates*. The top-left is the view's location and the bottom-right is the view's size added to its location.

Focusing Methods

```
FUNCTION TView.Focus: BOOLEAN;
```

This method establishes the drawing environment for the view. A view must be focused before any graphical operations can be performed it. Because the location of a view is relative to its superview, the method `FocusOnSuperView` is called to establish the superview's drawing environment. Then the current grafport's origin and clipping is set so that drawing can be done in the view's coordinate system. If it is not possible to focus the view (e.g. because its `fShown` flag is false or it isn't visible in its grafport), then `Focus` returns false and the current port's clipping region is set to an empty rectangle. The algorithm for focusing is complicated by the fact that views may have a larger coordinate space than QuickDraw, and this method takes that into account when setting the port's origin. `Focus` is rarely overridden.

```
FUNCTION TView.FocusOnSuperView: BOOLEAN;
```

This method focuses on the view's superview by calling the superview's `Focus` method and returning its result. If there is no superview, this method returns false.

```
PROCEDURE TView.ClipFurtherTo (r: Rect; hDeltaOrg, vDeltaOrg: INTEGER);
```

This is an internal utility method that sets the window clipping region when a view is being focused.

Drawing Methods

```
PROCEDURE TView.DrawContents;
```

This method implements MacApp's drawing algorithm. If the view can be focused, and if any part of the view is visible, then 1) the view's `Draw` method is called; 2) if the application is not printing (`gPrinting` is false) and not writing to the desk scrap (`gDrawingPictScrap` is false) then the view's `DoHighlightSelection` and `DoDrawPrintFeedback` methods are called; and 3) `DrawContents` is called for each subview. Though it is rarely done, this method can be overridden to change the drawing algorithm.

```
PROCEDURE TView.Draw (area: Rect);
```

This method draws the view itself. It is nearly always overridden; otherwise the view is considered transparent. The parameter `area` indicates what part of the view needs to be drawn, in QuickDraw coordinates. If the view is larger than `kMaxCoord` (30,000) pixels then the method `QDToViewRect` can be used to convert `area` into view coordinates.

```
PROCEDURE TView.DoHighlightSelection (fromHL, toHL: HLState);
```

This method is intended to be overridden to highlight the view's selection, if any. It does nothing unless overridden. `fromHL` and `toHL` can be one of `hlOff`, `hlDim`, or `hlOn`. Your `DoHighlightSelection` method should be prepared to handle any plausable combination of these states. The common combinations are:

`fromHL, toHL:`

`hlOff, hlOn:`	The selection is being turned on while the view is active.
`hlOn, hlOff:`	The selection is being turned off while the view is active.
`hlOn, hlDim:`	The view's window has been made inactive so the highlighting should be changed to inactive highlighting. If this view doesn't support dim (inactive) highlighting then it can treate hlDim as hlOff.
`hlDim, hlOn:`	The view's window has been made active so the highlighting should bemade active. If this view doesn't support dim (inactive) highlighting then it can treate hlDim as hlOff.
`hlDim, hlOff:`	The selection is being turned off while the view is inactive.
hlOff, `hlDim:`	The selection is being turned on while the view is inactive.

```
PROCEDURE TView.Update;
```

This method redraws invalidated portions of a view (and its subviews) by calling the `DrawContents` method sandwiched between the Window Manager routines `BeginUpdate` and `EndUpdate`. This method is called for a window when responding to an update event. You can call this method yourself if you are careful: you should be aware that this method clears all pending updates in a window regardless of whether the invalid areas of the window are drawn or not. Therefore, only use this method when you know the portion of the window to be updated is completely contained in the view in which it is being called.

```
PROCEDURE TView.Adorn (area: Rect; itsPenSize: Point;
      itsAdornment: CntlAdornment);
```

This method adorns the view according to the given parameters. Adornment essentially consists of drawing a frame around a rectangle. `area` is the rectangle around which the frame is drawn. `itsPenSize` is the size of the pen used to draw the adornment. `itsAdornment` indicates the type of adornment to be drawn.

Display Validation/Invalidation Methods

```
PROCEDURE TView.InvalidVRect (viewRect: VRect);
```

This method invalidates the given rectangle in view coordinates.

```
PROCEDURE TView.ValidateVRect (viewRect: VRect);
```

This method validates the given rectangle in view coordinates.

```
PROCEDURE TView.InvalidRect (r: Rect);
```

This method invalidates the given rectangle in QuickDraw coordinates.

```
PROCEDURE TView.ValidateRect (r: Rect);
```

This method validates the given rectangle in QuickDraw coordinates.

```
PROCEDURE TView.ForceRedraw;
```

This method invalidates the entire view by calling `InvalidVRect`(*the view's extent*).

```
PROCEDURE TView.RevealRect (rectToReveal: VRect; minToSee: Point; isVisible:
      BOOLEAN);
```

This method is intended to ensure that a specific part of the view's extent is visible. (It may be invisible because the view is clipped to its superview's margins.) `rectToReveal` is the rectangle to be made visible, in view coordinates. `minToSee` is the minimum amount of the rectangle to be made visible. `TView.RevealRect` calls its superview's `RevealRect`. As you will see later, if the superview is a scroller, then it scrolls the given rectangle into view. If `isVisible` is false, then the scroll is not immediately made visible. You should set `isVisible` to false only in situations in which you know the view will be redrawn and you want to avoid extraneous drawing.

```
PROCEDURE TView.RevealTop (isVisible: BOOLEAN);
```

This method attempts to reveal the top-left corner of the view's extent by calling `RevealRect` with a rectangle of (0, 0)/(0, 0). If `isVisible` is false, then the scroll is not immediately made visible. You should set `isVisible` to false only in situations in which you know the view will be redrawn and you want to avoid extraneous drawing.

```
PROCEDURE TView.RevealBottom (isVisible: BOOLEAN);
```

This method attempts to reveal the bottom-right corner of the view's extent. If `isVisible` is false, then

the scroll is not immediately made visible. You should set `isVisible` to false only in situations in which you know the view will be redrawn and you want to avoid extraneous drawing.

Mouse-Handling Methods

```
FUNCTION TView.ContainsMouse (theMouse: VPoint): BOOLEAN;
```

This method returns true if the view is shown and the given point in view coordinates intersects the view's extent. However, this method can be overridden for views that respond to mouse clicks only in *part* of the extent, rather than *anywhere* in the extent. For example, if a view is circular, then it could override `ContainsMouse` to return true when the point lies within the circle, rather than within its rectangular extent.

```
FUNCTION TView.HandleMouseDown ( theMouse: VPoint; VAR info: PEventInfo;
    VAR hysteresis: Point; VAR theCommand: TCommand): BOOLEAN;
```

This method implements MacApp's mouse-handling logic. It returns true if this view or one of its subviews handles the mouse click. The logic is such that each of the subviews is given the opportunity to handle the click first. If no subview handles the click, then the view itself may handle it and return true. The following pseudo-code shows how this works:

```
FUNCTION TestMouse (theSubView: TNewView): BOOLEAN;
BEGIN
  {Convert theMouse to view's QuickDraw coords};
  TestMouse := theSubView.HandleMouseDown( theMouse, info,
                                           hysteresis, theCommand);
END;

BEGIN
  HandleMouseDown := FALSE;
  theCommand := gNoChanges;
  IF ContainsMouse(theMouse) THEN
       BEGIN
       viewThatHandledMouse := LastSubViewThat(TestMouse);
       IF viewThatHandledMouse = NIL THEN
            theCommand := DoMouseCommand(theMouse);
       HandleMouseDown := TRUE;
       END;
END;
```

```
FUNCTION TView.DoMouseCommand (VAR theMouse: Point; VAR info: EventInfo;
      VAR hysteresis: Point): TCommand;
```

Once HandleMouseDown has determined which view should handle the mouse click, it calls the view's DoMouseCommand method. DoMouseCommand simply returns gNoChanges, so it must be overridden by views that respond to mouse clicks. Note that theMouse is in QuickDraw coordinates. info is the EventInfo record for the mouse-down event.

Mouse clicks are handled in one of two ways. For clicks that do not require tracking and are not undoable, DoMouseCommand takes the appropriate action and returns gNoChanges. For clicks that require tracking or *are* undoable, DoMouseCommand should create a command object and return the command as its result. In this case, hysteresis can be set to the number of pixels the mouse must move before tracking begins.

Cursor Handling Methods

```
FUNCTION TView.HandleCursor (theMouse: VPoint; cursorRgn: RgnHandle): TView;
```

This method implements the algorithm used to determine which view sets the cursor and the cursor region. It simply finds the most deeply nested, enabled subview that contains the mouse and calls its DoSetCursor method. If DoSetCursor returns false, then the DoSetCursor method for the view's superview is called, and so on, until DoSetCursor returns true. You can override HandleCursor for a view that needs to force the cursor to a given shape regardless of its subviews.

```
FUNCTION TView.DoSetCursor (localPoint: Point; cursorRgn: RgnHandle): BOOLEAN;
```

This method is used to set the cursor shape. localPoint is the cursor's current location in QuickDraw coordinates. To set the cursor, call the Toolbox routine SetCursor (or SetCCursor for a color cursor), return true, and optionally set cursorRgn to a region in which the cursor shape will remain the same. If this view does not set the cursor, then DoSetCursor should return false, in which case the view's superview will be given a chance to set the cursor. Upon entry, cursorRgn is set to an empty region.

Clipboard-Handling Methods

```
FUNCTION TView.ContainsClipType (aType: ResType): BOOLEAN;
```

This method is used to determine whether a view installed in the Clipboard represents a particular kind of data. It simply returns true if the desk scrap contains data of the given type by calling the Scrap Manager routine GetScrap. This method is intended to be overridden by views that can be installed in the Clipboard and may maintain a local version of the Clipboard data rather than installing it in the desk scrap.

```
FUNCTION TView.GivePasteData (aDataHandle: Handle; dataType: ResType):
      LONGINT;
```

This method returns a handle to data that is represented by a view installed in the Clipboard. dataType indicates the type of data to be returned, and aDataHandle is a handle in which to store the data. Note that aDataHandle is either NIL, in which case GivePasteData simply returns the size of the data, or is already allocated. GivePasteData returns the length of the data in the handle, or a negative value indicating an error. TView.GivePasteData returns the data by calling the Scrap Manager routine

GetScrap, but it may be overridden by views that can be installed in the Clipboard and may maintain a local version of the Clipboard data rather than installing it in the desk scrap.

```
PROCEDURE TView.WriteToDeskScrap;
```

This method is called on a view installed in the Clipboard, when the application is losing control to a desk accessory or another application. Its purpose is to write the contents of the Clipboard to the desk scrap for Clipboard views that maintain their data locally. TView.WriteToDeskScrap simply creates a QuickDraw picture of the clipboard view and writes the picture to the desk scrap. This method is intended to be overridden by views that can be installed in the Clipboard and that write data other than pictures to the desk scrap.

Printing Methods

The view printing methods are described in the *MacApp 2.0b5 Printing Release Notes*.

Miscellaneous Methods

```
FUNCTION TView.GetGrafPort: GrafPtr;
```

This method returns a pointer to the grafport in which the view is contained, or NIL if the view is not in a grafport.

```
FUNCTION TView.GetWindow: TWindow;
```

This method returns a reference to the window object in which this view is contained, or NIL if the view is not in a window.

```
FUNCTION TView.GetScroller (immediateSuperView: BOOLEAN): TScroller;
```

This method returns a reference to the scroller object in which the view is contained, or NIL if the view is not in a scroller. If immediateSuperView is true and the scroller is not the view's immediate superview then GetScroller returns NIL.

```
FUNCTION TView.GetDialogView: TView;
```

This method returns a reference to the dialog view object in which the view is contained, or NIL if the view is not in a dialog view. Note that the function result is of type TView, so you must cast the result in order to use it as a dialog view (for example, aDialogView := TDialogView(GetDialogView)). The reason for this is that the dialog views are not defined in the main MacApp unit.

```
PROCEDURE TView.GetVisibleRect (VAR visQDRect: Rect);
```

This method returns the part of the view's extent that is visible within its superview. This may not be the entire extent since views are clipped to their superview's margins. The rectangle returned is in QuickDraw coordinates. It can be converted to view coordinates with the `QDToViewRect` method.

```
PROCEDURE TView.DoChoice (origView: TView; itsChoice: INTEGER);
```

The purpose of calling `DoChoice` is to propogate information up the view hierarchy. Consider the case of clicking on a radio button. The radio button itself may take some action, such as toggling its value. But the radio button may be contained in a view that is responsible for ensuring that other related radio buttons are turned off. If the radio button calls `DoChoice`, its superview can take additional action above and beyond that taken by the radio button itself.

```
PROCEDURE TView.ViewEnable (state: BOOLEAN; redraw: BOOLEAN);
```

This method enables (`state` is true) or disables (`state` is false) the view's ability to handle mouse clicks by setting its `fViewEnable` flag to the value of `state`. If `redraw` is true, then the view is redrawn by calling its `ForceRedraw` method.

```
FUNCTION TView.IsViewEnabled: BOOLEAN;
```

This method returns true if the view is able to handle mouse clicks, or false if the view is unable to handle mouse clicks.

```
PROCEDURE TView.Show (state, redraw: BOOLEAN);
```

This method shows the view (`state` is true) or hides the view (`state` is false) and sets the view's `fShown` flag to `state`. It is analagous to the Window Manager routine `ShowHide`. If the `redraw` parameter is true then the view is redrawn to reflect its new state.

```
FUNCTION TView.IsShown: BOOLEAN;
```

This method returns true if the view is shown, and false if it is hidden. Note that the issue of whether a view is hidden or shown is different from whether the view is invisible or visible. The view may be shown but still not be visible, because it is clipped from view by its superview.

Resource Writing Methods

```
PROCEDURE TView.WRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
```

This method writes the `TView` portion of the view's resource template. It is the inverse of `TView.IRes` and is only used by programs that write `'view'` resources.

```
PROCEDURE TView.WriteRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
```

This method serves as a wrapper for `WRes`. If you want to write a view object as part of a `'view'` resource, then you would call this method, which calls the appropriate `WRes` methods.

Creation/Destruction Methods

```
PROCEDURE TView.IView ( itsDocument: TDocument;
                        itsSuperView: TView;
                        itsLocation, itsSize: VPoint;
                        itsHSizeDet, itsVSizeDet: SizeDeterminer);
```

This method initializes a view by calling `IEvtHandler(itsSuperView)`, setting its document, superview, location, size, and size determiners, setting `fHLDesired` to `hlDim`, and adding the view to its superview by calling its superview's `AddSubView` method.

```
PROCEDURE TView.IRes (itsDocument: TDocument; itsSuperView: TView;
      VAR itsParams: Ptr);
```

This method initializes a view from a resource. `itsDocument` is the view's document, `itsSuperView` is the view's superview, `itsParams` is a pointer to the `TView` section of a view's resource. `IRes` initilizes the view and offsets `itsParams` by the length of the `TView` parameters.

```
PROCEDURE TView.Free; OVERRIDE;
```

This method frees its subviews, removes itself from its superview (by calling `fSuperView.RemoveSubView`), and calls `INHERITED Free` to free itself. It also frees the view's print handler if it has one.

```
PROCEDURE TView.FreeFromClipboard;
```

Called when a view installed in the Clipboard is freed. If the view's document is not NIL then the document's `FreeFromClipboard` method is called (which will free the view). Otherwise the view's `Free` method is called.

The TScroller Class

The `TScroller` class implements a transparent view that performs coordinate translation. By changing the translation values, the subviews of a scroller are "scrolled" through the scroller. For example, given a scroller whose size is 100 pixels in both directions, and a subview located at (0, 0), whose size is 1000 pixels, then a translation value of (300, 500) will display the part of the subview defined by the rectangle (300, 500)/(400, 600). A scroller may refer to a vertical and horizontal scroll bar, whose values and are kept in sync with the scroller's translation values. Furthermore, it is possible to scroll without scroll bars by calling the appropriate scroller methods to change the translation value.

Fields

`fTranslation`	The number of pixels by which all coordinates in the scroller are translated, in both the horizontal and vertical direction.
`fScrollLimit`	The maximum point that can be displayed in the scroller.
`fMaxTranslation`	The maximum values that `fTranslation` can take. This is computed by

subtracting the scroller's size from `fScrollLimit`.

`fScrollBars` — A reference to the horizontal and vertical scroll bars, or `NIL` if they don't exist.

`fScrollUnit` — The amount of change to be made in the translation values when the scroller receives a "scroll-by-arrow" message, for both the horizontal and vertical directions.

`fConstrain` — Indicates whether the translation values should be constrained to even multiples of `fScrollUnits` in either the horizontal and vertical direction.

`fSBarOffsets` — When a scroller is resized, this field is used to determine how to resize the associated scroll bars. `fSBarOffsets.top` and `bottom` are offsets indicating where the top and bottom of the vertical scroll bar is in relation to the top and bottom of the scroller. `fSBarOffsets.left` and `right` are offsets indicating where the left and right of the horizontal scroll bar is in relation to the left and right of the scroller.

Coordinate Conversion Methods

```
PROCEDURE TScroller.LocalToSuper (VAR thePoint: VPoint); OVERRIDE;
```

This method converts the given point from local view coordinates to superview coordinates. It is overridden to take into account the scroller's coordinate translation.

```
PROCEDURE TScroller.SuperToLocal (VAR thePoint: VPoint); OVERRIDE;
```

This method converts the given point from superview coordinates to local view coordinates. It is overridden to take into account the scroller's coordinate translation.

Subview Management Methods

```
PROCEDURE TScroller.AddSubView (theSubView: TView); OVERRIDE;
```

This method calls `INHERITED AddSubView`, and then notifies the subview that it's in a scroller by calling `theSubView.BeInScroller(SELF)`.

```
PROCEDURE TScroller.RemoveSubView (theSubView: TView); OVERRIDE;
```

This method notifies the subview that it is no longer in a scroller by calling `theSubView.BeInScroller(NIL)`, and then calls `INHERITED RemoveSubView`.

Size/Location Management Methods

```
PROCEDURE TScroller.Resize (width, height: VCoordinate; invaldate: BOOLEAN);
      OVERRIDE;
```

This method calls `INHERITED Resize`, and then calls `AdjustScrollBars` to resize and move the scroller's scroll bars. It can be overridden if you don't want automatic adjustment of the scroll bars when a scroller is resized.

```
PROCEDURE TScroller.Locate (h, v: VCoordinate; invalidate: BOOLEAN); OVERRIDE;
```

This method calls `INHERITED Locate`, and then calls `AdjustScrollBars` to move the scroller's scroll bars. It can be overridden if you don't want automatic adjustment of the scroll bars when a scroller is moved.

```
PROCEDURE TScroller.GetExtent (VAR itsExtent: VRect); OVERRIDE;
```

This method calls `INHERITED GetExtent`, and then offsets `itsExtent` by `fTranslation`. Thus, `itsExtent` is the scroller's extent translated by the translation values.

```
PROCEDURE TScroller.AdjustScrollBars;
```

This method is called when a scroller moves or changes size. It erases the scroll bars, and then moves and resizes them appropriately. This method can be overridden for scrollers whose scroll bars don't depend on this scroller's location or size.

```
PROCEDURE TScroller.SetScrollLimits (scrollLimit: VPoint;
      drawScrollBars: BOOLEAN);
```

This method is called to change the scroller's maximum translation values. This entails setting the `fScrollLimit` field to `scrollLimit`, recomputing the `fMaxTranslation` field, and setting the scroll bar maximums by calling each scroll bar's `SetMaximum` method. The `drawScrollBars` parameter is passed to `SetMaximum` to indicate whether the control should be immediately redrawn. Ordinarily this is true. Furthermore, if the new maximum translation is less than the current translation, then the scroller's `ScrollTo` method is called to scroll to the maximum translation.

Focusing Methods

```
FUNCTION TScroller.Focus: BOOLEAN;
```

This method calls `INHERITED Focus` and puts into effect the coordinate translation of `fTranslation`. Thus, the scroller's coordinates and those of its subviews are translated by this amount.

Low-Level Scrolling Methods

These methods are rarely overridden, and even more rarely called from application code.

```
PROCEDURE TScroller.ScrollDraw (delta: VPoint);
```

This method carries out the graphic activity associated with a scroll, where `delta` is the number of pixels to scroll. The scroll is done by using QuickDraw's `ScrollRect` to scroll the contents of the scroller, and then simulating an update event to draw the vacated area. This is done by calling the Window Manager routine `BeginUpdate`, erasing the vacated area, calling the scroller's `DrawContents` method, and then calling the Window Manager's `EndUpdate`.

```
PROCEDURE TScroller.DoScroll (delta: VPoint; isVisible: BOOLEAN);
```

This method changes the scroller's translation by the `delta` amount. If `isVisible` is true, then `ScrollDraw` is called to carry out the scroll graphically. Furthermore, when `isVisible` is true `DoScroll` checks to see if there is a pending update event on the scroller's window. If so, the window is updated before changing the scroller's translation values.

Intermediate-Level Scrolling Routines

These methods are usually called from scroll bar objects, though they may also be called from application code to simulate the behavior of scroll bars. These methods are occasionally overridden as described below.

```
FUNCTION TScroller.ScrollStep (vhs: VHSelect; partCode: INTEGER): VCoordinate;
```

This method is called by one of the scroller's scroll bars, when the user clicks in an arrow or page area of the scroll bar. It may be called in other contexts to effect the same action as clicking a scroll bar arrow or page area. `vhs` indicates the scrolling direction (horizontal or vertical) and `partCode` indicates the part of the scroll bar that was clicked (`inUpButton`, `inDownButton`, `inPageUp`, or `inPageDown`).

This method determines how many pixels to scroll the view, calls `DoScroll` to carry out the scroll, and returns the change in the scroll bar's value. For `inUpButton` and `inDownButton`, the amount of scrolling is the same as the scroller's `fScrollUnit[vhs]` value. For `inPageUp` and `inPageDown`, the amount of scrolling is equal to the scroller's size in the `vhs` direction, constrained to a multiple of `fScrollUnit` if `fConstrain[vhs]` is true. This method may be overridden to change the scrolling behavior, or if the scroll bar units are not the same as the scroller's translation units.

```
FUNCTION TScroller.ScrollRelative (vhs: VHSelect; sBarValue: VCoordinate):
    VCoordinate;
```

This method is called by a scroll bar object when its thumb is released. (It may also be called from other parts of the application in order to simulate thumb scrolling.) `vhs` indicates the scrolling direction and `sBarValue` indicates the scroll bar's value when the thumb was released. This method determines the scroller's new translation values, calls `DoScroll` to carry out the scroll, and may return a nonzero value which is added to the scroll bar's value as described below.

The new translation value is set to `sBarValue`, unless `fConstrain[vhs]` is true, in which case the translation value is rounded to an even multiple of `fScrollUnit[vhs]`, and the difference between this value and `sBarValue` is returned as `ScrollRelative`'s result. This method may be overridden to change the scrolling behavior, or if the scroll bar units are not the same as the scroller's translation units.

High Level Scrolling Methods

These methods are frequently called by application code. They are rarely overridden.

```
PROCEDURE TScroller.ScrollBy (deltaH, deltaV: VCoordinate; isVisble: BOOLEAN);
```

This method changes the scroller's translation and scroll bar values by `deltaH` and `deltaV` pixels. This is done by calling each scroll bar's `DeltaValue` method and then calling the scroller's `DoScroll` method. If `isVisible` is true then the contents of the scroller are redrawn to reflect the new translation. Passing false for `isVisible` is useful in situations where you wish to avoid screen flashing (that is, drawing the same view twice). This method must be overridden if the scroll bar units are not the same as the scroller's translation units.

```
PROCEDURE TScroller.ScrollTo (h, v: VCoordinate; isVisible: BOOLEAN);
```

This method sets the scroller's translation and scroll bar values to the horizontal and vertical values in `h` and `v`. This is done by calling each scroll bar's `SetValue` method and then calling the scroller's `DoScroll` method. If `isVisible` is true then the contents of the scroller are redrawn to reflect the new translation. Passing false for `isVisible` is useful in situations where you wish to avoid screen flashing (that is, drawing the same view twice).

```
PROCEDURE TScroller.RevealRect (rectToReveal: VRect; minToSee: Point;
      isVisible: BOOLEAN); OVERRIDE;
```

This method ensures at least `minToSee` pixels of the given rectangle are visible in the scroller. This is done by calling `ScrollBy` if necessary to change the translation values such that the given rectangle is made visible. This method is typically used to ensure that a selection is visible (for example, making the caret visible in a text edit view when the user starts typing). If `isVisible` is false, then the scroll is not immediately made visible. You should set `isVisible` to false only in situations in which you know the view will be redrawn and you want to avoid extraneous drawing.

```
PROCEDURE TScroller.AutoScroll (viewPt: VPoint; VAR delta: VPoint);
```

This method is used to implement automatic scrolling when the mouse is being tracked and has strayed outside a scroller. `viewPt` is the location of the mouse in view coordinates. `delta` returns the number of pixels to scroll in the horziontal and vertical directions. It is computed by determining where `viewPt` is in relation to the scroller. If `viewPt` is "above" the scroller (that is, `viewPt.v < extent.top`), then `delta.v` is set such that the contents of the scroller are scrolled down. If `viewPt` is "below" the scroller, then `delta.v` is set such that the contents of the scroller are scrolled up. `delta.h` is set similarly. The number of pixels to scroll is determined by the scroller's `fScrollUnit` values. `AutoScroll` can be overridden to use a different method to determine the value of `delta`.

Miscellaneous Methods

```
PROCEDURE TScroller.SubViewChangedSize (theSubView: TView; delta: VPoint);
      OVERRIDE;
```

This method calls `SetScrollLimits` to set the scroller's `fMaxTranslation` and `fScrollLimit` to reflect the size of `theSubView`. This method should be overridden in cases where a scroller has subviews of varying size.

```
PROCEDURE TScroller.HaveScrollBar (theScrollBar: TSScrollBar; direction:
      VHSelect);
```

This method attaches a scroll bar to a scroller. It is called from the scroll bar's initialization method. You can also use it to detach a scroll bar by passing `NIL` in `theScrollBar`.

```
PROCEDURE TScroller.SetScrollParameters (horzUnits, vertUnits: VCoordinate;
      horzConstraint, vertConstraint: BOOLEAN);
```

This method sets the scroller's `fScrollUnits` and `fConstrain` fields to the given parameters.

```
FUNCTION TScroller.GetScroller (immediateSuperView: BOOLEAN): TScroller;
      OVERRIDE;
```

This method returns `SELF`.

```
PROCEDURE TScroller.CreateScrollBars (wantHorzSBar, wantVertSBar: BOOLEAN);
```

This method is an internal utility that creates the scroll bars as indicated by the parameters. It is called from `IScrollBar` and `IRes`.

Resource-Writing Methods

```
PROCEDURE TScroller.WRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method writes the `TScroller` portion of the view's resource template. It is the inverse of `TScroller.IRes` and is only used by programs that write `'view'` resources.

```
PROCEDURE TScroller.WriteRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method serves as a wrapper for `WRes`. If you want to write a scroller object as part of a `'view'` resource you would call this method, which calls the appropriate `WRes` methods.

Creation/Destruction Methods

```
PROCEDURE TScroller.IScroller ( itsSuperView: TView;
                                itsLocation, itsSize: VPoint;
                                itsHSizeDet, itsVSizeDet: SizeDeterminer;
                                itsHorzMax, itsVertMax: VCoordinate;
                                wantHorzSBar, wantVertSBar: BOOLEAN);
```

This method initializes a scroller object. it calls `IView` to set the scroller's superview, location, size, and size determiners. (The scroller's document is set to NIL.) It sets `fTranslation` and `fMaxTranslation` to (0, 0), creates the scroll bars if necessary, sets `fSBarOffsets` to (0, 0, 0, 0), sets the scroll units to `kStdScrollUnit` (16), sets `fConstrain` to false, and calls `SetScrollLimits` with `itsHorzMax` and `itsVertMax`.

```
PROCEDURE TScroller.IRes (itsDocument: TDocument; itsSuperView: TView;
      VAR itsParams: Ptr);
```

This method initializes a scroller from a resource. `itsDocument` is ignored: the scroller's document is set to `NIL`. `itsSuperView` is the view's superview, while `itsParams` is a pointer to the `TView` section of a view's resource. `IRes` calls `INHERITED IRes` to initialize the inherited data (that of `TView`), initilizes the `TScroller` data, and offsets `itsParams` by the length of the `TScroller` parameters.

```
PROCEDURE TScroller.Free; OVERRIDE;
```

If the scroller has scroll bars, then the scroll bars are freed. Then `INHERITED Free` is called.

The TWindow Class

The `TWindow` class implements windows. This section describes those fields and methods that are in addition to or that override `TView`.

Fields

`fWMgrWindow`	The Window Manager window pointer.
`fIsActive`	Indicates whether this window is active. This may differ from testing if `fWMgrWindow = FrontWindow` because of a pending activate event that hasn't been processed yet.
`fIsResizable`	True if the window has a grow box.
`fIsClosable`	If true, the Close menu item is enabled when this window is active. If the window has a close box then this field is initially set to true.
`fFreeOnClosing`	If true, then this object is freed when the window is closed. The default setting is false.
`fDisposeOnFree`	If true, then `DisposeWindow(fWMgrWindow)` is called when this window is freed.
`fClosesDocument`	If true, then when this window is closed, it also calls `fDocument.Close`. The default setting is true.

`fOpenInitially`	If true, then when the window's document is opened, this window is opened as well. The default setting is true.
`fMoveBounds`	A rectangle defining this window's move bounds. It is passed to the Window Manager routine `DragWindow` and is initially set to `gStdWMoveBounds`.
`fResizeLimits`	A rectangle defining the minimum and maximum size of the window. It is passed to the Window Manager routine `GrowWindow` and is initially set to `gStdWSizeRect`.
`fTarget`	The event handler, usually a view, that becomes the application's target (`gTarget`) when this window is active.
`fPreDocname`	Used to parse the window title into a constant part and a part replaced by the window's document name.
`fConstTitle`	Used to parse the window title into a constant part and a part replaced by the window's document name.
`fIsModal`	True indicates that the window is modal (that is, no other window can be activated while this one is active, although the menu bar is available). False indicates that the window is modeless.
`fDoFirstClick`	True indicates that the window responds to mouse clicks even when it isn't the front window. The default value is false.
`fTargetId`	The identifier of the view that will be the window's initial target when it is created from a resource template.
`fAdapted`	Set to true if the window is "adapted to the screen" by calling `TWindow.AdaptToScreen`.
`fHorzCentered`	Set to true if the window is horizontally centered by its `Center` method.
`fVertCentered`	Set to true if the window is vertically centered by its `Center` method.
`fStaggered`	Set to true if the window is staggered by its `Stagger` method.
`fForcedOnScreen`	Set to true if the window is "forced onto the screen" by its `ForceOnScreen` method.
`fProcId`	The proc ID of the Window Manager window that represents this window.

Open/Close/Activation Methods

```
PROCEDURE TWindow.Activate (entering: BOOLEAN); OVERRIDE;
```

This method is called to activate (`entering` is true) or deactivate (`entering` is false) a window. First, if entering is different from the window's `fIsActive` field, then the window's `DrawContents (TRUE)` method is called. Then `INHERITED Activate` is called to activate or deactivate the window's subviews. What happens next depends on the value of `entering`.

If `entering` is true, then `gFrontWindow` is set to `SELF`, the application's target is set to the window's `fTarget`, and `gDocument` is set to the window's `fDocument`.

If `entering` is false, then the window's `fTarget` is set to the application's current target, the

application's target is set to gApplication, gFrontWindow and gDocument are set to NIL, and the cursor is set to the arrow.

Finally, the window's grow box is redrawn and fIsActive is set to the value of entering. This method can be overridden for windows that require additional activation or deactivation logic, in which case you usually call INHERITED Activate.

```
PROCEDURE TWindow.Open; OVERRIDE;
```

This method calls the window's Resize method, resizing the window to its portRect, calls its Show(TRUE, kRedraw) method to show the window, and calls INHERITED Open to inform all of its subviews that they are now in an open window. This method can be overridden for windows that need extra processing when they're opened, in which case INHERITED Open is usually called.

```
PROCEDURE TWindow.Close; OVERRIDE;
```

This method calls its Show(FALSE, kRedraw) method to hide the window and then deactivates the window by calling its Activate(FALSE) method. Next it calls INHERITED Close to inform its subviews that their window is being closed. And finally it calls its Free method if fFreeOnClosing is true. This method can be overridden for windows that need extra processing when they're opened, in which case INHERITED Close is usually called.

```
PROCEDURE TWindow.CloseByUser;
```

This method is called when the user clicks the window's go-away box or chooses the Close menu command while this window is active. Before calling Close, CloseByUser determines whether it should also close the window's document. It closes the document if the window's fClosesDocument field is true, or if this is the document's only window.

Size Management Methods

```
PROCEDURE TWindow.Resize (width, height: VCoordinate; invalidate: BOOLEAN);
      OVERRIDE;
```

This method calls the Toolbox routine SizeWindow to change the window size, invalidates the old and new positions of the window's grow box, and then calls INHERITED Resize.

```
PROCEDURE TWindow.ResizeByUser (globalMouse: Point);
```

This method calls the Window Manager routine GrowWindow and, if the result is non-zero calls Resize to resize the window.

```
PROCEDURE TWindow.Zoom (partCode: INTEGER);
```

This method zooms a window from it existing size to fill the screen, or from screen size back to its previous size. partCode is either inZoomIn or inZoomOut, indicating which way to zoom. The window is zoomed by focusing itself and erasing its portRect, calling the Window Manager routine ZoomWindow, and then calling its Resize method to set the window's size.

Location Management Methods

```
PROCEDURE TWindow.MoveByUser (globalMouse: Point);
```

This method calls the Window Manager routine `DragWindow`. MacApp calls this method when the user clicks in the window's title bar.

```
PROCEDURE TWindow.Locate (h, v: VCoordinate; invalidate: BOOLEAN); OVERRIDE;
```

This method moves the window to the horizontal and vertical global coordinates specified by `h` and `v`, by calling the Window Manager routine `MoveWindow`.

Focusing Methods

```
FUNCTION TWindow.Focus: BOOLEAN; OVERRIDE;
```

If the window's `fWMgrWindow` is not `NIL`, this method focuses the window by setting the port to the window's port, setting the port's origin to (0, 0), setting `gLongOffset` to (0, 0), and setting the port's clipping region to the entire `portRect`. It returns true if the window could be focused and the window is visible. Returns false if the window's `fWMgrWindow` is `NIL` or the window is invisible.

```
FUNCTION TWindow.FocusOnSuperView: BOOLEAN; OVERRIDE;
```

This method simply calls `Focus`.

Drawing Methods

```
PROCEDURE TWindow.DrawContents; OVERRIDE;
```

This method calls `INHERITED DrawContents` and then draws the grow box (if the window has one) by calling `DrawResizeIcon`.

```
PROCEDURE TWindow.Draw (area: Rect); OVERRIDE;
```

This method calls the QuickDraw routine `EraseRect` on the window's `portRect`.

```
PROCEDURE TWindow.DrawResizeIcon;
```

This method draws the window's grow box.

Mouse-Handling Methods

```
FUNCTION TWindow.HandleMouseDown (theMouse: VPoint; VAR info: EventInfo;
      VAR hysteresis: Point; VAR theCommand: TCommand): BOOLEAN; OVERRIDE;
```

If the window is currently the front window, then `INHERITED HandleMouseDown` is called. Otherwise the window is brought to the front by calling its `Select` method and if `fDoFirstClick` is true, the window is updated and `INHERITED HandleMouseDown` is called.

Menu Handling Methods

```
FUNCTION TWindow.AllowsMenuAcces: BOOLEAN;
```

Indicates whether the menus are accessable when this window is the front window. The standard behavior is to always return true, regardless of whether the window is modal or modeless.

```
PROCEDURE TWindow.DoSetupMenus; OVERRIDE;
```

Calls `INHERITED DoSetupMenus` only if the window is not modal (i.e. `fIsModal` is false). For modal windows this prevents enabling menus whose scope is beyond that of the window itself.

Miscellaneous Methods

```
FUNCTION TWindow.GetGrafPort: GrafPtr; OVERRIDE;
```

This method returns the window's `fWMgrWindow` field.

```
FUNCTION TWindow.GetWindow: TWindow; OVERRIDE;
```

This method returns a reference to itself.

```
FUNCTION TWindow.IsShown: BOOLEAN; OVERRIDE;
```

This method returns true if the window is currently shown on the screen. This is determined by checking the `fWMgrWindow^.visible` flag.

```
PROCEDURE TWindow.Show (state, redraw: BOOLEAN); OVERRIDE;
```

This method shows (or hides if `state` is false) the window on the screen by calling the Window Manager routine `ShowWindow` (or `HideWindow`) and sets the window's `fShown` flag to `state`. The redraw parameter is ignored.

```
PROCEDURE TWindow.GetGlobalBounds (VAR globalBounds: Rect);
```

This method returns the window's bounding rectangle in global coordinates.

```
PROCEDURE TWindow.SetTarget (newTarget: TEvtHandler);
```

This method sets the event handler object that is to be the target when this window is activated by setting the window's `fTarget` field to `newTarget`. Furthermore, if the window is already the front window, then the application's `SetTarget` method is called to make `newTarget` the application's target.

```
PROCEDURE TWindow.SetTitleForDoc (newDocTitle: Str255);
```

This method sets the window's title to reflect the name of its document.

```
PROCEDURE TWindow.SimpleStagger (dh, dv: INTEGER; VAR counter: INTEGER);
```

This method is used to produce a "stagger effect" when creating windows. It basically offsets the location of the window by `dh` and `dv` pixels multiplied by counter.

```
PROCEDURE TWindow.Center (horizontally, vertically: BOOLEAN);
```

This method horizontally and/or vertically centers the window on the main screen.

```
PROCEDURE TWindow.AdaptToScreen;
```

This method adjusts the size of the window to the primary screen of the computer on which the application is being run. This is done by adding to the window size the difference between the primary screen and the standard screen size (512*x*342 pixels). This method is intended to be called before the window is opened, so it does not change the window's `fSize` field. Rather, it calls the Toolbox routine `SizeWindow` and adjusts `fSize` during `Open`.

```
PROCEDURE TWindow.ForceOnScreen;
```

This method ensures that at least some part of the window is visible on the primary screen. This is useful for applications that save window locations and attempt to reopen windows in the last saved location. Problems can occur when the application is run on a system with a large screen, and then run on a system with a small screen. In such cases, it is possible for a window to be visible on the large screen, but not visible on the small screen. `ForceOnScreen` prevents this problem. This method is intended to be called before the window is opened, so it does not change the window's `fSize` field. Rather, it calls the Toolbox routine `SizeWindow` and adjusts `fSize` during `Open`.

```
PROCEDURE TWindow.SetResizeLimits (minSize, maxSize: Point);
```

This methods sets the minimum and maximum sizes for the window so that the user cannot grow the window larger than `maxSize` nor shrink it smaller than `minSize`. `maxSize` also determines what the window's size will be if it is zoomed.

```
PROCEDURE TWindow.Select;
```

This method selects this window (brings it to the front) by calling the application's `SelectWMgrWindow` method.

```
PROCEDURE TWindow.InstallDocument (itsDocument: TDocument);
```

Associates the given document with the window. This is called from `IWindow` and `IRes`. If `itsDocument` is not nil then the window is added to the document's window list. Otherwise the window is added to the application's free window list.

Resource-Writing Methods

```
PROCEDURE TWindow.WRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method writes the `TWindow` portion of the view's resource template. It is the inverse of `TWindow.IRes` and is only used by programs that write `'view'` resources.

```
PROCEDURE TWindow.WriteRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method serves as a wrapper for `WRes`. If you want to write a window object as part of a `'view'` resource you would call this method, which calls the appropriate `WRes` methods.

Creation/Destruction

```
PROCEDURE TWindow.IWindow (itsDocument: TDocument; itsWMgrWindow: WindowPtr;
      canResize, canClose, disposeOnFree: BOOLEAN);
```

This method calls `IView`, sets its fields to their default values, and either adds the window to its document's window list, or adds the window to the application's free window list if `itsDocument` is `NIL`. Note that the window's `fSize` is set to (0, 0): `TWindow.Open` resizes the window to the size of its `fWMgrWindow^.portrect`.

```
PROCEDURE TWindow.IRes (itsDocument: TDocument; itsSuperView: TView;
      VAR itsParams: Ptr);
```

This method initializes a window from a resource. `itsDocument` is the window's document. `itsSuperView` should be `NIL`, and `itsParams` is a pointer to the `TView` section of a view's resource. `INHERITED IRes` is called to initialize the inherited data (in this case the `TView` data), then the `TWindow` data is initialized, and finally `itsParams` is offset by the length of the `TWindow` data.

```
PROCEDURE TWindow.Free; OVERRIDE;
```

This method removes the window from either its document's window list, or the application's free window list (if `fDocument` is `NIL`), then calls `INHERITED Free` and calls the global routine `FreeWMgrWindow`.

The TControl Class

The `TControl` class represents the MacApp notion of a control. MacApp controls include Control Manager controls (i.e. radio buttons, push buttons, check boxes, and scroll bars) as well as non-Control Manager controls (either of your own creation or as defined in the UDialog unit.).

Controls have the following properties:

- They are limited to QuickDraw's coordinate space.
- The can be tracked with the mouse without you having to create a command object. `TControl` methods can be overridden to implement behavior while tracking the mouse rather than implementing the behavior in a separate command object.
- They can be adorned with one of a set of standard adornments, such as a rectangular frame.
- They have a text style used to draw the control's label if it has one.
- They have insets or margins in which mouse clicks are ignored.

This section describes those fields and methods that are in addition to or that override `TView`.

Fields

`fDefChoice`	The code that is passed to `DoChoice`. The default value is the constant `mOKHit`.
`fInset`	Used to define which part of the view is considered to be the active control area. The active control area is determined by insetting the view's extent by `fInset`. Generally, controls respond to mouse clicks only in the active area, and adornments are drawn in the inactive area. The default inset is (0, 0, 0, 0).
`fHilite`	Whether the control is highlighted or not. The default setting is false.
`fDimmed`	Whether the control is dimmed or not. The default setting is false.
`fSizeable`	False indicates that the active control area is of a fixed size, regardless of whether the view's size changes. (In this case, changing the view size simply changes the bottom and right insets from the view's size to the control's size.) True indicates that the active control area changes in proportion to the view's size. The default setting is true.
`fDismissesDialog`	Whether clicking on this control dismisses a dialog. The default setting is false.
`fAdornment`	The type of adornment for this control. The default setting is no adornment.
`fPenSize`	The pen size used to draw the control's adornment. The default setting is (1, 1).
`fTextStyle`	The font style used to draw the control's label, if any. The default setting is the system style.

Open/Close/Activate Methods

```
PROCEDURE TControl.Activate (entering: BOOLEAN); OVERRIDE;
```

This method is overridden to do nothing since controls have no subviews.

Size Management Methods

```
PROCEDURE TControl.ComputeSize (VAR newSize: VPoint); OVERRIDE;
```

This method is overridden to adjust the control insets appropriately.

```
PROCEDURE TControl.Resize (width, height: VCoordinate; invalidate: BOOLEAN);
      OVERRIDE;
```

Overridden to force the entire view to be redrawn if `invalidate` is true.

Drawing Methods

```
PROCEDURE TControl.Draw (area: Rect); OVERRIDE;
```

This method draws the control's adornment, calls its `Dim` method the control if `fDimmed` is true, and calls its `Hilite` method if `fHilite` is true.

Focusing Methods

```
FUNCTION TControl.Focus: BOOLEAN; OVERRIDE;
```

This method is overridden to set the current port's text style when the control is focused.

Mouse-Handling Methods

```
FUNCTION TControl.ContainsMouse (theMouse: VPoint): BOOLEAN; OVERRIDE;
```

This method is overridden to return true only if `theMouse` is in the control's active area (that is, the view's extent is inset by `fInsets`).

```
FUNCTION TControl.DoMouseCommand (VAR theMouse: Point; VAR info: EventInfo;
      VAR hysteresis: Point): TCommand); OVERRIDE;
```

This method tracks the control using a `TControlTracker` object. It in turn calls the `TControl` methods `TrackConstrain`, `TrackFeedback`, and `TrackMouse`. Returns the tracker object.

The following methods are used to implement custom-control mouse tracking.

```
PROCEDURE TControl.TrackConstrain (anchorPoint, previousPoint: VPoint;
      VAR nextPoint: VPoint);
```

As with `TCommand.TrackConstrain`, overriding this method allows a control to constrain the mouse while it is being tracked. `anchorPoint` is the location of the mouse when it was clicked in the control. `previousPoint` is the location of the mouse when it was last tracked. `nextPoint` is the current location of the mouse. Constraint of the mouse is accomplished by changing `nextPoint` appropriately. The standard behavior of `TrackConstrain` is to not constrain the mouse.

```
PROCEDURE TControl.TrackFeedback (anchorPoint, nextPoint: VPoint;
      turnItOn, mouseDidMove: BOOLEAN);
```

This method may be overridden to provide mouse-tracking feedback. This method is called twice for every call to `TrackConstrain` and `TrackMouse`. The first time it is called with `turnItOn` false, the second time with `turnItOn` true. This technique makes it easy to implement feedback such as a drawing the outline of a box from the anchor point to the current mouse location. (Note that not all feedback is done this way. For example, tracking a push button calls for highlighting the button when the mouse enters the button, and unhighlighting the button when the mouse leaves it. Sometimes it is more convenient to implement feedback in the `TrackMouse` method.) The standard behavior of this method is to provide no feedback.

```
PROCEDURE TControl.TrackMouse (aTrackPhase: TrackPhase; VAR anchorPoint,
      previousPoint, nextPoint: VPoint; mouseDidMove: BOOLEAN);
```

This method is intended to be overridden to carry out any mouse-tracking activity required by the control, other than that implemented in `TrackConstrain` and `TrackFeedback`. `aTrackPhase` has one of three values: `TrackPress`, which indicates that mouse tracking has just begun; `TrackMove`, which indiates that mouse tracking is in progress; and `TrackRelease`, which indicates that the mouse button has been released. `anchorPoint` is the location of the mouse when it was clicked. `previousPoint` is the location of the mouse the last time `TrackMouse` was called. `nextPoint` is the current location of the mouse. `mouseDidMove` is true if the mouse has moved since the last time `TrackMouse` was called. The standard behavior of `TrackMouse` is to do nothing. Therefore you must override this method to implement mouse tracking behavior specific to this control.

Dimming Methods

```
PROCEDURE TControl.DimState (state, redraw: BOOLEAN);
```

This method dims (`state` is true) or undims (`state` is false) the control. It sets `fDimmed` to state and, if `redraw` is true the control is redrawn.

```
PROCEDURE TControl.Dim;
```

This method draws the control's dim effect by painting a gray pattern over the control. It is called only when the control is dimmed. This method can be overridden to change the way the dim effect is drawn for a control.

```
FUNCTION TControl.IsDimmed: BOOLEAN;
```

This method returns true if the control is currently dimmed; otherwise it returns false.

Highlighting Methods

```
PROCEDURE TControl.HiliteState (state, redraw: BOOLEAN);
```

This method highlights (`state` is true) or unhighlights (`state` is false) the control. It sets `fHilite` to state and, if `redraw` is true the control is focused and calls its `Hilite` method is called.

```
PROCEDURE TControl.Hilite;
```

This method draws the control's highlighting effect. It is called from `HiliteState` to highlight or unhighlight the control, and from `Draw` when the control's `fHilite` is true. Since this method is responsible for both highlighting and unhighlighting it can examine the `fHilite` field to see whether it should highlight or unhighlight. The standard behavior is to simply invert the control, but you can override this method to change the way highlighting is drawn.

Miscellaneous Methods

```
PROCEDURE TControl.ControlArea (VAR theArea: Rect);
```

This method returns the active control area of the view as a QuickDraw rectangle. This is computed by taking the view's extent and insetting it by `fInset`.

```
PROCEDURE TControl.InstallColor (theColor: RGBColor; redraw: BOOLEAN);
```

This method sets the color of the control's text to `theColor`. If `redraw` if true, then the control will be redrawn to reflect its new color. If `redraw` is false, then the control will not be redrawn even though the new color affects its appearance. Set `redraw` to false only in situations in which you know the control will eventually be redrawn and you want to avoid flickering or flashing.

```
PROCEDURE TControl.InstallTextStyle (theTextStyle: TextStyle; redraw:
      BOOLEAN);
```

This method sets the text style of the control to `theTextStyle`. If `redraw` if true, then the control will be redrawn to reflect its new text style. If `redraw` is false, then the control will not be redrawn even though the new style affects its appearance. Set `redraw` to false only in situations in which you know the control will eventually be redrawn and you want to avoid flickering or flashing.

```
PROCEDURE TControl.Inset (dh, dv: INTEGER; redraw: BOOLEAN);
```

This method insets the active control area of the control by `dh` and `dv` pixels. If `redraw` if true, then the control will be redrawn to reflect its new text style. If `redraw` is false, then the control will not be redrawn even though the new insets may affect its appearance. Set `redraw` to false only in situations in which you know the control will eventually be redrawn and you want to avoid flickering or flashing.

```
PROCEDURE TControl.SetInset (newInset: Rect; redraw: BOOLEAN);
```

This method sets the control's insets to those defiined in `newInset`. If `redraw` if true, then the control will be redrawn to reflect its new text style. If `redraw` is false, then the control will not be redrawn even though the new insets may affect its appearance. Set `redraw` to false only in situations in which you know the control will eventually be redrawn and you want to avoid flickering or flashing.

```
FUNCTION TControl.Validate: BOOLEAN;
```

This method returns true if the control's contents are valid. Validation generally applies to controls that accept keyboard entry, but it is not necessarily restricted as such. The default behavior is to return true.

Resource-Writing Methods

```
PROCEDURE TControl.WRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method writes the `TControl` portion of the view's resource template. It is the inverse of `TControl.IRes` and is only used by programs that write `'view'` resources.

```
PROCEDURE TControl.WriteRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method serves as a wrapper for `WRes`. If you want to write a window object as part of a `'view'` resource you would call this method, which calls the appropriate `WRes` methods.

Creation/Destruction Methods

```
PROCEDURE TControl.IControl (itsSuperView: TView; itsLocation,
      itsSize: VPoint; itsHSizeDet, itsVSizeDet: SizeDeterminer);
```

This method initializes a `TControl` object. It calls `IView` to initialize the fields inherited from `TView`, and then sets the control fields to their default values as described in the section on `TControl` fields above.

```
PROCEDURE TControl.IRes (itsDocument: TDocument; itsSuperView: TView;
      VAR itsParams: Ptr); OVERRIDE;
```

This method initializes a control from a `'view'` resource. `itsDocument` is the document for which this control is being created. `itsSuperView` is the control's superview, and `itsParams` is a pointer to the `TView` section of the control's resource. `INHERITED IRes` is called to initialize the inherited data (in this case the `TView` data), then the `TControl` data is initialized, and finally `itsParams` is offset by the length of the `TControl` data.

The TCtlMgr Class

This class is an abstract class used to represent Control Manager controls (push buttons, radio buttons, check boxes, and scroll bars). It is a subclass of `TControl`.

Fields

`fCMgrControl` The Control Manager control represented by this view.

Drawing Methods

```
PROCEDURE TCtlMgr.Draw (area: Rect); OVERRIDE;
```

If the control is visible, this method draws the control by setting the control's `contrlVis` to 0, and calling the Control Manager routine `ShowControl` to draw it. (It would be easier to use `DrawControl` but it isn't supported on 64K ROMs.) Furthermore, the `fCMgrControl`'s `contrlOwner` is temporarily set to the current port in case the control is being printed.

Mouse Handling Methods

```
FUNCTION TCtlMgr.DoMouseCommand (VAR theMouse: Point; VAR info: EventInfo;
    VAR hysteresis: Point): TCommand; OVERRIDE;
```

This method calls the Control Manager routine `TrackControl` to track the control. If `TrackControl` returns a nonzero result, then `DoChoice(SELF, fDefChoice)` is called so that any of the control's superviews can respond to the choice code. You may also override this method to implement specific behavior for specific Control Manager controls.

Size Management Methods

```
PROCEDURE TCtlMgr.Resize (width, height: VCoordinate; invalidate: BOOLEAN);
    OVERRIDE;
```

If `fSizeable` is true then `fCMgrControl` is resized by calling the Control Manager routine `SizeControl`. This is done after setting the clipping rectangle to (0, 0, 0, 0) to prevent the Control Manager from drawing when `SizeControl` is called. If `invalidate` is true, then the `ForceRedraw` method is called to ensure that the entire control is redraw. Then `INHERITED Resize` is called.

Dimming Methods

```
PROCEDURE TCtlMgr.DimState (state, redraw: BOOLEAN); OVERRIDE;
```

This method is overridden to dim the control using the Control Manager routine `HiliteControl`.

Highlighting Methods

```
PROCEDURE TCtlMgr.HiliteState (state, redraw: BOOLEAN); OVERRIDE;
```

This method hilites the control by calling the Control Manager routine HiliteControl and passing either 0 if `state` if false or 10 if `state` is true. This has the effect of hiliting the control as though the mouse is clicked in the control.

Current, Minimum, and Maximum Value Methods

```
FUNCTION TCtlMgr.GetVal: INTEGER;
```

This method returns the control's current value.

```
PROCEDURE TCtlMgr.SetVal (newVal: INTEGER; redraw: BOOLEAN);
```

This method sets the control's current value. If `redraw` is true, then the control will be redrawn to reflect its new value. If `redraw` is false, then the control will not be redrawn even though the new value may affect its appearance. Set `redraw` to false only in situations where you know the control will eventually be redrawn and you want to avoid flickering or flashing.

```
FUNCTION TCtlMgr.GetMin: INTEGER;
```

This method returns the control's minimum value.

```
PROCEDURE TCtlMgr.SetMin (itsMin: INTEGER; redraw: BOOLEAN);
```

This method sets the control's minimum value. If `redraw` is true, then the control will be redrawn to reflect its new minimum. If `redraw` is false, then the control will not be redrawn even though the new minimum may affect its appearance. Set `redraw` to false only in situations where you know the control will eventually be redrawn and you want to avoid flickering or flashing.

```
FUNCTION TCtlMgr.GetMax: INTEGER;
```

This method returns the control's maximum value.

```
PROCEDURE TCtlMgr.SetMax (itsMax: INTEGER; redraw: BOOLEAN);
```

This method sets the control's maximum value. If `redraw` is true, then the control may be redrawn to reflect its new maximum. If `redraw` is false, then the control will not be redrawn even though the new maximum may affect its appearance. Set `redraw` to false only in situations where you know the control will eventually be redrawn and you want to avoid flickering or flashing.

```
PROCEDURE TCtlMgr.SetValues (itsVal, itsMin, itsMax: INTEGER; redraw:
      BOOLEAN);
```

This method sets the control's current, minimum, and maximum values. If `redraw` if true then the control will be redrawn to reflect its new values. If `redraw` is false then the control will not be redrawn even though the new values may affect its appearance. Set `redraw` to false only in situations where you know the control will eventually be redrawn and you want to avoid flickering or flashing.

Miscellaneous Methods

```
PROCEDURE TCtlMgr.SetText (itsText: Str255; redraw: BOOLEAN);
```

This method sets the control's title. If `redraw` if true, then the control will be redrawn to reflect its new title. If `redraw` is false, then the control will not be redrawn even though the new title will affect its appearance. Set `redraw` to false only in situations where you know the control will eventually be redrawn and you want to avoid flickering or flashing.

```
PROCEDURE TCtlMgr.GetText (VAR theText: Str255);
```

This method returns the control's title in `theText`.

```
PROCEDURE TCtlMgr.BeInPort (itsPort: GrafPtr); OVERRIDE;
```

This method keeps the Control Manager control in sync with the port in which the control is being placed. It sets `fCMgrControl^^.contrlOwner` to `itsPort`, or to `gWorkPort` if `itsPort` is `NIL`. `contrlOwner` is the Control Manager field that indicates in which port the control is to be drawn.

```
PROCEDURE TCtlMgr.WhileFocused (PROCEDURE DoToControl; redraw: BOOLEAN);
```

IF `beVisible` is true, then `WhileFocused` attempts to focus the control, and then calls `DoToControl`. If `redraw` is false, then `DoToControl` is called with clipping temporarily set to an empty rectangle so that any drawing done by `DoToControl` will not be visible.

```
PROCEDURE TCtlMgr.CreateCMgrControl (itsBounds: Rect; itsTitle: Str255;
      itsValue, itsMin, itsMax, itsProcID: INTEGER);
```

This is an internal method used to create the Control Manager control associated with this view. The given parameters are passed to the Control Manager routine `NewControl`, the result of which is assigned to `fCMgrControl`.

Resource-Writing Methods

```
PROCEDURE TCtlMgr.WRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method writes the `TCtlMgr` portion of the view's resource template. It is the inverse of `TCtlMgr.IRes` and is only used by programs that write `'view'` resources.

```
PROCEDURE TCtlMgr.WriteRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method serves as a wrapper for `WRes`. If you want to write a window object as part of a `'view'` resource you would call this method, which calls the appropriate WRes methods.

Creation/Destruction Methods

```
PROCEDURE TCtlMgr.ICtlMgr (itsSuperView: TView; itsLocation, itsSize: VPoint;
      itsHSizeDet, itsVSizeDet: SizeDeterminer; itsTitle: Str255;
      itsVal, itsMin, itsMax: INTEGER; itsProcId: INTEGER);
```

This method initializes a `TCtlMgr` object. It calls `IControl` to initialize its inherited data, and then calls `CreateCMgrControl` to create the Control Manager control.

```
PROCEDURE TCtlMgr.IRes (itsDocument: TDocument; itsSuperView: TView;
      VAR itsParams: Ptr); OVERRIDE;
```

This method initializes a `TCtlMgr` object from a `'view'` resource. `itsDocument` is the document for which the control is being created. `itsSuperView` is the control's superview, and `itsParams` is a pointer to the `TView` section of the object's resource. `INHERITED IRes` is called to initialize the inherited data (in this case the `TView` and `TControl` data), then the `TCtlMgr` data is initialized, and finally `itsParams` is offset by the length of the `TCtlMgr` data.

```
PROCEDURE TCtlMgr.Free; OVERRIDE;
```

This method frees `fCMgrControl`, and then calls `INHERITED Free`.

The TScrollBar Class

The `TScrollBar` class represents standard Macintosh scroll bars, but whose values are expressed as `VCoordinates` rather than integers.

Fields

`fDirection`	The scroll bar's direction (horizontal or vertical).
`fLongVal`	The scroll bar's current value in `VCoordinate` units.
`fLongMin`	The scroll bar's minimum value in `VCoordinate` units. The default value is zero.
`fLongMax`	The scroll bar's maximum value in `VCoordinate` units.
`fBitsToShift`	The number of bits to shift to convert the scroll bar's long values into Control Manager control values.

Mouse-Handling Methods

```
PROCEDURE TScrollBar.DoMouseCommand (VAR theMouse: Point; VAR info: EventInfo;
      VAR hysteresis: Point): TCommand; OVERRIDE;
```

This method differentiates between tracking the thumb and tracking any other part of the scroll bar. For the thumb, the Control Manager routine `TrackControl` is called to track the mouse, and the scroll bar's new value is set to reflect the final position of the thumb. For the arrows and page areas, `TrackControl` is called with an action procedure that calls the scroll bar's `TrackScrollBar` method.

```
PROCEDURE TScrollBar.TrackScrollBar (partCode: INTEGER);
```

This method is intended to be overridden so that action can be taken while tracking the scroll bar's arrows or page areas. Typically the action taken is to change the scroll bar's value appropriately. The standard behavior is to do nothing.

Current/Minimum/Maximum Methods

```
PROCEDURE TScrollBar.SetLongVal (itsVal: VCoordinate; redraw: BOOLEAN);
```

This method sets the scroll bar's value to `newValue`. It sets `fLongVal` to `itsVal` and calls `SetValue` with `itsVal` scaled to an integer. If `redraw` if true, then the scroll bar will be redrawn to reflect its new value. If `redraw` is false, then the scroll bar will not be redrawn even though the new value may affect its appearance. Set `redraw` to false only in situations in which you know the scroll bar will eventually be redrawn and you want to avoid flickering or flashing.

```
FUNCTION TScrollBar.GetLongVal: VCoordinate;
```

This method returns the scroll bar's long value.

```
PROCEDURE TScrollBar.SetLongMin (itsMin: VCoordinate; redraw: BOOLEAN);
```

This method sets the scroll bar's long minimum. It sets `fLongMin` to `itsMin` and calls `SetMin` with `itsMin` scaled to an integer. If `redraw` if true, then the scroll bar will be redrawn to reflect its new minimum. If `redraw` is false, then the scroll bar will not be redrawn even though the new minimum may affect its appearance. Set `redraw` to false only in situations in which you know the scroll bar will eventually be redrawn and you want to avoid flickering or flashing.

```
FUNCTION TScrollBar.GetLongMin: VCoordinate;
```

This method returns the scroll bar's long minimum.

```
PROCEDURE TScrollBar.SetLongMax (itsMax: VCoordinate; redraw: BOOLEAN);
```

This method sets the scroll bar's long maximum. It sets `fLongMax` to `itsMax` and calls `SetMax` with `itsMax` scaled to an integer. If `redraw` if true, then the scroll bar will be redrawn to reflect its new maximum. If `redraw` is false, then the scroll bar will not be redrawn even though the new maximum may affect its appearance. Set `redraw` to false only in situations in which you know the scroll bar will

eventually be redrawn and you want to avoid flickering or flashing.

```
FUNCTION TScrollBar.GetLongMax: VCoordinate;
```

This method returns the scroll bar's long maximum.

```
PROCEDURE TScrollBar.SetLongValues (itsVal, itsMin, itsMax: VCoordinate;
      redraw: BOOLEAN);
```

This method sets the scroll bar's current, minimum, and maximum long values. If `redraw` if true, then the scroll bar will be redrawn to reflect its new values. If `redraw` is false, then the scroll bar will not be redrawn even though the new values may affect its appearance. Set `redraw` to false only in situations in which you know the scroll bar will eventually be redrawn and you want to avoid flickering or flashing.

```
PROCEDURE TScrollBar.DeltaValue (delta: VCoordinate);
```

This method offsets the current value of the scroll bar by `delta`, by ensuring that `delta` is within the scroll bar's minimum and maximum and calling `SetVal`.

Resource-Writing Methods

```
PROCEDURE TScrollBar.WRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method writes the `TScrollBar` portion of the view's resource template. It is the inverse of `TScrollBar.IRes` and is only used by programs that write `'view'` resources.

```
PROCEDURE TScrollBar.WriteRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method serves as a wrapper for `WRes`. If you want to write a window object as part of a `'view'` resource you would call this method, which calls the appropriate `WRes` methods.

Creation/Destruction Methods

```
PROCEDURE TScrollBar.IScrollBar (itsSuperView: TView; itsLocation, itsSize:
      VPoint; itsHSizeDet, itsVSizeDet: SizeDeterminer; itsDirection:
      VHSelect; itsVal, itsMin, itsMax: VCoordinate);
```

This method initializes a scroll bar by calling `ICtlMgr` to initialize the inherited data, and then sets its fields to the given parameters.

```
PROCEDURE TScrollBar.IRes (itsDocument: TDocument; itsSuperView: TView;
      VAR itsParams: Ptr); OVERRIDE;
```

This method initializes a `TScrollBar` object from a `'view'` resource. `itsDocument` is the document for which the scroll bar is being created. `itsSuperView` is the scroll bar's superview, and `itsParams` is a pointer to the `TView` section of the object's resource. `INHERITED IRes` is called to initialize the

inherited data (in this case the TView, TControl, and TCtlMgr data), then the TScrollBar data is initialized, and finally itsParams is offset by the length of the TScrollBar data.

The TSScrollBar Class

The TSScrollBar class represents scroll bars used in conjunction with TScroller objects. A single scroll bar can be associated with any number of scrollers. Each scroller is responsible for scrolling itself based upon messages sent to it from its scroll bars.

Fields

fScrollers	A list of the scroller views associated with this scroll bar.

Activation Methods

```
PROCEDURE TSScrollBar.Activate (entering: BOOLEAN); OVERRIDE;
```

After focusing the scroll bar, this method either draws the scroll bar (entering is true) by calling the Control Manager routine ShowControl (fCMgrControl) or hides the scroll bar (entering is false) by calling the Control Manager routine HideControl and the Draw method to frame its rectangle. In both cases, the scroll bar's rectangle is validated to avoid unnecessary window updates.

Drawing Methods

```
PROCEDURE TSScrollBar.Draw (area: Rect); OVERRIDE;
```

This method is overridden so that if the scroll bar is invisible (and assumed to be inactive), its rectangle is framed. Otherwise, INHERITED Draw is called.

Mouse-Handling Methods

```
FUNCTION TSScrollBar.DoMouseCommand (VAR theMouse: Point; VAR info: EventInfo;
      VAR hysteresis: Point): TCommand; OVERRIDE;
```

This method is overridden to handle the thumb. If the thumb is tracked, then when the thumb is released, the scroll bar's value is passed to each controlled view's ScrollRelative method. The values returned by each ScrollRelative method is added to the scroll bar's value. If a part other than the thumb is clicked, then INHERITED DoMouseCommand is called.

```
PROCEDURE TSScrollBar.TrackScrollBar (partCode: INTEGER);
```

This method is used to track the cursor while it's in an arrow or page area of a scroll bar. It simply calls each scroller's ScrollStep method and adds the values returned by each ScrollStep to the scroll bar's value.

Resource-Writing Methods

```
PROCEDURE TSScrollBar.WRes (theResource: ViewRsrcHndl; VAR itsParams: Ptr);
      OVERRIDE;
```

This method writes the `TSScrollBar` portion of the view's resource template. It is the inverse of `TSScrollBar.IRes` and is only used by programs that write `'view'` resources.

```
PROCEDURE TSScrollBar.WriteRes (theResource: ViewRsrcHndl; VAR itsParams:
      Ptr); OVERRIDE;
```

This method serves as a wrapper for `WRes`. If you want to write a window object as part of a `'view'` resource you would call this method, which calls the appropriate `WRes` methods.

Creation/Destruction Methods

```
PROCEDURE TSScrollBar.ISScrollBar (itsSuperView: TView; itsLocation, itsSize:
      VPoint; itsHSizeDet, itsVSizeDet: SizeDeterminer; itsDirection:
      VHSelect; itsMax: LONGINT; itsScroller: TScroller);
```

This method initializes a scroll bar. It calls `IControl`, and then sets its fields to the given parameters. `fLongMin` is set to zero.

```
PROCEDURE TSScrollBar.IRes (itsDocument: TDocument; itsSuperView: TView; VAR
      itsParams: Ptr); OVERRIDE;
```

This method initializes a `TSScrollBar` object from a `'view'` resource. `itsDocument` is the document for which the scroll bar is being created. `itsSuperView` is the scroll bar's superview, and `itsParams` is a pointer to the `TView` section of the object's resource. `INHERITED IRes` is called to initialize the inherited data (in this case, the `TView`, `TControl`, and `TCtlMgr` data), then the `TScrollBar` data is initialized, and finally itsParams is offset by the length of the `TScrollBar` data.

```
PROCEDURE TSScrollBar.Free; OVERRIDE;
```

This method frees the `fScrollers` list, and then calls `INHERITED Free`.