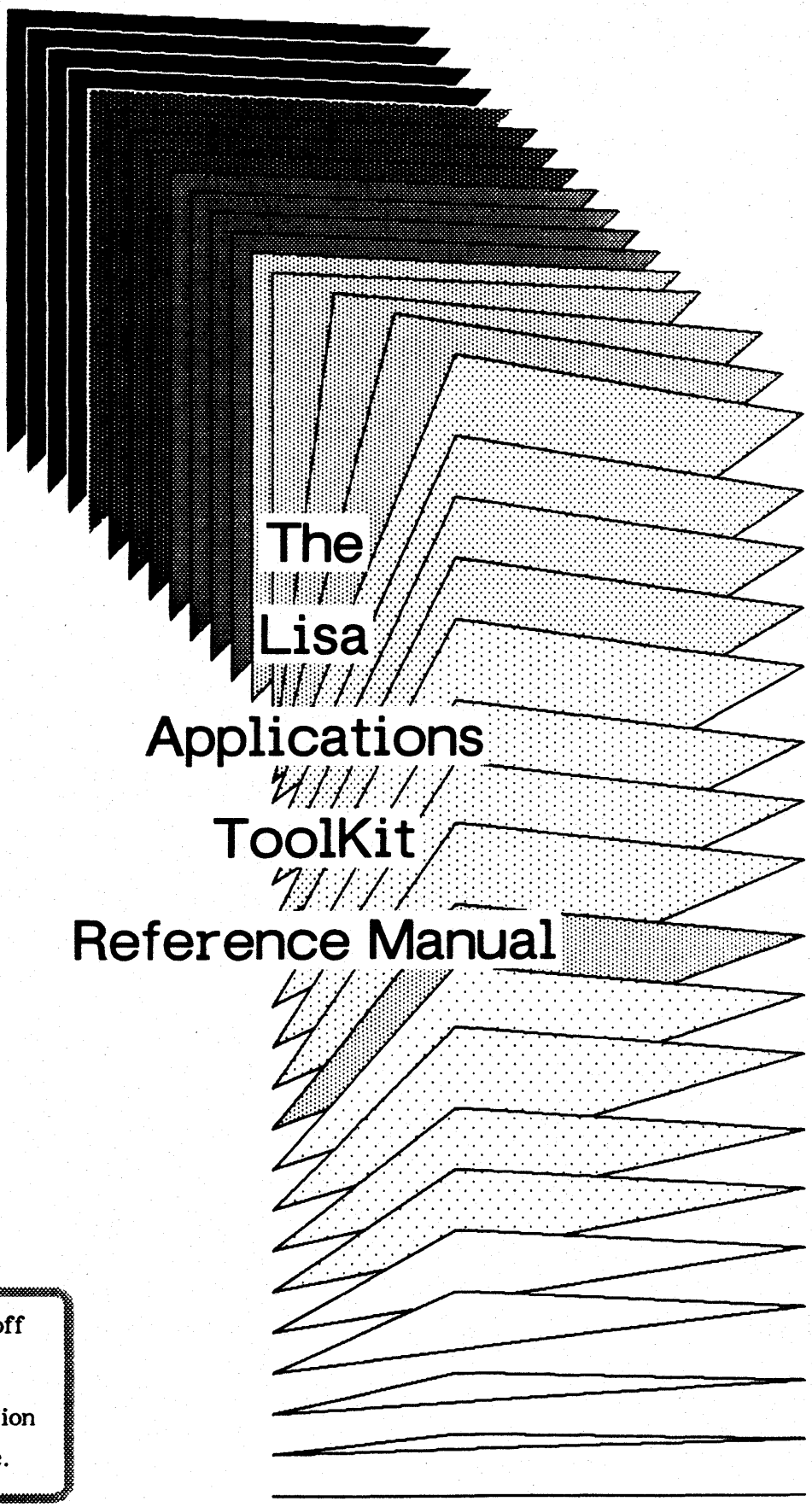


The Lisa Applications ToolKit Reference Manual

by Jonathan D. Simonoff

for

Macintosh User Education

Apple Computer, Inc.

The Lisa Applications ToolKit Reference Manual

Jonathan D. Simonoff

Licensing Requirements for Software Developers

Apple has a low-cost licensing program, which permits developers of software for the Lisa to incorporate Apple-developed libraries and object code files into their products. Both in-house and external distribution require a license. Before distributing any products that incorporate Apple software, please contact Software Licensing at the address below for both licensing and technical information.

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Simultaneously published in the USA and Canada.

Customer Satisfaction

If you discover physical defects in the manuals distributed with a Lisa product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

In addition, if Apple releases a corrective update to a software product during the 90-day period after you purchased the software, Apple will replace the applicable diskettes and documentation with the revised version at no charge to you during the six months after the date of purchase.

In some countries the replacement period may be different; check with your authorized Lisa dealer. Return any item to be replaced with proof of purchase to Apple or to an authorized Lisa dealer.

Limitation on Warranties and Liability

All implied warranties concerning this manual and media, including implied warranties of merchantability and fitness for a particular purpose, are limited in duration to ninety (90) days from the date of original retail purchase of this product.

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is," and you the purchaser are assuming the entire risk as to their quality and performance.

In no event will Apple or its software suppliers be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data.

The warranty and remedies set forth above are exclusive and in lieu of all others, oral or written, express or implied. No Apple dealer, agent or employee is authorized to make any modification, extension or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

License and Copyright

PREFACE

This manual documents the Lisa Applications ToolKit. It is intended for experienced programmers. At a minimum, readers should be familiar with Lisa Pascal, Lisa Clascal, the WorkShop, the *Lisa User Interface Guidelines*, and the Lisa office system. In particular, the reader is assumed to be familiar with the meaning of the following terms:

Clascal	class	class-type	object	object reference
method	pointer	handle	field	tool
window	panel	pane	mouse	mouse-pointer
view control	scroll bar	scroll arrow	resize icon	elevator

Chapter 1 describes the ToolKit in general. In particular it describes the different parts of the ToolKit, and the way those parts fit together.

Chapter 2 describes UObject. That unit includes the basic definitions of objects and definitions of classes for the creation and manipulation of files and complex data structures.

Chapter 3 describes UDraw, which defines classes used to convert displayed information from formats used by the ToolKit to formats used by QuickDraw, and vice versa.

Chapter 4 describes UABC, which implements most of the functions common to all Lisa applications.

Chapter 5 describes the debugging facilities available with the ToolKit.

Chapter 6 documents reference information for ToolKit units, including global variables, procedures, and functions, and non-class type definitions.

Part II contains reference sheets on all of the important ToolKit classes.

The following typographic and linguistic conventions are observed in this manual:

1. All program fragments, including variables and reserved words, are in bold type wherever they appear. Reserved words are, additionally, printed entirely with capital letters.
2. All method, object, and class names are in bold type wherever they appear. This is to avoid confusion; for example, when the word "view" is shown in bold (**view**), it refers to an object called **view**. When "view" is used in a more casual context, it is not bolded. An example is with "window," where there is the window as the box that appears on the display and the **window** as the instance of **TWindow** or instance of a descendant of **TWindow** that controls the window on the display.
3. All class names begin with a capital T. The letter following the T is also capitalized.

4. Method names are always capitalized.
5. Object-reference variable names and field names are not capitalized, even if they appear at the beginning of a sentence. (The one exception to this rule is in section titles, where object names are capitalized.)
6. Any words, or abbreviated words, appearing as parts of a class name, method name, or field name begin with a capital letter. For example, allowMouseOutside is a field name containing the words allow, mouse, and outside.
7. When speaking of objects of some particular class, rather than of the class itself, the object is usually given the class name without the T. This is often used in a generic sense; for example, when speaking of characteristics of objects of class TProcess or of some descendent of TProcess, the object is called the process.
8. The word *document* refers to the data associated with an icon on the desktop.

Contents

Part I
Chapter 1
Introduction to the ToolKit

1.1 The Parts of the ToolKit 1-3
1.2 The Parts of a ToolKit Application 1-5
1.3 Objects, Classes, and Space Allocation 1-9
1.4 How the Parts Fit Together 1-13
1.5 Failure Conditions 1-15

Chapter 2
UObject

2.1 About UObject 2-2
2.2 Objects and Heaps 2-2
2.3 Collections 2-6
2.4 Scanners 2-8

Chapter 3
UDraw

3.1 Introduction 3-2
3.2 The Purposes of UDraw 3-2
3.3 The UDraw Classes 3-3
3.4 The UDraw Routines 3-4
3.5 When To Use the Conversion Routines 3-4
3.6 Variables Declared in UDraw 3-4
3.7 The UDraw Utility Routines 3-5

Chapter 4
The Application Base Classes

4.1 What the ABC's Are 4-2
4.2 Which ABC's Contain What Functions 4-2
4.3 Global Variables 4-7
4.4 Global Procedures and Functions 4-8

Chapter 5
The ToolKit Debugger

5.1 The Debugging Facilites 5-2
5.2 The Pulldown Debug Menu 5-3
5.3 The Interactive Debugger 5-8

Chapter 6
Reference Information for ToolKit Classes

6.1 About This Chapter 6-2
6.2 Data Types 6-2
6.3 Global Variables 6-5
6.4 Global Procedures and Functions 6-10

Part II
Class Reference Sheets

Chapter 1
Introduction to the ToolKit

1.1 The Parts of the ToolKit 1-3
1.1.1 UObject 1-3
1.1.2 UDraw 1-4
1.1.3 UABC 1-4
1.1.4 Building Blocks 1-5
1.2 The Parts of a ToolKit Application 1-5
1.2.1 The Window 1-5
1.2.2 The View 1-6
1.2.3 The Panel 1-6
1.2.4 The Pad 1-8
1.2.5 The Selection 1-8
1.2.6 The Menu and Commands 1-8
1.3 Objects, Classes, and Space Allocation 1-9
1.4 How the Parts Fit Together 1-13
1.5 Failure Conditions 1-15

Introduction to the ToolKit

Lisa applications are characterized by the following design principles:

- Extensive use of graphics, including windows and the mouse pointer.
- Use of pull-down menus for commands.
- Few operating modes, or none at all.
- Data transfer between documents by simple cut and paste operations.

The ToolKit is a set of libraries that, wherever possible, provide standard behavior that follows these principles. For example, all Lisa applications have windows that can be moved around the screen; usually the windows can be resized and scrolled. The ToolKit provides all of these functions. The ToolKit also displays a menu bar for the active application, and provides a number of standard menu functions, such as saving, printing, and setting aside.

However, the ToolKit is more than a set of libraries. Because the ToolKit is written using Clascal, the ToolKit is almost a complete program by itself. (See the *Introduction to Clascal*.) You can, in fact, write a five-line main program, compile it, link it with the ToolKit, and run it. What results is the Generic Application.

The Generic Application has many of the standard Lisa application characteristics. A piece of Generic Application stationery can be torn off, and, when the new document is opened, it presents the user with a window with scroll bars, split controls, size control, and a title bar. The window can be moved, resized, and split into multiple panes. There is a menu bar with a few standard functions, so that the generic document can be saved, printed, and set aside. The single Generic Application process can manage any number of documents. You cannot, however, do anything within the window, aside from creating panes. The space within the window, along with the additional menu functions, is the responsibility of the real application.

Therefore, when you write a Lisa application using the ToolKit, you essentially write extensions to the Generic Application. It is easy to write extensions to any Clascal program. In order to create your application from the basic Generic Application, you create a set of subclasses, including methods to perform the work of your application. You then write a simple main program and compile and link it with the ToolKit.

Whenever necessary, the ToolKit calls your application's routines. In particular, although the ToolKit draws the scroll bars and the rest of the frame of the window, your application must draw the contents of the window. For example, if the user scrolls the document, the ToolKit tells your program

to redraw the changed areas. The ToolKit tells your program to redraw in other situations as well, such as when the user opens the window, chooses "Revert to Previous Version" from the menu, or enlarges the window.

One advantage of Clascal is that you can write applications in steps. You can begin by doing the least amount possible, and get an application that does very little, but will run. You can then extend your application bit by bit, checking as you go. This characteristic of Clascal makes it easy to extend the capabilities of ToolKit programs, even years after you write the original program.

1.1 The Parts of the ToolKit

The basic ToolKit is contained in three units:

- UObject defines class **TObject**, which is the ancestor class for all Clascal objects. It implements debugging, copying, and heap management for all objects used in the ToolKit. It also implements a number of complex data structures. These data structures -- arrays, strings, files, and lists -- allow your program to easily store groups of data. (This unit is also used when writing Clascal programs without the rest of the ToolKit.)
- UDraw reimplements all QuickDraw procedures, using long integers in place of QuickDraw's regular integers, so that documents can be as large as necessary. (Ordinary QuickDraw procedures used with the ToolKit limit documents to 41 pages.) It also defines a number of graphics routines that are not defined by QuickDraw.
- UABC contains the bulk of the ToolKit: the Application Base Classes.

A set of other units, called building blocks, implement standard, but not universal, behavior. These include building blocks for text editing, creating rulers, and managing dialog boxes.

1.1.1 UObject

UObject defines class **TObject**. All classes are descendants of **TObject**.

UObject implements the following characteristics for **TObject**:

- Objects are stored on a compactable heap.
- Objects have certain debugging facilities available.
- Objects can be freed; that is, have their heap space released.
- Objects can be copied (cloned).

UObject also implements a number of standard data structures for use with Clascal and the ToolKit. It implements a group of utility routines for use with the data structures. The data structures are

- arrays, which are one-dimensional arrays of records.
- lists, which are one-dimensional arrays of object-references.

- strings, which are one-dimensional strings of characters.
- files, which are disk-based one-dimensional strings of characters.

The utility methods provided for these classes allow your program to add, delete, and retrieve objects from the structures.

In addition, UObject provides a set of utility classes that scan through the structures.

The structures are all quite similar to each other.

Any files created and handled by file objects are separate from the files used to store the application's documents. The document's files are created and managed entirely by the ToolKit, and you generally never deal with them yourself. Every document has a docManager, an object of type TDocManager, which handles the document's files.

Chapter 2 describes UObject more fully.

1.1.2 UDraw

QuickDraw is based on simple (16-bit) integers, and, as a result, has fairly severe limitations on how large documents can be. The ToolKit therefore provides unit UDraw, which defines drawing routines that use long (32-bit) integers, which can be used in place of QuickDraw routines that use simple integers.

This change increases the maximum document length and width from 41 pages in each dimension to something like four million pages in each dimension.

UDraw also provides a small set of additional routines. These routines convert LRects (based on long integers) to Rects (based on simple integers) and Rects to LRects, LPoints to Points and Points to LPoints. In addition, there are routines that provide other useful functions.

Chapter 3 describes UDraw more fully.

1.1.3 UABC

UABC, which defines the Application Base Classes, is the heart of the ToolKit. Most of the standard behavior provided by the ToolKit is implemented in this unit. It defines TProcess, TDocManager, TWindow, TPanel, TPane, TCommand, among other classes. The routines that handle the menus and the mouse are defined here. Most of the work you must do to get your application running involves subclassing UABC classes.

For example, the class TDocManager contains methods to manage the files and data segments used by an application. You must always subclass TDocManager, so that your own files, rather than the generic files, are managed.

Chapter 4 describes UABC more fully.

1.1.4 Building Blocks

The building blocks are units that provide useful behavior beyond what the basic ToolKit does. The most commonly used building blocks are UText, the Text Building Block, which provides simple text editing and formatting functions, and UDialog, which provides for program dialogs with the user.

Another building block implements universal text for clipboard communication with non-ToolKit programs.

This document does not discuss the building blocks. Sample programs using the existing building blocks, as well as some documentation, are available.

Because building blocks are units that are added to programs on top of the existing ToolKit units, any developer can create them. Useful building blocks might implement graphics editing, bitmap editing, data entry and formatting, or table editing.

1.2 The Parts of a ToolKit Application

One of the characteristics of Clascal is that programs should always deal with objects. Almost everything in an application is represented by an object, from the data to the scroll bars. For example, the window you see on the display is controlled by an instance of the ToolKit class TWindow. In this manual, the window object is usually called **window**.

A number of the objects your program will use are members of classes defined by the ToolKit. The most important ToolKit classes for the programmer are

- TWindow
- TView
- TPanel
- TSelection
- TDocManager
- TCommand

There are also a number you never need to worry about. For example, the scroll bars are defined by ToolKit classes, but the ToolKit handles them. In addition, the panes are controlled by instances of classes defined by the ToolKit, but the programmer does not have to deal directly with them.

The objects that control the application's display are associated with the parts of the display that they control in Figure 1-1.

1.2.1 The Window

A window object controls the region of the Lisa's display that belongs to one window of a particular application. The image produced on the Lisa display by the window objects is called a window. Windows have title bars, borders, and resize icons. (The scroll bars belong to the panel object. See Subsection 1.2.3.)

A sample window is shown in the bottom of Figure 1-1. Notice that the window surrounds the panels and panes, but the panels and panes are controlled by separate objects.

Your application can manage several documents at a time, each in its own window. At most, one window and its associated window object is bound to the address space of the process at a time.

Your application does not need to do much to make each window function correctly. Almost all window functions are handled by the ToolKit.

1.2.2 The View

The view object creates a graphical representation of the data used in an application. That graphical representation, the view's picture, is called the view.

What you see in an application's window is usually just a part of a view. If you scroll the window, you are essentially moving the window around on top of the view. Figure 1-1 shows how two noncontiguous pieces of the view are shown in the window. The shaded area of the view shown in the figure does not appear in either of the panes, and hence does not appear in the window. The user uses the scroll icons to show desired parts of the view.

A window can have more than one view object. There can be several views of a single set of data. A LisaGraph window, which shows a numerical and a graphical representation of the same set of data, has two views and one data set. In other applications, separate views may display independent sets of data.

The application does not need to be concerned with what part of the view is actually shown in the window. The ToolKit can handle that. Your application can, however, find out exactly what is visible in the window, to improve program efficiency.

A view is created separately from the display. In order to show a view in a window, you associate the view object with a panel object.

1.2.3 The Panel

The panel object controls the scroll icons and shows a portion of the view. Every view that is displayed is shown in a panel.

Some applications, such as LisaWrite, have only one panel per window. Others, such as LisaGraph, have more than one.

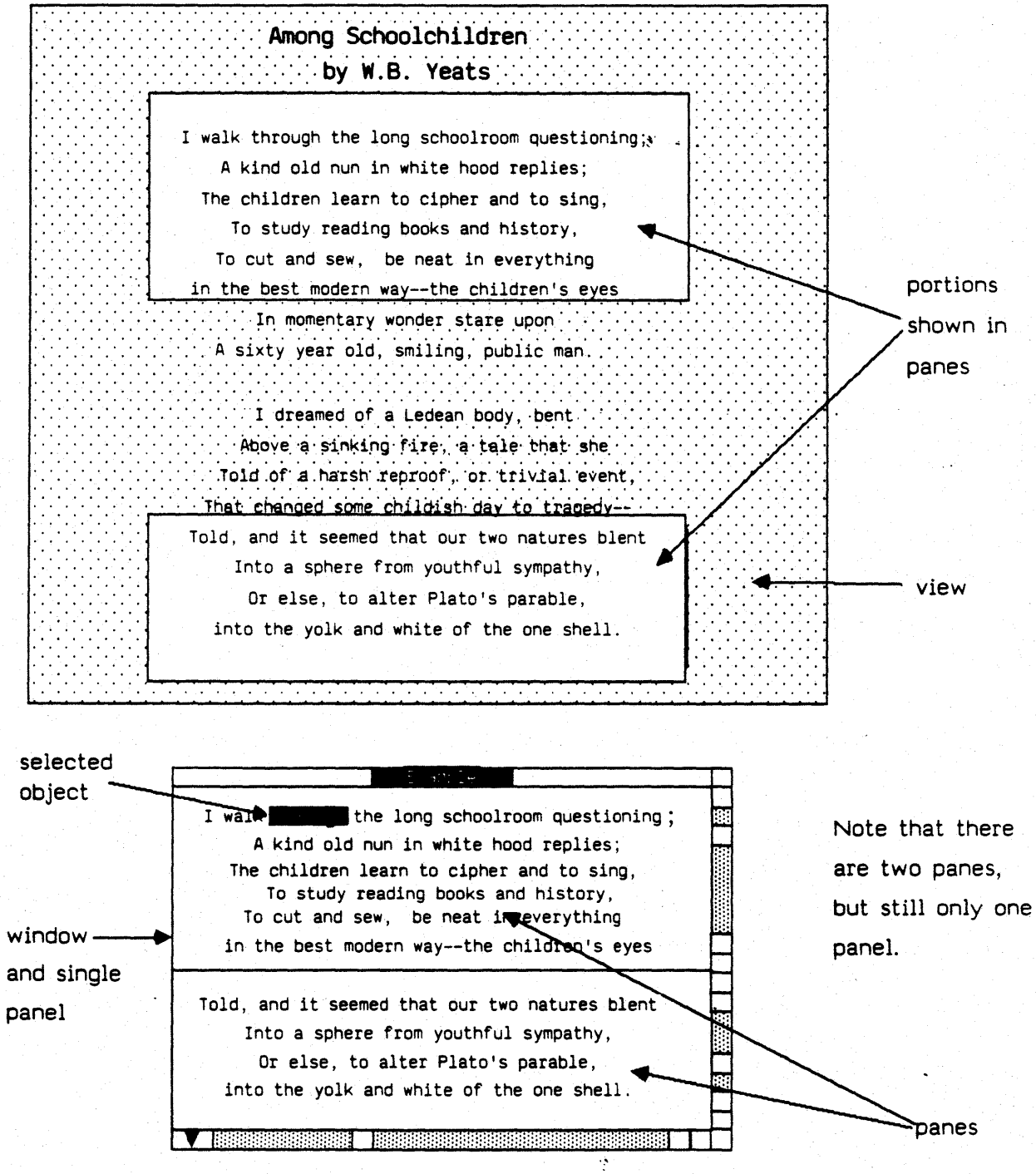


Figure 1-1 Window, Panel, Panes, and View

Each panel usually looks at a different view from every other panel.

Figure 1-1 shows a one-panel window. Notice that the panel's vertical scroll bar is divided into two parts, one for each pane. Every panel object is associated with a single view object, so any panes in the panel can only show portions of that particular view. In Figure 1-1, both of the panes show portions of the single view represented in the top of the figure.

1.2.4 The Pad

The ToolKit's graphics are computed using long integers. QuickDraw actually does the on-screen drawing using regular integers. pad objects are used to convert from long integers to regular integers, and the other way around.

Just as QuickDraw provides a global variable thePort, the ToolKit provides a global variable thePad to represent the current destination of graphic output. The value of thePad changes as the destination of output changes, but those changes generally are transparent to your program.

Programs often call the TPanel method OnAllPadsDo, which assures that any changes made to a panel's display are shown in all parts of the panel.

Programs also occasionally call methods of thePad.

pane objects are members of class TPad, because TPane is a subclass of TPad. The effect of this is that any pane can be thePad, and, so, any pane can be the destination for graphic output.

1.2.5 The Selection

Often, a command operates on an object that is currently selected. For example, in order to change the type style in LisaWrite, you select some characters and then choose a menu item.

Every panel object has a selection object associated with it. When a menu command is chosen, a command number is sent to the selection object, which then decides whether or not the command can be done. If it is possible to do, the selection object performs the operation, creates a command object that performs the operation, or tells another object to perform the operation.

Every panel object has one and only one selection object. If there is more than one panel in a given window, one of the panel objects is defined to be the selectPanel for the window object. The selection in the selectPanel is given the application's commands.

Any selection object can point to any number of display and/or data objects, from zero on up. Those objects are the *selected objects*. Selected objects generally are highlighted in some way to draw attention to them. Figure 1-1 shows a single selected object in the window in the lower part of the figure. The selected object is highlighted by reverse video.

1.2.6 The Menu and Commands

All commands available to an application user appear in pull-down menus that are listed in the menu bar. You create these menus by putting the menu names and item names in the phrase file for the application. Whenever the

user pulls down a menu, the application can enable or disable menu items, and it can also mark menu items as checked off. If an Apple-key equivalent to a menu command is allowed, that is also in the phrase file.

When a menu command is chosen, the command is given to the selection object in the selectPanel of the active window.

If the command is not one of the application's commands, it is passed to the ToolKit, which carries out the command if it is a legal ToolKit command, or notifies the user that the command cannot be done.

1.3 Objects, Classes, and Space Allocation

Every ToolKit class is a descendant of class TObject. See the section on UObject (the unit that defines TObject) for more information.

Pieces of data used in ToolKit applications are often stored in instances of classes, and so, are themselves objects. Data might be stored in objects of class types defined in the application or in the collection classes defined in UObject. It is important to store your program's information in objects, rather than in global variables, because only objects are saved and restored by the ToolKit. If you do not store your information in objects, you must make certain that the information is saved and restored correctly.

Class-type variables and other object references are not class objects, but actually are *handles* (double-indirect pointers) on objects. Note that it is not necessary (or even permissible) to use pointer variables or carets (^) to resolve these handles; when fields of the class are used, the variables are automatically resolved to return the stored values.

Figure 1-2 shows how object references in programs are handles on objects stored in the document heap.

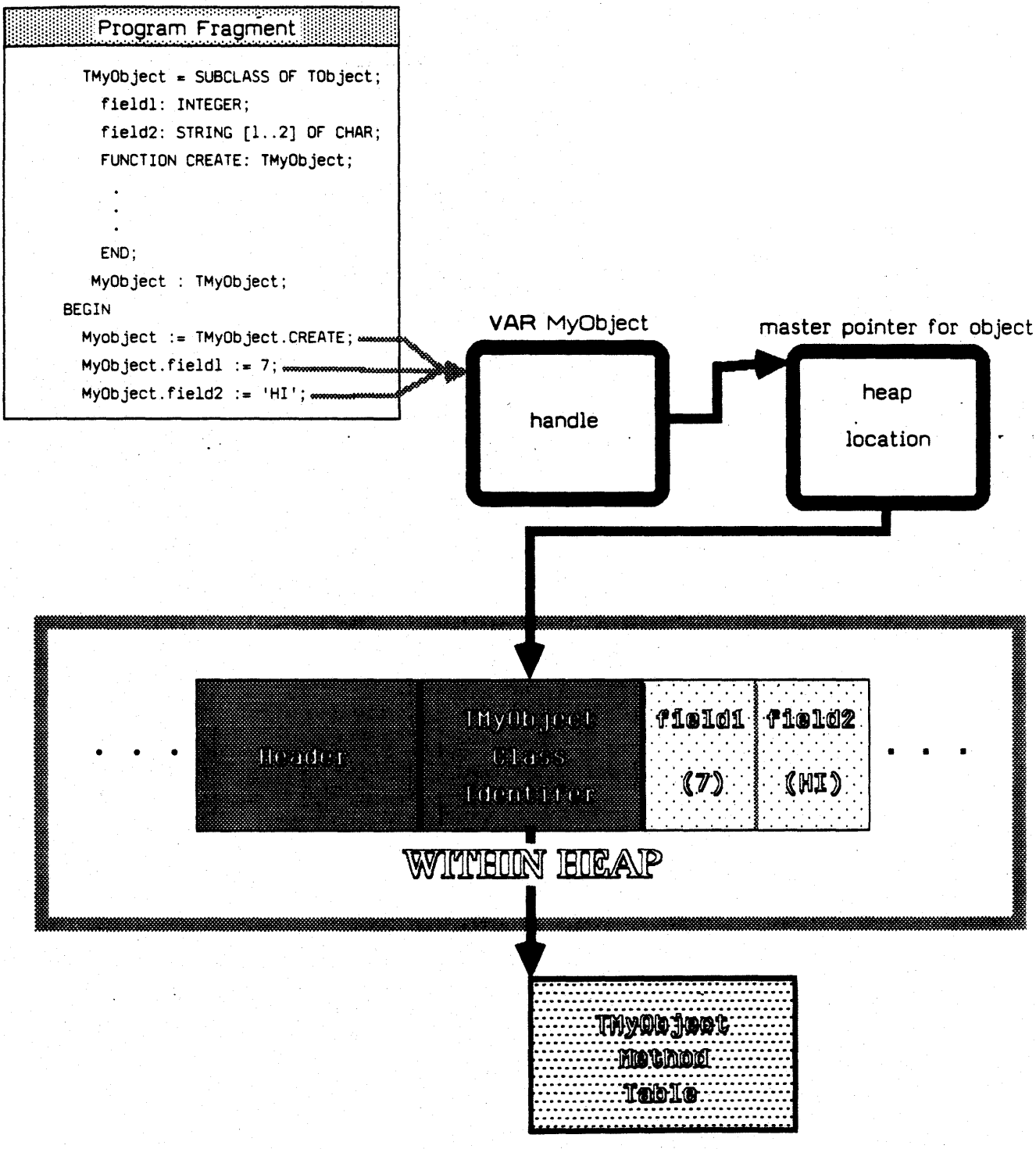


Figure 1-2 Object References

Every subclass must have a CREATE method written specifically for that particular subclass. When a class's CREATE method is used, a new instance of the class (an object) is created. That is, CREATE allocates heap storage for the data stored in the object, and for pointers to the methods of the class. CREATE is a function that returns a handle on the new class object.

When a class-type variable's value is assigned to another class-type variable, the target class-type variable and the source variable both point to the same object. Here is an example, where red and green end up pointing at the same object:

```
TYPE
  TColor = SUBCLASS OF TObject        {Defines a new class.}
    FUNCTION CREATE (object: TObject; itsHeap: THeap):TColor;
    END;
VAR
  green : TColor;    {Creates a TColor-reference variable.}
  red : TColor;      {Creates another TColor-reference variable.}
BEGIN
  green:=TColor.CREATE(object,heap); {Creates an instance of TColor.}
  red := green;  {Makes red point to the same object as green.}
    .
    .
    .
```

Notice that there is never a CREATE call for the variable red. The CREATE method allocates heap space for the object. In the example above, the heap space is allocated by using CREATE to initialize the variable green. When the value of green is assigned to red, red becomes a handle on the same heap space on which green is a handle. CREATE methods create new storage space; a direct assignment to a class-type variable creates a new access path to the same storage space.

Suppose you used a sequence like this:

```
green := TColor.CREATE (object, heap);  {Creates an instance of
                                          TColor.}
red := TColor.CREATE (object, heap);    {Creates another instance
                                          of TColor.}
red := green;     {Makes red point to the same object as green.}
```

The storage space created when you used TColor.CREATE for red would be lost, because red becomes a handle on the storage space created for green. (The storage space created for red may be permanently lost to the program's use, because the heap manager maintains the space until it is deallocated with a Free method. TObject has a Free method, and, therefore, all descendants of TObject also have one. If you wanted to get rid of the object pointed to by red, you would have used the method call red.Free before the line red := green.)

Figure 1-3 represents the way different assignments work in a number of different cases.

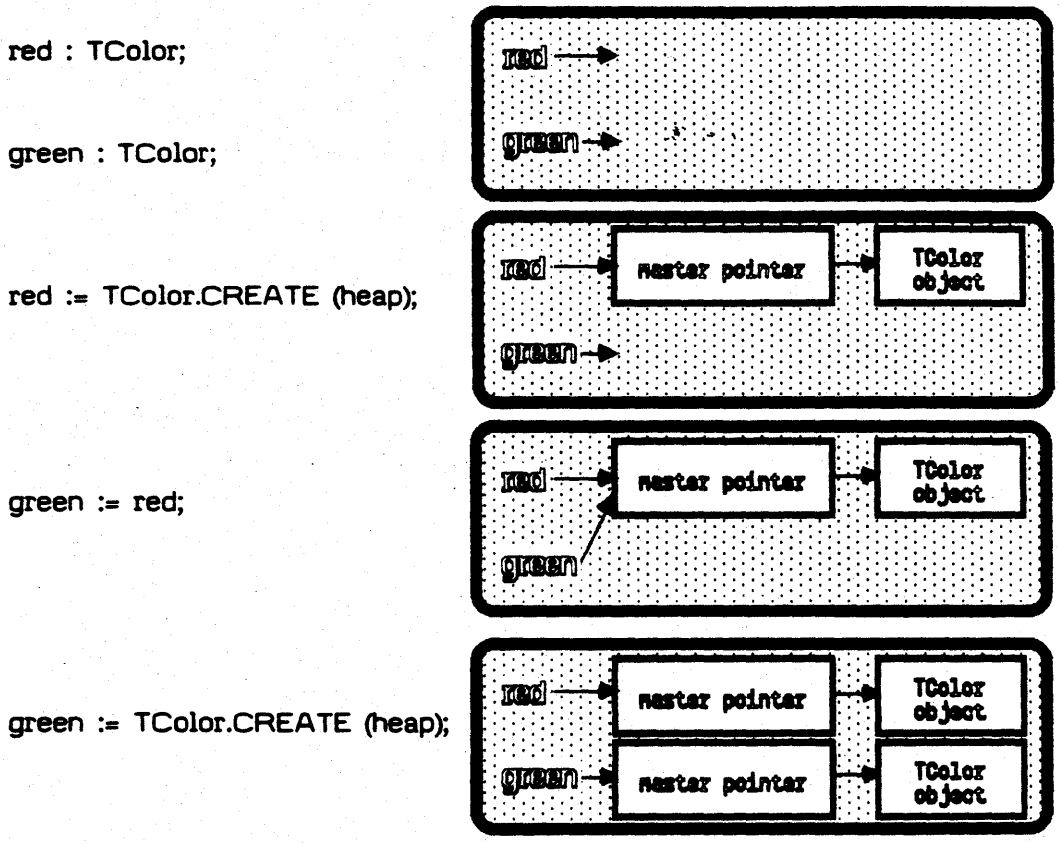


Figure 1-3 Class-type Variable Assignments

When the variables green and red are first defined, they do not point anywhere. After TColor.CREATE is used to create an object for red, red points to a master pointer, which points to the new object. Green still points nowhere. Assigning the value of red to green makes green point at the same master pointer, hence the same object, as red. When TColor.CREATE is called again to create a new object for green, green points to the master pointer for the new object.

Notice the crucial difference between a class-type variable and an ordinary Pascal variable, such as a RECORD. A class-type variable is actually a handle on a piece of storage. Therefore, when the value of one class-type

variable is assigned to another, the second variable and the first point to the same piece of storage. When the value of a RECORD is assigned to another RECORD, the second RECORD contains a duplicate of the first.

1.4 How the Parts Fit Together

In an object-oriented program, the objects act on themselves when some object tells them to act. In the ToolKit, the ultimate controlling object is the process.

The process object contains a method called Run. Run consists of an event loop that checks the display, updates it when necessary, and checks for a command or other event. When a command is received, the command is sent to the active selection object.

The event loop repeatedly checks for user events: key strokes, menu commands, and mouse clicks. When an event is received, an action is taken depending on what type of event it is and where it happens.

For example, when the mouse button is pressed, the Window Manager, the part of the Lisa office system that manages the display, first checks to see if the mouse was on the desktop, in one of the application windows, in the title bar for a window, or in the menu bar. If the mouse was within the outer boundary of an application's window (exclusive of the title bar), the Window Manager gives the event to the process object for that application. The process gives the event to the window object. The window checks to see if the mouse was clicked on the resize icon. If so, the window tracks the mouse's movement until the button is released, and grows or shrinks the displayed window appropriately. If the mouse was not on the resize icon when the button was pressed, the event is passed to the panel objects for the application. Each of the panels checks to see if the event was within its outer boundaries. When one of the panels discovers that the area in which the mouse was clicked is within its boundaries, that panel tells its scrollBar objects to see if the mouse was there. If it was in a scroll bar, the panel scrolls appropriately. If the event was not in a scroll bar, the panel tests whether the event was in a panel resize icon. If it was in such an icon, it resizes appropriately. Otherwise, the panel asks each of the pane objects, which control the visible contents of the panel, to check if the event was within the area displayed by that pane. When one of the panes claims the mouse click, it informs the view object that controls the objects displayed by the pane. The view determines on which part of the view the mouse was clicked, and then takes some action, usually beginning a new selection. Then the panel is given control again. The panel tracks the movement of the mouse, informing the pane, view, and selection as the mouse moves.

As another example, the mouse may have been clicked in the menu bar. In that case, the Window Manager gives the event to the process, which passes it to the window, which gives it to the menuBar object. The menuBar waits until a command is selected, and then calls window.DoCommand and sends it the command number. The window object calls selection.CreateCommand, which creates a command object of the right type for that command. The

command object is then told to carry out the command. It may call ABC methods, building block methods, or application specific methods.

Many of the higher level objects, such as the window and the selection, have fields that point to other objects. TWindow defines a field, TWindow.selectPanel, which contains a handle on the selectPanel. Because all handles are equivalent object-references, and all references to objects are through handles, the window object can easily call a method of the selectPanel object. In turn, the selectPanel has fields for any objects it needs to reach.

The way you structure these field references is central to how your program operates.

As a result of the fact that you often reach methods through fields, the calls to methods in ToolKit programs may occasionally seem quite complex.

When myObject is an object with the following definition:

```
TYPE
 TMyObject = SUBCLASS OF TObject      {Creates a new class.}
  qualities: TQuark;          {Defines a field of class TMyObject.}
  FUNCTION CREATE(object:TObject; itsHeap:THeap): TMyObject;
  END;
 VAR
  myobject: TMyObject; {Defines a TMyObject-reference variable.}
```

and TQuark and quark have this definition:

```
TYPE
 TQuark = SUBCLASS OF TList;   {Creates a new class.}
   FUNCTION CREATE (object:TObject;
                    itsHeap:THeap;initialSize:INTEGER):TQuark;
   FUNCTION ChangeSpin (i: INTEGER);
   END;
VAR
 quark: TQuark;      {Defines a TQuark-reference variable.}
```

you can invoke method TQuark.ChangeSpin like this:

```
quark := TQuark.CREATE (object, itsHeap, initialSize); {Creates an
                                                        instance of
                                                        TQuark.}
myObject := TMyObject.CREATE (object, itsHeap); {Creates an instance
                                                 of TMyObject.}
myObject.qualities := quark;  {Makes the qualities field of myObject
                               point to the same object as quark.}
myObject.qualities.ChangeSpin (i);  {Calls a method of TQuark through
                                     the qualities field of myObject.}
```

The effect of the last line is to change the spin in the ith element of the list object for which both myObject.qualities and quark are handles. There is only

one object, although it is shown here with two separate handles. The following line has the same effect:

```
quark.ChangeSpin (1);
```

1.5 Failure Conditions

Some ToolKit methods and procedures return error codes when there is a failure, but many do not. For example, if **NewObject** (a global procedure defined in UObject) fails to allocate because the swapping space on the disk is full, it does not return an error code. Instead, when there is a failure, UObject's procedure **ABCBreak** is called. In the debugging version of the ToolKit, the Lisa beeps, and enters the debugger. Press the right-option key and the Enter key at the same time to display the alternate screen, which shows the error message. In the production version of the ToolKit, the Technical Difficulties alert is displayed. The application terminates when the user responds to that alert.

To facilitate debugging, some error checks are enabled by commands in the Debug Menu, notably "Check List Indices," which enables subscript bounds checking in lists.

Chapter 2
UObject

2.1 About UObject 2-2
2.2 Objects and Heaps 2-2
2.2.1 How UObject Manages a Heap 2-2
2.2.2 Creating Objects 2-2
2.2.3 TObject 2-3
2.3 Collections 2-6
2.3.1 TList 2-7
2.3.2 TArray 2-7
2.3.3 TString 2-7
2.3.4 TFile 2-7
2.4 Scanners 2-8
2.3.4 FileScanners 2-10

UObject

2.1 About UObject

UObject provides the basic structures for objects, object debugging, and object organization. Sections 2.2 and 2.3 deal with the structure and concepts underlying these units.

The type definitions provided by UObject provide capabilities in two categories:

- Heap and memory management and debugging. This includes the definition of type TObject, which is the ancestor of all classes. TObject and heap management are discussed in Section 2.2. The ToolKit debugger is discussed in Chapter 5.
- Organization, inspection, and manipulation of groups of objects and other data. The collections provided allow for the creation and manipulation of lists, strings, arrays, and files. These are discussed in Section 2.3.

2.2 Objects and Heaps

UObject implements the most important and basic parts of the ToolKit, namely heap management and the basic object definitions. The most basic object definition, that for TObject, is discussed in subsection 2.2.2. ToolKit heap management is discussed in Section 2.2.1.

2.2.1 How UObject Manages a Heap

A heap consists of a header, an area for allocation of master pointers to objects, and a variable length storage area in which frequent allocation and deallocation occur.

The value of any class-type variable is a double-indirect pointer: a 4-byte handle on a 4-byte master pointer to an object consisting of a *storage block* (containing the fields of the object) and an 8-byte header comprised of:

- A 2-byte indicator of the size of the object.
- A 2-byte locator for the master pointer.
- A 4-byte indicator of the object's class.

The master pointer and header together produce a 12-byte overhead for each object.

Objects are relocatable. When a heap fills up, it is compacted and the master pointers are updated. The master pointers are never moved. Because object-references are pointers to the master pointers, they do not have to be changed. If compacting the data on the heap does not yield enough space, the heap is expanded.

2.2.2 Creating Objects

Every class must have its own, unique CREATE function. When you want to create a new object of some class, invoke the CREATE function for that class by calling TThisClass.CREATE, where TThisClass is the name of the class.

The CREATE function allocates space on the heap for a new object of that class-type, and returns a handle on the new object. Each subclass is required by Clascal to define its own CREATE function; CREATE is not a method, and is not overridden by the subclass, because the CREATE function for each subclass often has different arguments.

Every CREATE function should, by convention, have a TObject-type reference as its first parameter, and a heap as its second parameter. The remainder of the parameters depend on what you need to initialize the class. In general, CREATE functions follow the following model, where TMyObject is the class you are defining, and TMyObject is a subclass of TSuperObject.

```
FUNCTION TMyObject.CREATE (obj: TObject; heap: THeap;
                           parameters needed for this
                           object);
BEGIN
   IF obj=NIL THEN obj:= NewObject(heap, THISCLASS);
   SELF:= TMyObject (TSuperObject.CREATE(obj, heap, parameters
                     needed by TSuperObject));
         {If TMyObject is a subclass of TObject, the previous line
         is: SELF:= TMyObject(obj);}
   {initialize myObject's parameters here}
END;
```

When you create a new object of type TMyObject, pass in NIL as the value of obj. The CREATE function sets the class pointer to TMyObject, and then calls each succeeding superclass, until TObject is reached. TObject allocates the proper amount of heap space, and then each subclass in turn initializes its fields.

Note that because the class pointer is set at the beginning (by NewObject), all SELF method calls in any CREATE function in the superclass chain call methods of TMyObject. If any of the superclasses's CREATE functions call a method through SELF and the method is overridden by TMyObject, or by some other descendant of the calling class that is an ancestor of TMyObject, and that method expects to have fields initialized in the object, you should initialize the fields before initializing SELF. You can initialize those fields between the call to NewObject and the call to TSuperObject.CREATE.

2.2.3 TObject

All class types are descendants of the basic type TObject, defined in UObject. TObject is the only class that has no superclass. It has no data fields, but it provides the basic methods used by all other classes.

Any descendant of TObject

- Inherits all methods and fields of its superclass.
- Can define methods itself.
- Can override any of its superclass's methods.

A subclass cannot, however, change the interface definition for methods inherited from its superclass. The single exception for this is for CREATE methods, the methods that create new objects of a given class-type. Every class defines its own interface to CREATE, as well as implementing the method. (A few methods listed in the ToolKit interface units are defined as CONCEPTUAL methods. These methods, which are commented out of the interface, are listed only for information, and are not implemented or defined. Subclasses can therefore define the interface for CONCEPTUAL methods in any way.)

The following is a brief discussion of the most important methods of TObject. See the reference sheet for TObject in Part II of this manual for more complete information.

- Heap

  Returns the heap on which this object is allocated. This method is never overridden.

- FreeObject

  Deallocates the heap space taken by the object. The master pointer and all references for the object are invalid after this method is executed. Note that this method is rarely called directly. You usually call Free. FreeObject is almost never overridden.

- Free

  Free calls the Free method for every object referred to in the fields of the original object. Free then calls FreeObject, to deallocate the space used by the original object. TObject.Free simply calls FreeObject, because TObject defines no fields. ToolKit classes override Free so that it frees all the objects referred to by fields defined by that subclass. When you create a new subclass and add fields to it, you override Free so that it frees the objects in the fields you have added. You end by calling SUPERCLASS.Free. Eventually, this chain of SUPERCLASS calls reaches TObject.Free, which calls TObject.FreeObject, and frees your object. The following example assumes that head and tail are fields of class TPair, and are object-references.

```
PROCEDURE TPair.Free;
  BEGIN
    SELF.head.Free;
    SELF.tail.Free;
    SUPERSELF.Free;
  END;
```

- CloneObject

  Creates a copy of the object and returns a handle on the new object. This method is almost never overridden.

- **Clone**

  Creates a copy of the object and all its fields and returns a handle on the new object. Similar to CloneObject, but Clone also creates copies of objects referred to by fields of the original object. As with Free, you must re-implement Clone in your subclasses so that it copies field-referenced objects; TObject.Clone simply calls CloneObject. See the description of Free for more information.

- **Debug**

  Prints details about the object, such as its class and the contents of its fields, on the alternate screen. It only exists in the debugging version of the ToolKit. This method may be overridden, but it seldom needs to be.

- **Fields**

  Enumerates the contents of the fields of the objects. This is called by Debug. Because TObject has no fields, TObject.Fields does nothing. You must implement Fields if you add fields to a subclass you create. The ToolKit implements Fields for all ToolKit classes. You simply write a routine that calls SUPERSELF.Fields and then enumerates the information for the fields you have added. Fields methods only exist in the debugging version of the ToolKit. You should compile your units {$IFC fDbg OK}. If you want to put your debugging code in a separate segment from the rest of your program, you can use $S. See the reference sheet for TObject in Part II of this manual for instructions for writing a Field method. You should follow those instructions so that the information appears in the correct format.

  Here is a model for a Fields method.

```
PROCEDURE TMyClass.Fields(PROCEDURE Field(nameAndType:
S255));
BEGIN {The fields must be listed in declared order, with none
omitted and none added.}
{Call the superclass's Fields method first (unnecessary if the
superclass is TObject)}
SUPERSELF.Fields(Field);
The following type names are recognized by the parser.
Field('flag: BOOLEAN');
Field('coCode: Byte');
Field('inputChar: CHAR');
Field('version: INTEGER');
Field('width: LONGINT');
Field('viewLPt: LPoint');
Field('boundLRect: LRect');
Field('size: Point');
Field('ptr: Ptr');
Field('boundRect: Rect');
Field('someName: STRING[100]');
```

```
Field('someByte: HexByte');
Field('someInt: HexInteger');
Field('someNumber: Real');
```

If the last field is a Byte or a BOOLEAN, force padding to a word boundary by inserting:

```
Field('');
```

For safety, you can put that in every Fields method. It can appear anywhere in the listing.

Every class name is recognized. For example:

```
Field('miscObj: TObject');
Field('myPanel: TPanel');
Field('mySel: TMySelection');
Field('appSpecific: TAppSpecific');
```

You may report more than one field in a single call to reduce code space.

```
Field('boundLRect:LRect;size:Point;ptr:Ptr;mySel:TMySelection');
```

Unpacked invariant RECORDs are recognized.

```
Field('info: RECORD version: INTEGER; size: Point END');
```

If the record has variants, use a CASE statement to choose between them.

```
CASE SELF.variant OF
  flavor1: Field('RECORD version: INTEGER; size: Point END');
  flavor2: Field('RECORD viewLPt: LPoint END');
END;
```

If SELF.variant does not exist, choose one of the variant fields.

Unpacked ARRAYs with literal bounds are recognized.

```
Field('desc: ARRAY [1..99] OF RECORD version: INTEGER; id:
ARRAY [1..2] OF CHAR END');
```

Other constructs and type names are NOT recognized; substitute one of the above forms.

As a last resort, use ARRAY [1..SIZEOF(SELF.fieldName)] OF Byte.

```
END;
```

2.3 Collections

UObject implements these basic organizational collections, all of which are objects, and descendants of type TCollection:

- **lists** are indexed lists of object-references.
- **arrays** are one-dimensional lists of records.
- **strings** are one-dimensional lists of characters.
- **files** are one-dimensional lists of characters stored in disk files.

Notice that **lists** are made up of object-references. The objects themselves, like all objects, are stored separately on the heap. **arrays** and **strings** store their data within themselves. They themselves are objects stored on the heap, however. **files** store their data on disk; the **file** object is used by your program as an interface to the disk file.

Each of these class-types (except for **files**, as explained below) has a set of methods that allow you to

- Add members to the **collection**.
- Delete members from the **collection**.
- Allocate new space for the **collection**.
- Remove unused space allocated for the **collection**.

In addition, each **collection** type has a corresponding class-type that defines a **scanner** for that type of **collection**. A **scanner** is an object that is used to move through a **collection**, acting in some way on each member in turn. **scanners** are used, for example, to search for particular **collection** members.

You can also use **scanners** to add and delete **collection** members, but those operations are typically done through methods of the collections. The advantage of using **scanners** is that your current position in the **collection** is maintained for you.

files, unlike other **collections**, can only be accessed through **fileScanners**.

2.3.1 TList

TList defines an object that maintains a list of objects-references.

A **list** is similar to a Pascal ARRAY [1..size] OF object-references. The entire **list** is stored as one variable-length, contiguous object of type TList. Because **list** elements are object-references, elements of different **lists** (or multiple elements of a single **list**) can refer to the same object.

2.3.2 TArray

An **array** is similar to a **list**, but where a **list** contains object-references, an **array** contains records. It is similar to a Pascal ARRAY [1..SIZE] OF any type. While an object can have references to it in several **lists**, an item can normally be in only one **array** at a time. (Actually, you could put object-references in an **array**, and then the **array** would be equivalent to a **list**. The interface to **lists** is designed for use with object-references, however, while the interface to **arrays** is not.)

2.3.3 TString

A **string** is a one-dimensional array of characters, similar to a Pascal ARRAY [1..size] OF CHAR. The character string itself is stored in the **string** object; any particular string of characters can exist in only one **string** object.

Each element of a **satring** occupies one byte of memory. An element of an **array** occupies at least two bytes of memory.

2.3.3 TFile

A **file** is similar to a **string** in the sense that it is used to handle a string of characters. A **file**, however, acts as an interface to a disk file. When you create a **file** object, you specify an OS pathname. The ToolKit checks to see if a file with that name exists. If it does, the length of the file in bytes is put into the size field of the **file** object, and a call to **file**.Exists returns

TRUE. The file is not opened, however. If the file does not exist, a 0 (zero) is put into the **file.size** field, and a call to **file.Exists** returns FALSE.

The disk file is opened when you create a **fileScanner** for the **file** object.

The method **file.Scanner** opens the file for read and write access. To open the file for more limited access, use **file.ScannerFrom**. See the *Operating System Reference Manual for the Lisa* for details on file access options and error codes.

2.4 Scanners

scanners are objects used to scan **lists**, **strings**, **arrays**, or **files** and operate on every member, search for something, or assist in locating the correct point to insert, replace, delete, and so on. This section discusses how to use **scanners** to manipulate your **collections**.

All **scanners** are instances of descendants of **TScanner**. Every type of **collection** has a corresponding type of **scanner**: you use an instance of **TListScanner**, **TArrayScanner**, **TFileScanner**, or **TStringScanner**. Each of the **collection** classes has a **Scanner** method, which is a function that returns a **scanner** object of the correct type. **scanners** are used and then thrown away; they generally are used to go once through a **collection**. In fact, when a **scanner** reaches the end of a **collection**, the **scanner** automatically frees itself.

Every **scanner** object has a **position** field. **scanner.position** is the point in the **collection** presently occupied by the **scanner**. The **position** is the index number of the last member of the **collection** returned by the **scanner**, unless the **scanner** is new. By default, new **scanners** have a **position** of 0, which is the position before the first member. **scanners** can, in general, start in any position, however. Reverse scanners begin with a **position** just past the end of the **collection**.

When you create a **scanner**, you can specify its **scanDirection**. A **scanDirection** of **scanBackward** means that the **scanner** moves towards the beginning of the **collection**; a **scanDirection** of **scanForward** means that the **scanner** moves towards the end of the **collection**. Every time you call the method **scanner.Scan**, the **position** moves one member in **scanDirection**. **fileScanners** can only have a positive **scanDirection**; **files** can only be scanned toward the end. The default **scanDirection** for all **scanners** is positive.

When a **scanner** is first created you can specify the initial **postion**. You do that by specifying the index number of the first member to be returned by the **scanner**. When the **scanner** is created, **scanner.position** is that index number plus or minus one, depending on **scanDirection**.

Every time you call **scanner.Scan**, which is a function, you get a **BOOLEAN** return value, and a return parameter. The return value indicates whether or not the scan is finished. It is finished when **scanner.position** is past the last member of the **collection**, if **scanDirection** is positive; or when **scanner.position** is before the first member of the **collection**, if **scanDirection** is negative. The return parameter is the next member in the **collection**, in **scanDirection**.

Because scanner.position is updated when scanner.Scan is called, scanner.position indicates the index number of the returned member. The returned member is referred to as the *current member.*

When scanner.Scan is called after the last member was returned, the return value is FALSE, and the returned member parameter is NIL or CHR(0), depending on the type of the collection. When you are done with a scanner before the last member has been reached, call scanner.Done. After that call, the next call to scanner.Scan returns FALSE, but the returned member parameter is left unchanged. The scanner then frees itself.

During a scan, objects can be inserted before or after the current position, and the current member can be deleted or replaced.

You can have more than one scanner on the same collection at the same time, but if you do, you cannot insert or delete using these scanners.

All types of scanners are used in the same way. The general format is:

```
VAR   s: TScanner;
      member: TMember;
      l: TCollection;
         .
         .
         .
s := l.Scanner;
WHILE s.Scan(member) DO
 BEGIN
 {code here accessing member}
 END;
```

Creating the scanner with s := l.Scanner sets the scan position to a point just before the first element in the list. If you want to set an inital position and scanDirection, use the method ScannerFrom to create the scanner. Each execution of the WHILE loop

- Advances value of l.scanner.position by 1.
- Sets member to the next member of the collection. When the collection is a list, member is an object-reference. When the collection is a string or a file, member is a character. When the collection is an array, member is a pointer to a record.
- Runs the code in the WHILE loop.

When the scan position reaches a point just after the last element in the list

- s.Scan returns FALSE.
- obj is set to NIL, for lists and arrays, or CHR(0) for strings and files.
- The scanner frees itself.

When the purpose of the loop is to search for a certain element, call s.Done when the element being searched for is found. Then

- The next s.Scan returns FALSE.
- The object variable retains its assigned value.
- The scanner frees itself.

The following example illustrates the use of a listScanner to peel every banana in a bunch of bananas and remove the rotten ones. If an underripe banana is encountered, no more bananas are peeled.

```
PROCEDURE CheckBananas (bunch: TList {OF Banana});
VAR  s: TListScanner;
     b: TBanana;
BEGIN
   s := bunch.Scanner;          {Create a scanner to scan bunch.}
   WHILE s.Scan(b) DO           {For each b in bunch do:}
   BEGIN
      b.Peel;                   {Tell b to Peel itself.}
      If b.underripe THEN       {ask b if it is underripe}
         s.Done                 {If so, force the scan to terminate.}
      ELSE IF b.rotten THEN     {Ask b if it is rotten.}
           s.Delete (TRUE);     {If so, tell s to delete b from bunch and
                                 free it.}
   END;
END;
```

2.4.1 FileScanners

file objects, unlike other collection objects, can only be accessed through scanners.

Once the fileScanner is created, further references to the file object act as if the contents of the file were a string stored on the heap. The contents of the disk file are not moved to the heap, however (they are paged through buffers in the OS), so file operations can take longer than string operations. If no disk file with the given name exists, the file is created when the fileScanner is created, and the file object acts like an empty string. Note that the ToolKit does not explicitly tell you whether or not the file existed before you created the fileScanner.

Chapter 3
UDraw

3.1 Introduction 3-2

3.2 The Purposes of UDraw 3-2

3.3 The UDraw Classes 3-3

3.3.1 TArea 3-3
3.3.2 TPad 3-3

3.4 The UDraw Routines 3-4

3.5 When To Use the Conversion Routines 3-4

3.6 Variables Declared in UDraw 3-4

3.6.1 Constants 3-4
3.6.2 Orthogonal Conversions 3-4
3.6.3 Pen States for Highlighting 3-5
3.6.4 Patterns in View Coordinates 3-5
3.6.5 Graphics Areas 3-5
3.6.6 Conversion Pad (noPad) 3-5

3.7 The UDraw Utility Routines 3-5

3.7.1 Arithmetic and Utility Operations 3-6
3.7.2 Graph Functions 3-7
3.7.2.1 Arithmetic on Points 3-7
3.7.2.2 Arithmetic on Rectangles 3-9
3.7.2.3 Operations on Rectangles 3-10
3.7.3 Calculations with Rectangles and Points - View Coordinates 3-11
3.7.4 Drawing Operations for View Coordinates 3-12
3.7.4.1 Line-Drawing Routines 3-12
3.7.4.2 Graphic Operations on Rectangles 3-13
3.7.4.3 Graphic Operations on Ovals 3-13
3.7.4.4 Graphic Operations on Rounded-Corner Rectangles 3-14
3.7.4.5 Graphic Operations on Arcs and Wedges 3-15

UDraw

3.1 Introduction

UDraw provides graphics routines for drawing in ToolKit programs. It must be used with all ToolKit applications. The functions UDraw provides include most of the functions provided by QuickDraw, except that UDraw uses 32-bit long integer (LONGINT) coordinates instead of 16-bit integer (INTEGER) coordinates. UDraw also provides a few extra functions.

Using long integer coordinates provides coordinates that can range from -2147483648 (-2^{31}) to 2147483647 ($2^{31}-1$). However, no single object can have a size greater than 2^{15} in either dimension.

This chapter assumes that you are familiar with the QuickDraw documentation, which is contained in the Pascal manual.

UDraw defines TArea, TPad, and TBranchArea.

TArea implements methods for all parts of an application that draw on the screen, including bordered areas, such as windows and panels.

TPad, which is a subclass of TArea, implements methods that take information drawn in an application's view, and display it on the screen.

TBranchArea, which is also a subclass of TArea, is used internally to maintain splittable panels. There is a reference sheet for TBranchArea in Part II of this manual. Because you do not deal directly with this class, it is not discussed further in this chapter.

3.2 The Purposes of UDraw

UDraw exists for two main purposes. First, it defines the basic graphical structures that characterize ToolKit applications: bordered and unbordered screen areas. Second, UDraw translates graphics from ToolKit formats to forms that can be used by QuickDraw. QuickDraw does all on screen drawing on the Lisa.

QuickDraw always draws in a *grafPort*. A grafPort is a software-defined port that connects your application's drawing routines to the screen. When you want to draw, you *focus* the grafPort on some part of the screen.

You normally do not need to worry about focusing in ToolKit programs. You draw in your view, using the view coordinate drawing routines implemented in UDraw and described in this chapter. Some of your application's methods are called with the grafPort already focused and ready to draw in a pane or on a page. Other methods are called for a whole panel, and must call the method TPanelOnAllPadsDo, which focuses on each pane in turn.

You use the drawing routines defined in UDraw, rather than the similar routines in QuickDraw, because the UDraw routines use long integers, allowing specification of larger coordinates than QuickDraw can handle. UDraw

converts your long integer coordinates to integer coordinates, scaling the coordinates and adjusting the grafPort so that the drawing appears correctly. Alternately, a set of conversion routines is provided so that you can convert long integers to integers yourself, which in certain circumstances is more efficient. UDraw can scale coordinates automatically to handle pads with difference resolutions (primarily used for low and high resolution printing) or for zoom factors.

You can call QuickDraw routines directly, as long as you observe the following restrictions:

- Your application's view does not print or zoom.
- The view must be small enough to use 16-bit integer coordinates.

If you want, you can use QuickDraw directly for drawing on the screen, and use UDraw when printing, but you will then need different code for each operation.

3.3 The UDraw Classes

UDraw provides the classes TArea and TPad. TArea defines a screen area with a border. TPad provides drawing and conversion routines used to display part of a view on the screen. Reference Sheets for these classes are provided in Chapter 7.

3.3.1 TArea

You generally do not use TArea or its methods directly. Instead, you use TWindow or TPanel, which are subclasses of TArea. TBand, a class that is rarely used in applications, is also a subclass of TArea. These classes are defined in UABC and are described in Chapter 4.

TArea defines an area on the screen with a border. This area is defined by two rectangles: **outerRect**, which describes the whole area including the border, and **innerRect**, which excludes the border. You can look at these variables to find the size of an area, but do not change their values. To change the size of an area, use the methods SetOuterRect and SetInnerRect.

Methods of TArea take care of functions such as setting the size of an area, drawing the frame, erasing the contents of the area, and focusing.

3.3.2 TPad

Tpad is a subclass of TArea. It provides methods to convert between view coordinates and GrafPort coordinates. These methods are called by the utility routines that do drawing in view coordinates (see Section 3.5 UDraw Routines). You can use these methods directly to increase efficiency by avoiding conversion operations for every routine called.

The grafPort is always focused on some instance of a descendant of TPad, whose handle is stored in the global variable thePad. In essence, a pad is the place where drawing actually occurs. Drawing is only done on a single pad at a time. Which pad it is, however, is taken care of by the ToolKit. Instances of TPane, a subclass of TPad, are usually used as thePad.

3.4 The UDraw Routines

The Udraw routines provide functions similar to QuickDraw, using long integer view coordinates in place of QuickDraw's integer grafPort coordinates. Udraw also adds some other functions for both grafPort and view coordinates.

These routines can be called to perform drawing operations in the view. All necessary conversions between view and grafPort coordinates are done by the view coordinate drawing routines. You can do conversions yourself to avoid unnecessary calculations, and then use the grafPort coordinate drawing routines in QuickDraw.

3.5 When To Use the Conversion Routines

The conversion routines can be used when your application uses grafPort coordinates for a series of operations. Since all on-screen drawing occurs in grafPort coordinates, every view coordinate UDraw routine first converts from view to grafPort coordinates, and then calls a grafPort coordinate routine in QuickDraw. You can, if you wish, call the conversion routines yourself, and then use the converted values in calling grafPort coordinate routines directly.

3.6 Variables Declared in UDraw

A number of variables are declared in the unit UDraw. This section describes the most useful of them.

3.6.1 Constants

The following variables are used as constants in UDraw, and are shown here for your use.

zeroPt:	Point	point with grafPort coordinates of (0,0)
zeroRect:	Rect;	grafPort coordinates for a rectangle with topLeft and botRight both (0,0)
hugeRect:	Rect;	rectangle with topLeft (0,0) and botRight (MAXINT/2,MAXINT/2)
zeroLPt:	LPoint;	point with view coordinates of (0,0)
zeroLRect:	LRect;	view coordinates for a rectangle with topLeft and botRight both (0,0)
hugeLRect:	LRect	rectangle with topLeft (0,0) and botRight (MAXLINT/2,MAXLINT/2)

3.6.2 Orthogonal Conversions

The following array is for performing orthogonal conversions by mapping v to h and vice versa.

```
orthogonal:  ARRAY [v..h] OF VHSelect
```

The value of orthogonal is an enumerated type: either v or h. orthogonal[v] has a value of h, and orthogonal[h] has a value of v.

3.6.3 Pen States for Highlighting

The following array is used to set the QuickDraw pen state for highlighting.

```
highPen:    ARRAY [TH-ighTransit] OF PenState;
```

The value of highPen[highTransit] is a pen state. highTransit has a value that depends on the state of highlighting at that time, and on what highlighting is desired, from the set (hNone, hOffToDim, hOffToOn, hDimToOn, hDimToOff, hOnToOff, hOnToDim). highPen[highTransit] has as its value the correct pen state setting to produce the required highlighting.

3.6.4 Patterns in View Coordinates

Pen patterns are central to drawing on the Lisa display. QuickDraw defines a number of standard patterns. The following provide the same patterns in view coordinates.

```
lPatWhite:   LPattern    Maps to QuickDraw pattern white.
lPatBlack:   LPattern;   Maps to QuickDraw pattern black.
lPatGray:    LPattern    Maps to QuickDraw pattern gray.
lPatLtGray:  LPattern;   Maps to QuickDraw pattern ltGray.
lPatDkGray:  LPattern;   Maps to QuickDraw pattern dkGray.
```

3.6.5 Graphics Areas

A region in QuickDraw is an part or group of parts of a graphical object. A region does not have to be contiguous, although in the ToolKit all of a single region must be in a single view.

```
focusRgn:   RgnHandle   The part of the pad that needs to be updated.
thePad:     TPad;       The pad currently focused on.
```

3.6.6 Conversion Pad (noPad)

The conversion pad is provided so that you can call cetain methods of TPad when you need to, without needing to create a new pad object, and without changing any values in thePad.

```
noPad:      TPad;       Dummy pad that can be used to convert
                        between short and long integer coordinates
                        without arithmetic on the coordinates. For
                        example:
                            noPad.RectToLRect(rect1, lRect1);
                        sets lRect1.top to rect1.top. It performs
                        similar operations on the other three
                        coordinates.
```

3.7 The UDraw Utility Routines

This section describes the utility procedures and functions implemented in UDraw. These routines are not methods; methods of the classes defined in UDraw are described in the reference sheets for the classes: TArea, TBranchArea, and TPad.

Since these are global procedures and functions, you can use them at any time, from any part of your program.

3.7.1 Arithmetic and Utility Operations

UDraw defines a number of arithmetic and utility procedures and functions that may be useful in your program. Most of these correspond to standard Pascal language functions. UDraw reimplements those to use long integers. A few others are additions you may find useful.

```
PROCEDURE BoolToStr (bool: BOOLEAN; str: TPstring);
```

BoolToStr returns a string (str) containing TRUE if bool is true, FALSE if it is false.

```
FUNCTION  IsSmallPt (srcPt: LPoint): BOOLEAN;
```

IsSmallPt returns TRUE if both coordinates of the point can be expressed in grafPort (INTEGER) coordinates.

```
FUNCTION  IsSmallRect (srcRect: LRect): BOOLEAN;
```

IsSmallRect returns TRUE if the coordinates of the rectangle can be expressed in grafPort (INTEGER) coordinates.

```
FUNCTION  LIntDivInt (i: LONGINT; j: INTEGER): LONGINT;

FUNCTION  LIntDivLInt (i, j: LONGINT): LONGINT;
```

These functions return the result of a long integer, i, divided by j. LIntDivInt takes an integer as a divisor; LIntDivLInt takes a long integer as a divisor. This acts like the Pascal DIV operator. LIntDivInt(i,j) is identical to i DIV j in results and similar in performance. However, InitDivLInt(i,j) is much faster than i DIV j (when i and j are LONGINT), but it may not produce the correct result in all cases.

```
FUNCTION  LIntMulInt (i: LONGINT; j: INTEGER): LONGINT;
```

LIntMulInt returns the result of i multiplied by j. It is faster than i*j when i or j is a LONGINT.

```
FUNCTION  LIntOvrInt (i: LONGINT; j: INTEGER): LONGINT;
```

Uses the formula (i+j/2)/j to round i/j to the nearest LONGINT.

```
FUNCTION  Max (i, j: LONGINT): LONGINT;
```

Max returns the larger of two numbers.

```
FUNCTION  Min (i, j: LONGINT): LONGINT;
```

Min returns the smaller of two numbers.

```
PROCEDURE PopFocus;
```

PopFocus restores the last focus pushed onto the focus stack with PushFocus. The current focus is lost.

```
PROCEDURE PushFocus;
```

PushFocus stores the current grafPort focus on the focus stack. Use PopFocus to restore it.

```
PROCEDURE Reduce (VAR numerator, denominator: INTEGER);
```

Reduce reduces the fraction numerator/denominator to its lowest whole number terms. Note that the values of numerator and denominator may change.

```
PROCEDURE ResizeFeedback (mousePt: Point; minPt, maxPt: Point; outerRect:
Rect; tabHeight, sbWidth, sbHeight: INTEGER; VAR newPt: Point);
```

This procedure is used to draw the temporary lines displayed when the user is resizing a panel, pane, or window. It is called automatically in those cases but you can call it yourself to implement similar feedback.

3.7.2 Graph Functions

Most functions and procedures defined in this section have two versions: one for grafPort coordinates and one for view coordinates. Each version operates identically, except that operands and results in the view coordinate routines are LONGINTs, and operands and results for the grafPort coordinate routines are INTs. In every case, the first routine given is the grafPort coordinate routine. You can tell the routines apart by their names: the names for grafPort coordinate routines usually contain Pt (for point) or Rect (for rectangle), while the names for view coordinate routines usually contain LPt (for long point) or LRect (for long rectangle).

3.7.2.1 Arithmetic on Points

```
PROCEDURE PtPlusPt (operand1, operand2: Point; VAR result: Point);

PROCEDURE LPtPlusLPt (operand1, operand2: LPoint; VAR result: LPoint);

FUNCTION FPtPlusPt (operand1, operand2: Point): LONGINT;
```

The two procedures add the coordinates of operand1 to the coordinates of operand2, and return the result in result. The function adds the two points, and gives the result as the return value.

Because functions cannot return a Point, the result is returned as a LONGINT, which can be coerced into a Point. For example, where pt is of type POINT, you can use pt:= Point(FPtPlusPt(pt1, pt2)); to convert the result to a POINT.

In all cases, operand1 and operand2 are unchanged.

```
PROCEDURE PtMinusPt (operand1, operand2: Point; VAR result: Point);

PROCEDURE LPtMinusLPt (operand1, operand2: LPoint; VAR result: LPoint);

FUNCTION FPtMinusPt (operand1, operand2: Point): LONGINT;
```

The two procedures subtract the coordinates of operand2 from the coordinates of operand1, and return the result in result. The function performs the same operation, but gives the result as the return value.

Because functions cannot return a Point, the result is returned as a LONGINT, which can be coerced into a Point. See FPtPlusPt for an example of coercion.

In all cases, operand1 and operand2 are unchanged.

PROCEDURE PtMulInt (operand1: Point; operand2: INTEGER; VAR result: Point);

PROCEDURE LPtMulInt (operand1: LPoint; operand2: INTEGER; VAR result: LPoint);

FUNCTION FPtMulInt (operand1: Point; operand2: INTEGER): LONGINT;

The two procedures multiply both coordinates of operand1 by operand2 and return the result in result. The function multiplies the two operands, and gives the result as the return value.

Because functions cannot return a Point, the result is returned as a LONGINT, which can be coerced into a Point. See FPtPlusPt for an example of coercion.

In all cases, operand1 and operand2 are unchanged.

PROCEDURE PtDivInt (operand1: Point; operand2: INTEGER; VAR result: Point);

PROCEDURE LPtDivInt (operand1: LPoint; operand2: INTEGER; VAR result: LPoint);

FUNCTION FPtDivInt (operand1: Point; operand2: INTEGER): LONGINT;

The two procedures divide both coordinates of operand1 by operand2 and return the dividend in result. The function divides the two operands, and gives the dividend as the return value.

Because functions cannot return a Point, the result is returned as a LONGINT, which can be coerced into a Point. See FPtPlusPt for an example of coercion.

In all cases, operand1 and operand2 are unchanged.

FUNCTION EqualPt (operand1, operand2: Point): BOOLEAN;

FUNCTION EqualLPt (operand1, operand2: LPoint): BOOLEAN;

These procedures compare the two points and return TRUE if they are equal or FALSE if not.

FUNCTION FPtMaxPt (operand1, operand2: Point): LONGINT;

This function returns a point with coordinates equal to the largest of the two corresponding coordinates in the operands. The maximum of each coordinate is taken separately; therefore the return value may not be either of the two points given as operands. It may be a combination of the two.

FUNCTION FPtMinPt (operand1, operand2: Point): LONGINT;

This function returns a point with coordinates equal to the smallest of the two corresponding coordinates in the operands. The minimum of each coordinate is taken separately; therefore the return value may not be either of the two points given as operands. It may be a combination of the two.

FUNCTION FDiagRect (rectangle: Rect): LONGINT;

This function returns a point designating the lower right corner of rectangle as if the upper right corner is at (0,0).

Because functions cannot return a Point, the result is returned as a LONGINT, which can be coerced into a Point. See FPtPlusPt for an example of coercion.

PROCEDURE PointToStr (pt: Point; str: TPstring);

PROCEDURE LPointToStr (pt: LPoint; str: TPstring);

These procedures convert pt to the string str.

3.7.2.2 Arithmetic on Rectangles

FUNCTION EmptyRect (r: Rect): BOOLEAN;

FUNCTION EmptyLRect (r: LRect): BOOLEAN;

These procedures return TRUE if the given rectangle is an empty rectangle or FALSE if not. A rectangle is considered empty if the bottom coordinate is equal to or less than the top or the right coordinate is equal to or less than the left.

FUNCTION EqualRect (rectA, rectB: Rect): BOOLEAN;

FUNCTION EqualLRect (rectA, rectB: LRect): BOOLEAN;

These procedures compare the two rectangles and return TRUE if they are equal or FALSE if not. The two rectangles must have identical boundary coordinates to be considered equal.

PROCEDURE RectMinusRect (operand1, operand2: Rect; VAR result: Rect);

PROCEDURE LRectMinusLRect (operand1, operand2: LRect; VAR result: LRect);

These procedures subtract the operand2 rectangle from the operand1 rectangle, returning the result in result.

PROCEDURE RectPlusRect (operand1, operand2: Rect; VAR result: Rect);

PROCEDURE LRectPlusLRect (operand1, operand2: LRect; VAR result: LRect);

These procedures add two rectangles and return the sum in result. Rectangles are added by adding their corresponding top left and bottom right coordinates.

3.7.2.3 Operations on Rectangles

PROCEDURE AlignRect (VAR dstRect: Rect; srcRect: Rect; vhs: VHSelect);

PROCEDURE AlignLRect (VAR destLRect: LRect; srcLRect: LRect; vhs: VHSelect);

These procedures change dstRect to match srcRect in the vhs direction.

FUNCTION LengthRect (r: Rect; vhs: VHSelect): INTEGER;

FUNCTION LengthLRect (r: LRect; vhs: VHSelect): LONGINT;

These procedures return the length of the rectangle side in the vhs direction.

FUNCTION RectHasPt (destRect: Rect; pt: Point): BOOLEAN;

FUNCTION LRectHasLPt (destLRect: LRect; pt: LPoint): BOOLEAN;

These procedures determine if the point is inside the rectangle and return TRUE if so or FALSE if not. These procedures check if pt is greater than or equal to the top left point and less than or equal to the bottom right point of the destination rectangle. (QuickDraw's PtInRect procedure returns FALSE if the topleft point is equal to the bottom right point.)

PROCEDURE RectHavePt (dstRect: Rect; VAR pt: Point);

PROCEDURE LRectHaveLPt (destLRect: LRect; VAR pt: LPoint);

These procedures force the point to be inside the rectangle if it is not, by changing the point so that pt is greater than or equal to the top left coordinate of the rectangle, and less than or equal to the bottom right coordinate. (topLeft <= pt <= botRight)

FUNCTION RectsNest (outer, inner: Rect): BOOLEAN;

FUNCTION LRectsNest (outer, inner: LRect): BOOLEAN;

These procedures determine if the inner rectangle is inside (or touches) the outer rectangle and return TRUE if so, FALSE if not.

PROCEDURE RectifyRect (VAR dstRect: Rect);

PROCEDURE RectifyLRect (VAR destLRect: LRect);

These procedures exchange opposing coordinates of a rectangle until the top left coordinate is less than or equal to the bottom right coordinate.

FUNCTION RectIsVisible (rectInPort: Rect): BOOLEAN;

FUNCTION LRectIsVisible (srcLRect: LRect): BOOLEAN;

These procedures return TRUE if the rectangle intersects the current update region. The grafPort must be focused on the view containing the rectangle. You can call LRectIsVisible to test whether a part of your document requires drawing. You can speed up your application by not drawing parts that cannot be seen by the user.

PROCEDURE RectToStr (r: Rect; str: TPstring);

PROCEDURE LRectToStr (r: LRect; str: TPstring);

These procedures convert the rectangle r to the string str.

3.7.3 Calculations with Rectangles and Points - View Coordinates

PROCEDURE SetLPt (VAR destPt: LPoint; itsH, itsV: LONGINT);

SetLPt assigns two LONGINT coordinates to a variable of type LPoint.

PROCEDURE SetLRect (VAR dstRect: LRect; itsLeft, itsTop, itsRight, itsBottom: LONGINT);

SetLRect assigns the four boundary coordinates to the rectangle dstRect. The result is a rectangle with coordinates (itsLeft,itsTop,itsRight,itsBottom).

This procedure is supplied as a utility to help you shorten your program text. If you want a more readable text at the expense of length, you can assign LONGINTs (or Lpoints) directly into the rectangle's fields. There is no significant code size or execution speed advantage to either method.

PROCEDURE OffsetLRect (VAR dstRect: LRect; dh, dv: LONGINT);

OffsetLRect moves the rectangle by adding dh to each horizontal coordinate and dv to each vertical coordinate. If dh and dv are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The rectangle retains its shape and size; it is moved on the coordinate plane. This does not affect the screen unless you subsequently call a routine to draw the rectangle.

PROCEDURE InsetLRect (VAR dstRect: LRect; dh, dv: LONGINT);

InsetLRect shrinks or expands the rectangle. The left and right sides are moved in by the amount specified by dh; the top and bottom are moved toward the center by the amount specified by dv. If dh or dv is negative, the appropriate pair of sides is moved outward instead of inward. The effect is to alter the size by 2*dh horizontally and 2*dv vertically, with the rectangle remaining centered in the same place on the coordinate plane.

If the resulting width or height becomes less than 1, the rectangle is set to the empty rectangle (0,0,0,0).

FUNCTION SectLRect (srcRectA, srcRectB: LRect; VAR dstRect: LRect): BOOLEAN;

SectLRect calculates the rectangle that is the intersection of the two input rectangles, and returns TRUE if they indeed intersect or FALSE if they do not. Rectangles that "touch" at a line or a point are not considered intersecting, because their intersection rectangle (really, in this case, an intersection line or point) does not enclose any bits on the bitmap.

If the rectangles do not intersect, the destination rectangle is set to (0,0,0,0). SectRect works correctly even if one of the source rectangles is also the destination.

PROCEDURE UnionLRect (srcRectA, srcRectB: LRect; VAR dstRect: LRect);

UnionLRect calculates the smallest rectangle that encloses both input rectangles. It works correctly even if one of the source rectangles is also the destination.

FUNCTION LPtInLRect (pt: LPoint; r: LRect): BOOLEAN;

PtInLRect determines whether the pixel below and to the right of the given coordinate point is enclosed in the specified rectangle, and returns TRUE if so or FALSE if not.

3.7.4 Drawing Operations for View Coordinates

The following routines are similar to QuickDraw calls, but use long integers (view coordinates) so that they can perform graphics in the view. See the Lisa Pascal appendix on QuickDraw for the corresponding routines for grafPort coordinates. In general, the grafPort routines are identical to these, except that the routine name does not have the L that indicates these are for long integers.

You must focus the grafPort on your view before calling these routines.

3.7.4.1 Line-Drawing Routines

PROCEDURE MoveToL (h, v: LONGINT);

MoveToL moves the pen to location (h,v) in view coordinates. No drawing is performed.

PROCEDURE MoveL (dh, dv: LONGINT);

MoveL moves the pen a distance of dh horizontally and dv vertically from its current location; it calls MoveToL(h+dh,v+dv), where (h,v) is the current location in view coordinates. The positive directions are to the right and down. No drawing is performed.

PROCEDURE LineToL (h, v: LONGINT);

LineToL draws a line from the current pen location to the location specified (in view coordinates) by h and v. The new pen location is (h,v) after the line is drawn.

PROCEDURE LineL (dh, dv: LONGINT);

LineL draws a line to the location that is a distance of dh horizontally and dv vertically from the current pen location; it calls LineToL(h+dh,v+dv), where (h,v) is the current location. The positive directions are to the right and down. The pen location becomes the coordinates of the end of the line after the line is drawn.

3.7.A.2 Graphic Operations on Rectangles

PROCEDURE FrameLRect (r: LRect);

FrameLRect draws an outline just inside the specified rectangle, using the current views pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the the current pen pattern (**pnPat**), according to the pattern transfer mode (**pnMode**). The pen location is not changed by this procedure.

PROCEDURE PaintLRect (r: LRect);

PaintLRect paints the specified rectangle with the current pen pattern and mode. The rectangle on the bitmap is filled with the **pnPat**, according to the pattern transfer mode specified by **pnMode**. The pen location is not changed by this procedure.

PROCEDURE EraseLRect (r: LRect);

EraseLRect paints the specified rectangle with the current GrafPort's background pattern **bkPat** (in **patCopy** mode). The GrafPort's **pnPat** and **pnMode** are ignored; the pen location is not changed.

PROCEDURE InvrtLRect (r: LRect);

InvrtLRect inverts the pixels enclosed by the specified rectangle: every white pixel becomes black and every black pixel becomes white. The GrafPort's **pnPat**, **pnMode**, and **bkPat** are all ignored; the pen location is not changed.

PROCEDURE FillLRect (r: LRect; lPat: LPattern);

FillLRect fills the specified rectangle with the given pattern (in **patCopy** mode). The GrafPort's **pnPat**, **pnMode**, and **bkPat** are all ignored; the pen location is not changed.

3.7.A.3 Graphic Operations on Ovals

PROCEDURE FrameLOval (r: LRect);

FrameLOval draws an outline just inside the oval that fits inside the specified rectangle. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the **pnPat**, according to the pattern transfer mode specified by **pnMode**. The pen location is not changed by this procedure.

PROCEDURE PaintLOval (r: LRect);

PaintLOval paints an oval just inside the specified rectangle. The oval is filled with the pen pattern **pnPat**, according to the pattern transfer mode specified by **pnMode**. The pen location is not changed by this procedure.

PROCEDURE EraseLOval (r: LRect);

EraseLOval paints an oval just inside the specified rectangle with the current background pattern **bkPat** (in **patCopy** mode). The pen pattern **pnPat** and the pattern transfer mode **pnMode** are ignored; the pen location is not changed.

PROCEDURE InvrtLOval (r: LRect);

InvrtLOval inverts the pixels enclosed by an oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. The pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

PROCEDURE FillLOval (r: LRect; pat: LPattern);

FillLOval fills an oval just inside the specified rectangle with the given pattern (in patCopy mode). The GrafPort's pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

3.7.4.4 Graphic Operations on Rounded-Corner Rectangles

PROCEDURE FrameLRRect (r: LRect; ovalWidth, ovalHeight: INTEGER);

FrameLRRect draws an outline just inside the specified rounded-corner rectangle, using the current pen pattern, mode, and size. OvalWidth and ovalHeight specify the diameters of curvature for the corners. The outline is as wide as the pen width and as tall as the pen height. The pen location is not changed by this procedure.

PROCEDURE PaintLRRect (r: LRect; ovalWidth, ovalHeight: INTEGER);

PaintLRRect paints the specified rounded-corner rectangle with the current GrafPort's pen pattern and mode. OvalWidth and ovalHeight specify the diameters of curvature for the corners. The rounded-corner rectangle on the bitmap is filled with the pnPat, according to the pattern transfer mode specified by pnMode. The pen location is not changed by this procedure.

PROCEDURE EraseLRRect (r: LRect; ovalWidth, ovalHeight: INTEGER);

EraseLRRect paints the specified rounded-corner rectangle with the current GrafPort's background pattern bkPat (in patCopy mode). OvalWidth and ovalHeight specify the diameters of curvature for the corners. The GrafPort's pnPat and pnMode are ignored; the pen location is not changed.

PROCEDURE InvrtLRRect (r: LRect; ovalWidth, ovalHeight: INTEGER);

InvrtLRRect inverts the pixels enclosed by the specified rounded-corner rectangle: every white pixel becomes black and every black pixel becomes white. OvalWidth and ovalHeight specify the diameters of curvature for the corners. The pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

PROCEDURE FillLRRect (r: LRect; ovalWidth, ovalHeight: INTEGER; pat: LPattern);

FillLRRect fills the specified rounded-corner rectangle with the given pattern (in patCopy mode). OvalWidth and ovalHeight specify the diameters of curvature for the corners. The pnPat, pnMode, and bkPat are all ignored; the pen location is not changed.

3.7.4.5 Graphic Operations on Arcs and Wedges

These procedures perform graphic operations on arcs and wedge-shaped sections of ovals.

PROCEDURE FrameLArc (r: LRect; startAngle, arcAngle: INTEGER);

FrameLArc draws an arc of the oval that fits inside the specified rectangle, using the current GrafPort's pen pattern, mode, and size. StartAngle indicates where the arc begins and is treated mod 360. ArcAngle defines the extent of the arc. The angles are given in positive or negative degrees; a positive angle goes clockwise, while a negative angle goes counterclockwise. Zero degrees is at 12 o'clock high, 90° (or -270°) is at 3 o'clock, 180° (or -180°) is at 6 o'clock, and 270° (or -90°) is at 9 o'clock. Other angles are measured relative to the enclosing rectangle: a line from the center of the rectangle through its top right corner is at 45 degrees, even if the rectangle is not square; a line through the bottom right corner is at 135 degrees, and so on .

The arc is as wide as the pen width and as tall as the pen height. It is drawn with the prnPat, according to the pattern transfer mode specified by prnMode. The pen location is not changed by this procedure.

PROCEDURE PaintLArc (r: LRect; startAngle, arcAngle: INTEGER);

PaintLArc paints a wedge of the oval just inside the specified rectangle with the current GrafPort's pen pattern and mode. StartAngle and arcAngle define the arc of the wedge as in FrameArc. The wedge on the bitmap is filled with the prnPat, according to the pattern transfer mode specified by prnMode. The pen location is not changed by this procedure.

PROCEDURE EraseLArc (r: LRect; startAngle, arcAngle: INTEGER);

EraseLArc paints a wedge of the oval just inside the specified rectangle with the current GrafPort's background pattern bkPat (in patCopy mode). StartAngle and arcAngle define the arc of the wedge as in FrameArc. The GrafPort's prnPat and prnMode are ignored; the pen location is not changed.

PROCEDURE InvrtLArc (r: LRect; startAngle, arcAngle: INTEGER);

InvrtLArc inverts the pixels enclosed by a wedge of the oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. StartAngle and arcAngle define the arc of the wedge as in FrameArc. The GrafPort's prnPat, prnMode, and bkPat are all ignored; the pen location is not changed.

PROCEDURE FillLArc (r: LRect; startAngle, arcAngle: INTEGER; lPat: LPattern);

FillLArc fills a wedge of the oval just inside the specified rectangle with the given pattern (in patCopy mode). StartAngle and arcAngle define the arc of the wedge as in FrameArc. The GrafPort's prnPat, prnMode, and bkPat are all ignored; the pen location is not changed.

Chapter 4
The Application Base Classes

4.1 What the ABC's Are 4-2
4.2 Which ABC's Contain What Functions 4-2
4.2.1 TView 4-2
4.2.2 TWindow, TPanel, and TPane 4-3
4.2.3 TProcess 4-4
4.2.4 TDocManager 4-4
4.2.5 TSelection 4-5
4.2.6 TCommand 4-5
4.3 Global Variables 4-7
4.4 Global Procedures and Functions 4-8

The Application Base Classes

4.1 What the ABC's Are

The Application Base Classes (ABC's) consist of predefined Clascal class-type definitions.

The methods defined in UABC perform most of the automatic functions necessary for a Lisa application. They control screen display, scrolling, window size and resizing, menu display, command input, keyboard input, and mouse handling.

Aside from defining the objects necessary for the standard Lisa functions needed by applications, the ABC's structure the way an application is written. When a ToolKit program is running, the ToolKit calls certain application methods. You must write your application so that the correct methods are in the right places. This chapter contains descriptions of the important ABC's and also tells you what you are required to do, and what you can do, with each ABC.

Several of the most important classes in UABC are always subclassed for use in application programs. When you create a subclass of one of these classes, you may add methods that add application-specific functions, and you may add data fields for the program's use.

The following ABC classes are always or, in the case of TCommand, nearly always subclassed for use in application programs.

- TView
- TWindow
- TProcess
- TDocManager
- TSelection
- TCommand

Each of these ABCs is fully described in its reference sheet in Part II of this manual.

4.2 Which ABC's Contain What Functions

This section describes the most important functions handled by the ABC's that every ToolKit application uses.

4.2.1 TView

Class TView is always subclassed at least once for use by your application. You can have multiple view objects in a single program. The views you create produce a complete visual representation of the entire set of objects that make up the visible data of the program. Each view object may be thought of as producing a picture that is something like a sheet of paper, which often is larger than the Lisa display, but which has definite boundaries. In some of

the original Lisa applications, such as LisaDraw, and in all ToolKit applications, you can scroll to a point where the white background stops, and a gray background is visible. That border is the border of the picture created by the view.

What is visible on the display is normally only a part (or possibly several parts) of the view's entire picture. Which part of the view's picture is displayed is determined by the user, through the use of the scroll bars and the window and panel size control boxes. The application does not need to be concerned with what is actually seen on the screen, since the ToolKit handles the display once the view is defined. The application can, though, ask the ToolKit what parts of the view's picture are visible on the display, and use that information to speed display processing.

Because Lisa applications are visually oriented, the view often contains the most important references to the data. In addition, all of the drawing is always ordered through the view object.

4.2.2 TWindow, TPanel, and TPane

When writing an application, you always subclass TWindow, and then define the window's BlankStationery method to initialize a new document.

The window object and panel objects control the automatic screen functions. The portion of the display that they create is essentially the entire border of the window that you see on the screen. The window object separates off a section of the display screen surrounding the entire visible portion of an application, including the title bar and the scroll bars. The window object creates and controls the title bar and the window size control box.

The panel object creates the scroll bars, as well as the panel size control box, if there is one. Each panel object has two scroll bars, one on the right and one on the bottom. The scroll bars can be suppressed, unless the panel is at the right or bottom of a window with a size control box, or has a size control box itself.

A window can contain a number of panels.

Different panel objects usually display portions of different views. The views may be different representations of the same set of objects. For example, LisaGraph has two panels, one that displays a chart of the values and one that displays a graphical representation of the values. Panels in the same window may also display views of different objects; the Calculator, when used with its transaction tape, contains two panels. The calculator panel shows the calculation going on at that moment; the tape panel shows a record of all the calculations performed.

Creating a panel object also creates one pane object.

Each time the user splits a panel, one or more pane objects are added and one of the scroll bars is divided in two. The panes can each display a

different portion of the view looked on by that panel. The number of panes in a window is equal to the number of divisions of the window.

The pane objects control the visible contents of the panel objects. It is the panes that determine what parts of the view objects are actually displayed, although the user may change what can be seen in a pane by using the panel's scroll bars or the window's size control box. Your application, however, never deals directly with panes. The current pane is assigned to the global variable thePad before your application is told to draw. You call routines defined in UDraw, and the ToolKit takes zooming factors into account, converts your long integers into simple integers, and calls QuickDraw to draw on the screen.

4.2.3 TProcess

There is always one and only one instance of a descendant of TProcess in any ToolKit application.

The methods of TProcess are the methods of an application that do not apply to any one document. For example, the routines that initialize libraries and communicate with the Desktop Manager are methods of TProcess.

The most important function of the process object is in controlling the main event loop. ToolKit programs operate primarily in response to user events, such as menu commands, mouse clicks, and keyboard entry. Most of the time in a ToolKit program is usually spent waiting for some event. When an event is received, appropriate methods from UABC, UObject, building blocks, or the application's units are called. When the methods complete, the event loop continues, unless the event caused a serious error (usually one that results in a "technical difficulties" display), or ended the session.

The process object also handles errors, warnings, and termination. It is the process object that displays messages from the phrase file.

The process object for an application can still exist even when there is no view object, window object, docManager object, or selection object for that application. In this case, the process is waiting for the user to activate a document. (You may notice that every Lisa application takes longer to start up the first time it is used after the Lisa office system is started. This is because the process for an application is created the first time a document that uses that application is opened after start up. When a document is put away, every object except the process is deallocatted. The process for that application is kept until the Lisa turns off, waiting for the user to open a document.)

4.2.4 TDocManager

There is one and only one instance of a descendant of TDocManager for every document icon or window managed by a process. It is created when the user first opens or displays the document.

The docManager object controls opening and closing the files used to store the state of the document, and also manages memory used for that document.

Aside from creating a subclass and defining a CREATE method and a NewWindow method, you usually do not need to deal with the docManager.

4.2.5 TSelection

Many actions or commands in Lisa applications act on some particular object within the set of objects used by the program. The object (or group of objects) acted on in such a case is called the selected object.

The selection is the instance of TSelection or instance of a descendant of TSelection that receives commands and has access to any selected objects.

There is always one selection for every panel object. One panel in each window is designated the selectPanel, which means that it contains the active selection object. When a command is given, that command is sent by the ToolKit to the selection in the active window's selectPanel. Normally, the descendant of TSelection that you create declares a field to reference the selected object, or a list of the selected objects. Your command routines use that field to act on the selected object.

Note that a selection is not the same as a selected object. The selection is an instance of TSelection or a descendant of TSelection, at least one of which always exists for each panel object. Selected objects are objects displayed in the panel containing the selection. The selection may, however, be null, meaning that there is no selected object. TSelection defines a field that indicates the kind of selection. A null selection is designated nothingKind.

If an object is selected, it normally is highlighted in the view. Every TSelection descendant class should implement a Highlight method to visibly mark every selected object, and also to remove the mark when the object is no longer selected.

Because so many menu and key commands involve the selection object, commands are always handled through the selection. The selection may, however, delegate responsibility for a command to the window, the view, or to some other object.

4.2.6 TCommand

When the user selects a command from a menu, or gives an Apple-key equivalent, the command is translated into a command number for use by the ToolKit and the application. Some commands change the contents of the document, and some just change the display, such as when a ruler is displayed. When the command changes the document, unless the user is asking that a command be undone (that is, to have the effects of the last command reversed), the command number is used to create an instance of a descendant of TCommand or, occasionally, an instance of TCommand itself. The new command object is then told to Perform the command.

The command object is kept until another command is given. If that command is an undo command, the command object is told to undo itself. Some commands cannot be undone, in which case the ToolKit gives the user a message to that effect, but most can be undone.

If another undo command is given, the command object is told to redo itself. This can continue indefinitely, with undo and redo alternating.

In order to make undoing a command easier, commands are often implemented so that they do not change any data, but only change the view so that it appears as if the data has been changed. This is called applying a *filter* to the data. After it is too late to undo the command, a separate step, performed by the method **Commit**, actually carries out the operation on the data, and makes the effect of the command permanent.

When a command is implemented so that it uses a filter, the way the view is displayed is changed. The original set of objects that make up the view is not altered. When the objects are displayed, however, a *virtual* image of the objects is shown in place of the *actual* image. Figure 4-1 illustrates this. A command to recolor all the objects in the view was given by the user. When the view is displayed, the objects are first filtered so that they appear in the virtual view with the correct color. The actual view has not been changed.

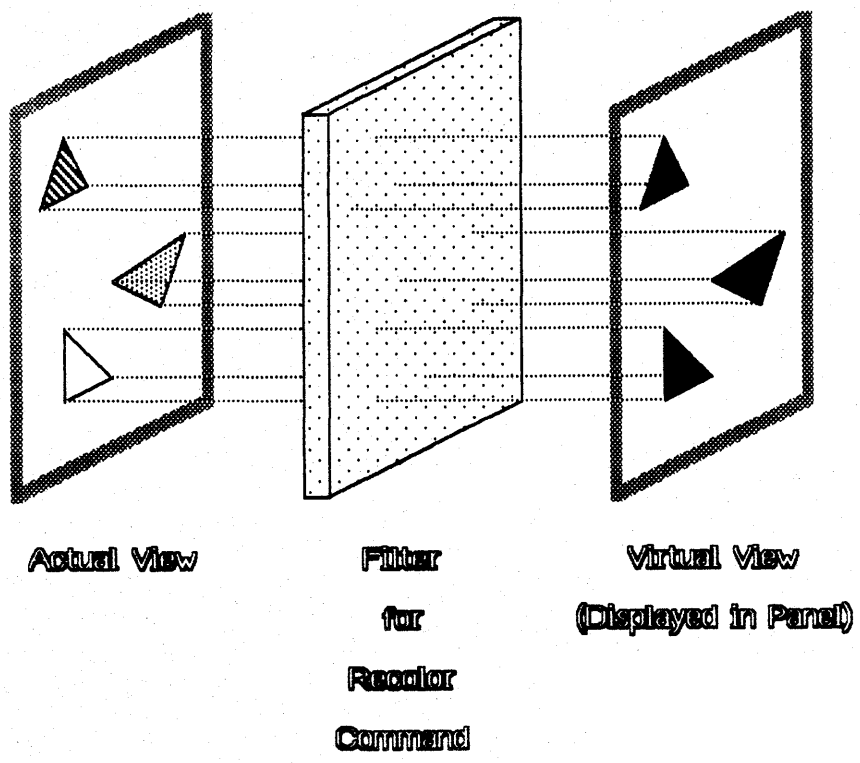


Figure 4-1
Filtering

There is a method of TView called EachVirtualPart. EachVirtualPart is one of a number of methods in the ToolKit that take a procedure as an argument. These *iterating* methods apply the input procedure to a whole set of objects, one at a time. Window. and View.EachVirtualPart do not themselves apply the input procedure; they each call other iterating methods, which then apply the procedure.

TWindow. or TView.EachVirtualPart tests the most recent command object to see if it has been done. (Remember that command objects can in general be done or undone. Undoing a command object removes its effects.) If the command has been undone, EachActualPart is called. EachActualPart, which exists in both TView and TWindow, applies the input procedure to each object in the document's data set.

If the command has been done, TWindow. or TView.EachVirtualPart calls command.EachVirtualPart for the most recent command object. command.EachVirtualPart applies the method command.FilterAndDo to each object. command.FilterAndDo alters the object in the same way the command would, calls the procedure that was passed in as an argument to EachVirtualPart, and then changes the object back to its original state.

For example, if the last command was to recolor the object, and the procedure given as an argument to EachVirtualPart is a Draw method, FilterAndDo first checks if the object was selected. (When commands like this one are implemented, you must have some way of telling which objects were selected when the last command object was created.) If the object was not selected, it is simply told to Draw itself. If the object was selected, the color is changed to what it should be after the command, and then the object is told to Draw itself. The object's color is then changed back.

When a command that is not an undo command is given, the previous command object is told to Commit itself (unless the command was undone and not redone), which makes the new actual view match the old virtual view. In the example given above, the object's color is changed as the old command required. In Figure 4-1, the actual view would be changed to match the virtual view shown. Note that it is not necessary to invalidate any of the display at this point, because it already reflects the changed state of the document.

The old command object is then deallocated. A new command object is created for the new command.

4.3 Global Variables

UABC defines a number of global variables. These variables hold important values for the process. The global variables indicate the active window object (currentWindow), identify the tool and the process, and hold a wide range of other values. These variables are documented in Chapter 6.

4.4 Global Procedures and Functions

UABC defines a number of global procedures and functions. These provide services and utility functions needed throughout the ToolKit, and by application programs. The routines that might be used by applications are documented in Chapter 6.

Chapter 5
The ToolKit Debugger

5.1 The Debugging Facilites 5-2

5.2 The Pulldown Debug Menu 5-3

5.2.1 Report Every Event 5-4
5.2.2 Count Heaps after Commands 5-5
5.2.3 Check List Indices 5-5
5.2.4 Dump Process Globals 5-6
5.2.5 Dump Active Document Prelude 5-6
5.2.6 Enable Experimental Features 5-6
5.2.7 Report Garbage in Document Heap 5-7
5.2.8 Free Garbage in Document Heap 5-7
5.2.9 Edit Dialog 5-7
5.2.10 Stop Editing Dialog 5-7

5.3 The Interactive Debugger 5-8

5.3.1 Breakpoint 5-10
5.3.2 ClearBreakpoints 5-12
5.3.3 DebugStatus 5-12
5.3.4 EnterLisaBug 5-13
5.3.5 FrameDump 5-13
5.3.6 Go 5-13
5.3.7 HeapDump 5-13
5.3.8 InspectObject 5-13
5.3.9 LevelsToWatch 5-14
5.3.10 MemoryDump 5-14
5.3.11 OutputTo 5-14
5.3.12 Prompt 5-15
5.3.13 RefsToObject 5-15
5.3.14 StackCrawl 5-15
5.3.15 Tally & Time 5-15
5.3.16 Watch 5-16

The ToolKit Debugger

5.1 The Debugging Facilites

The Lisa Applications ToolKit provides two basic debugging facilities. One is a set of pull-down menu options that can be turned on while the program is running. Those options

- Produce information on the dynamic state of the program.
- Allow you to edit dialog boxes.
- Enable or disable experimental program features.
- Free objects left on the heap that have no remaining references.

In addition, there is a separate interactive debugging menu, displayed on the alternate screen. The interactive debugger allows you to stop the application and

- Examine the state of the process, document, objects, and object fields.
- Step through the program at any pace you desire.
- Dynamically trace the action of the program.
- Analyze your program's performance.

The ToolKit interactive debugging menu allows you to also use the basic Lisa debugging facilites provided by Lisabug. However, although Lisabug has some functions similar to those provided by the ToolKit interactive debugger, Lisabug operates on the assembler level, while the interactive debugger operates on the Clascal level. Lisabug is not discussed in this chapter. See the dubugger chapter of the Workshop manual for more information on Lisabug.

The ToolKit debugging facilites work on ToolKit programs that are running on the desktop. You can also use the interactive menu to debug Clascal programs running under the Workshop. Note that you cannot access the pull-down menu options from the Workshop.

Another point to remember is that when any program has a fatal error on the Lisa, you drop into Lisabug, not into the ToolKit interactive debugger. If you want to find fatal bugs using the ToolKit debugger, you must do so by using breakpoints or by stepping through the program to determine exactly where the crash occurs. Lisabug will, however, allow you to find out some information about the state of the program after the crash.

Some errors that are peculiar to ToolKit programs halt your program and bring up the ToolKit interactive debugger. In those cases, an explanatory message is given. You can try to analyze the bug by examining the stack, the heap, registers, and so on. You can also try to ontinue progrom execution, by giving the Go command, although the process or document may be damaged. It is generally better to halt your program and try to trace the

error with another process and document. You can halt the program by using the Enter Lisabug command to reach Lisabug, and then giving a Go 0 command. That command causes a bus error. Type Go in response to the bus error, and the process is terminated.

You can also use the ToolKit debugger to reach Lisabug, if you want access to the program on the assembler level. The ToolKit debugger makes certain that you enter Lisabug while in user code; you cannot debug system code with either Lisabug or the ToolKit debugger.

Whether you are using Lisabug or either of the ToolKit debuggers, the results of your actions show on the alternate screen. This display, maintained separately from the desktop, can be reached by holding down the Option key and pressing the Enter key in the numeric pad on the right side of the keyboard. To stop a ToolKit program and see the interactive debugging menu, press any key while the alternate screen is displayed. To return to the desktop, press Option/Enter again.

A fully debugged ToolKit program normally is linked with a version of the ToolKit that has no debugging facilites. In order to use the ToolKit debuggers, you must link with the debugging version of the ToolKit. The debugging version slows your application down by 20% to 40%.

Lisabug also is only available in development versions of the Lisa system.

5.2 The Pulldown Debug Menu

The ToolKit pulldown debug menu allows you to check that your application is running correctly in certain respects. It does not give you any information about fatal errors, computational errors, or the state of fields and objects. It does, however, let you enable range checking on list accesses. You can also enable checks on objects in the heap, to identify objects that have no references and should be freed. Additionally, functions in this menu print out information about the process as a whole, about the document, and about events as they occur.

In all cases, the information is printed on the alternate screen. You can display the alternate screen at any time by holding down an Option key and pressing the Enter key that appears on the numeric keypad on the right side of the keyboard. Press Option/Enter again to return to the main screen.

Here is a list of the commands in the pulldown menu, along with a brief description of each. More detail about each command is given in the subsections that follow. Four commands are flags that are turned on when you select them. They remain in effect until you select them again. When they are in effect they are checked off in the menu. The other commands print information on the alternate screen each time the command is chosen.

- Report Every Event prints brief information about every event. It slows your program slightly. Disable this function by choosing it a second time.

- **Count Heaps after Commands** prints the number of objects in each heap every time any command is given. This also invokes the Report Garbage in Document Heap menu command. It slows your program slightly. Disable this function by choosing it a second time.
- **Check List Indices** enables range checking on list indices. When an error is found, the application is halted and a message is printed on the alternate screen. It slows your program slightly. Disable this function by choosing it a second time.
- **Dump Process Globals** prints the values of various important global variables.
- **Dump Active Document Prelude** prints some information about the currently active document.
- **Enable Experimental Features** enables features of your application that you have defined to be provisional, and do not want to appear in demonstrations, or in intermediate versions of your application. Disable this function by choosing it a second time.
- **Report Garbage in Document Heap** prints information about objects in the document heap to which their are no references.
- **Free Garbage in Document Heap** frees all objects in the document heap to which there are no references.
- **Edit Dialog** allows you to edit a dialog box, if one is currently displayed. The changes you make are not saved at this time.
- **Stop Editing Dialog** stops dialog box editing.

5.2.1 Report Every Event

When this option is checked off, the debugger prints brief information about every event. Like all information printed by the ToolKit debugger, this printout appears on the alternate screen.

Any keyboard or mouse button use triggers a printout by this function. For example, when the application user presses the mouse button, that is considered an event. Releasing the mouse button is another event. Clicking the mouse button is two events: one when the button is pressed, and one when it is released. Pressing a key is also an event. Moving the mouse, however, is not an event. Clicking in a different window causes a deactivate event and an activate event instead of mouse events. Choosing a menu command is an event only in that it involves a mouse button press and release event.

The event type is printed out, along with the tool name, the process ID, the "folder" identification (that is, whether the event occured in a clipboard, dialog box, or window), and, if appropriate, the window title. Additional

information is printed for desktop events, including the kind of event, the password, and the document prefix.

When this option is in effect, it is checked off in the debug menu. To disable this function, choose the menu item a second time.

5.2.2 Count Heaps after Commands

When this option is checked off, the debugger prints the number of objects in each heap every time any command is given. Like all information printed out by the ToolKit debugger, this printout appears on the alternate screen.

Note that only commands trigger the printout; simple mouse button presses do not. Key presses are considered commands. In addition, many mouse actions (such as click and drag) are considered commands. All menu items are considered commands for the purposes of this function.

This function is useful for checking that your program is not leaving unreferenced objects on the heap. The number of objects on the heap should not grow unrestrictedly, unless objects are actually added to the program's data set. You can use the **Report Garbage in Document Heap** function to find out if there actually are unreferenced objects on the heap.

This function is also useful for generally checking the impact of commands on the heap.

When this option is in effect, it is checked off in the debug menu. To disable this function, choose the menu item a second time.

5.2.3 Check List Indices

When this option is checked off, the debugger performs a range check whenever a list is accessed. When an error is found, the application is halted and a message is printed on the alternate screen.

When this option is not checked off, the effect of an out-of-range list access is unpredictable. Such an access attempt could store data outside the object and damage other objects or heap structures.

Enabling range checking slows your application somewhat.

When this option is in effect, it is checked off in the debug menu. To disable this function, choose the menu item a second time.

5.2.4 Dump Process Globals

When you choose this option, the debugger prints the values of various important global variables on the alternate screen. The variables whose values are printed are:

activeWindowID	currentDocument	menuBar
allowAbort	currentWindow	myProcessID
boundClipboard	cursorShape	myTool
boundDocument	deferUpdate	process
clickState	docList	toolName
clipboard	genClipPic	toolPrefix
closedBySuspend	idleTime	toolVolume
closedDocument	inBackground	

The debugger only prints the first eight characters of each name.

These values are only printed once. To print the values again, choose the option again.

See Chapter 6, Section 6.3.2, UABC Global Variables for more information on these variables.

5.2.5 Dump Active Document Prelude

When you choose this option, the debugger prints some information about the currently active document. The values and types of the following variables are printed.

```
password     -- A key number used to identify documents.
version      -- The version sequence number of this application.
country      -- From the phrase file.
language     -- From the phrase file.
preludeSize  -- The size of the document prelude, in bytes.
docSize      -- The size of the document, in bytes.
numSegments  -- The number of data segments in use.
docDirectory -- The docDirectory object for this document.
```

These values are only printed once. To print the values again, choose the option again. Only the docSize and numSegments values should change.

5.2.6 Enable Experimental Features

When this option is checked off, features of your application that you have defined to be provisional are enabled. The ToolKit toggles the global boolean variable fExperimenting on the basis of this command. To implement experimental features, simply surround code dealing with those features with a test on fExperimenting. When you check off this item in the menu, that variable is TRUE, and your provisional code is executed. Because the debug

menus do not appear in non-debug versions of ToolKit program, your experimental code can never be executed in non-debug versions of your program.

When this option is in effect, it is checked off in the debug menu. To disable this function, choose the menu item a second time.

5.2.7 Report Garbage in Document Heap

When you choose this option, the debugger prints information about objects in the document heap to which their are no references. The information printed consists of the object's handle and class-type. To determine what command caused the problem, free the garbage by using the Free Garbage in Document Heap option, and try out various application commands. Invoke the Report Garbage in Document Heap option after each command.

5.2.8 Free Garbage in Document Heap

When you choose this option, the debugger frees all objects in the document heap to which there are no references. The objects freed are the same ones that would be indicated by Report Garbage in Document Heap.

This command only acts once. To free garbage later, choose the option again.

5.2.9 Edit Dialog

This function is only enabled when one of your application's dialog boxes is displayed and the current selection is in the dialog box. When you choose this command, the dialog box is re-displayed so that each separate piece of the dialog has a surrounding box and a title bar. You can use the mouse to grab any of the title bars, and move the piece around within the dialog box. A size icon for the whole dialog box is also displayed, and you can use that to grow or shrink the box.

You can change any of the text that appears in the dialog box.

You cannot add elements to the dialog box. All elements must be defined when you create the dialog box. See the documentation on dialog boxes (not covered in this manual) for more information.

When you are done editing the dialog box, choose Stop Editing Dialog from the debug menu. The changes you made to the dialog box are stored with that document.

5.2.10 Stop Editing Dialog

This function is only enabled when a dialog box is displayed, and you have chosen Edit Dialog. It stops dialog box editing. See subsection 5.2.10 Edit Dialog for more information.

5.3 The Interactive Debugger

The ToolKit interactive debugger allows you to stop your ToolKit program and access a menu of interactive debug functions. You can use the interactive debugger to find out

- What methods are called when.
- The values of variables, objects, and fields.
- The current sequence of routine calls, back to the main program.

In addition, you can set breakpoints anywhere in the program or step through the program at any pace.

You can reach the interactive debugger by keying Option/Enter to display the alternate screen, and then pressing the space key. (You can actually press any key, but a key other than the space key is taken as a command.) Your application is halted, and the interactive debugger menu and prompt are displayed. Figure 6-1 shows a sample of the debugger menu and prompt.

Alternately, you can insert an ABCBreak call in your program, which halts the program and starts the interactive debugger. You can also call EntDebugger. ABCBreak turns off the debugger flags from the menu, while EntDebugger does not.

Occasionally, an error in your program may cause the ToolKit to call ABCBreak. In that case, a message describing the type of error is displayed on the alternate screen along with the interactive debugger menu and prompt. ABCBreak automatically switches the display to the alternate screen.

The interactive debugger is also invoked when a breakpoint is reached. In that case, the alternate screen is automatically displayed.

Switching to the alternate screen does not affect the operation of a running application, unless you purposely stop the program by typing a key. The program does not receive any events that occur when the alternate screen is displayed; the single exception is when the mouse button is depressed when you switch to the alternate screen. In that case, mouse move and mouse release events are given to the running application. (If you halt the program while the mouse button is pressed, however, further mouse events are ignored.)

The debugger acts on the Clascal method, procedure, and function level. You can find out information about objects and methods. In particular, assuming you have used the BP and EP procedure calls to begin and end all your methods, you can find out what method is called when.

One important detail of using the BP and EP procedures concerns the level you assign to each of your methods. BP takes a level parameter. When you trace the action of your program, you can give the Watch command (the tracing command) a minimum level. Methods below that level are ignored by

the trace. Nothing in the ToolKit enforces level conventions. However, the following conventions are observed by the ToolKit and the sample programs, and are recommended:

- 1, 2, and 3 are low level ToolKit routines.
- 4, 5, and 6 are somewhat higher level ToolKit routines.
- 7 and 8 are major ToolKit routines.
- 9, 10, and 11 are building block routines.
- 12 and higher are application routines.

You can also set levels in and trace the action of procedures and functions that are not methods. Use BP and EP in the same way you do with methods. Note that, in general, procedures and function in the ToolKit that are not methods do not call BP and EP, and, therefore, do not show up in traces, and cannot be traced. However, there is nothing stopping you from using BP and EP in your non-method procedures and functions.

Here is a brief list of the functions in the menu. They are described in more detail in the subsections that follow.

- Breakpoint allows you to set breakpoints on the entrance and exit of a method, procedure, or function that has BP and EP calls. In addition, you can set a breakpoint on any reference to an object of some class. Note that Lisabug has a spearate breakpoint capability that can set a breakpoint at any instruction in memory.
- ClearBreakpoints removes breakpoints set with the ToolKit debugger. (Lisabug breakpoints are not affected.)
- DebugStatus prints the current watch level and watch count and also lists the current ToolKit debugger breakpoints.
- EnterLisabug brings you immediately into Lisabug, making certain that the application's code (and not system code) is running.
- Go continues your program's execution.
- Prompt turns the interactive menu display on or off. It does not otherwise affect the operation of the debugger or your program.
- StackCrawl shows the call sequence currently in effect, all the way back to your main program. It is similar to the SC command is Lisabug.
- FrameDump prints out information about one of the stack frames displayed by StackCrawl.
- InspectObject prints information about any Clascal object currently used by your program.
- HeapDump shows you the objects in the heaps.

- **Memory Dump** displays a portion of memory. This function works like Lisabug's DM command. See the debugger chapter of the Workshop manual for more information.
- **LevelsToWatch** sets the minimum level routine that is printed when you use the **Watch** command to trace your program.
- **OutputTo** directs a copy of the interactive debugger's output to a file or to a printer, in addition to the output on the alternate screen. Anything you type on the alternate screen is also printed.
- **Watch** displays the names of routines that contain BP and EP calls as they begin and end. This command can also be used to step through the program.

In all cases, you can invoke the command by giving the first letter of the command name in response to the interactive debugger prompt. You can type the whole command name if you want -- everything after the first letter is ignored, until a space or a <RETURN> is reached. Some commands take optional command line parameters. In all of those cases, the debugger will prompt you for any parameters it needs that you do not give.

5.3.1 Breakpoint

This command allows you to set breakpoints on the entrance and exit of a method, procedure, or function. In addition, you can set a breakpoint on any class, or to any method of a given class regardless of the method's name.

Up to 10 breakpoints can be set in any one process at one time.

Breakpoints are attached to the process, not to the document. Therefore, if you open a new document that uses the same process as one where you previously set breakpoints, the breakpoints operate when that document is processed.

You give the **Breakpoint** command by typing at least a B on the alternate screen in response to the interactive debugger prompt. (You can type any amount of the word breakpoint that you want -- as long as you give the B.) Then type <RETURN>.

When you invoke this command, the debugger prompts you for two pieces of information -- a class and a method name. You can give both, or either alone.

You are first prompted with the message

Class?

You can supply a class-type (only the first eight characters are significant), or you can ignore the message and simply type <RETURN>.

You are then prompted with the message

Method?

You can supply a method, procedure, or function name (only the first eight characters are significant), or you can ignore the message and simply type <RETURN>.

If you give a period after the class name in response to the first prompt, you are not asked for a method name.

Depending on the responses you gave to the prompts, there are four possibilites to what kind of breakpoint is set.

- If you gave a class-type and no method, procedure, or function name, the breakpoint occurs on any call to a method defined by that class-type. Breakpoints are not set on calls to methods defined by an ancestor or descendant of that class-type, even if the call is made through a class-reference of the given class-type.
- If you gave a method, procedure, or function name with no class-type, the breakpoint occurs on the entry and exit to any routine with that name. The breakpoint occurs whether or not the routine is a method or a global or local procedure or function, and regardless of the level of the routine.
- If you gave a class-type and a method name, the breakpoint occurs only on the entry and exit of the given method of the specifies class type, regardless of the class of the object referenced.
- If you gave neither, no breakpoint is set.

If you want, you can give the breakpoint on the command line, separated from the Breakpoint command by one or more spaces. You can give the class-type and method name, in the form

```
classJMethod
```

or you can give the class-type alone, in which case you are prompted for a method name. Alternately, you can give the class-type and a period (.), in which case you are not prompted for the method name. You can also give the method, procedure, or function name alone, but it must be preceded by a period (.), even if the routine is not a method, or the debugger will mistake it for a class-type.

For a breakpoint to occur, the routine must call the BP and EP procedures. Note that the debugger does not check if a routine with the given name exists, or if the routine calls BP and EP.

A breakpoint can be set on a routine that is swapped in or out. (In Lisabug, a breakpoint can only be set on a routine that is swapped in.)

Execution slows significantly when breakpoints are set.

To see what breakpoints are set, use the DebugStatus command.

5.3.2 ClearBreakpoints

This command removes breakpoints set with the ToolKit debugger. Lisabug breakpoints can not be changed with this command.

You can remove one breakpoint at a time, or you can remove all breakpoints at once.

To remove all breakpoints, type ALL, either on the command line, separated from the ClearBreakpoints command by at least one space, or in response to the prompt that appears after you give the ClearBreakpoints command.

To remove a single breakpoint, type the sequence number of the breakpoint, either on the command line, separated from the ClearBreakpoints command by at least one space, or in response to the prompt that appears after you give the ClearBreakpoints command.

The breakpoint sequence numbers appear in the display produced by the DebugStatus command.

5.3.3 DebugStatus

This command prints the current watch level and watch count and also lists the current ToolKit debugger breakpoints. See the Level command for an explanation of watch level; see the Watch command for an explanation of watch count.

The breakpoints appear in the following format:

```
1: class    .method
2: class    .method
      .
      .
      .
n: class    .method
```

n is the largest number of breakpoints set at any one time in that particular process, up to a maximum of 10. If a breakpoint is set on a class-type (that is, without a method name) the method space is blank, although the period (.) still appears. If a breakpoint is set on a method, procedure, or function name (that is, without a class-type) the class space is blank. Note that the period always appears, even when the breakpoint is set on a procedure or function that is not a method.

The sequence numbers that appear on the left are used in the ClearBreakpoints command. When you delete a breakpoint, the sequence numbers of the other breakpoints are not changed. The next breakpoint set uses the lowest open sequence number.

5.3.4 EnterLisabug

When you give this command, the ToolKit debugger brings you immediately into Lisabug. You can then use Lisabug in the normal way.

To get back to the ToolKit debugger, type G in Lisabug. Your program does not start running. The interactive debugger menu is immediately displayed.

There are a few special features of Lisabug when used with ToolKit applications. See the Lisabug documentation in the workshop manual for more information.

5.3.5 FrameDump

This command prints out information about one of the stack frames. It prompts for a number. Give it the number appearing after the # sign in the StackCrawl output.

The information given includes the section of memory containing SELF, and a range of memory containing local variables, and a range of memory containing the parameters for the call.

5.3.6 Go

This command continues your program's execution from where it entered the debugger. The normal office system screen is automatically displayed.

5.3.7 HeapDump

This command shows you the contents of the process, document, and clipboard heaps.

If you type H alone to invoke this command, you are prompted for several options. Alternately, you can type H *classtype* <RETURN>. In that case, only objects on the document heap of the specified class and its descendant classes are shown, and no prompts appear.

5.3.8 InspectObject

This command prints detailed information about any Clascal object that currently exists on your document or process heaps. Give the hexadecimal value for the object's handle, with or without the preceding $. Here is a sample of output from this command.

```
TSWHVIEW [ extentLR: LRect = [(0,0),(631,555)] ; view: TView = TSWHVIEW — $00DA060C
; panel: TPanel = TPANEL — $00DA0604 ; clickLPt: LPoint = (212,110) ; printMan:
TPrintMa = TSTDPRIN — $00DA0690 ; hMax: INTEGER = 90 ; vMax: INTEGER = 60 ;
minExten: LRect = [(0,0),(50,50)] ; isPrinta: BOOLEAN = TRUE ; isMainWi: BOOLEAN =
TRUE ; bevList: TList = TLINKLIS — $00DA0698 ]
```

Note that the exact format of the fields display depends on how you implement the Fields method for this class. If you do not implement a Fields method for a subclass that you define, you will not be able to see the contents of any field that your subclass added.

5.3.9 LevelsToWatch

This command sets the minimum level routine that is printed when you use the Watch command to trace your program. The level refers to the number given in the BP procedure call at the beginning of every method, procedure, and function that you wish to show up in the trace, or on which you wish to set breakpoints.

When you execute this command, the debugger prompts you for a level number. The number you give is the lowest level routine that appears in the watch trace.

The can be set to any whole number from 1 to 9999. It is 1 by default. Use the DebugStatus command to find the currently set watch level.

5.3.10 MemoryDump

This command works the same as the DM command in Lisabug. See the Workshop documentation for more information.

5.3.11 OutputTo

This command is used to direct a copy of the interactive debugger's output to a file or to a printer, in addition to the output on the alternate screen.

When you invoke this command, you are asked where you want to direct output to. Give the destination by identifying the device and, if appropriate, giving a file name. Give the device and file names preceeded by a dash, just as you do in the Workshop.

For example, if you want to direct output to the printer, type -printer. If you want to send output to a file on a diskette in the upper drive, type -upper-filename. If you direct output to an existing file, you destroy the current contents of the file. If the file does not exist, the debugger creates the file. If you want to examine the file in the Workshop editor, it must have a name ending in .TEXT.

The debugger output always shows on the alternate screen; OutputTo merely directs a duplicate to the given destination.

Output can only go to the console and one additional destination. Naming a third destination closes the path to the second destination.

You should always cancel the new output direction before going on to do something else. Otherwise, all other messages that go to the alternate screen are sent to the named destination, until you turn off the Lisa. Other alternate screen messages include the environments window display and the Workshop command line. In addition, if you directed output to the printer, the printer is not available for any other use.

Cancel the additional output direction by giving the OutputTo command a second time. You can type -Console, or simply type <RETURN>, as -Console is the default. Turning off the Lisa always cancels the output redirection, and closes any open files.

5.3.12 Prompt

This command determines whether or not the interactive debugger menu is displayed. It takes a Yes or No (Y or N) as a parameter. If you indicate Yes (either on the command line, separated from the Prompt command by at least one space, or in response to the prompt that appears if you do not specify Y or N) the menu is displayed. If you indicate No, the debugger prompt (-->) is displayed without showing the menu.

The Prompt command does not otherwise affect the operation of the debugger or your program.

5.3.13 RefsToObject

The RefsToObject command displays the names of objects that refer to a given object. RefsToObject prompts you for an object handle. It then searches the document heap for all occurances of that handle.

5.3.14 StackCrawl

This command displayes the current calling sequence in terms of stack frames. Here is a sample of a stack crawl.

```
Frame #   1 a $00F79FF8  TPROCESS.CHANGECU+ $0042  (TSANPROC: $000615EE)
Frame #   2 a $00F7A010  TPROCESS.TRACKCUR+ $007C  (TSANPROC: $000615EE)
Frame #   3 a $00F7A01C  TVDIBOU .IDLECONT+ $0026  (TSANWIND: $00DA0090)
Frame #   4 a $00F7A02C  TSELECTI.IDLECONT+ $0042  (TPICKSEL: $00DA062C)
Frame #   5 a $00F7A05C  TPROCESS.OBEYEVEN+ $0164  (TSANPROC: $000615EE)
Frame #   6 a $00F7A070  TPROCESS.RUN      + $001E  (TSANPROC: $000615EE)
Frame #   7 a $00F7FA22           ??SAMPLE + $01FE
```

5.3.15 Tally & Time

The Tally & Time command allows you to measure the performance of your program.

When you first invoke the command, you are asked if you want to continue execution and measure performance. If you answer no, the command is cancelled. If you answer yes, your application begins running, and the ToolKit records information on the methods and segments used. Every entry into a routine that begins with a BP call is recorded, and the time until the following EP is recorded.

When you next enter the debugger (in any manner except through ABCBreak), you are asked if you want to see performace measurements. If you respond no, you are asked if you want to zero the tallies and times. If you answer yes, all tally and time records are removed, and you are ready to start with a clean slate. If you answer no, any further performance measurments are added to the measurements up to that point.

Finally, you are asked if you want to continue performance measurement. Once again, if you answer yes, your program starts immediately. The only way to see the debugger menu is to answer no. In that case the menu is displayed. Note that in order to continue measuring performance, you must restart your program through the Tally & Time command.

To see the performance data, answer yes when you are asked if you want to see performance measurements. When you have such measurements recorded (that is, as long as you ran your program through the Tally & Time command and did not zero the tallies and times) you are asked if you want to see the results whenever you invoke the Tally & Time command, or when you invoke the debugger having run the program through Tally & Time.

When you ask to see the performance measurements, you are given some general information, and then a chart of segment usage. You are then asked if you want to see the procedure statistics. Note that these statistics actually show all routines containing BP and EP calls, whether they are procedures or functions. Next, you are asked if you want to see a list of procedures that were and those that weren't called in a particular segment. The "procedures that weren't called" once again include functions, and actually show all rotuines that do not contain BP and EP calls, in addition to routines that were not called.

You can stop any listing at any time by hitting a key on the keyboard. In that case, the previous question is repeated.

To skip a question and go to the next one, type a <RETURN>.

5.3.16 Watch

The Watch command displays the names of routines that begin with BP, end with EP, and have level numbers greater than or equal to the current watch level as they begin and end. This command can also be used to step through the program.

Chapter 6
Reference Information for ToolKit Classes

6.1 About This Chapter 6-2

6.2 Data Types 6-2

6.2.1 UObject Data Types 6-2
6.2.2 UABC Data Types 6-4

6.3 Global Varibles 6-5

6.3.1 UObject Global Variables 6-5
6.3.2 UABC Global Variables 6-6

6.4 Global Procedures and Functions 6-10

6.4.1 UObject Global Procedures and Functions 6-10
6.4.2 UABC Global Procedures and Functions 6-14

Reference Information for ToolKit Classes

6.1 About this Chapter

The ToolKit uses a number of predefined data types and global variables, procedures, and functions. Those that are of interest or use to programmers using the ToolKit are documented in this chapter.

The ToolKit uses a number of libraries that are not available to the general programming public. A few of the items in this chapter depend on definitions in these secret libraries. In addition, a few of the items depend on definitions in SYSCALL, the unit that defines the operating system used on the Lisa. You should never need to deal directly with either the secret libraries or SYSCALL. SYSCALL is documented in the *Operating System Reference Manual for the Lisa*

6.2 Data Types

The ToolKit contains a large number of type definitions. The definitions of class types are described in the class reference sheets in Part II of this manual. This section describes data types used in those class definitions. Refer to these pages when you encounter a data type that is not a class. The types defined in UABC begin with the letter T.

6.2.1 UObject Data Types

The data types in this subsection are defined in UObject.

The following five types are aliases for commonly used types.

```
Ptr = ^LONGINT;
ProcPtr = Ptr;
Handle = ^Ptr;
S8 = STRING[8];
S255 = STRING[255];
```

The following data types are used in many places throughout the ToolKit.

`TFilePath = S255;`	This corresponds to Pathname in SYSCALL.
`TFilePart = STRING[32];`	This defines the length of each level in a pathname; corresponds to e_name in SYSCALL.
`TPassword = TFilePart;`	
`THeap = Ptr;`	A pointer to a heap. Given to NewObject when defining a new object.
`TClass = Ptr;`	A pointer for a class. THISCLASS, given a parameter to NewObject when defining a new object, has a value of this type.
`Byte = -128..127;`	A signed, eight-bit value.

```
TPString = ^S255;
TpINTEGER = ^INTEGER;
TpLONGINT = ^LONGINT;
TAuthorName = STRING[32];          An argument of UnitAuthor and ClassAuthor.
TClassName = STRING[8];            A string for holding a class name.
TCollecHeader = RECORD             An alias for TCollection^^, which can be used
                                   for faster access to collections.
        classPtr: TClass;
        size: LONGINT;             The number of real elements, not counting
                                   the hole.
        dynStart: INTEGER;         The number of bytes from the classPtr to
                                   the dynamic data; MAXINT if none are
                                   allowed.
        holeStart: INTEGER;        0 = at the beginning, size = at the end;
                                   MAXINT = none allowed.
        holeSize: INTEGER;         Measured in MemberBytes units.
        holeStd: INTEGER;          If the holeSize goes to 0, this is how much
                                   to grow the collection by.
END;
TFastString = RECORD               An alias for a TString^^, which can be used for
                                   faster access to strings.
        header: TCollecHeader;
        ch: PACKED ARRAY[1..32740] OF CHAR;
END;
TPFAstString = ^TFastString;
THFastString = ^TPFastString;
TArrayHeader = RECORD              An alias for TArray^^, which can be used for
                                   faster access to arrays.
        classPtr: TClass;
        size: LONGINT;             The number of real elements, not counting
                                   the hole.
        dynStart: INTEGER;         The number of bytes from the classPtr to
                                   the dynamic data; MAXINT if none are
                                   allowed.
        holeStart: INTEGER;        0 = at the beginning, size = at the end;
                                   MAXINT = none allowed.
        holeSize: INTEGER;         Measured in MemberBytes units.
        holeStd: INTEGER;          This is how much to grow the collection by
                                   if the holeSize goes to 0.
        recordBytes: INTEGER;
END;
TScanDirection = (scanForward, scanBackward);
```

6.2.2 UABC Data Types

The following data types are defined in UABC.

TDlResponse = (dlAccept, dlDismissDialogBox, dlGiveToMainWindow, dlRefuse); Defines the response given when the user clicks in the main window or in the menu bar, or types when a dialog box is displayed.

TEnumAbilities = (aBar, aScroll, aSplit);
TAbilities = SET OF TEnumAbilities; Defines whether or not there is a scroll bar on the side of a panel and, if there is, whether or not there are icons for scrolling and splitting.

TUnitsFromEdge = (pixelsFromEdge, percentFromEdge); Used to define parameters for some TPanel methods.

TAlertArg = 1..5; Used to define parameters for some TProcess methods.

TAlertCounter = 6..9; Used to define parameters for some TProcess methods.

TAlignment = (aLeft, aRight, aCenter, aJustify);

TPageAlignment = (aTopLeft, aTopCenter, aTopRight, aBottomLeft, aBottomCenter, aBottomRight);

TClickState = Defines a record that contains information about the most recent mouse click.

```
RECORD
    where: Point;        The window-relative point where the pointer was
                         located.
    when: LONGINT;       The time in hundredths of a second from last
                         boot when the mouse button was last clicked.
    clickCount: INTEGER; The number of times the mouse button
                         was pressed in the period set by the
                         user through preferences.
    fShift, fOption, fApple: BOOLEAN; Whether these buttons were
                                      held down while the mouse
                                      button was clicked.
END;
```

TCmdNumber = INTEGER; Defines a command number.

TCmdPhase = (doPhase, undoPhase, redoPhase); Defines the command phase given to commandPerform.

TCursorNumber = INTEGER;

TEnumIcons = (lSkewer, lScrollBack, lFlipBack, lGrayA, lThumb, lGrayB, lFlipFwd, lScrollFwd); Used internally to define scroll bar icons.

TMousePhase = (mPress, mMove, mRelease); Used internally for mouse tracking.

TRevelation = (revealNone, revealSome, revealAll); Defines how much of the selected objects should be scrolled into the view if the user tries to operate on them.

TPrReserve = ARRAY [0..127] OF Byte; Defines an array of printer information.

TPrelude = Defines a prelude stored at the beginning of every document's heap.

```
RECORD
    password:      [2] INTEGER; Presently unused.
    version:       [2] INTEGER; Presently always 1.
    country:       [2] INTEGER; From phrase file.
    language:      [2] INTEGER; From phrase file.
    preludeSize:   [2] INTEGER; Contains SIZEOF (TPrelude).
    unused:        [6] ARRAY [0..5] OF Byte;
    printPref:     [128] TPrReserve; Format for printer information.
    docSize:       [4] LONGINT; Number of bytes in document.
    numSegments:   [2] INTEGER; Number of 128K files in document.
    docDirectory:  [4] TDocDirectory; Root object in document
                                      heap.
END;
```

TPPrelude = ^TPrelude;

TSBoxID = LONGINT; An alias for a secret library type.

TWindowID = LONGINT; An alias for a secret library type.

6.3 Global Variables

The ToolKit defines a number of global variables. Unlike variables which exist as fields of classes, these variables can be accessed from any part of an application that USES the ToolKit.

If an R appears before the variable name in the following lists, you might sensibly want your to application read the value of the variable. If a W appears before the variable name, you can alter the value of the variable.

6.3.1 UObject Global Variables

The following variables are defined in UObject.

R amDying: BOOLEAN; If TRUE, this process is terminating.

fCheckIndices: BOOLEAN; If TRUE, list indices are checked.

RW fDebugRecursion: BOOLEAN;	TRUE if you want want to inhibit tracing; you must save and restore its value; normally this is needed only if you override the Debug method. If you do that, save the old value of this variable in your Debug method. Then set the value to TRUE to inhibit tracing. Finally, before you method completes, restore the old value.
R isInitialized: BOOLEAN;	If TRUE, the application is initialized, and it cannot tell the desktop manager that initFailed any more.
W keyPresLimit: INTEGER;	How often to call KeyPress from the debugger to check for user interrupt.
mainDsRefnum: INTEGER;	The file system reference number (refnum) of the process data segment.
R mainHeap: THeap;	The heap of the process.
mainLdsn: INTEGER;	The ldsn of the process data segment.
R onDesktop: BOOLEAN;	If TRUE, the application is installed in the desktop. Nearly always true if UABC is used, because programs using UABC usually run on the desktop.
wmIsInitialized: BOOLEAN;	If TRUE, the window manager is open.

6.3.2 UABC Global Variables

The following global variables are defined in UABC for use by the ToolKit. They are set in TProcess.Commence, and can be considered fields of the process. In particular, changes to these variables affect all documents using that process.

You can examine the values of any of these variables, but only the variables whose name is preceded by an R are expected to be meaningful to you.

You should not change the value of any of them, except for a few that are preceded by a W.

activeWindowID: TWindowID;	The window manager ID of this process's active document window, or 0 (zero), if there is no active document.
allowAbort: BOOLEAN;	If TRUE, aborts of the process are allowed. If FALSE, process.Abort requests always return FALSE.
R amPrinting: BOOLEAN;	TRUE if this process is presently printing rather than drawing.

autoBreakPen: PenState;	The pen state to use to draw automatic page breaks.
blinkOffCentiSecs: LONGINT;	The time in hundredths of a second that the insertion point is off.
blinkOnCentiSecs: LONGINT;	The time in hundredths of a second that the insertion point is on.
boundClipboard: TClipboard;	The clipboard whose data segment is bound, or NIL, if no clipboard is presently bound.
boundDocument: TDocManager;	The document whose data segment is bound, or NIL. Usually the active document, if there is one, although applications can temporally bind inactive documents.
cancelString: STRING[20];	The word "Cancel" for use in buttons, given in the current language.
R clickState: TClickState;	The place where the mouse was last clicked, the number of times it was clicked, and whether the shift key was pressed, the option key was pressed, and the Apple key was pressed.
R clipboard: TClipboard;	The clipboard document manager.
clipPrintPref: TPrReserve;	The print-preference for the clipboard.
closedBySuspend: BOOLEAN;	If TRUE, closedDocument was just suspended, and not saved.
closedDocument: TDocManager;	If not NIL, this document was just closed.
cornerNumberStyle: TTypeStyle;	The type style used for page numbers in page preview mode.
currentDocument: TDocManager;	The active document if the process is running in foreground. If the process is running in background, this contains the docManager that is managing the background document. Otherwise contains NIL.
R currentWindow: TWindow;	The window for the current document (currentDocument.window), or NIL, if there is no current document or there is no window because the process is running in the background.
cursorShape: TCursorNumber;	The current cursor shape, as recorded by TProcess.ChangeCursor.

W deferUpdate: BOOLEAN; If TRUE, display update is deferred while the user is typing until some non-typing event occurs.

docList: TList {OF TDocManager}; The open docManagers using this process.

R eventTime: LONGINT; The time of the most recent event, in hundredths of a second from system boot.

R eventType: INTEGER; The type number of the most recent event. The available types are:

0 - nil event
1 - mouse button down
2 - mouse button up
3 - key down
4 - folder activate
5 - folder deactivate
6 - folder update
7 - folder moved
8 - desktop manager event
9 - abort event
10 - died event
11 - private
12 - private
13 - private

R fExperimenting: BOOLEAN; If TRUE, enable experimental functions. This value is changed by choosing ENABLE EXPERIMENTAL FUNCTIONS from the debug menu. You implement experimental functions by testing this value to see if the experimental sections should be executed.

fCountHeap: BOOLEAN; If TRUE and IFC fCheckHeap, count objects once per command. This value is changed by choosing Check and Count Heaps after Commands from the debug menu.

R genClipPic: BOOLEAN; If TRUE, this process is now drawing the Universal Graph component of the clipboard.

R highLevel: ARRAY [BOOLEAN] OF THighTransit;
If TRUE, the value is the highlighting transition from not active to active (hOffToOn); if FALSE, the value is the highlighting transition from active to not active (hOnToDim).

R highToggle: ARRAY [BOOLEAN] OF THighTransit;
If TRUE, the value is the highlighting transition from not selected to selected (hOffToOn). If FALSE, the value is the highlighting transition from selected to not selected (hOnToOff).

R idleTime: LONGINT; The time since the application finished processing the last user input, in hundredths of a second.

RW inBackground: BOOLEAN; TRUE if the process is allowed to run in the background. The program must set this value.

limboPen: PenState; The pen state used to fill limbo area in paginated view.

manualBreakPen: PenState; The pen state used to draw manual page breaks.

marginPattern: LPattern; The pen pattern used to fill margins in paginated view.

R menuBar: TMenuBar; The menuBar object for this document.

myProcessID: LONGINT; The OS process ID for this process object.

myTool: LONGINT; The tool number of this application.

normalPen: PenState; The pen state resulting from PenNormal.

okString: STRING[20]; The word for "OK" for use in buttons, in the current language.

phraseFile: TFileScanner; The filescanner for the main phrase file.

R process: TProcess; The process object presently being used by this application.

R screenRightEdge: INTEGER; The horizontal size of the screen in pixels. 720 for Lisa 1.0 and 2.0 screen.

scrollRgn: RgnHandle; The region that needs to be refreshed because of a scroll.

R stdMargins: LRect; The standard page margin size, in pixels. One inch on each side.

suspendSuffix: ARRAY [1..maxSegments] OF STRING[3];
A suffix string ($S1, $S2, ...) for each data segment used by the document.

theBodyPad: TBodyPad; The bodyPad used for printing and paginated views.

theMarginPad: TMarginPad; The marginPad used for printing and paginated views.

toolName: STRING[67];	The name of this application as specified by the InstallTool program.
R **toolPrefix: TFilePath;**	The Desktop Manager and OS file prefix used for this application. Has the form {Tnnn}.
R **toolVolume: TFilePath;**	The name of the volume on which this application resides. Has the form -volume-.
wordDelimiters: STRING[67];	The delimiters of a word in the current language. These delimiters are defined in the phrase file, as are all language-specific values, and indicate what character sequences define the beginning and end of a word.

6.4 Global Procedures and Functions

The ToolKit defines a number of procedures and functions that are not methods of any class, and can be called from any unit that USES the ToolKit.

6.4.1 UObject Global Procedures and Functions

The following are the global procedures and functions that can be used in application programs and are defined in UObject.

PROCEDURE IntToStr (int: INTEGER; str: TPstring);
PROCEDURE LIntToStr (int: LONGINT; str: TPstring);
These functions return a string containing the integer **int**. IntToStr takes an integer operand; LIntToStr takes a long integer operand.

PROCEDURE SplitFilePath (fullPath, itsCatalog, itsFilePart: TFilePath);
Given **fullPath** returns **itsCatalog** and **itsFilePart**. Levels in the Lisa's tree-structures file system are separated by dashes. This procedure searches backwards from the end of **fullPath**. Everything after the last dash in **fullPath** is placed in **itsFilePart**. The rest of the string is placed in **itsCatalog**.

FUNCTION Min(i, j: LONGINT): LONGINT;
Returns the smaller of the two numbers.

FUNCTION Max(i, j: LONGINT): LONGINT;
Returns the larger of the two numbers.

PROCEDURE XferLeft(source, dest: Ptr; nBytes: INTEGER);
Moves bytes from memory from **source** to **dest**, starting with the leftmost byte.

PROCEDURE XferRight(source, dest: Ptr; nBytes: INTEGER);
Moves bytes from memory from **source** to **dest**, starting with the rightmost byte.

FUNCTION EqualBytes(source, dest: Ptr; nBytes: INTEGER): BOOLEAN;
Returns TRUE if all the bytes in source and destination have the same value.

FUNCTION LIntAndLInt(i, j: LONGINT): LONGINT;
Returns the logical AND of the two numbers.

FUNCTION LIntOrLInt(i, j: LONGINT): LONGINT;
Returns the logical OR of the two numbers.

FUNCTION LIntXorLInt(i, j: LONGINT): LONGINT;
Returns the logical exclusive OR of the two numbers.

FUNCTION NewObject(heap: THeap; itsClass: TClass): TObject;
Creates a new object of type itsClass, of a size defined by itsClass.

FUNCTION NewDynObject(heap: THeap; itsClass: TClass; dynBytes: INTEGER): TObject;
Creates a new dynamic object of type TClass, of a size defined by itsClass plus dynBytes. This is used to create dynamically allocated objects, such as arrays.

PROCEDURE ResizeDynObject(object: TObject; newTotalBytes: INTEGER);
Changes the size of a given dynamic object to the new size. newTotalBytes must include the class size plus the size of the dynamic part.

FUNCTION NewOrRecycledObject(heap: THeap; itsClass: TClass; VAR chainHead: TObject): TObject;
Checks the given chainHead to see if there are any objects to be recycled. If there are, this procedure reuses the object, so that new space does not have to be allocated. If there are no objects to be recycled, NewObject is called to get a new object. This is used when many objects of a particular type are created and freed quickly. chainHead must be of the same type as itsClass, and must also be on the same heap.

PROCEDURE RecycleObject(object: TObject; VAR chainHead: TObject);
Adds the given object to the group stored by chainHead. This procedure is called in place of Free when you are creating and freeing objects of a particular type quickly. In that case, use this procedure in place of Free, and NewOrRecycledObject in place of NewObject. chainHead must be of the same class as the recycled objects, and must also be on the same heap.

PROCEDURE Free (object: TObject);
Deallocates the heap space used by the given object. All references to the object become invalid. If the object is NIL (indicating it has alreading been freed), this procedure does nothing.

FUNCTION Superclass(class: TClass): TClass;
Returns the class pointer for the superclass of the given class, or NIL.

FUNCTION ClassDescendsFrom(descendant, ancestor: TClass): BOOLEAN;
Returns TRUE if ancestor is an ancestor class for descendant.

PROCEDURE NameOfClass(class: TClass; VAR className: TClassName);
Gets the classname of the class using the given class pointer.

FUNCTION SizeOfClass(class: TClass): INTEGER;
Returns the size, in bytes, of an object of type class.

The next 3 procedures can only be called from a class-init block or a subroutine of a class-init block.

PROCEDURE UnitAuthor(companyAndAuthor: TAuthorName);
Registers the company and author name for all classes defined in a unit. This procedure must be called once in every unit.

PROCEDURE ClassAuthor(companyAndAuthor: TAuthorName; classAlias: TClassName);
Overrides a registered company and author name, for use when you declare a methodless dummy of someone else's class for version conversion.

PROCEDURE ClassVersion(itsVersion, oldestItCanRead: Byte);
Used in releases past first release of your application. This procedure is added to every class that requires data conversion.

PROCEDURE ABCBreak(s: S255; errCode: LONGINT);
Stops the application. If the application was linked with the debugging version of the ToolKit, the debugger is invoked. Otherwise, a technical difficulties dialog box is displayed or, if called early in the application, the system may hang. This procedure should generally be surrounded with IFC compiler directives, so that non-debug versions of your program do not contain breaks. You can use ABCBreak calls in non-debug versions of your program. In that case, the program halts.

PROCEDURE LIntToHex(decNumber: LONGINT; hexNumber: TPString);
Converts a LONGINT to a hexadecimal number.

PROCEDURE EntDebugger(inputStr, enterReason: S255);
Invokes the ToolKit interactive debugger. The enterReason string is displayed. inputStr is ignored.

PROCEDURE DumpVar(pVariable: Ptr; nameAndType: S255);
Writes the value of a variable on the alternate screen.

PROCEDURE WrStr(str: S255);
Like the required Pascal function WRITE, but performs word wrap when necessary. (Warning: This routine may have bugs.)

PROCEDURE WrLn;
Like the required Pascal function WRITELN, but performs word wrap when necessary. Every time this is called, the next line is indented by the number of characters indicated by outputIndent, an INTEGER global variable of UObject.

PROCEDURE WrObj(object: TObject; numLevels: INTEGER; memberTypeStr: S255);
Like objectLDebug, but it formats using the outputIndent global variable to define the left margin.

FUNCTION NewHeap(VAR error: INTEGER; heapStart, numBytes: LONGINT;
numObjects: INTEGER): THeap;
Creates a new heap. The ToolKit calls this for you. You should never call this function.

FUNCTION MakeDataSegment(VAR error, dsRefnum: INTEGER; firstTryVolume,
thenTryVolume: TFilePath;idsn, memBytes,
diskBytes: INTEGER): LONGINT;
Creates a new data segment. The ToolKit calls this for you. You should never call this function.

PROCEDURE SetHeap(heap: THeap);
PROCEDURE GetHeap(VAR heap: THeap);
These routines deal with the heap used by QuickDraw for regions. You can use these to but QuickDraw regions on other heaps.

PROCEDURE MarkHeap(heap: THeap; mpAddress: LONGINT);
PROCEDURE SweepHeap(heap: THeap; report: BOOLEAN);
These routines implement garbage collection. They are called by the functions in the debug menu that deal with garbage collection. If report is TRUE, the garbage is reported and not freed. If report is FALSE, the garbage is freed and not reported. You should never call these routines directly; they are called when you choose commands from the debug menu.

PROCEDURE BP(MyTraceLevel:integer);
Marks the beginning of a method or other routine for debugging purposes. You should begin all your methods with a call to this procedure, since the ToolKit debugger will otherwise ignor your methods. Also, see PROCEDURE EP.

PROCEDURE EP; {Trace entry from method and write SELF (unless CREATE,
Debug, FreeObject, or Free)}
Marks the end of a method or other routine for debugging purposes. See PROCEDURE BP.

PROCEDURE HexStrToLint(hexString: TPString; VAR decNumber: LONGINT; VAR
result: TConvResult);
Converts a hexadecimal string to a long integer. result tells something about the result and can be: cvValid, cvNoNumber (null input), cvBadNumber (invalid character in input), or cvOverflow.

PROCEDURE StrToLInt(str: TPString; VAR decNumber: LONGINT; VAR result: TConvResult);
Converts a string to a long integer. result tells something about the result and can be: cvValid, cvNoNumber, cvBadNumber, or cvOverflow.

PROCEDURE StrToInt(str: TPString; VAR decNumber: INTEGER; VAR result: TConvResult);
Converts a string to an integer. result tells something about the result and can be: cvValid, cvNoNumber, cvBadNumber, or cvOverflow.

PROCEDURE TrimBlanks(str: TPString);
Removes leading and trailing blanks from str.

FUNCTION CharUpperCased(ch: CHAR): CHAR;
Converts the character to its corresponding uppercase character.

PROCEDURE StrUpperCased(str: TPString);
Converts the characters in the string to their corresponding uppercase characters.

PROCEDURE LatestError(newError: INTEGER; VAR previousError: INTEGER);
This is used to handle error codes returned by multiple operations, so that you end up with the first error number or warning number (error code < 0) if there was no error.

You should pass in the latest error as newError and the variable that is to be the final error code as previousError. Here is the actual code of LatestError:

```
IF ((newError > 0) AND (previousError <= 0) OR
(newError < 0) AND (previousError = 0)) THEN
previousError := newError;
```

6.4.2 UABC Global Procedures and Functions

The following are the global procedures and functions that can be used in application programs and are defined in UABC.

FUNCTION ToolOfFile (wholeName: S255): LONGINT;
Given the name of the tool, this returns the tool number. When given a string of the form {Txxx}job}, returns xxx.

FUNCTION ToolOfProcess (processId: LONGINT): LONGINT;
Given an OS process ID, this returns the tool number using that process.

The following routines are used to insert comments into the Universal Graph of the clipboard, so LisaDraw can understand it.

```
PROCEDURE PicTextBegin (alignment: TAlignment);
PROCEDURE PicTextEnd;
```
These two routines surround a sequence of text drawing calls that should be considered one field in LisaDraw.

```
PROCEDURE PicGrpBegin;
PROCEDURE PicGrpEnd;
```
These two routines surround a sequence of drawing calls that should be grouped in LisaDraw.

Part II
Class Reference Sheets

1 About the Reference Sheets

This part contains reference sheets for the classes in UABC, UDraw, and UObject. Not all classes are included, nor are all their methods; however, you do not need to know about any that are not included here. The reference sheets document methods, subclasses, superclasses, data fields and other relevant information for the classes.

All of the classes and all of their methods can be found in the interface units, copies of which are given in the appendices.

There is one reference sheet for each class.

All reference sheets follow this format:

CLASS: TThisClass

SUPERCLASS: TSuperclass

DEFINED SUBCLASS(ES): TSubclass(es)

INHERITED DATA FIELDS: Data fields inherited from TSuperclass, and their significance.

DATA FIELDS: Data fields defined by TThisClass, and their significance.

METHODS: Methods implemented by TThisClass or inherited from TSuperClass, divided into a number of explanatory catagories.

FAILURE CONDITIONS: Typical causes of failure of methods of TThisClass, if any. (See Section 1.4.)

NOTES: Pertinent notes on TThisClass and its use.

In these reference sheets, the following symbols are used:

**** to indicate classes that must be subclassed for use.

R before a data field to indicate that the contents of the data field can be sensibly read.

W before a data field to indicate that the contents of the data field may be modified by the application.

2 CREATE Methods

Every class must have its own CREATE method. In every case, the first two parameters of CREATE must be an object and a heap pointer. See Chapter 2, UObject for an explanation of the use of these two parameters.

3 The Reference Sheets

The remainder of this chapter is an alphabetical listing of the reference sheets.

Each reference sheet begins on a new page. Please check the explanation of symbols above before reading the reference sheets. The sheets are in alphabetical order.

CLASS: TArea

SUPERCLASS: TObject

DEFINED SUBCLASSES: TBand
TBranchArea (not documented)
TPad
TPanel
TWindow

INHERITED DATA FIELDS: none

DATA FIELDS:

R	innerRect: Rect;	Window-relative bounds excluding borders.
R	outerRect: Rect;	Window-relative bounds including borders.
RW	parentBranch: TBranchArea;	The branchArea that indicates the parent of this area.

IMPLEMENTED METHODS:

ChildWithPt (pt: Point; childList: TList; VAR nearestPt: Point): TArea;

pt is a point in window-relative coordinates.
childList is a list of areas contained within this area.
nearestPt is the point in the returned area closest to pt.
The return value is one of the areas from childList.

TArea.ChildWithPt first finds the point in area.innerRect closest to pt. It then the compares the new point to the outerRects of all the areas in childList. When it finds the area that contains the point, it finds the point in that area's innerRect that is closest to the point. That value is returned as nearestPt. The area containing nearestPt is the return value.

Erase;

TArea.Erase erases the interior. This assumes the focus was set by calling Focus.

Frame; DEFAULT;

Overridden versions of Frame draw outlines, scroll bars outside the inner rectangle. TArea.Frame draws a one-pixel wide box around the area. Frame assumes the focus was set by calling Focus.

GetBorder (VAR border: Rect); DEFAULT;

border is the width of the border.

GetBorder returns the widths of the border bars. Each of the fields of the Rect defines the width of the border on the corresponding side. The standard sizes returned by overridden versions of

GetBorder are: for windows, bands, panes: 1 all around; for panels: 1 on left/top, scrollBars on right/bottom. TArea.GetBorder returns a border that is one-pixel wide on all sides.

SetInnerRect (newInnerRect: Rect);

newInnerRect is the area bounds excluding borders.

TArea.SetInnerRect sets the inner rectangle of an area. This method also resets the size of the outerRect based on the border size given by GetBorder.

SetOuterRect (newOuterRect: Rect);

newOuterRect is the bounding box.

TArea.SetOuterRect sets the outer rectangle to a new size. This method also resets the size of the innerRect based on the border size given by GetBorder.

ABSTRACT METHODS:

DownAt (mousePt: Point): BOOLEAN; ABSTRACT;

mousePt is the place the mouse button was last pressed.

TArea.DownAt does nothing; subclasses should implement DownAt so it returns TRUE if mousePt is in the area. It is implemented for you by TWindow, TPane, TPad, and TPanel, the subclasses of TArea that need it.

Focus; ABSTRACT;

TArea.Focus focuses the grafPort on this area. Focus is an abstract method here, but is implemented by TWindow, TPane, and TPad, subclasses of TArea.

Refresh (rActions: TActions; highTransit: THighTransit); ABSTRACT;

rActions is what to do in the area, from the set of TActions: rErase, rFrame, rBackground, rDraw.
highTransit is the change in highlighting in the area. The standard choices are: hNone, hOffToDim, hOffToOn, hDimToOn, hDimToOff, hOnToOff, and hOnToDim.

TArea.Refresh does nothing; subclasses should implement Refresh so it redraws any changed parts of the area. It is implemented for you by TWindow, TPane, TPad, TMarginPad, and TBodyPad, subclasses of TArea.

ResizeInside (newInnerRect: Rect); ABSTRACT;

newInnerRect is the area bounds excluding borders.

ResizeInside is intended to change the size of area.innerRect. It should be implemented to do higher-level operations on the area,

such as resizing areas contained within this area, and then call SELF.SetInnerRect.

ResizeOutside (newOuterRect: Rect); ABSTRACT;

newOuterRect is the bounding box.

ResizeOutside is intended to change the size of area.outerRect. It should be implemented to do higher-level operations on the area, such as resizing areas contained within this area, and then call SELF.SetOuterRect.

NOTES:

1. TArea implements area objects that are used to define areas on the screen in a general way. The areas may or may not have borders. TWindow, TPanel, TPane, TPad, TBand, TBranchArea, and TDialogBox are all descendents of TArea.
2. You never create a TArea object. You always use some descendant of TArea.
3. You will not usually not subclass TArea.
4. You will subclass one descendant of this class, TWindow.
5. The boundaries of an area are set with the methods SetOuterRect and SetInnerRect. Do not change innerRect or outerRect directly.

CLASS: TArray

SUPERCLASS: TCollection

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

R size: LONGINT;	The number of real elements in this array, not counting the hole. This is a LONGINT for the benefit of huge collections, such as remote data bases. It is always in the INTEGER range for instances of TArray.
dynStart: INTEGER;	The number of bytes from the class pointer to the dynamic data area.
holeStart: INTEGER;	0 means hole at the beginning; value of size means hole at the end.
holeSize: INTEGER;	The initial size of the hole, measured in number of members.
holeStd: INTEGER;	How much to grow the array by if the holeSize goes to 0.

DATA FIELDS: R recordBytes: INTEGER; Bytes per record.

METHODS:

CREATE (object: TObject; itsHeap: THeap; initialSlack: INTEGER; bytesPerRecord: INTEGER): TArray;

initialSlack is the size of the initial hole, in member-sized units. bytesPerRecord is the number of bytes per record you wish the array to have. bytesPerRecord must be an even number. SIZEOF(*recordType*) is often used.

TArray.CREATE creates an empty array on itsHeap that has bytesPerRecord bytes per record (array.recordBytes:= bytesPerRecord). initialSlack is a hole for array members, expressed as the number of members the hole could hold. If initialSlack is greater than 0, the insert methods can be used to initialize the list without allocating any additional space.

At (i: LONGINT): Ptr; DEFAULT;

i is an ordinal position in the array.
The return value is an address for the element.

At returns the address of the element at the given position in the array. Because the address is in relocatable storage, it is valid only until the next time the heap is compacted. For that reason, GetAt is preferable.

This method is used for a simple subscripted fetch. Instead of using

```
x:= ordArray[i];
```

as you would to fetch element i from the Pascal ARRAY ordArray, you write

```
x:= TpRecord(myArray.At(i))^;
```

to fetch element i from the TArray myArray.

Debug (numLevels: INTEGER; memberTypeStr: S255); OVERRIDE;

numLevels is the number of levels of detail printed. memberTypeStr indicates how the array elements should be treated. This value is a data type; the debugger prints out the values of the elements as if they were of this type.

Debug is called by the ToolKit debugger to print details about this array. numLevels determines how many levels of detail are printed. When numLevels equals:

0 (zero), the debugger prints just the class of array;

1, the debugger also prints size of array;

2, the debugger also prints a compacted list of member classes

greater than or equal to 3, the debugger also prints class, size, and calls Debug(numLevels-1) on members of the array.

DelAll; DEFAULT;

DelAll deletes all the elements of an array. The array itself is not deleted. array.size becomes 0. There is a hole of size array.holeSize.

Each (PROCEDURE DoToObject(pRecord: Ptr)); DEFAULT;

DoToObject is a PROCEDURE that takes a pointer as its argument.

Each applies the given PROCEDURE to each element in the array. For example:

```
myArray.Each (Tally);
```

is equivalent to:

```
FOR i:= 1 TO myArray.size DO
    Tally (myArray.At(i));
```

DelAt (i: LONGINT); DEFAULT;

i is an ordinal position in the array.

DelAt deletes the element at position i; elements after i are renumbered to fill in the empty space.

DelFirst;

DelFirst deletes the element at the beginning of the array; elements after the first are renumbered to fill in the empty space. This is equivalent to array.DelAt(1).

DelLast;

DelLast deletes the element at the end of the array. This is equivalent to DelAt(array.Size).

DelManyAt (i: LONGINT; howMany: LONGINT); DEFAULT;

i is an ordinal position in the array.
howMany indicates the number of array elements to be deleted.

DelManyAt deletes a number of elements from the array. The first deleted element is at position i; elements after i+howMany are renumbered to fill in the empty space.

First: Ptr;

The return value is a pointer to a record whose size is array.recordBytes.

First returns a pointer to the first element in the array. This is equivalent to array.At(1).

GetAt (i: LONGINT; pElement: Ptr);

i is an ordinal position in the array.
pElement is an address for the element.

GetAt returns the address of the element at the given position in the array. It is similar to At, except that the syntax of the call is

```
myArry.GetAt (i, @x);
```

instead of

```
x:= myArray.At(i)^;
```

GetAt is safer, because your program does not deal with a pointer to relocatable storage.

InsAt (i: LONGINT; pRecord: Ptr); DEFAULT;

i is an ordinal position in the array.
pRecord is a pointer to a record whose size is array.recordBytes.

InsAt inserts an element in the array. The newly inserted element occupies position i.

InsFirst (pRecord: Ptr);

pRecord is a pointer to a record whose size is array.recordBytes.

InsFirst inserts an element at the beginning of the array. The new element occupies the first position in the array. This is equivalent to array.InsAt(1, pRecord).

InsLast (pRecord: Ptr);

pRecord is a pointer to a record whose size is array.recordBytes.

InsLast inserts an element at the end of the array. The new element occupies the last position in the array. This is equivalent to array.InsAt(array.Size+1, pRecord).

Last: Ptr;

The return value is a pointer to a record whose size is array.recordBytes.

Last returns a pointer to the element at the end of the array. This is equivalent to array.At(array.size).

ManyAt (i, howMany: LONGINT): TArray;

i is an ordinal position in the array.
howMany is a number of elements.

ManyAt returns an array of size howMany containing copies of the elements from the original array beginning at position i and continuing through howMany elements.

MemberBytes: INTEGER; OVERRIDE;

MemberBytes returns array.recordBytes, the number of bytes in each element of the array, as was specified in the bytesPerRecord parameter of the TArray.CREATE call.

Pos (after: LONGINT; pRecord: Ptr): LONGINT;

after is a the position in the array.
pRecord is a pointer to a record.
The return value is a position in the array.

Pos searches the array, beginning at position after+1. It returns the position of the first element found that is equivalent to the record pointed to by pRecord. If no such record is found, Pos returns after.

PutAt (i: LONGINT; pRecord: Ptr); DEFAULT;

i is an ordinal position in the array.
pRecord is a pointer to a record whose size is bytesPerRecord.

PutAt deletes the element at position i, and replaces it with the record pointed to by pRecord.

This method is used for a simple subscripted fetch. Instead of using

```
ordArray[i]:= x;
```

as you would to replace element i in the Pascal ARRAY ordArray, you write

```
myArray.PutAt(i, @x);
```

to replace element i in the TArray myArray.

Scanner: TArrayScanner;

The return value is an arrayScanner for this array.

Scanner returns an object of class TArrayScanner, allowing use of arrayScanner methods. This is equivalent to

```
array.ScannerFrom (1, scanForward);
```

ScannerFrom (firstToScan: LONGINT; scanDirection: TScanDirection): TArrayScanner; DEFAULT;

firstToScan is a position in the array.
scanDirection scanForward or scanBackWard.
The return value is a new arrayScanner.

ScannerFrom returns an object of class TArrayScanner, with arrayScanner.Position equal to firstToScan minus one, so that the first call to arrayScanner.Scan returns the address of the record at firstToScan.

StartEdit (withSlack: INTEGER);

withSlack is the new value for holeStd.

StartEdit changes thje value of holeStd to withSlack. The next edit call creates a hole of size withSlack, so that subsequent edit calls act more quickly.

StopEdit;

StopEdit removes the hole from the array and sets holeStd to 0 (zero). Any subsequent edit call removes any hole it forms.

FAILURE CONDITIONS: Heap can't grow.
Array > 32K bytes.
Deleting from empty array (only checked if the debug menu item CHECK LIST INDICES is on).
Subscript out of range (only checked if the debug menu item CHECK LIST INDICES is on).

NOTES:

1. An array is a space-optimized list. It is similar to a list object, but where a list contains object handles, an array contains records.
2. While an object can have referrences to it in several lists, a record can be in only one array at a time.
3. Use instances of TString when you need to store a string of characters or one-byte values.
4. list operations are more convenient and often run faster than array operations: TArray has a relatively limited interface.
5. Arrays have three modes: *create mode*, *edit mode*, and *static mode*.
6. When you first create an array, it is in create mode. You define an initialSlack value in the CREATE call. The array is given enough empty space to hold initialSlack records. That space is the hole. As you add members, the space in the hole is used for the new members. The amount of space allocated to the array does not change until you fill up the hole with array members. You can also call array.StopEdit, which removes the hole from the array.
7. When the hole is filled, the array enters static mode. In static mode, no hole is ever maintained. If you add a member, the array is copied into a space large enough to hold the additional member. If you delete a member, the extra space is freed.
8. Enter edit mode by calling array.StartEdit(withSlack). In edit mode, a hole big enough for withSlack elements is initially created. As you add members, the size of the hole decreases. The hole is always positioned after the last element inserted or after the position occupied by the last element deleted. When the hole is entirely filled, the space allocated to the array is increased, so there is a new hole big enough for withSlack members. If you delete members, the extra space is added to the hole. Call array.StopEdit to stop editing. Any space taken up by the hole is then freed.
9. array.size * array.recordBytes cannot exceed 32767.
10. You can access array elements faster by creating an alias for the array, and then using double-indirect pointers. The array must have no hole, or a hole at the end of the array, for this method to work.

Assume you have created an array similar to this one, where

```
TYPE T= RECORD
    {fields}
    END;
VAR myArray: TArray[OF T];
```

Also, assume you have created your array in some way such as this:

```
myArray:= TArray.CREATE (NIL, heap, initalSize, SIZEOF(T));
```

To access the array more quickly than you could with At and PutAt, declare some variables like these:

```
TFastT= RECORD
    header: TArrayHeader;          {A pre-defined ToolKit type.}
    records: ARRAY[1..6000] OF T;  {The number in italics is given
                                    here as an example.  It should
                                    be 32000/SIZEOF(T)}
    END;
TPFastT= ^TFastT;
THFastT= ^TPFastT;
```

Then you can access the array like this:

```
THFastT(a)^^.records[i]:= tRec;
tRec:= THFastT(a)^^.records[i];
```

where tRec is a record of type T. The array bounds in TFastT should be chosen to be large enough to avoid range-check errors at run time, but must be less that about 32000 DIV SIZEOF(T).

CLASS: TArrayScanner

SUPERCLASS: TScanner

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

R	collection: TCollection;	The array being scanned.
R	position: LONGINT;	The current position of the scanner. The scanner position is always between, before, or after members: 0=before first, size+1=after last.
	increment: INTEGER;	The change in position for every scan: 1 if scanning forward, -1 if scanning backward.
	scanDone: BOOLEAN;	When this is TRUE, the next Scan call returns FALSE, signalling the end of the scan. The VAR parameter of the Scan method, which normally contains the next record in the array, is unchanged after that Scan call.
R	atEnd: BOOLEAN;	TRUE if the end of the array is imminent, so that the next Scan call will return FALSE.

DATA FIELDS: none

METHODS:

CREATE (object: TObject; itsArray: TArray; itsInitialPosition: LONGINT;
itsScanDirection: TScanDirection): TArrayScanner;

itsArray is an array object.
itsInitialPosition is the initial ordinal position in itsArray.
itsScanDirection is scanForward or scanBackward.

TArrayScanner.CREATE creates an object of type TArrayScanner, which is used to access itsArray. You never call this method directly; instead, you use array.Scanner or array.ScannerFrom. Those methods call TArrayScanner.CREATE.

Append (pRecord: Ptr); DEFAULT;

pRecord is a pointer to a new member of the array.

Append adds a copy of the record pointed to by pRecord to the array immediately after the current position. position is adjusted by adding 1, regardless of the current increment.

Delete; DEFAULT;

Delete deletes the record at the current position. position is adjusted by adding arrayScanner.increment.

DeleteRest; DEFAULT;

Delete deletes all records after the current position, if arrayScanner.increment is +1, or before the current position, if arrayScanner.increment is -1. position is unchanged. Any subsequence arrayScanner.Scan call returns FALSE.

Done;

Done signals that you are finished using this arrayScanner. The next call to arrayScanner.Scan returns FALSE, as if the end of the array had been reached. Unlike when the end of the array is reached, however, the returned record in Scan remains unchanged.

Free; OVERRIDE;

Free deallocates this arrayScanner object.

Obtain: Ptr; DEFAULT;

The return value is a pointer to a record in the array.

Obtain returns a pointer to the current record. For example, where s is an arrayScanner created with myArray.Scanner

```
x← s.Obtain^;
```

is the same as

```
x← myArray.At;
```

Replace (pRecord: Ptr); DEFAULT;

pRecord is a pointer to a record.

Replace removes the current record from the array and replaces it with a copy of the record pointed to by pRecord. The new record is now the current record.

Scan (VAR pNextRecord: Ptr): BOOLEAN; DEFAULT;

pNextRecord is a pointer to the next record from the array. The return value is FALSE if the end of the array has been reached or arrayScanner.Done has been called.

Scan begins at the beginning of the array and, each time it is called, returns a pointer to the next record. The record pointed to by the returned pointer is referred to as the *current* record. The value of Scan is TRUE until Scan is called after the last record in the array is reached, or until arrayScanner.Done has been called. If the end of the array was reached, and Done was not called, the value of pNextRecord is NIL. If Done was called, pNextRecord keeps the last value it had. When Scan returns FALSE, the scanner is freed.

Seek (arrayPos: LONGINT);

arrayPos is the location to which you wish the scanner to move. arrayPos = 0 is the beginning of the array.

Seek moves the current scan position to arrayPos. It transfers no data.

Skip (deltaPos: LONGINT);

deltaPos is the number of elements you wish the scan position to move within the array. A positive value moves the position toward the end of the array; a negative value moves the position toward the beginning of the array.

Skip moves the scan position deltaPos bytes forward or backward within the array. arrayScanner.increment is ignored. Skip transfers no data.

NOTES:

1. An arrayScanner is used to access a stored data array, to scan it, and to manipulate its individual elements. Use TArrayScanner and its subclasses to scan arrays.
2. You may have more than one arrayScanner for the same array at the same time. However, if you do, you cannot use Append, Delete, or DeleteRest.

CLASS: TBand

SUPERCLASS: TArea

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: None that are significant. See TArea for information.

DATA FIELDS:

window: TWindow;	The window containing this band.
panes: TList {of TPane};	The panes contained in this band.
panel: TPanel;	The panel containing this band.
scroller: TScroller;	The scroller for this band.
scrollDir: VHSelect;	The orientation of this band. v if this is a row of panes with a vertical scroll bar; h if this is a column of panes with a horizontal scroll bar.

METHOD YOUR APPLICATION MIGHT CALL:

ChildWithPt (pt: Point; childList: TList; VAR nearestPt: Point): TArea;

pt is a point in window-relative coordinates.
childList is a list of areas contained within this band.
nearestPt is the point in the returned area closest to pt.
The return value is one of the areas from childList.

ChildWithPt first finds the point in bandInnerRect closest to pt. It then the compares the new point to the outerRects of all the areas in childList. When it finds the area that contains the point, it finds the point in that area's innerRect that is closest to the point. That value is returned as nearestPt. The area containing nearestPt is the return value.

ResizeOutside (newOuterRect: Rect);

newOuterRect is the bounding box.

ResizeOutside is used to change the size of bandOuterRect. It sets the new value of bandOuterRect, invalidates as needed, adjusts the size of the scoller, and resized all panes in the band (by calling bandResizePanes). It can be re-implemented to do higher-level operations on the band, such as resizing areas contained within this band.

ScrollBy (deltaLCd: LONGINT);

deltaLCd is the amount the band is scrolled, in view coordinates.

TBandScrollBy scrolls the band by the given amount. Bands only scroll in one orientation, bandScrollDir. The thumb is also moved. A positive deltaLCd scrolls towards the end of the view, while a negative value scrolls towards the beginning. When deltaLCd is 0

(zero), the ScrollBy does not scroll, but, instead, invalidates the entire band. The next time Update is called, the band is redrawn. This feature is used for resizing bands.

ScrollTo (viewLCd: LONGINT);

viewLCd is the position in the view the band is scrolled to, in view coordinates.

TBand.ScrollTo scrolls the band by the given amount. viewLCd is placed in the top left corner of the band. The thumb icon is also moved.

ThumbTo (newThumbPos: INTEGER);

newThumbPos is the desired position of the thumb icon. 0 is the position at the beginning of the document, and 1000 is the position at the end.

TBand.ThumbTo moves the thumb to the given position, and scrolls the band by the resulting amount.

NOTES:

1. TBand is never subclassed or instantiated by application programs, and you almost never call it.
2. You can call the methods listed above in order to implement scrolling in your own way.
3. Methods of TBand are not normally used for scrolling. Instead, use TPanel.RevealLRect to bring information in the panel into view.

CLASS: TBodyPad

SUPERCLASS: TPad

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: See TPad.

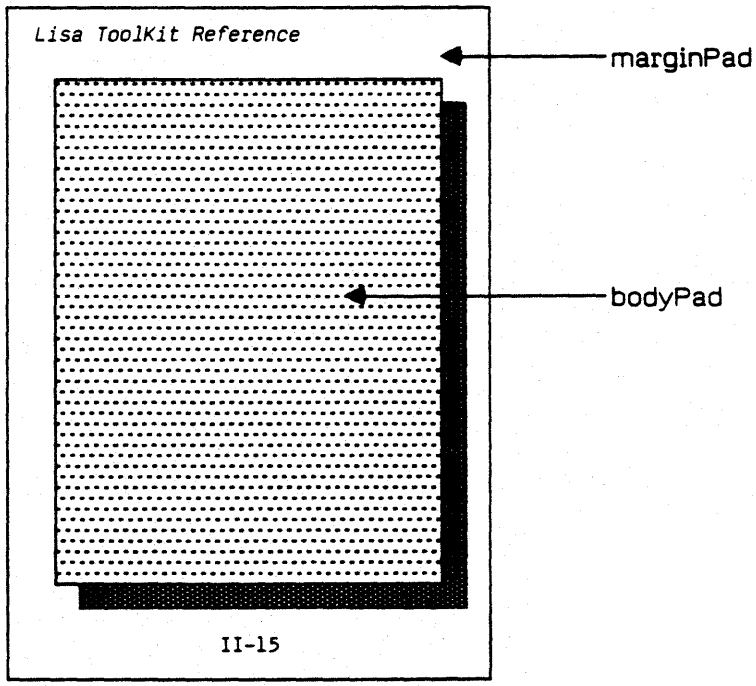
DATA FIELDS:

marginPad: TMarginPad;	The marginPad that defines the margins surrounding this bodyPad
nonNullBody: Rect;	The section of the view that appears in this bodyPad.

METHODS: None that are important for applications.

NOTES:

1. This class defines a pad used in creating the printable picture of a section of a view.
2. A bodyPad contains a printable picture of a section of a view. Each page of the entire printable image is made up of the bodyPad and a marginPad.



CLASS: TClipboard

SUPERCLASS: TDocManager

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: None are important. See TDocManager for more information.

DATA FIELDS:

R	hasView: BOOLEAN;	FALSE indicates there is no ToolKit representation of the data in the clipboard.
R	hasPicture: BOOLEAN;	FALSE indicates there is no universal graphics picture of the data in the clipboard.
R	hasUniversalText: BOOLEAN;	FALSE indicates there is no universal text version of the data in the clipboard. Note that you need to use the Universal Text Building Block to use universal text.
	hasIcon: BOOLEAN;	TRUE if there is an icon reference available.
R	cuttingTool: LONGINT;	Contains the tool number of the application that last loaded the clipboard.
R	cuttingProcessID: LONGINT;	Contains the OS process ID of the process that last loaded the clipboard.
	clipCopy: TFileScanner;	A scanner on a file containing a copy of the clipboard before conversion.

METHODS:

None you need.

NOTES:

1. The clipboard object acts as a directory to the clipboard, which stores the last data cut or copied from any document on the desktop.
2. Every process has its own clipboard.
3. You may want to inspect the clipboard's data fields to find out whether or not you can copy information from the clipboard.

CLASS: TCollection

SUPERCLASS: TObject

DEFINED SUBCLASSES: TArray
TFile
TList
TString

INHERITED DATA FIELDS: none

DATA FIELDS:

R size: LONGINT;
The number of real elements in this collection, not counting the hole. This is a LONGINT for the benefit of huge collections, such as remote data bases. It is always in the INTEGER range for instances of TList, TArray, and TString.

RW dynStart: INTEGER;
The number of bytes from the class pointer to the dynamic data area.

holeStart: INTEGER;
0 means hole at the beginning;value of size means hole at the end.

holeSize: INTEGER;
The initial size of the hole, measured in number of collection members.

holeStd: INTEGER;
How much to grow the collection by if the holeSize goes to 0.

METHODS:

CREATE (object: TObject; heap: THeap; initialSlack: INTEGER):
TCollection;

object, in this case, must be an object that was allocated by the subclass's CREATE method by calling NewDynObject.
initialSlack is the size of the initial hole, in members.

TCollection.CREATE creates an object of type TCollection. initialSlack is a hole for collection members. If initialSlack is greater than 0, the insert methods can be used to initialize the collection, without allocating any storage. This is never called directly; it is called by CREATE methods of subclasses.

Clone (heap: THeap): TObject; OVERRIDE;

heap is an available heap.
The return value is a new collection that has the same contents and characteristics as the calling collection.

TCollection.Clone creates a copy of the collection.

MemberBytes: INTEGER; ABSTRACT;

The return value is a size, in bytes.

TCollection.MemberBytes does nothing. In subclasses, MemberBytes returns the number of bytes used to store a collection member.

Equals (otherCollection: TCollection): BOOLEAN;

otherCollection is another collection.
The return value tells whether or not the two collections are equal.

TCollection.Equals tests whether this collection contains the same elements as otherCollection.

StartEdit (withSlack: INTEGER);

withSlack is the new value for holeStd.

TCollection.StartEdit changes the value of holeStd to withSlack. The next edit call creates a hole of size withSlack, so that subsequent edit calls act more quickly. Holes that form from edit calls are not closed until the next call to collection.StopEdit.

StopEdit;

TCollection.StopEdit removes the hole from the collection and sets holeStd to 0 (zero). Any subsequent edit call that has no hole removes any hole it forms, until the next call to StartEdit.

InsManyAt (i: LONGINT; otherCollection: TCollection; index, howMany: LONGINT);

i is an index number for an element of the collection.
otherCollection is a collection that contains the elements to be inserted.
index is an index number for an element of the otherCollection.
howMany is the number of elements to be inserted.

TCollection.InsManyAt inserts elements from a collection into this collection.

InsNullsAt (i, howMany: LONGINT);

i is an index number for an element of the collection.
howMany is the number of null elements to be inserted.

TCollection.InsNullsAt inserts null elements into the collection after the given point.

PRIVATE METHODS (Only called by subclasses):

CheckIndex (index: LONGINT);

index is an index number for an element of the collection.

TCollection.CheckIndex tests to be sure that index is greater than 1 and less than or equal to collection.size, if the dubug menu CHECK LIST INDICES command was chosen.

AddrMember (i: LONGINT): LONGINT;

i is an index number for an element of the collection.
The return value is the address of a member of the collection.

TCollection.AddrMember returns a value that can be interpreted as a pointer to a given element of the collection. The address is valid only momentarily, as collections (like all objects) are stored dynamically, and can move at any time.

CopyMembers (dstAddr, startIndex, howMany: LONGINT);

dstAddr is an address.
startIndex is the index number for the first element of this collection to be copied.
howMany is the number of elements to be copied.

TCollection.CopyMembers copies elements from the collection to dstAddr.

EditAt (atIndex: LONGINT; deltaMembers: INTEGER);

atIndex is the index number for the place in the collection where the edit is to occur.
deltaMembers is the number of members to insert or delete.

TCollection.EditAt adds or deletes members of the collection. If deltaMembers is greater than 0, the members are added. If deltaMembers is less than 0, members are deleted. This method does all the work (resizing the collection, moving members around, relocating the hole) except actually copying members into the collection. It is called by other edit methods.

ResizeColl (membersPlusHole: INTEGER);

membersPlusHole is the capacity of the collection after this method is called. Its value is equal to collection.size plus the hole.

TCollection.ResizeColl changes the amount of space allocated to this collection to membersPlusHole*memberBytes.

ShiftColl (afterSrcIndex, afterDstIndex, howMany: INTEGER);

afterSrcIndex is a position in the collection.
afterDstIndex is a position in the collection.
howMany is a number of elements.

TCollection.ShiftColl moves the hole by moving howMany elements right afterSrcIndex to be right afterDstIndex.

NOTES:

1. TCollection defines the basic structure for all groups of objects used in the ToolKit. This includes files, arrays, lists, and strings.

2. TCollection is an abstract class; no objects of type TCollection are ever created. Instead, you create objects of the subclasses of TCollection: TArray, TList, TFile, and TString.
3. collection.size is the number of actual members, not counting the hole.
4. You can ignore the hole when using most methods. However, if you use the private methods, you must take the hole into account.
5. Since collections may be given more space than is needed to hold the data in them, there may be an unused "hole" somewhere in the storage block. The fields holeStart and holeSize specify (in member-sized units) the starting index of the hole and the length of the hole. When holeSize is zero, there is no hole. If members are added when there is no hole, the storage block is expanded to allow for at least another holeStd members.
6. collections have three modes: *create mode*, *edit mode*, and *static mode*.
7. When you first create an collection, it is in create mode. You define an initialSlack value in the CREATE call. The collection is given enough empty space to hold initialSlack elements. That space is the hole. As you add members, the space in the hole is used for the new members. The amount of space allocated to the collection does not change until you fill up the hole with collection members. You can also call collection.StopEdit, which removes the hole from the collection.
8. When the hole is filled, the collection enters static mode. In static mode, no hole is ever maintained. If you add a member, the collection is copied into a space large enough to hold the additional member. If you delete a member, the extra space is freed.
9. Enter edit mode by calling collection.StartEdit(withSlack). In edit mode, a hole big enough for withSlack elements is initially created. As you add members, the size of the hole decreases. When the hole is entirely filled, the space allocated to the collection is increased, so there is a new hole big enough for withSlack members. If you delete members, the extra space is added to the hole. Call collection.StopEdit to stop editing. Any space taken up by the hole is then freed. The hole is always positioned after the last member inserted or after the position that was occupied by the last member deleted.

CLASS: TCommand

SUPERCLASS: TObject

DEFINED SUBCLASSES: TCutCopyCommand
TPasteCommand

INHERITED DATA FIELDS: none

DATA FIELDS:

R cmdNumber: TCmdNumber; The command number of the menu item that describes the command.

R image: TImage; The image (usually a view) to which the command applies, or NIL if the data is stored in the window.

R undoable: BOOLEAN; TRUE means that this command can be undone.

R doing: BOOLEAN; This value is TRUE if the last call to Perform was in doPhase or redoPhase.

revelation: TRevelation; Determines how much of the selection should be automatically scrolled into a pane when the command is performed. Can be none (no scrolling), some (scroll to show some part of the selection), or all (scroll to display the entire selection, if possible).

W unHiliteBefore: ARRAY [TCmdPhase] OF BOOLEAN;
TRUE tells the ToolKit to remove the highlighting from all selections before Perform(cmdPhase) is called. The defaults are TRUE.

W hiliteAfter: ARRAY [TCmdPhase] OF BOOLEAN;
TRUE tells the ToolKit to add highlighting to all selections after Perform(cmdPhase) is called. The defaults are TRUE.

METHODS YOUR APPLICATION MUST OVERRIDE:

CREATE (object: TObject; itHeap: THeap; itsCmdNumber: TCmdNumber; itsImage: TImage; isUndoable: BOOLEAN; itsRevelation: TRevelation): TCommand;

itsCmdNumber is the command number of a menu command.
itsImage is the image to which the command applies, usually a view, or NIL if the data is stored in the window.
isUndoable indicates whether or not the command can be reversed.
itsRevelation determines how much of the selection is scrolled into view when Perform is called. The value can be revealNone (no scrolling), revealSome (scroll to show some part of the selection), or revealAll (scroll to display the entire selection, if possible). If the

selection is already fully displayed, this field has no effect. See TPanel.RevealRect for more information.

TCommand.CREATE creates an object of type TCommand.

Perform (cmdPhase: TCmdPhase); DEFAULT;

cmdPhase is either doPhase, undoPhase, or redoPhase. The first time Perform is called, the cmdPhase is doPhase. After that, undoPhase and redoPhase alternate.

Perform is called when the command is done, undone, or redone. If the command is implemented using a filter, Perform often just invalidates (see TPanel.InvalRect) the changed areas of the view. If the command is not implemented with a filter, Perform should carry out the action of the command.

METHOD YOUR APPLICATION WILL CALL:

CREATE (object: TObject; itHeap: THeap; itsCmdNumber: TCmdNumber; itsImage: TImage; isUndoable: BOOLEAN; itsRevelation: TRevelation): TCommand;

See above.

METHODS YOUR APPLICATION MIGHT OVERRIDE:

Commit; DEFAULT;

Commit is called when a command has been done and can not any longer be undone. (Commit is usually called when another command is given by the user.) See note 4. TCommand.Commit does nothing.

EachVirtualPart (PROCEDURE DoToObject (obj: TObject));

PROCEDURE DoToObject is a procedure that takes an object-reference of type TObject as an argument. DoToObject cannot be a method. It is usually defined as a local procedure within some method.

EachVirtualPart is used in implementing a filter. A *virtual part* is a selectable part of the set of objects. Every virtual part must be an object. DoToObject is a procedure that acts on the virtual part. When taking some action that depends on the action of undoable commands, call image or view.EachVirtualPart, giving as the argument the method you want to apply to the objects. If the last command has been done (that is, if command.Perform was last called with doPhase or redoPhase), command.EachVirtualPart is called. command.EachVirtualPart calls FilterAndDo, which alters the object before calling DoToObject.

You do not have to reimplement TCommand.EachVirtualPart except for commands that act on more than one virtual object at a time.

For example, if the command involves reordering the objects in your data set, implement **commandEachVirtualPart** so that it does the reordering.

When you do not reimplement **TCommandEachVirtualPart**, implement **commandFilterAndDo** for commands that act on one virtual object at a time.

TCommandEachVirtualPart calls **view** or **image.EachActualPart** to get the actual set of data. (You have to implement **EachActualPart**. See the reference sheet for **TView** or **TImage**.) **EachActualPart** must return a list of objects that together make up the data set. **TCommandEachVirtualPart** calls **FilterAndDo** once for each object in the list.

Do not call **commandEachVirtualPart** directly. However, when you are implementing your own version of **commandEachVirtualPart**, you may call **TCommandEachVirtualPart** from your method.

FilterAndDo (actualObj: TObject; PROCEDURE DoToObject(filteredObj:TObject));

actualObj is an actual part of the document.

DoToObject is a procedure that takes an object-reference of type **TObject** as an argument. **DoToObject** cannot be a method.

FilterAndDo is used with filtered commands. It is intended to check **actualObj**, and see if the command would have changed it. If so, **FilterAndDo** should change the object according to the action of the command, call **DoToObject** giving the **filteredObj** as an argument, and then change the object back to its unaltered state. **TCommandFilterAndDo** calls **DoToObject**, but does not filter the object first.

Free;

Free is called by the ToolKit when a command can no longer be done or undone. **TCommandFree** deallocates the command object. Subclasses should override **TCommandFree** if there is anything other than the command object which should be deallocated at the same time.

NOTES:

1. All commands that make significant changes to the document should return an instance of **TCommand** or a descendant of **TCommand** to the ToolKit. See the *Lisa User Interface Standards* for standards.
2. The ToolKit tells the command object to Perform with **cmdPhase** equal to **doPhase**. If the user chooses undo from the Edit menu, and the command object can be undone, the ToolKit tells the command object to Perform with **cmdPhase** equal to **undoPhase**. If the user again requests undo, the

ToolKit tells the command object to Perform with cmdPhase equal to redoPhase. Subsequent undo requests alternate between undoPhase and redoPhase.

3. When another command is sent to the ToolKit, the ToolKit first tells the last command object to Commit (unless the command has been undone) and then to Free. If the last Perform was with undoPhase, the Commit call is omitted.

4. The Commit method is intended for use with commands that can not be easily reversed, but that you want the user to be able to undo. You implement such a command so that the action of the command is not actually carried out the when the command is given. Instead, you make the display appear as if the command was carried out. This is referred to as *filtering* the display. If the command is then undone, you can simply display the true state of the data. If another command is given, call Commit to actually change the data.

5. If the command is implemented using a filter, the action of the command is not carried out by Perform. Perform usually simply invalidates regions of the view that would be changed by the command.

6. All methods that would be affected by the action of filtered commands are implemented so that they call TView.EachVirtualPart. Give a procedure that acts on a TObject-type reference as an argument to TView.EachVirtualPart. If the command has been done (that is, command.Perform was last called with cmdPhase = doPhase or redoPhase), window.FilterDispatch is called. window.FilterDispatch calls command.EachVirtualPart. If the command alters the set of objects displayed in the view, you should reimplement command.EachVirtualPart so that it produces the correct set of virtual parts. TCommand.EachVirtualPart simply passes each object from view.EachActualPart to command.FilterAndDo. If the command alters the individual parts of the view, you should implement command.FilterAndDo so that it checks whether or not the command should have changed that particular object and, if it should have, changes the object so that it appears correctly. The procedure given to EachVirtualPart is then called, with the altered object as its argument. When that procedure completes, the object is changed back to its original state, and FilterAndDo is given the next object.

7. Note that you or a building block must implement Perform, and often also implement Commit, for all your application's commands, except for cutCopyCommand and pasteCommand, whose Commit and Perform methods are implemented in the ToolKit. For cutCopyCommand, you (or a building block) implement DoCutCopy and for pasteCommand you (or a building block) implement DoPaste.

8. Some commands do not appear in a menu in the menu bar. An example of this would be where, in a graphics program, the user selects a box and

then drags it across the window. In that case, the user is essentially giving a move command. According to the *User Interface Standards*, because such a command makes a significant change to the document, it must be implemented using a command object. In addition, there must be a name for the command, so that the Edit menu can tell the user what command can be undone. See note 9 below.

9. To implement a command that does not appear in the menu bar, put the command name in your phrase file with a menu number greater than 100. This is referred to as a *buzzword* menu. You put the name in the phrase file so that you can use it later to display the name of the command in the Undo menu item. When the action that is referred to by the buzzword menu occurs, create a command object for the command from within your application. Call window.PerformCommand(command) where command was created by selection.NewCommand or window.NewCommand.

10. If a command makes a significant change to the document, but you do not want to, or cannot, implement undo, create an instance of TCommand (rather than an instance of a descendant of TCommand) with the unDoable field set to FALSE and implement the command entirely within NewCommand, after first calling window.CommitLast.

11. If commands are not yet clear to you, do not dispair. To fully understand commands and command flow, you will probably need to study some sample applications. Please refer to the ToolKit Segments on commands. After you have studied that, examine the sample programs carefully.

CLASS: TCutCopyCommand

SUPERCLASS: TCommand

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

R	cmdNumber: TCmdNumber;	The command number of the menu item that describes the command.
R	view: TView;	The view to which the command applies, or NIL if the data is stored in the window.
R	undoable: BOOLEAN;	TRUE means that this command can be reversed.
R	doing: BOOLEAN;	This value is TRUE if the last call to Perform was in doPhase or redoPhase.
	revelation: TRevelation;	Determines how much of the selection should be automatically scrolled into a pane when the command is performed. Can be none (no scrolling), some (scroll to show some part of the selection), or all (scroll to display the entire selection, if possible).
W	unHiliteBefore: ARRAY [TCmdPhase] OF BOOLEAN;	TRUE tells the ToolKit to remove the highlighting from all selections before Perform is called.
W	hiliteAfter: ARRAY [TCmdPhase] OF BOOLEAN;	TRUE tells the ToolKit to add highlighting to all selections after Perform is called.

DATA FIELDS: R isCut: BOOLEAN; Tells whether the data is to be cut (TRUE) or just copied to the clipboard (FALSE).

METHODS YOUR APPLICATION MUST OVERRIDE:

CREATE (object: TObject; itHeap: THeap; itsCmdNumber: TCmdNumber; itsImage: TImage; isCutCmd: BOOLEAN): TCutCopyCommand;

itsCmdNumber is the command number of a menu command.
itsImage the image to which the command applies (usually a view), or NIL if the data is stored in the window.
isCutCmd indicates whether this is a cut (TRUE) or copy (FALSE).

TCutCopyCommand.CREATE creates an object of type TCutCopyCommand.

DoCutCopy (clipSelection: TSelection; deleteOriginal: BOOLEAN; cmdPhase: TCmdPhase);

clipSelection, when cmdPhase=doPhase, is a selection in the clipboard that you can replace with a copy of the selectPanel's selection. When cmdPhase=undoPhase or redoPhase, clipSelection is the selection the cutCopyCommand inserted in the clipboard.
deleteOriginal indicates whether the objects are to be cut from the document (TRUE) or just copied to the clipboard (FALSE). It is equivalent to the value of SELF.isCut.
cmdPhase is either doPhase, undoPhase, or redoPhase.

DoCutCopy executes the cut or copy. The first time DoCutCopy is used, the cmdPhase is doPhase. After that, undoPhase and redoPhase alternate. This routine is called by Perform after Perform sets up the clipboard data segment. You must implement DoCutCopy so that it does the following:

- When cmdPhase is doPhase, clone selected objects and put them in the clipboard and, if deleteOriginal is TRUE, delete the selected objects from the document.
- When cmdPhase is undoPhase, and deleteOriginal is TRUE, restore the selected objects to the document. You do not have the change the clipboard; the ToolKit does that for you.
- When cmdPhase is redoPhase, and deleteOriginal is TRUE, delete the original objects from the document. You do not have the change the clipboard; the ToolKit does that for you.

When deleteOriginal is FALSE (that is, this is a copy command), nothing is done by DoCutCopy on undoPhase or redoPhase.

METHOD YOUR APPLICATION MIGHT OVERRIDE:

Commit;

You can override Commit if you want to implement cut so that it uses a filter when Perform is called. In that case, cutCopy.Commit removes the information from the document. If you override this method, you should call TCutCopyCommand.Commit at the beginning of your method.

METHOD YOUR APPLICATION WILL CALL:

CREATE (object: TObject; itHeap: THeap; itsCmdNumber: TCmdNumber; itsView: TView; isCutCmd: BOOLEAN): TCutCopyCommand;

See above.

INTERNAL METHOD:

Perform (cmdPhase: TCmdPhase);

cmdPhase is either doPhase, undoPhase, or redoPhase.

TCutCopyCommand.Perform calls cutCopyCommand.DoCutCopy to execute the cutCopyCommand. The first time Perform is used, the cmdPhase is doPhase. After that, undoPhase and redoPhase alternate. You will have to override Perform if you want it to put a filter over the data rather than performing the cut immediately. In that case, use Commit to call DoCutCopy.

NOTES:

1. See TCommand for general information on command implementation.
2. TCutCopyCommand is used to cut the selected objects from the display and copy them into the clipboard. The objects are always copied into the clipboard, no matter whether the operation was a cut or a copy.
3. Building blocks often override and implement its action. Otherwise, you must override this class to create cut/copy command objects. At a minimum, you must implement DoCutCopy in every subclass you define.
4. See the ToolKit Segments and sample programs for examples of use of the cutCopyCommand.

CLASS: TDialogBox

SUPERCLASS: TWindow

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

R	panels: TList {OF TPanel};	The panels in the dialogBox. There is always at least one. See class TPanel.
	panelTree: TArea;	When the window contains no panels (which happens momentarily during BlankStationary) contains NIL. When the window contains one panel, contains the panel. When the window contains more than one panel, contains a TBranchArea that is the root of a tree of panels.
	dialogBox: TDialogBox;	Always NIL, since this is a dialog box window.
R	selectPanel: TPanel;	The panel with the active selection.
	undoSelPanel: TPanel;	The selectPanel during the last command.
R	clickPanel: TPanel;	The panel where the mouse button was last clicked.
	undoClickPanel: TPanel;	The clickPanel during the last command.
R	selectWindow: TWindow;	The window with the active selection. Either SELF or its window.
	undoSelWindow: TWindow;	The window with the active selection during the last command.
	wmgrID: TWindowID	ORD of the pointer to the Window Manager's grafPort.
R	isResizable: BOOLEAN;	Tells whether or not there is a Resize Box.
	believeWmgr: BOOLEAN;	If TRUE, the Toolkit should believe the window manager's idea of the size of the dialog box; this is FALSE (for example) when

			the application creates the dialogBox object before the dialog box is put on the screen.
		maxInnerSize: Point;	The dialogBox size the user explicitly set by using the grow icon.
		changes: LONGINT;	Not significant for dialogBoxes.
	R	lastCmd: TCommand;	Not significant for dialogBoxes.
		printerMetrics: TPrinterMetrics;	Not significant for dialogBoxes.
		pgSzOK: BOOLEAN;	Not significant for dialogBoxes.
		pgRgOK: BOOLEAN;	Not significant for dialogBoxes.
		panelToPrint: TPanel;	Not significant for dialogBoxes.
		objectToFree: TObject;	Not significant for dialogBoxes.
	R	innerRect: Rect;	Contains the size of the dialog box, excluding the entire frame.
	R	outerRect: Rect;	Contains the size of the dialog box, including the frame.
	R	parentBranch: TBranchArea;	Not significant for dialogBoxes.
DATA FIELDS:		keyResponse: TDlResponse;	What to do with this dialogBox when the user types a key while the box is displayed. From the set dlAccept, dlDismissDialogBox, dlGiveToMainWindow, dlRefuse.
		menuResponse: TDlResponse;	What to do with this dialogBox when the user presses the mouse button in the menu bar while the box is displayed. From the set dlAccept, dlDismissDialogBox, dlGiveToMainWindow, dlRefuse.

downInMainWindowResponse: TDlResponse;
What to do with this dialogBox when the user presses the mouse button in main window while the box is displayed. From the set dlAccept, dlDismissDialogBox, dlGiveToMainWindow, dlRefuse.

freeOnDismissal: BOOLEAN;
If TRUE, the dialogBox object is automatically freed by the ToolKit when the dialogBox is taken down.

METHODS:

CREATE (object: TObject; heap: THeap; itsResizability: BOOLEAN; itsHeight: INTEGER; itsKeyResponse, itsMenuResponse, itsDownInMainWindowResponse: TDlResponse): TDialogBox;

itsResizability indicates whether or not there is a size control box.
itsHeight is the vertical size of the dialog box.
itsKeyResponse is the dialog's response to keystrokes, from the set: dlAccept, dlDismissDialogBox, dlGiveToMainWindow, and dlRefuse.
itsMenuResponse is the dialog's response to the user pressing the mouse button in the menu bar, from the set: dlAccept, dlDismissDialogBox, dlGiveToMainWindow, and dlRefuse.
itsDownInMainWindowResponse is the dialog's response to the user pressing the mouse button in the main window, from the set: dlAccept, dlDismissDialogBox, dlGiveToMainWindow, and dlRefuse.

TDialogBox.CREATE creates an object of type TDialogBox.

Appear;

TDialogBox.Appear displays the dialog box.

BeDismissed;

TDialogBox.BeDismissed removes the dialog box from the display in the default manner, usually by the user pushing the default button.

Disappear;

TDialogBox.Disappear removes the dialog box from the display.

NOTES:

1. Although this class must be subclassed to be used, that is usually taken care of by the Dialog Building Block, which defines class TDialogWindow.
2. Dialog boxes are used to gather command parameters and settings from the user.

3. Your application should never call methods of TDialogBox directly. Call TWindow.PutUpDialogBox and TWindow.TakeDownDialogBox when you need to control a dialog box.
4. To arrange for a dialogBox to be freed automatically when it is taken down, set the field freeOnDismissal to TRUE after creating the dialogBox object.

CLASS: TDocDirectory

SUPERCLASS: TObject

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: none

DATA FIELDS:

window: TWindow;	The window object of this document.
classWorld: TClassWorld;	Contains information on all classes used by this process.

METHOD YOUR APPLICATION MUST OVERRIDE:

CREATE (object: TObject; itsHeap: THeap; itsWindow: TWindow; itsClassWorld: TClassWorld): TDocDirectory;

itsHeap is the document's heap.
itWindow is the window object of this document.
itsClassWorld contains information on all classes used by this process.

TDocDirectory.CREATE creates an object of type TDocDirectory.

NOTES:

1. TDocDirectory is only used by the ToolKit. It is documented here for your information.
2. TDocDirectory is used to create an object that lists all the classes used by a process, and also points to the document's window object.

****CLASS: TDocManager

SUPERCLASS: TObject

DEFINED SUBCLASS: TClipboard

INHERITED DATA FIELDS: none

DATA FIELDS: files:

RECORD

R	volumePrefix: TFilePath;	Desktop Manager volume and prefix of OS files. For example, a possible volumePrefix is -PARAPORT-[D101T35].
R	volume: TFilePath;	Desktop Manager volume of OS files.
R	password: TPassword;	The password assigned by the user to this docManager's files.
	saveExists: BOOLEAN;	Whether Save file is known to exist and seem readable.
W	shouldSuspend: BOOLEAN;	If TRUE (the default), the docManager should create and read suspend files.
W	shouldToolSave: BOOLEAN;	If TRUE (default is FALSE) and the tool is opened (rather than a document being opened), it is allowed to create or read files.

END;

dataSegment:

RECORD

refnum: ARRAY [1..maxSegments] OF INTEGER;
File System reference numbers (refnums) of the data segments used.

R	preludePtr: TPPrelude;	A pointer to the prelude of the data segment, where certain information is kept, such as a handle that enables the process to find the window object and the

		formatting for printer information.
	R changes: LONGINT;	Number of changes since the last suspend or save.
	END;	
R	docHeap: THeap;	The heap, which begins after the prelude in the data segment.
R	window: TWindow;	The window object for this docManager.
WR	pendingNote: INTEGER;	If <> 0, NOTE alert that should be displayed when the document contolled by this docManager is next activated.
R	openedAsTool: BOOLEAN;	TRUE if the tool was opened instead of a separate document.

METHOD YOUR APPLICATION MUST OVERRIDE:

CREATE (object: TObject; itsHeap: THeap; itsPathPrefix: TFilePath): TDocManager;

itsPathPrefix is the name of the volume containing the Desktop Manager. Supplied by the ToolKit.

TDocManager.CREATE creates a docManager object.

NewWindow (heap: THeap; wMgrId: TWindowId): TWindow; DEFAULT;

heap is the document heap.
wMgrID is the identifier assigned to your window by the Window Manager.

NewWindow creates a window object for the document. You should have this method call CREATE for your descendant of TWindow.

METHOD YOUR APPLICATION MIGHT OVERRIDE:

DfltHeapSize: LONGINT; DEFAULT;

The return value is the default heap size.

DfltHeapSize returns the default heap size. This value is docDsBytes, a constant defined in UABC, currently 5120. You might want to override this method to increase that value.

OTHER METHODS:

Complete (allIsWell: BOOLEAN);

allIsWell tells whether or not there is an error. TRUE indicates there is no error.

Complete signals that the document controlled by this docManager is finished processing.

Close (afterSuspend: BOOLEAN);

afterSuspend tells whether or not this docManager was suspended. TRUE indicates it was suspended.

Close closes the docManager's files. afterSuspend is TRUE if the document was set aside by the user. TDocManager.Close calls TDocManager.CloseFiles.

CloseFiles;

TDocManager.CloseFiles closes the files controlled by this docManager. You can override this method if your application has its own files that must be closed. In that case, end your method by calling SUPERSELF.CloseFiles.

Open (VAR error: INTEGER; wmgrID: TWindowID; VAR OpenedSuspended:BOOLEAN);

error is an error number.
wMgrID is the identifier assigned to your window by the Window Manager.
OpenedSuspended is TRUE if the document was not closed, but only suspended (set aside).

Open opens the files controlled by this docManager. You should never override this method.

OpenBlank (VAR error: INTEGER; wmgrID: TWindowID);

error is an error number.
wMgrID is the identifier assigned to your window by the Window Manager.

OpenBlank opens new files for a new document. You might want to override this method if you have non-standard files to open.

OpenSaved (VAR error: INTEGER; wmgrID: TWindowID);

error is an error number.
wMgrID is the identifier assigned to your window by the Window Manager.

OpenSaved opens the saved files managed by this docManager. You might want to override this method if you have non-standard files to open.

OpenSuspended (VAR error: INTEGER; wmgrID: TWindowID);

error is an error number.
wMgrID is the identifier assigned to your window by the Window Manager.

OpenSuspended opens suspended files managed by this docManager. You might want to override this method if you have non-standard files to open.

RevertVersion (VAR error: INTEGER; wmgrID: TWindowID);

error is an error number.
wMgrID is the identifier assigned to your window by the Window Manager.

RevertVersion reverts to the most recently saved versions of the files controlled by this docManager. Any changes made since the files were last saved are lost. You should never override this method.

SaveVersion (VAR error: INTEGER; volumePrefix: TFilePath; andContinue: BOOLEAN);

error is an error number.
volumePrefix is the ToolKit-assigned prefix for this docManager's files.
andContinue indicates whether or not the files are closed after being saved. TRUE leaves the files open.

SaveVersion saves the files controlled by this docManager. You might want to override this method in order to save extra files. If you do, save your files in two stages. First, write out your files under temporary names. Then call SUPERSELF.SaveVersion. The ToolKit writes out its files under temporary names and, when that operation returns successfully, renames the temporary files to the permanent names, thus destroying the old versions. Once SUPERSELF.SaveVersion returns successfully, change the names of your temporary files to their permanent names.

Suspend (VAR error: INTEGER);

error is an error number.

Suspend suspends the document controlled by this docManager. Suspension is equivalent to setting aside a document. You might want to override this method to put your extra files in some special state.

Bind; DEFAULT;

Bind inserts the data segments controlled by this docManager into memory.

Unbind; DEFAULT;

Unbind removes the data segments controlled by this docManager from memory.

NOTES:

1. The application must declare a subclass of TDocManager.
2. The docManager object keeps tracks of files and datasegments used by the process. Every ToolKit process must have a docManager for each open window or icon.
3. The docManager handles document suspension, save, and restore commands. It saves the document by writing the contents of the heap to a file. When the document is later reopened, the saved file is read back into the heap. If you want to save your files in another way, you will have to reimplement SaveVersion, Suspend, OpenSaved, and OpenSuspend. See the interface unit UABC for more information on these methods.
4. Generally, applications do not need to reimplement any routines except for CREATE and NewWindow, and also do not need to deal with any of the data fields of TDocManager.
5. If you create additional files that need to be managed, you might need to reimplement OpenBlank, OpenSaved, and OpenSuspended.
6. By default, when your process is not active, and it receives a note, the active process is interrupted and the note is displayed. If you want, you can intercept the note, and set pendingNote to TRUE. The note is displayed when your process is activated by the user.
7. You might need to change the default values of shouldSuspend and shouldToolSave, if the user can open the tool directly. Sometimes you do not need to create a suspend file because you do not need to save any state information. In that case, you might want to set shouldSuspend to FALSE. When that is done, the effect of setting aside the tool is the same as putting the tool away. As there are no documents to suspend, it makes more sense to put the tool away. In addition, you might want to set shouldToolSave to TRUE. In that case, when the tool is put away, the

data segments it uses are saved with it. When the user opens the tool again, the data segments are re-loaded, and the tool is in the state in which it was left. See the following notes for more information.

8. Set **shouldSuspend** and **shouldToolSave** to FALSE if you do not want to save any state information when the user sets aside the tool and turns off the Lisa.
9. Set **shouldSuspend** and **shouldToolSave** to TRUE if you want to save state information when user saves the tool. Note that in this case, the user will not be able to go back to a blank document unless you provide a menu command to do so.

CLASS: TFile.

SUPERCLASS: TCollection

DEFINED SUBCLASSES: none

INHERITED DATA FIELD: size: LONGINT; The number of bytes in the disk file.

Other fields are inherited, but are irrelevant for files.

DATA FIELDS:
path: TFilePath; The pathname of this file.
password: TPassword; The password for this file.
scanners: TList {OF Scanner}; The fileScanners for this file.

METHODS:

CREATE (object: TObject; heap: THeap; itsPath: TFilePath; itsPassword: TPassword): TFile;

itsPath is the file system pathname for this file.
itsPassword is the password for this file.

TFile.CREATE creates an object of type TFile. If itsFilePath is in the catalog, the ToolKit finds out the size of the file, and puts that value in file.size. If the disk file does not exist, file.size is set to 0.

Free; OVERRIDE;

Free closes the file and releases the heap space used for this file. It also frees this file's fileScanners.

Clone (heap: THeap): TObject; OVERRIDE; {Illegal}

heap is the document's heap.
The return value is a file.

You should never use this method. It is called by the ToolKit occasionally. TFile overrides the TObject implementation of Clone so that Clone checks file system errors.

MemberBytes: INTEGER; OVERRIDE;

TFile.MemberBytes always returns 1.

Scanner: TFileScanner;

Scanner creates a fileScanner for this file.

ScannerFrom (firstToScan: LONGINT; manip: TAccesses): TFileScanner;

firstToScan is the place in the file where reading or writing should begin, measured in bytes.
manip is fRead or fWrite.

ScannerFrom scans the file from the given point.

ChangePassword (VAR error: INTEGER; newPassword: TPassword);

error is an error number.
newPassword is the new password for the file.

ChangePassword attempts to give the file a new password. If the operation is successful, file.password is changed to the new value and error equals 0 (zero). If the operation is not successful, error indicates the reason.

Delete (VAR error: INTEGER);

error is an error number.

Delete attempts to delete the file. If the operation is successful, error equals 0 (zero). If the operation is not successful, error indicates the reason.

Exists (VAR error: INTEGER): BOOLEAN;

error is an error number.
The return value indicates whether or not the file exists.

Exists attempts to find out if the file exists. If the file does exist, the return value is TRUE; otherwise the value is FALSE. If file.Exists cannot find out whether or not the file exists, error indicates the reason.

WhenModified (VAR error: INTEGER): LONGINT;

error is an error number.
The return value indicates when the file was last modified.

WhenModified attempts to find out when the file was last modified. If the operation is successful, error equals 0 (zero). If the operation is not successful, error indicates the reason.

Rename (VAR error: INTEGER; newFileName: TFilePath);

error is an error number.
newFileName is a pathname.

Rename attempts to change the name of the file. If the operation is successful, error equals 0 (zero). If the operation is not successful, error indicates the reason.

VerifyPassword (VAR error: INTEGER; password: TPassword): BOOLEAN;

error is an error number.
password is a string of type TPassword.
The return value indicates whether the password is correct.

VerifyPassword finds out whether or not password is valid for the file. This ignores SELF.password.

NOTES:

1. This class is used to manipulate the catalog entry for disk files.
2. In order to read or write the file, create a fileScanner. See the reference sheet on TFileScanner for more information.

CLASS: TFileScanner

SUPERCLASS: TStringScanner

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

R	collection: TCollection;	The file being scanned.
R	position: LONGINT;	The current position of the scanner. The scanner position is always between, before, or after members: 0=before first, size+1=after last.
R	increment: INTEGER;	The change in position for every scan: always +1 for fileScanners.
	scanDone: BOOLEAN;	When this is TRUE, the next Scan call returns FALSE, signalling the end of the scan. The VAR parameter of the Scan method, which normally contains the next object in the file, is unchanged after the Scan call.
R	atEnd: BOOLEAN;	TRUE if the end of the list is imminent, so that the next Scan call will return FALSE.
R	actual: LONGINT;	The number of bytes transferred in the last transfer operation.

DATA FIELDS:

acesseses: TAccesses;	The set of allowable accesses to the file, from the set (fRead, fWrite, fAppend, fPrivate).
refnum: INTEGER;	The file system reference number for this file.
error: INTEGER;	The first error or warning encountered.

METHODS:

CREATE (object: TObject; itsFile: TFile; manip: TAccesses):
TFileScanner;

itsFile is a file object.
manip is a set of allowable accesses to the file, from the set (fRead, fWrite, fAppend, fPrivate). More than one can be used at a time.

TFileScanner.CREATE creates an object of type TFileScanner, which is used to access itsFile. itsFile must exist before a fileScanner can be created for it; however, you can use a TFile.Create call in your TFileScanner.Create call, which will create the necessary file object.

Allocate (slack: LONGINT); OVERRIDE;

slack is the amount of space added to the OS file, in bytes.

Allocate adds empty space to the OS file, so that subsequence write operations work more quickly.

Append (character: CHAR); OVERRIDE;

character is a new member of the file.

TFile.Append can only be called when the fileScanner is at the end of the file; that is, when fileScanner.position is equal to file.size. Append adds character to the end of the file. position is adjusted by adding 1, so that the fileScanner remains at the end of the file.

Close; OVERRIDE;

Close closes the OS file scanned by this fileScanner object. The file object and the fileScanner object are not deallocated; you can use the fileScanner.Open method to reopen the OS file.

Compact; OVERRIDE;

Compact rewrites the file so that it uses the minimum amount of space necessary. Any unused disk pages are returned.

Delete; OVERRIDE;

TFile.Delete can only be called when the fileScanner is at the end of the file; that is, when fileScanner.position is equal to file.size. Delete deletes the last character in the file. position is adjusted by subtracting 1.

DeleteRest; OVERRIDE;

Delete deletes all characters after the current position. position is unchanged.

Done; DEFAULT;

Done signals that you are finished using this fileScanner. The next call to fileScanner.Scan returns FALSE, as if the end of the file had been reached. Unlike when the end of the file is reached, however, the returned character in Scan remains unchanged.

Free; OVERRIDE;

Free deallocates this fileScanner object. If this is the last fileScanner for the file, file.Free is called.

FreeObject; OVERRIDE;

Free deallocates this fileScanner object and closes the OS file accessed by this fileScanner. The disk file is closed.

Obtain: CHAR; OVERRIDE;

The return value is a character from the file.

Obtain returns the current character.

Open; OVERRIDE;

Open opens an OS file previously closed by fileScanner.Close. The file must have been closed by this fileScanner.

Replace (character: CHAR); OVERRIDE;

character is a character that replaces the current character.

Replace removes the current character from the file and replaces it with the given character. The new character is now the current character.

Scan (VAR nextChar: CHAR): BOOLEAN; OVERRIDE;

nextChar is the next character from the file.
The return value is FALSE if the end of the file has been reached or fileScanner.Done has been called.

Scan begins at the beginning of the file and, each time it is called, returns the next character. The returned character is referred to as the *current* character. The value of Scan is TRUE until Scan is called after the last character in the file is reached, or until fileScanner.Done has been called. If the end of the file was reached, and Done was not called, the value of nextChar is NIL. If Done was called, nextChar keeps the last value it had. When method Done is called or when the scanner reaches the end of the file, the scanner is freed at the next call to scanner.Scan.

Seek (newPosition: LONGINT); OVERRIDE;

newPosition is the location to which you wish the scanner to

move. filePos = 0 is the beginning of the file.

Seek moves the current scan position to newPosition. It transfers no data.

Skip (deltaPos: LONGINT); OVERRIDE;

deltaPos is the number of bytes you wish the scan position to move within the file. A positive value moves the position toward the end of the file; a negative value moves the position toward the beginning of the file.

Skip moves the scan position deltaPos bytes forward or backward within the file. It transfers no data.

XferRandom (whichWay: xReadWrite; pFirst: Ptr; numBytes: LONGINT; mode: TIOMode; offset: LONGINT); OVERRIDE;

whichWay is either xRead, xWrite or xSkip.
pFirst points at the first byte to read data into or write data out of. It is ignored with xSkip.
numBytes is the number of bytes read, written or skipped.
mode is mAbsolute, mRelative or mSequential.
offset is a byte position from the beginning of the file when mode is mAbsolute, or from the current file.position if mRelative.

XferRandom reads, writes or skips numBytes bytes of the file, starting at the indicated position. All the other I/O methods call this method. XferRandom calls the Lisa OS directly. Reading and writing are always done left to right (from the beginning towards the end) regardless of the scanDirection.

XferSequential (whichWay: xReadWrite; pFirst: Ptr; numBytes: INTEGER); OVERRIDE;

whichWay is either xRead, xWrite or xSkip.
pFirst points at the first byte to read data into or write data out of. It is ignored when whichWay equals xSkip.
numBytes is the number of bytes read, written or skipped.

XferSequential reads, writes or skips the next numBytes bytes of the file. Reading and writing are always done left to right (from the beginning towards the end) regardless of the scanDirection.

XferPString (whichWay: xReadWrite; pStr: TPString);

whichWay is either xRead, xWrite or xSkip.
pStr is a pointer to a string to read or write.

XferFile reads a string pStr^ from a file or writes it to a file. If whichWay is xRead, the destination string must be long enough. The entire string or file is transferred, regardless of the current fileScanner.position.

FAILURE CONDITIONS: Reported as Lisa OS error codes in the field **fileScanner.error**.

NOTES:

1. A **fileScanner** is used to access a stored data file, to scan it, and to manipulate its individual elements.
2. Before you can use a **fileScanner** to access a file, the file must be opened by TFile.CREATE.
3. The Refnum variable is used by the ToolKit and the Lisa OS and should not be altered.
4. The Actual variable contains the number of bytes of data most recently transferred to or from the file. It is the same as the number requested unless there is an error. In that case, **fileScanner.error** is greater than 0 (zero).
5. **fileScanner.error** contains the code of the first error (or the first warning, if there was no error) encountered since the error variable was set to 0. Warning codes are negative; error codes are positive.
6. The Xfer methods are designed so that the same procedure may transfer data to or from a file.
7. You may have more than one **fileScanner** open on the same file at the same time.
8. WARNING: No checking is done that sufficient space has been allotted for data reads at **pFirst^** or **pStr^**.
9. The application does not need to use TFileScanner to read and write ToolKit document files; the application's **docManager** does that for you.

CLASS: THeading

SUPERCLASS: TImage

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:
R extentLRect: LRect; The size of the entire image.
R view: TView; The view to which this header or footer is attached.
allowMouseOutside: BOOLEAN;
Not significant for headings.

DATA FIELDS:

printManager: TPrintManager; The printManager for the view with which this heading is associated.

pageAlignment: TPageAlignment;
From the set: aTopLeft, aTopCenter, aTopRight, aBottomLeft, aBottomCenter, aBottomRight.

offsetFromAlignment: LPoint; The vertical and horizontal offset from the headings home edge or corner. See the notes below.

oddOnly: BOOLEAN; If TRUE, this header or footer is printed only on odd numbered pages.

evenOnly: BOOLEAN; If TRUE, this header or footer is printed only on even numbered pages.

minPage: LONGINT; The first page to have this header or footer.

maxPage: LONGINT; The last page to have this header or footer.

METHODS:

ChangePageAlignment (newPageAlignment: TPageAlignment);

newPageAlignment is a new value for heading.pageAlignment.

THeading.ChangePageAlignment installs newPageAlignment as the new page alignment for the heading.

AdjustForPage (pageNumber: LONGINT; editing: BOOLEAN); DEFAULT;

pageNumber is the number of the page currently being prepared for printing.
editing indicates whether or not this method is being called from the Headings and Margins dialog, or a similar application-defined dialog.

THeading.AdjustForPage does nothing. You can reimplement it so that it makes any necessary modifications to the heading that should

be made before printing it on the given page. AdjustForPage is called just before the heading is printed on the page.

Draw;

THeading.Draw draws the heading in the pageView.

LaunchLayoutBox (view: TView): TImage;

view is a view in which layout will take place.
The return value is an image, usually a layout box.

THeading.LaunchLayoutBox does nothing. If you want the user to be able to manipulate the headings of the printed version of your view using the Headings and Margins dialog, you must redefine this method. The simplest way of doing so is to return TPageLayoutBox.CREATE (see the interface of UDialog for particulars). If you want editing to be possible, you need to define your own subclass of TPageLayoutBox, and have LaunchLayoutBox return and instance of that class.

LocateOnPage (editing: BOOLEAN);

editing indicates whether or not this method is being called from the Headings and Margins dialog, or a similar application-defined dialog.

You should never need to override this method.
THeading.LocateOnPage places the heading properly on the page.

OffSetBy (deltaLPt: LPoint);

deltaLPt a vertical and horizontal change, expressed as a point relative to (0,0).

OffSetBy shifts the heading's extentLRect by the values given in deltaLPt. If your subclass of THeading has data structures in it that use view-relative locations, you should redefine OffSetBy so that those locations get adjusted. Call SUPERSELF.OffSetBy after you do your application-specific offsets.

NOTES:

1. THeading defines header, footer, and margin images for use in printing.
2. Applications rarely need to subclass or to refer directly to instances of THeading.
3. Unlike with most images, heading.extentLRect only defines the size of the heading. The position of the heading is determined by heading.pageAlignment and heading.offsetFromAlignment.
4. The value in heading.pageAlignment determines, in general, where on the page the heading is located. A heading can have a pageAlignment that puts the base point for the heading at the top center, top left, top right, bottom center, bottom left, or bottom right of the page. The values

contained in offsetFromAlignment are used to offset the base point of the heading from the given pageAlignment point.

5. Where the base point of the heading is located depends on the pageAlignment point. The base point is always the point in the heading coresponding to the pageAlignment point. For example, if the pageAlignment is aTopLeft, the base point of the heading is its top left corner. If the pageAlignment is aTopCenter, the base point of the heading is its top center. In effect, the pageAlignment defines a justification. A pageAlignment of aTopCenter or aBottomCenter defines a heading that is always centered around the point defined by pageAlignment and offsetFromAlignment. A pageAlignment of aTopRight or aBottomRight defines a heading that has its right edge fixed, regardless of the length of the heading.
6. Because pageAlignment defines a point on the edge of a page, and many printers cannot print on the edges of pages, a heading generally has a non-zero value in offsetFromAlignment.
7. headings can be determined dynamically, so their size can change from page to page. AdjustForPage is used to get the heading's extent correct (and its contents as desired); LocateOnPage offsets the heading to the position on the page indicated by its pageAlignment, offsetFromAlignment, and extentLRect.
8. ToolKit applications generally use TLegendHeading, defined in the Dialog Building Block to produce headings. If you use that, the user can use a menu command to call up a dialog box that allows definition of headers. TLegendHeading headings can only be one line long. See the documentation on the Dialog Building Block for more information.

CLASS: TImage

SUPERCLASS: TObject

DEFINED SUBCLASSES: TView
THeading

INHERITED DATA FIELDS: none

DATA FIELDS:

R extentLRect: LRect; — The bounding box of the entire image, in view coordinates.

R view: TView; — The view in which this image appears.

allowMouseOutside: BOOLEAN; — FALSE by default. When FALSE, and the mouse point is outside the image, the point is converted to the closest point within the image. If TRUE, the point is not converted. In that case, mouse points can be outside extentLRect and, particularly, can be negative. This feature exists primarily for use with sidebands. Note that your program must be prepared to handle mousepoints outside extentLRect.

METHODS YOUR APPLICATION MUST OVERRIDE:

CREATE (object: TObject; heap: THeap; itsExtentLRect; itsView: TView): TImage;

itsExtent is the size of the image.
itsView is the view in which this image appears.

TImage.CREATE creates an object of type TImage. Normally, your application does not deal directly with TImage or its methods; you need to subclass TView, and implement a CREATE method for your subclass.

Draw;

You should implement a Draw method in each descendant of TImage to draw the image. Your Draw method should assume that thePad is set up to draw in one pane. (See TPad in Chapter 3 for an explanation of thePad.)

MouseRelease; DEFAULT;

You may write a MouseRelease routine that takes some application-dependent action when the mouse button is released. TImage.MouseRelease calls selection.MouseRelease.

MouseTrack (mPhase: TMousePhase; mouseLPt: LPoint); DEFAULT;

mphase is the mouse state: mPress, mMove, or mRelease. mouseLPt is the view-relative point where the pointer is located.

TImage.MouseTrack dispatches mouse events to MousePress, MouseMove, or MouseRelease.

ReactToPrinterChange; DEFAULT;

ReactToPrinterChange is meant to allow an image to react to a change in the choice of properties of the printer for which the document is formatted. TImage.ReactToPrinterChange does nothing. (This method is implemented for you by TView.)

RecalcExtent; DEFAULT;

RecalcExtent is intended to instruct an image to recompute its extentLRect. TImage.RecalcExtent does nothing.

Resize (newExtent: LRect); DEFAULT;

newExtent is the new image size.

Resize sets the extentLRect of the image to newExtent.

SeesSameAs (image: TImage): BOOLEAN; DEFAULT;

image is an object of type TImage or a descendant of TImage. The return value indicates whether or not image is the same as SELF.

TImage.SeesSameAs compares image with SELF. If they are the same object, SeesSameAs returns TRUE. You can re-implement SeesSameAs to return TRUE whenever you think that is sensible. For example, you might want to return TRUE if the two images are derived from the same set of objects.

FAILURE CONDITIONS: none

NOTES:

1. You do not usually deal directly with TImage. You normally subclass TView and then use the subclass to create the application's view, or one of the application's views.
2. An image defines a portion of a view.
3. The image's extentLRect describes, in 32-bit view coordinates, the bounding box of the portion of the view occupied by the image.

4. Aside from the subclasses of TImage listed above, subclasses of TImage are used extensively in the text and dialog building blocks. Those subclasses include: TTextImage and TParaImage from the text building block, and TDialogImage, TDialog, and a number of other dialog components from the dialog building block. See the documentation of those building blocks for more information.

CLASS: TList

SUPERCLASS: TCollection

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

R size: LONGINT;	The number of real elements in this list, not counting the hole. This is a LONGINT for the benefit of huge collections, such as remote data bases. It is always in the INTEGER range for instances of TList.
dynStart: INTEGER;	The number of bytes from the class pointer to the dynamic data area.
holeStart: INTEGER;	0 means hole at the beginning; value of size means hole at the end.
holeSize: INTEGER;	The initial size of the hole, measured in a number of members.
holeStd: INTEGER;	How much to grow the collection by if the holeSize goes to 0.

DATA FIELDS: none

METHODS:

CREATE (object: TObject; itsheap: THeap; initialSlack: INTEGER): TList;

initialSlack is the size of the initial hole, in member-sized units.

TListCREATE creates a new object of type TList. initialSlack is a hole for list members. Because the list is created with a hole, the insert methods can be used to initialize the list, without allocating any extra space.

At (i: LONGINT): TObject; DEFAULT;

i is an ordinal position in the list.
The return value is an element from the list.

At returns the element at the given position in the list. Subclasses generally override At.

Debug (numLevels: INTEGER; memberTypeStr: S255); OVERRIDE;

numLevels is the number of levels of detail printed.
memberTypeStr is only relevent for arrays.

Debug is called by the ToolKit debugger to print details about this list. numLevels determines how many levels of detail are printed. When numLevels is:

0 (zero), the debugger prints just the class of list;

1, the debugger also prints size of list;

2, the debugger also prints a compacted list of member classes

greater than or equal to 3, the debugger also prints class, size, and calls Debug(numLevels-1) on members of the list.

DebugMembers;

DebugMembers is present in debugging version only; it prints elements on the alternate screen.

DelAll (freeOld: BOOLEAN); DEFAULT;

freeOld indicates whether or not the objects in the list are to be freed.

DelAll deletes all the elements of a list. If freeOld is TRUE, the objects to which the list element point are also deallocated by calling Free. The list itself is not deleted.

DelAt (i: LONGINT; freeOld: BOOLEAN); DEFAULT;

i is an ordinal position in the list.
freeOld indicates whether or not the original object is to be deleted.

DelAt deletes the element at position i. If freeOld is TRUE, the object to which the list element points is also deallocated by calling Free.

DelFirst (freeOld: BOOLEAN);

freeOld indicates whether or not the object in the list is to be freed.

DelFirst deletes the element at the beginning of the list. If freeOld is TRUE, the object to which the list element points is also deallocated by calling Free.

DelLast (freeOld: BOOLEAN);

freeOld indicates whether or not the object in the list is to be freed.

DelLast deletes the element at the end of the list. If freeOld is TRUE, the object to which the list element points is also deallocated by calling Free.

DelManyAt (i: INTEGER; howMany: INTEGER; freeOld: BOOLEAN);
DEFAULT;

i is an ordinal position in the list.
howMany indicates the number of list elements to be deleted.
freeOld indicates whether or not the objects in the list are to be freed.

DelManyAt deletes a number of elements from the list. The first

deleted element is at position i. If freeOld is TRUE, the objects to which the list elements point are also deallocated by calling Free.

DelObject (object: TObject; freeOld: BOOLEAN);

object is an object.
freeOld indicates whether or not the object in the list is to be freed.

DelObject deletes all occurrences of object in the list. If ffree is TRUE, the object is deallocated by calling Free.

Each (PROCEDURE DoToObject(object: TObject)); DEFAULT;

DoToObject is a PROCEDURE that takes an object as its argument.

Each applies the given PROCEDURE to each element in the list.

First: TObject; DEFAULT;

First returns the first element in the list.

InsAt (i: INTEGER; object: TObject) ABSTRACT;

i is an ordinal position in the list.
object is an object.

InsAt inserts an element in the list. The newly inserted element occupies position i.

InsFirst (object: TObject); DEFAULT;

object is an object.

InsFirst inserts an element at the beginning of the list. The newly inserted element occupies the first position in the list.

InsLast (object: TObject);

object is an object.

InsLast inserts an element at the end of the list. The newly inserted element occupies the last position in the list.

Last: TObject; DEFAULT;

Last returns the element at the end of the list.

ManyAt (i, howMany: LONGINT): TList;

i is an ordinal position in the list.
howMany is a number of elements.

ManyAt returns a list with the elements from the original list beginning at position i and continuing through howMany elements.

MemberBytes: INTEGER;

MemberBytes returns the amount of space, in bytes, taken up by each element in the list. As lists contain only handles, this method always returns 4. (This method is more useful in other types of collections.)

PopLast: TObject;

Pop returns the element at the end of the list, and then deletes that element from the list. element:=SELF.PopLast is equivalent to:

```
popLast:= SELF.Last;
SELF.DelLast(FALSE);
```

Pos (after: LONGINT; object: TObject): LONGINT;

after is an index number in the list.
object is an object that might be in the list.
The return value is the index number of object, or, if object is not found, after.

Pos searches the list from after to the end. If object is found, the index number of object is returned. To seach the entire list, use after of 0 (zero).

PutAt (i: LONGINT; object: TObject; freeOld: BOOLEAN); DEFAULT;

i is an ordinal position in the list.
object is an object handle.
freeOld indicates whether or not the original object is to be deleted.

PutAt deletes the element at position i and replaces it with object. If freeOld is TRUE, the object to which the original list element points is deleted. Subclasses generally override PutAt.

Scanner: TListScanner;

The return value is a listScanner for this list.

Scanner returns an object of class TListScanner, allowing use of listScanner methods. This is equivalent to

```
listScannerFrom (1, scanForward);
```

ScannerFrom (firstToScan: LONGINT; scanDirection: TScanDirection): TListScanner; DEFAULT;

firstToScan is a position in the list.
scanDirection scanForward or scanBackWard.
The return value is a new arrayScanner.

ScannerFrom returns an object of class TListScanner, with listScanner.Position equal to firstToScan minus one (or plus one, if

the scanDirection is scanBackward), so that the first call to listScanner.Scan returns the address of the record at firstToScan.

StartEdit (withSlack: INTEGER);

withSlack is the new value for holeStd.

StartEdit changes the value of holeStd to withSlack so the next edit call creates a hole of size withSlack, so that subsequent edit calls act more quickly.

StopEdit;

StopEdit removes the hole from the list and sets holeStd to 0 (zero). Any subsequent edit call removes any hole it forms.

FAILURE CONDITIONS: Heap can't grow.
Object size > 32K bytes.
Deleting from an empty list.
Subscript out of range.

NOTES:

1. This defines all lists of objects used in the ToolKit.
2. It is faster to traverse a list using Each than it is to use a listScanner. However, insertion and deletion are not possible during traverses using Each. An example of proper use of Each to traverse a list is:

```
FUNCTION PeelGoodBananas(bunch: TList);

  FUNCTION GoodBanana(ob: TObject);
    VAR banana: TBanana;
  BEGIN
    banana:= TBanana(ob);
    If banana.good THEN banana.Peel;
  END;

BEGIN
  bunch.Each(GoodBanana);
END;
```

3. lists have three modes: *create mode*, *edit mode*, and *static mode*.
4. When you first create a list, it is in create mode. You define an initialSlack value in the CREATE call. The list is given enough empty space to hold initialSlack object-references. That space is the hole. As you add members, the space in the hole is used for the new object-references. The amount of space allocated to the list does not change until you fill up the hole with object-references. You can also call list.StopEdit, which removes the hole from the list.
5. When the hole is filled, the list enters static mode. In static mode, no hole is ever maintained. If you add a member, the list is copied into a space large enough to hold the object-reference for the additional

member. If you delete a member, the extra space is freed.

6. Enter edit mode by calling list.StartEdit(withSlack). In edit mode, a hole big enough for withSlack object-references is initially created. As you add members, the size of the hole decreases. When the hole is entirely filled, the space allocated to the list is increased, so there is a new hole big enough for withSlack object-references. If you delete members, the extra space is added to the hole. Call list.StopEdit to stop editing. Any space taken up by the hole is then freed.

CLASS: TListScanner

SUPERCLASS: TScanner

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

R	collection: TCollection;	The list being scanned.
R	position: LONGINT;	The current position of the scanner. The scanner position is always between, before, or after members: 0=before first, size+1=after last.
R	increment: INTEGER;	The change in position for every scan: 1 if scanning forward, -1 if scanning backward.
	scanDone: BOOLEAN;	When this is TRUE, the next Scan call returns FALSE, signalling the end of the scan. The VAR parameter of the Scan method, which normally contains the next object in the list, is unchanged after the Scan call.
R	atEnd: BOOLEAN;	TRUE if the end of the list is eminent, so that the next Scan call will return FALSE.

METHODS:

CREATE (object: TObject; itsList: TList; itsInitialPosition: LONGINT;
itsScanDirection: TScanDirection): TListScanner;

itsList is a list object.
itsInitialPosition is the initial ordinal position in the list.
itsScanDirection is scanForward or scanBackward.

TListScanner.CREATE creates an object of type TListScanner, which is used to access itsList. itsList must exist before a listScanner can be created for it; however, you can use a TList.CREATE call in your TListScanner.CREATE call, which will create the necessary list object.

Append (object: TObject); DEFAULT;

object is a new member of the list.

Append adds object to the list immediately after the current position. position is adjusted by adding 1, so that the next Scan returns the object after the new object.

Delete (freeOld: BOOLEAN); DEFAULT;

freeOld indicates whether or not the object deleted from the list is to be freed.

Delete deletes the object at the current position. position is adjusted by subtracting 1.

DeleteRest (freeOld: BOOLEAN); DEFAULT;

freeOld indicates whether or not the objects deleted from the list are to be freed.

Delete deletes all objects after the current position. position is unchanged.

Done; DEFAULT;

Done signals that you are finished using this listScanner. The next call to listScanner.Scan returns FALSE, as if the end of the list had been reached. Unlike when the end of the list is reached, however, the returned object in Scan remains unchanged.

Free; OVERRIDE;

Free deallocates this listScanner object.

Obtain: TObject; DEFAULT;

The return value is a object from the list.

Obtain returns the current object without advancing position.

Replace (object: TObject; freeOld: BOOLEAN); DEFAULT;

object is a object that replaces the current object.
freeOld indicates whether or not the object deleted from the list is to be freed.

Replace removes the current object from the list and replaces it with the given object. The new object is now the current object.

Scan (VAR nextObj: TObject): BOOLEAN; DEFAULT;

nextObj is the next object from the list.
The return value is FALSE if the end of the list has been reached or listScanner.Done has been called.

Scan begins at the beginning of the list (or the end, if scanDirection is scanBackward) and, each time it is called, returns

the next object. The returned object is referred to as the *current* object. The value of Scan is TRUE until Scan is called after the last object in the list is reached, or until listScanner.Done has been called. If the end of the list was reached, and Done was not called, the value of nextObj is NIL. If Done was called, nextObj keeps the last value it had. When method Done is called or when the scanner reaches the end of the list, the scanner is freed the next time listScanner.Scan is called.

Seek (listPos: LONGINT);

listPos is the location to which you wish the scanner to move. listPos = 0 is the beginning of the list.

Seek moves the current scan position to listPos. It transfers no data.

Skip (deltaPos: LONGINT);

deltaPos is the number of bytes you wish the scan position to move within the list. A positive value moves the position toward the end of the list; a negative value moves the position toward the beginning of the list.

Skip moves the scan position deltaPos elements forward or backward within the list. It transfers no data.

FAILURE CONDITIONS: Attempt to insert, delete or replace after a delete has occurred at the same position.

NOTES:

1. A listScanner is for moving through a list and manipulating the individual elements. list objects are created through class TList.
2. You never allocate a scanner directly; one is allocated when you call list.Scanner.

CLASS: TMarginPad

SUPERCLASS: TPad

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: See TPad.

DATA FIELDS:

R	view: TView;	The view to which this marginPad relates.
R	pageNumber: LONGINT;	The page of the view that prints with this marginPad.
R	bodyPad: TBodyPad;	The bodyPad containing the printable version of the body of the page.

NOTES:

1. A marginPad is the output port for page headings and other page adornments that are printed on a page on the printer or displayed in the image of a page in page preview mode.
2. Application programs usually do not need to subclass or otherwise deal with TMarginPad.

CLASS: TMenuBar

SUPERCLASS: TObject

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: none

DATA FIELDS:

isLoaded: ARRAY [1..maxMenus] OF BOOLEAN;	Each element is TRUE if the corresponding menu has been inserted into the menu bar.
mapping: TArray;	Relates command numbers to menu and item indices.
numMenus: INTEGER;	The number of menus used by this application.
numCommands: INTEGER;	Total number of commands in all the application's menus.

METHODS YOUR APPLICATION MIGHT CALL:

BuildCmdName (destCmd, templateCmd: TCmdNumber; param: TPString);

destCmd is the number of a command in one of the menus. If destCmd does not exist, this method has no effect.
templateCmd is the command number associated with a command name template. If templateCmd does not exist, this method has no effect.
param is a pointer to a string, or NIL.

BuildCmdName replaces the string associated with destCmd with a new command name built with the string pointed to by param and the template associated with templateCmd. templateCmd corresponds to a string in a menu of the form:

text^tempstring^moretext

where text and moretext are optional. (If moretext does not exist, you do not have to give the second carat.) tempstring is replaced with the param string. The carats (^) are removed, and the resulting string is placed in the position in a menu corresponding with destCmd. If param is NIL, the carats are removed, tempstring is left alone, and the resulting string is put in the menu.

Check (cmdNumber: TCmdNumber; checked: BOOLEAN);

cmdNumber is the number of a command in one of the menus.
checked is whether (TRUE) or not (FALSE) the command should be checked off.

Check puts a check in a menu next to the command corresponding to cmdNumber. According to the *User Interface Standards*, the check

indicates that the item is presently in force. The ToolKit does not verify that the command should be checked.

Delete (menuID: INTEGER);

menuID is the identification number for one of the application's menus.

Delete removes the menu from the menu bar.

Draw;

Draw draws the menu. You should call this method after you call menuBar.Insert or menuBar.Delete.

Enable (cmdNumber: TCmdNumber; canBeChosen: BOOLEAN);

cmdNumber is the number of a command in one of the menus. canBeChosen is whether the command should be enabled (TRUE) or disabled (FALSE).

Enable determines whether or not the user is allowed to choose the command corresponding to cmdNumber. When a command is disabled, it appears in dim type in the menu, and the user cannot choose it. When a command is enabled, the command appears in normal type, and the user can choose it. The ToolKit does not make certain that the command should be enabled or disabled.

EndCmd;

EndCmd removes highlighting from the menu bar. See TMenuBar.HighlightMenu.

GetCmdName (cmdNumber: TCmdNumber; pName: TPString): BOOLEAN;

cmdNumber is the number of a command in one of the menus. pName is a pointer to a string location for the command name, or NIL, if you do not want to find out the command name.
The return value indicates whether or not the command was found.

GetCmdName checks to see if cmdNumber corresponds to an existing command. If it does, the return value is TRUE. In addition, if pName is not NIL, GetCmdName puts the command name that corresponds to the command number in the location pointed to by pName.

For example:

```
VAR x:STRING;  {Defines a variable for the command name.}
    .
    .
    .
menuBar.GetCmdName (cmdNumber, @x);
```

HighlightMenu (withCmd: TCmdNumber);

withCmd is the command number of the menu command you are about to process.

HighlightMenu highlights the title of the menu containing the given command. The ToolKit does this for you when the user chooses a command from a menu, uses an Apple key command, or uses a clear key command. You only need to call this method when the command was from somewhere else. First call menuBar.HighlightMenu. Then call TWindow.DoCommand to do your command, rather than creating the command object and calling command.Perform yourself. TWindow.DoCommand turns off highlighting when the command is finished processing. If you do not call TWindow.DoCommand, call menuBar.EndCmd when you are finished processing the command; that turns off highlighting.

Insert (menuID, beforeID: INTEGER);

menuID is the identification number for one of the application's menus that is not presently in the menu bar.
beforeID is the identification number for one of the menus in the application's menu bar.

Insert inserts the menu in the menu bar so that it appears immediately to the left of the menu indicated by beforeID. If beforeID is 0, the menu is placed at the end, to the right of all other menus in the menu bar.

PutCmdName (cmdNumber: TCmdNumber; pName: TPString);

cmdNumber is the number of a command in one of the menus.
pName is a pointer to a string location holding the command name.

PutCmdName replaces whatever is in the menu that corresponds with cmdNumber with the string pointed to by pName.

Unload;

Unload removes the application's entire menu bar.

NOTES:

1. There is only one instance of TMenuBar for any document, and that instance, accessed by the global variable menuBar, is created by the ToolKit.
2. You never subclass TMenuBar.
3. The menuBar controls the application's menus.

4. The methods given here are sometimes called from **window.SetUpMenus**, **window.CanDoCommand** and **selection.CanDoCommand**. Otherwise, you probably will not deal with this class.

5. These methods change the menu or menu bar until either another **menuBar** method effects another change, or a new process is created for the application. Remember that, unless you define your application to manage only one document, all documents belonging to a single application share the same process. Since every process has only one **menuBar** object, these methods change the menu bar for all documents using this process. In addition, since the process generally remains until the Lisa is powered off, even if there are no active documents using the process, you should be careful that the menu bar is in the correct condition when it is displayed.

CLASS: TObject

SUPERCLASS: NIL

DEFINED SUBCLASSES: All other classes are descendants.

INHERITED DATA FIELDS: none

DATA FIELDS: none

METHODS:

CREATE (object: TObject; heap: THeap): TObject; ABSTRACT;

TObject.CREATE is an ABSTRACT method, and does nothing. Each subclass of TObject has its own CREATE method.

Become (obj: TObject);

obj is an object-reference.

Become frees the calling object, by calling object.Free, and reuses the object's handle by making it to refer to obj. For example, ref1.Become(TMyObject.CREATE(object, heap)) frees the object pointed to by ref1 and makes ref1 point at the new object. All references to ref1 now refer to the new object.

Class: TClass;

Class returns the class pointer for this object's class.

Clone (heap: THeap): TObject;

heap is the heap for the new object.

Clone should create a copy of the object, and all its subordinate objects, on heap and returns a handle on the new object. TObject.Clone just calls CloneObject, which copies only the object itself. Override Clone to copy subordinate objects.

CloneObject (heap: THeap): TObject;

heap is the heap for the new object.

CloneObject allocates a copy of the object on heap and returns a handle on the new object. Overriding is rarely necessary.

Convert (fromVersion: Byte); ABSTRACT;

fromVersion is the old object's version number.

Convert is intended to update an object to the current version of the object.

Debug (numLevels: INTEGER; memberTypeString: S255); DEFAULT;

numLevels is the depth of detail printed.

Debug, present only in the debugging version of the ToolKit, prints details about the object on the alternate screen. It is called by the debugger in conjunction with the Fields method. When numLevels is equal to 0 (zero), Debug prints only the object's class. When numLevels is equal to 1, Debug prints the object's class, and the name, class (if any) and value of each field. When numLevels is 2, Debug prints all the information printed for numLevels=1, and also prints the same information about objects referred to by the fields of the original object. As the value of numlevels increases, additional levels of class reference fields are detailed. Overriding of Debug is rarely necessary.

Fields (PROCEDURE Field(nameAndType: S255)); DEFAULT;

Fields is present in the debugging version of the ToolKit only. At the present time (TK 8e), if you want the ToolKit debugger to be able to display the fields of your objects, you must define this method for every class you create. See Chapter 2 for information on writing Fields methods.

Free; DEFAULT;

objectFree should free object and all objects subordinate to object. TObjectFree simply calls FreeObject, which frees object and nothing else. If you create a descendant of TObject that has subordiate objects, you should override Free to free the subordinate objects. End your routine by calling SUPERSELF.FREE.

FreeObject; DEFAULT;

FreeObject removes the associated object and deallocates the heap space used by that object. Overriding is rarely necessary.

Heap: THeap

Heap returns the heap this on which this object is stored. Never override this method.

HeapBytes: INTEGER;

The return value is a number of bytes.

HeapBytes returns the number of bytes used by the object in the heap.

Read (s: TStringScanner);

s is a stringScanner.

Reads the data in the object, without the object's class pointer or the dynamic part, if any, by using the stringScanner to get the object's data.

Write (s: TStringScanner);

s is a stringScanner.

Writes the data in the object, without the object's class pointer and any dynamic part.

JoinClass (newClass: TClass): TObject;

newClass is the object's new class pointer.

JoinClass converts the object to a new class. It is called for you when version conversion is required.

FAILURE CONDITIONS:

Heap can't grow.
Object freed twice.
Tried to do x.Become(y) when x and y were not on the same heap.
Tried to do x.JoinClass(newClass) when the class of x and newClass are unrelated.

NOTES:

1. Class TObject is the ancestor of all other classes.
2. Instances of TObject are never created.
3. All objects are instances of descendants of TObject.

CLASS: TPad

SUPERCLASS: TArea

DEFINED SUBCLASSES: TPane
TBodyPad
TMarginPad

INHERITED DATA FIELDS:

innerRect: Rect;	GrafPort-relative bounds excluding borders.
outerRect: Rect;	GrafPort-relative bounds including border.
parentBranch: TBranchArea;	Not relevant for pads.

DATA FIELDS:

port: GrafPtr;	The grafPort used by this pad.
viewedLRect: LRect;	The portion of view that is displayed in innerRect.
visLRect: LRect;	viewedLRect sect visRgn while focused.
availLRect: LRect;	The larger part of view that fits in a grafPort coordinate Rect.
scrollOffset: LPoint;	The distance scrolled from the view topLeft.
origin: Point;	What to set the grafPort origin to when focused.
cdOffset: LPoint;	What to subtract from coordinates to get grafPort coordinates.
clippedRect: Rect;	Additional clipping to apply when focusing.
padRes: Point;	The pad's resolution, expressed as spots/inch in the pad coordinate space.
viewedRes: Point;	The view's resolution, expressed as spots/inch in the view coordinate space.
scaled: BOOLEAN;	TRUE indicates that the scaleFactor is not 1, which means that coordinate scaling must be used.
scaleFactor: TScaler;	The net scale factor, combining the effects of zooming and the different coordinate systems of the pad and its view.

zoomFactor: TScaler;	The zoom factors; The two zoom factors, one for vertical zoom and one for horizontal zoom, are stored as two Points, one of which holds the numerators, and the other the denominators of the zoom factors.

METHOD YOUR APPLICATION MUST OVERRRIDE:

CREATE (object: TObject; itsHeap: THeap; itsInnerRect: Rect; itsViewedLRect: LRect; itsPadRes, itsViewRes: Point; itsPort: GrafPtr): TPad;

itsInnerRect is the window-relative bounds excluding the border.
itsViewedLRect is the portion of the view visible in the pad.
itsPadRes is the resolution of this pad.
itsViewedRes is the resolution of the view, or other 32-bit space, looked on by this pad.
itsPort is the grafPort.

TPad.CREATE creates an object of type TPad.

METHODS YOUR APPLICATION WILL CALL:

InvalLRect (r: LRect);

r is the rectangle to be redrawn.

TPad.InvalLRect forces a redraw of r at next call to window.Update. r is in view coordinates.

InvalRect (r: Rect);

r is the rectangle to be redrawn.

TPad.InvalRect forces a redraw of r at next call to window.Update. r is in grafPort coordinates.

SetPen (pen: PenState);

pen is a pen state. See the QuickDraw documentation for the available pen states.

TPad.SetPen sets the pen state.

SetPenToHighlight (highTransit: THighTransit);

highTransit is the change in highlighting of the displayed object. The standard choices are: hNone, hOffToDim, hOffToOn, hDimToOn, hDimToOff, hOnToOff, and hOnToDim.

TPad.SetPenToHighlight sets the pen to the state appropriate for applying the indicated highlighting.

The following methods map coordinates from grafPort to view.

DistToLDist (distInPort: Point; VAR lDistInView: LPoint);

distInPort is a distance in grafPort coordinates.
lDistInView is the distance converted to view coordinates

TPadLDistToLDist converts a distance in grafPort coordinates to view coordinates.

PatToLPat (patInPort: Pattern; VAR lPatInView: LPattern);

patInPort is a pattern in grafPort coordinates
lPatInView is the same pattern converted to view coordinates

TPadLPatToLPat converts a pattern from grafPort to view coordinates.

PtToLPt (ptInPort: Point; VAR lPtInView: LPoint);

ptInPort is a point in grafPort coordinates.
lPtInView is the point converted to view coordinates.

TPadLPtToLPt converts a point from grafPort to view coordinates.

RectToLRect (rectInPort: Rect; VAR lRectInView: LRect);

rectInPort is a rectangle in grafPort coordinates.
lRectInView is the same rectangle converted to view coordinates.

TPadLRectToLRect converts a rectangle from grafPort to view coordinates.

The following methods do coordinate mapping from view to grafPort

LDistToDist (lDistInView: LPoint; VAR distInPort: Point);

lDistInView is a distance in view coordinates.
distInPort is the distance converted to grafPort coordinates.

TPadLDistToDist converts a distance in view coordinates to grafPort coordinates.

LPatToPat (lPatInView: LPattern; VAR patInPort: Pattern);

lPatInView is a pattern.
patInPort is the pattern converted to grafPort coordinates.

TPadLPatToPat converts a pattern from view to grafPort coordinates.

LPtToPt (lPtInView: LPoint; VAR ptInPort: Point);

lPtInView is a point in view coordinates.
ptInPort is the point converted to grafPort coordinates.

TPadLPtToPt converts a point from view to grafPort coordinates.

LRectToRect (lRectInView: LRect; VAR rectInPort: Rect);

lRectInView is a rectangle in view coordinates.
rectInPort is the same rectangle converted to grafPort coordinates

TPadLRectToRect converts a rectangle from view to grafPort coordinates.

The following method is used by the ToolKit to move the pad within the view.

OffsetBy (deltaLPt: LPoint);

deltaLPt is this

TPadOffsetBy offsets viewedLRect -- no effect on display.

The following method is used for display.

Focus; OVERRIDE;

TPadFocus sets the grafPort so that subsequenct QuickDraw calls are correctly sent to this pad.

The following methods perform drawing, and allow coordinates to be expressed in view coordinates. They function, for the most part, by converting the coordinates to grafPort coordinates, then invoking the corresponding QuickDraw call.

DrawLArc (verb: GrafVerb; r: LRect; startAngle, arcAngle: INTEGER);

verb indicates what to do with the arc (frame, paint, erase, invert, fill).
r is the bounding box of the arc.
startAngle is the angle of the start of the arc.
arcAngle is the angle of the arc.

TPadDrawLArc draws an arc in the view.

DrawLBits (VAR srcBits: BitMap; VAR srcRect: Rect; VAR dstLRect: LRect; mode: INTEGER; maskRgn: RgnHandle);

srcBits is a bitmap, part of which is copied to this pad.
srcRect is the area in srcBits copied to this pad.
dstLRect is the part of this pad that receives the copy from srcBits.
mode is the transfer mode used to draw in dstLRect. See the QuickDraw documentation for a discussion of transfer modes.
maskRgn is a region in this pad (presumably part of dstLRect) that is to be unchanged by this drawing operation.

TPadDrawLBits converts dstLRect to grafPort coordinates, and then calls QuickDraw's StdBit procedure. See the QuickDraw documentation for more information.

DrawLLine (newLPt: LPoint);

newLPt is the point to draw a line to.

TPadDrawLLine draws a line.

DrawLOval (verb: GrafVerb; r: LRect);

verb indicates what to do with the oval (frame, paint, erase, invert, fill).
r is the bounding box of the oval.

TPadDrawLOval draws an oval in the view.

DrawLPicture (pic: PicHandle; r:LRect);

pic is a pointer to a QuickDraw picture.
r is the desired bouding box, in view coordinates.

DrawLPicture draws the picture pointed to by pic in r. See the QuickDraw documentation for a discussion of pictures.

DrawLRect (verb: GrafVerb; r: LRect);

verb indicates what to do with the rectangle (frame, paint, erase, invert, fill).
r is the rectangle to draw.

TPadDrawLRect draws a rectangle.

DrawLRRect (verb: GrafVerb; r: LRect; ovalWidth, ovalHeight: INTEGER);

verb indicates what to do with the roundrect (frame, paint, erase, invert, fill).
r is the bounding box of the round rectangle.
ovalWidth is the width of the oval.
ovalHeight is the height of the oval.

TPadDrawLRRect draws a roundrect in the view.

DrawLText (textBuf: Ptr; startBytes, numBytes: INTEGER);

textBuf is a pointer to text.
startBytes the starting location in textBuf^.
numBytes the number of bytes.

TPadDrawLText draws text, scaling according the the current zoom factor, if necessary.

OTHER METHODS:

ChildWithPt (pt: Point; childList: TList; VAR nearestPt: Point): TArea;

pt is a point in window-relative coordinates.
childList is a list of areas contained within this pad.
nearestPt is the point in the returned area closest to pt.
The return value is one of the areas from childList.

ChildWithPt first finds the point in padInnerRect closest to pt. It then compares the new point to the outerRects of all the areas in childList. When it finds the area that contains the point, it finds the point in that area's innerRect that is closest to the point. That value is returned as nearestPt. The area containing nearestPt is the return value.

ClipFurtherTo (rBand: Rect);

rBand is a rectangle, in grafPort coordinates.

ClipFurtherTo narrows down the clip area at the next call to Focus. This method is used internally by the ToolKit.

Redefine (itsInnerRect: Rect; itsViewedLRect: LRect; itsPadRes,
itsViewRes: Point; itsZoomFactor: TScaler; itsPort: GrafPtr):
TPad;

itsInnerRect is the window relative bounds excluding the border.
itsViewedLRect is the portion of the view seen through this pad.
itsPadRes is the resolution of this pad.
itsViewRes is the resolution of the view looked on by this pad.
itsZoomFactor is a zoom factor for the pad.
itsPort is the grafPort.

TPad.Redefine is used to change the properties of a pad without having to free the old one and create a new one. In effect, this allows re-use of an allocated pad, with any or all of its parameters changed.

ResizeInside (newInnerRect: Rect);

newInnerRect is the pad bounds excluding borders.

ResizeInside changes the size of padInnerRect. It performs higher-level operations on the pad, such as resizing areas contained within this pad, and then calls SELF.SetInnerRect.

ResizeOutside (newOuterRect: Rect);

newOuterRect is the bounding box.

ResizeOutside changes the size of padOuterRect. It performs higher-level operations on the pad, such as resizing areas contained within this pad, and then calls SELF.SetOuterRect.

SetScrollOffset (VAR newOffset: LPoint);

newOffset is a new offset factor, with a vertical and horizontal component. Although this is a VAR argument, its value is not changed.

TPad.SetScrollOffset recalculates **pad.origin** and **pad.odOffset**.

SetZoomFactor (zoomNumerator, zoomDenominator: Point);

zoomNumerator is a set of two numbers that form the numerator of the zoom factors.
zoomDenominator is a set of two numbers that form the denominator of the zoom factors.

TPad.SetZoomFactor changes the zoom factors of the pad.

NOTES:

1. Methods of TPad fall into two categories:

   a) Those that convert coordinates between view and grafPort systems.

   b) Those that draw either to the screen or to a printer.

2. You never create instances of TPad, and you rarely create instances of descendants of TPad.
3. In general, the ToolKit creates instances of descendants of TPad in order to draw on the screen and print for you.
4. Panes, marginPads, and bodyPads are instances of descendants of TPad.

CLASS: TPageView

SUPERCLASS: TView

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: See TView.

DATA FIELDS: none

METHODS: None that are important for applications.

NOTES:

1. TPageView is used for the view associated with a marginPad when printing or previewing pages. (The application's view is associated with the corresponding bodyPad.)
2. TPageView.Draw is expected to draw the headings on a page.

CLASS: TPaginatedView

SUPERCLASS: TView

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: See TView.

DATA FIELDS:

R **unpaginatedView**: TView; The view of which this is the paginated version.

R **pageSize**: ARRAY[VHSelect] OF LONGINT; The size of one page in the paginated view, in view coordinates.

R **workingInMargins**: BOOLEAN; If TRUE, the application (in its own subclass of TPaginatedView) is currently fielding events in the margins of the page, rather than in the body.

METHODS: None that are important to applications.

NOTES:

1. Applications normally do not need to subclass or otherwise directly concern themselves with TPaginatedView.
2. There may be a different object of type TPaginatedView for every view used in the application.
3. A TPaginatedView object is created when the user asks to preview pages. When that happens, the view's printManager's NewPaginatedView method is called to launch the paginatedView. Because NewPaginatedView is called to create the paginatedView object, you can subclass TPaginatedView. One reason you may wish to do so is if you want to allow concurrent editing of headers, footers and main views, as is done in Lisawrite's Preview Pages mode.

CLASS: TPane

SUPERCLASS: TPad

DEFINED SUBCLASSES: TMarginPad
TBodyPad

INHERITED DATA FIELDS: See TPad.

DATA FIELDS:
R currentView: TView; The view displayed in this pane.
R panel: TPanel; The panel containing this pane.

METHODS YOUR APPLICATION MAY CALL:

ChildWithPt (pt: Point; childList: TList; VAR nearestPt: Point): TArea;

pt is a point in pane-relative coordinates.
childList is a list of areas contained within this pane.
nearestPt is the point in the returned area closest to pt.
The return value is one of the areas from childList.

ChildWithPt first finds the point in pane.innerRect closest to pt. It then the compares the new point to the outerRects of all the areas in childList. When it finds the area that contains the point, it finds the point in that area's innerRect that is closest to the point. That value is returned as nearestPt. The area containing nearestPt is the return value.

ScrollBy (VAR deltaLPt: LPoint);

deltaLPt is the amount of vertical (deltaLPt.v) and horizontal (deltaLPt.h) scrolling desired, in view coordinate units.

ScrollBy scrolls the pane by the distance given in deltaLPt. The scroll thumbs are moved accordingly.

ScrollToReveal (VAR anLRect: LRect; hMinToSee, vMinToSee: INTEGER);

anLRect an rectangle in the view.
hMinToSee a horizontal distance, in view units.
vMinToSee a vertical distance, in view units.

ScrollToReveal scrolls the pane so that some part of anLRect shows in the pane. hMinToSee and vMinToSee define the minimum part of anLRect that is displayed after the scroll.

NOTES:

1. TPane defines regions of the screen that display application data.
2. Applications never need to subclass TPane, and they also never create panes themselves.
3. The ToolKit creates at least one pane whenever a new panel is created, or when the user splits existing panes with a split control.

4. **Panes** occupy the region inside the borders of **panels**. Any part of an application **view** that is displayed is displayed in a **pane**.
5. Several **panes** may look on a single **view**. Every **pane** that looks on a particular **view** is normally contained in the same **panel**. Every **pane** in a particular **panel** looks on the same **view**.

CLASS: TPanel

SUPERCLASS: TArea

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

R	innerRect: Rect;	Window-relative bounds, excluding borders, but including rulers (if any).
R	outerRect: Rect;	Bounding box in window coordinates.

DATA FIELDS:

R	window: TWindow;	Window in which this panel appears.
R	panes: TList {OF TPane};	The panes of this panel, listed row-wise.
R	currentView: TView;	The paginated (page preview) or unpaginated view on which this panel looks.
R	view: TView;	The unpaginated view on which this panel looks when not in page preview mode. This is the view installed by your application.
R	paginatedView: TPaginatedView;	The paginated view, including margins, if the panel is currently in page preview mode. If not, contains NIL.
R	selection: TSelection;	The current selection.
R	undoSelection: TSelection;	The selection to be restored if the last command is undone.
	bands: ARRAY[VHSelect] OF TList;	The bands in this panel.
	scrollBars: ARRAY[VHSelect] OF TScrollBar;	The scrollbars of the panel. (scrollBars[(v,h)] -- the (vert,horiz) scroll bars).
R	abilities: ARRAY[VHSelect] OF TAbilites;	The abilites of the panel. See the CREATE method.
	minInnerDiagonal: Point;	The minimum size for innerRect.
	resizeBranch: TBranchArea;	Used for resizing the panel.
R	zoomed: BOOLEAN;	TRUE means that this panel is zoomed.
R	zoomFactor: RECORD numerator, denominator: POINT END;	The proportional zooming factor.
	previewMode: TPreviewMode;	Preview page breaks and page margins.

	lastClick: RECORD CASE gotPane: BOOLEAN OF TRUE: (clickPane: TPane); FALSE: (clickPoint: Point); END;	Describes the last mouse click. The pane of the last click. The innerRect.topLeft of lastClick.pane, for use when lastClick.pane was deleted.
	contentRect: Rect;	The part of innerRect excluding side bands (such as rulers).
	tlSideBandSize: Point;	The size of the horizontal side band (usually a ruler), which is displayed at the top of the panel.
	brSideBandSize: Point;	The size of the vertical side band (usually a ruler), which is displayed on the right side.
RW	deletedSplits: TArray;	If a TArray with recordBytes 2, the ToolKit stores the splits created by the user in the panel whenever the panel is shrunk so that it is too small to show the splits. Then, when the panel is enlarged so that it is large enough for the splits, the splits are redisplayed. If NIL, the ToolKit does not store splits, and they are lost. This is initialized to NIL in TPanel.CREATE; you can allocate an array and change the field if you desire.

METHODS YOUR APPLICATION WILL CALL:

CREATE (object: TObject; itsHeap: THeap; itsWindow: TWindow; minHeight, minWidth: INTEGER; itsVAbilities, itsHAbilities: TAbilities): TPanel;

itsWindow is the window containing this panel.
minHeight is the minimum height of this panel.
minWidth is the minimum width of this panel.
itsVAbilities indicates the vertical abilities of the panel from this set: aBar, aScroll, aSplit. See note 8 for more information.
itsHAbilities indicates the horizontal abilities of the panel from the same set as itsVAbilities.

TPanel.CREATE creates an object of type TPanel.

Highlight (selection: TSelection; highTransit: THighTransit);

selection is a selection object.

highTransit is the change in highlighting of the displayed object. The standard choices are: hNone, hOffToDim, hOffToOn, hDimToOn, hDimToOff, hOnToOff, and hOnToDim.

TPanelHighlight calls selectionHighlight once for each pad object (which includes panes and pages) in the panel object. (Uses OnAllPadsDo.)

NewView (object: TObject; itsExtent: LRect; itsPrintManager: TPrintManager; itsDfltMargins: LRect; itsFltPerfectlyOnPages: BOOLEAN): TView;

itsExtent is the size of the view, in view coordinates.
itsPrintManager is the print manager for the view.
itsDfltMargins defines the default margins when the view is printed (can be changed by the user).
itsFltPerfectlyOnPages tells whether or not the view should always divide into whole pages for printing. If TRUE, when the view size is changed, the ToolKit automatically increases its size (if necessary).

NewView creates the view object for the panel. This method or NewStatusView is called once and only once for every panel. Use NewView for printable views.

NewStatusView (object: TObject; itsExtent: LRect): TView;

itsExtent is the size of the view, in view coordinates.

NewStatusView creates the view object for the panel. This method or NewView is called once and only once for every panel. Use NewStatusView for views that cannot be printed.

OnAllPadsDo (PROCEDURE DoOnThePad);

PROCEDURE DoOnThePad is a procedure of no arguments.

TPanelOnAllPadsDo applies the given method to every pad object (which includes panes and pages) of the panel. It focuses the grafPort on each pad before applying the method to that pad. (The global variable thePad is the presently focused pad.) DoOnThePad is called once for every pane and page in the panel. In "Preview Page Margins" mode, DoOnThePad is called once for every page in each pane. DoOnThePad cannot be a method.

OnAllPadsDo should not be called from TViewDraw or TSelectionHighlight methods, but must be used to call those methods. On the other hand, you should use panelOnAllPadsDo in command, Mousepress, MouseMove, and MouseRelease methods.

METHODS YOUR APPLICATION MAY CALL:

BeginSelection;

BeginSelection sets selectPanel, and calls DeSelect for selections in all panels. In some cases, this method does the same for any current dialogBoxes.

BeSelectPanel (inSelectWindow: BOOLEAN);

inSelectWindow indicates whether panel is in the selectWindow.

TPanel.BeSelectPanel makes panel the selectPanel of its window. If inSelectWindow is TRUE, BeSelectPanel also makes that window, which might be a dialog box, be the selectWindow of the active window.

ChildWithPt (pt: Point; childList: TList; VAR nearestPt: Point): TArea;

pt is a point in window-relative coordinates.
childList is a list of areas contained within this panel.
nearestPt is the point in the returned area closest to pt.
The return value is one of the areas from childList.

ChildWithPt first finds the point in panel.innerRect closest to pt. It then the compares the new point to the outerRects of all the areas in childList. When it finds the area that contains the point, it finds the point in that area's innerRect that is closest to the point. That value is returned as nearestPt. The area containing nearestPt is the return value.

Divide (vhs: VHSelect; fromEdgeOfPanel: INTEGER; units: TUnitsFromEdge; whoCanResizeIt: TResizability; minSize: INTEGER; itsVAbilities, itsHAbilities: TAbilities);

vhs indicates the orientation of the new panel. v divides the panels so one is on top of the other; h divides the panels so they are side by side.
fromEdgeOfPanel, if positive, is the distance of the split from the top left corner; if negative, is the distance from the bottom right corner.
units indicates whether the fromEdgeOfPanel value is a percentage (percentFromEdge) or is in pixels (pixelsFromEdge).
whoCanResizeIt determines whether the size of the panel varies with the size of the window, whether the panel has a resize icon so that the user can change its size, and whether the application can change the panel's size.
minSize defines the minimum size of the panel.
itsVAbilities indicates the vertical abilities of the panel from this set: aBar, aScroll, aSplit. See note 8 for more information.
itsHAbilities indicates the horizontal abilities of the panel from the same set as itsVAbilities.

TPanel.Divide creates a new panel, after the first panel in the

window has already been created with TPanel.CREATE. Divide automatically calls TPanel.Insert.

Frame;

Frame draws the frame of the panel, including the scroll bars.

HaveView (view: TView);

view is a view object.

HaveView installs the given view in the panel.

Insert (panel: TPanel; vhs: VHSelect; fromEdgeOfPanel: INTEGER; units: TUnitsFromEdge; whoCanResizeIt: TResizability);

panel is the panel that is inserted.
vhs indicates the orientation of the new panel. v divides the panels so one is on top of the other; h divides the panels so they are side by side.
fromEdgeOfPanel, if positive, is the distance of the split from the top left corner; if negative, is the distance from the bottom right corner.
units indicates whether the fromEdgeOfPanel value is a percentage (percentFromEdge) or is in pixels (pixelsFromEdge).
whoCanResizeIt determines whether the size of the panel varies with the size of the window, whether the panel has a resize icon so that the user can change its size, and whether the application can change the panel's size.

TPanel.Insert inserts a previously created panel into the window, using the space occupied by SELF (the panel through which Insert is called). You normally do not call Insert directly; it is called for you when you use Divide to create a new panel. You can, however, use TPanel.Create to create your own panels, and then call Insert yourself.

Invalidate;

Invalidate invalidates the entire panel, so that it is redrawn the next time TWindow.Update is called.

InvalLRect (lRectInView: LRect);

lRectInView is a part of the view.

TPanel.InvalLRect tells all pad objects (which includes panes and pages) in the panel object to perform thePad.InvalLRect on themselves. (Uses OnAllPadsDo.)

OkToDrawIn (lRectInView: LRect): BOOLEAN;

lRectInView is a view-relative rectangle.

A return value of TRUE indicates that the application can safely call Draw methods or QuickDraw and UDraw routines, directly (rather than calling panel.InvalRect to cause drawing in the next Update) within the rectangle lRectInView. An application or building block should always call OkToDrawIn before attempting to draw or erase directly when giving feedback (except for XOR feedback) in response to a user action or command. If permission is denied, the application should call TPanel.InvalLRect (which may display more slowly), rather than a Draw method. TPanel.OkToDrawIn checks to see if lRectInView includes a page break, which will only occur when the document is in "Preview Page Breaks" mode. If it does, it is not safe to Draw, and the return value is FALSE. Otherwise, TPanel.OkToDrawIn calls view.OkToDrawIn. You can reimplement view.OkToDrawIn so that it actually checks to see if it is safe to draw in lRectInView. TView.OkToDrawIn always returns FALSE.

PaneShowing (anLRect: LRect): TPane;

anLRect is rectangle in the view looked on by this panel.
The return value is one of the panel's panes.

PaneShowing returns the first pane in panel.panes that shows some part of anLRect.

PaneToScroll (VAR anLRect: LRect; hMinToSee, vMinToSee: INTEGER):
TPane;

anLRect is rectangle in the view looked on by this panel.
hMinToSee is the minumim horizontal portion of anLRect that you want to see.
vMinToSee is the minumim vertical portion of anLRect that you want to see.
The return value is one of the panel's panes.

PaneToScroll is called by TPanel.RevealLRect to obtain the pane that should be scrolled to reveal the indicated minimum portions of anLRect. Your application might call this method directly.

Preview (newMode: TPreviewMode);

newMode is the new preview mode from the Page Layout menu.

Preview displays the pageView version of the view looked on by this panel.

PrintView (printPref: TPrReserve);

printPref are the printer preferences.

PrintView prints the view looked on by this panel.

Refresh (rActions: TActions; highTransit: THighTransit);

rActions is from the set (rErase, rFrame, rBackground, rDraw).
highTransit is the change in highlighting of the displayed object. The standard choices are: hNone, hOffToDim, hOffToOn, hDimToOn, hDimToOff, hOnToOff, and hOnToDim.

Refresh refreshes the panel's display. Update sets up the grafPort for the invalidated portion of the panel's view, and then calls Refresh. If there is no invalid portion, Refresh is not called.

Remove;

TPanelRemove takes the SELF panel out of the window. The panel that was divided to create space for this panel is expanded to fill the space. TPanelRemove does not free the panel.

Replace (panel: TPanel);

panel is the panel that is to be inserted.

TPanelReplace replaces the SELF panel with the given panel. The panel that was removed is not freed. The new panel is resized to fill the space occupied by the old panel.

ResizeInside (newInnerRect: Rect);

newInnerRect is the panel bounds excluding borders.

ResizeInside changes the size of panel.innerRect. It performs higher-level operations on the panel, such as resizing areas contained within this panel, including bands, and then calls SELF.SetInnerRect.

ResizeOutside (newOuterRect: Rect);

newOuterRect is the bounding box.

ResizeOutside changes the size of panel., and invalidates the changed areas. It performs higher-level operations on the panel, such as resizing areas contained within this panel, and then calls SELF.SetOuterRect. ResizeOutside calls ResizeInside.

Rescroll;

Rescroll is called when the view changes drastically. It fixes the elevators so that they are positioned correctly. The entire panel is invalidated.

RevealLRect (VAR anLRect: LRect; hMinToSee, vMinToSee: INTEGER);

anLRect an rectangle in the view.
hMinToSee a horizontal distance, in view units.
vMinToSee a vertical distance, in view units.

RevealLRect scrolls the panel so that some part of anLRect shows in a pane. hMinToSee and vMinToSee define the minimum part of

anLRect that is displayed after the scroll. This method calls TPanel.PaneToScroll to obtain the pane that should be scrolled to reveal anLRect.

SetZoomFactor (zoomNumerator, zoomDenominator: point);

zoomNumerator is a point containing the vertical (zoomNumerator.v) and horizontal(zoomNumerator.h) numerators of the zoom factor. zoomDenominator is a point containing the vertical (zoomDenominator.v) and horizontal(zoomDenominator.h) denominators of the zoom factor.

SetZoomFactor sets a zoom factor that translates sizes between the view and the panel. The factor is a set of two fractions, one for the horizontal zoom factor and one for the vertical zoom factor. Each fraction is broken up into two numbers, a numerator and a denominator. The set of numerators is stored in the point zoomNumerator and the set of denominators in the point zoomDenominator. (Warning: this feature has not been debugged.)

ShowSideBand (vhs: VHSelect; topOrLeft: BOOLEAN; size: INTEGER; viewLCd: LONGINT);

vhs indicates the orientation of the side band: vertical or horizontal. A vertical side band appears at the top or bottom of the panel; a horizontal sideband appears on the left or right side.
topOrLeft indicates whether the side band is on the top or the left side of the panel. For example, if TRUE, and vhs is v, the side band appears on the top of the panel.
size is the size of the side band, in vhs direction.
viewLCd is a view-coordinate number that indicates the part of the view that shows in the sideband. This is usually some negative number. The number does not change the appearance of the side band except that if the number is positive, it is part of the view, can be scrolled, and can be printed.

ShowSideBand displays a side band. The side band is always displayed with a 20 pixel gap between it and the panel.

SideBandRect (vhs: VHSelect; topOrLeft: BOOLEAN; VAR bandRect: Rect);

vhs indicates the orientation of the side band: vertical or horizontal. A vertical side band appears at the top or bottom of the panel; a horizontal sideband appears on the left or right side.
topOrLeft indicates whether the side band is on the top or the left side of the panel. For example, if TRUE, and vhs is v, the side band appears on the top of the panel.
bandRect is the innerRect of the side band.

SideBandRect returns the innerRect of the side band, given SELF.contentRect. You probably will never call this method.

FAILURE CONDITIONS: None

NOTES:

1. TPanel is never subclassed.
2. A panel is the part of a window that looks onto a particular view.
3. There may be several panels in a single window. Each one normally displays the contents of a different view object.
4. panelCREATE is normally called once for each window. To create more panels, call panelDivide once for each additional panel. Divide calls panelCreate and then panelInsert.
5. If you want to add a panel as quickly as possible, you can call CREATE when you initialize your window, and store the panel away. When you want the panel to appear on the display, call panelInsert.
6. The pane objects that control the panes within the panel are what actually display the visible portion of the view object. The panel object controls the scrolling of those panes.
7. Every panel initially gets one pane. The user may create additional panes if one of the TAbilities given is aSplit. It is not necessary for your application to mention pane objects.
8. The TAbilities are:

   aBar, which gives a gray bar with no scrolling ability.

   aScroll, which gives a scroll bar (a gray bar with flippers, scrollers, and elevators), and allows autoscroll.

   aSplit, which gives a pane splitter.

   These abilities are independent; give a list of the ones that you want.

CLASS: TPasteCommand

SUPERCLASS: TCommand

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

R	cmdNumber: TCmdNumber;	The command number of the menu item that describes the command.
R	view: TView;	The view to which the command applies, or NIL if the data is stored in the window.
R	undoable: BOOLEAN;	TRUE means that this command can be reversed.
R	doing: BOOLEAN;	This value is TRUE if the last call to Perform was in doPhase or redophase.
	revelation: TRevelation;	Determines how much of the selection should be automatically scrolled into a pane when the command is performed. Can be none (no scrolling), some (scroll to show some part of the selection), or all (scroll to display the entire selection, if possible).
W	unHiliteBefore: ARRAY [TCmdPhase] OF BOOLEAN;	TRUE tells the ToolKit to remove the highlighting from all selections before Perform is called.
W	hiliteAfter: ARRAY [TCmdPhase] OF BOOLEAN;	TRUE tells the ToolKit to add highlighting to all selections after Perform is called.

DATA FIELDS: none

METHOD YOUR APPLICATION MUST OVERRIDE:

CREATE(object: TObject; itHeap: THeap; itsCmdNumber: TCmdNumber; itsView: TView): TPasteCommand;

itsCmdNumber is the command number of a menu command.
itsView the view to which the command applies, or NIL if the data is stored in the window.

TPasteCommand.CREATE creates an object of type TPasteCommand.

DoPaste (clipSelection: TSelection; pic: picHandle; cmdPhase: TCmdPhase); DEFAULT;

clipSelection is, when cmdPhase=doPhase or redoPhase, the selection object that indicates what objects are to be inserted. When cmdPhase=undoPhase clipSelection is NIL.
pic is the handle of a QuickDraw-picture representation of the data in the clipboard, or NIL.
cmdPhase is either doPhase, undoPhase, or redoPhase.

DoPaste executes the paste. Either clipSelection or pic is used to indicate what is to be pasted into the document. If clipSelection is not NIL, it indicates a set of ToolKit objects in the clipboard. If clipSelection is NIL, pic references a QuickDraw picture that is to be pasted into the document. The first time DoPaste is called, the cmdPhase is doPhase. After that, undoPhase and redoPhase alternate. Do not call this routine directly; it is called by Perform.

When cmdPhase is doPhase or redoPhase, DoPaste must clone the contents of the clipboard, and put the copies into the document. When cmdPhase is undoPhase, DoPaste must delete the added information from the document. On doPhase and redoPhase, the clipSelection might be NIL if there is no view to paste or if the selection is a generic TSelection. In that case, information for the paste must be obtained from pic.

METHOD YOUR APPLICATION WILL CALL:

CREATE (object: TObject; itHeap: THeap; itsCmdNumber: TCmdNumber; itsView: TView): TPasteCommand;

See above.

METHODS YOUR APPLICATION MIGHT OVERRIDE:

Commit;

You can override Commit if you want to implement the paste so that it uses a filter when Perform is called. In that case, Commit changes the document, while DoPaste simply copies the contents of the clipboard into someplace Commit and EachVirtualPart can reach. TPasteCommand.Commit does nothing.

EachVirtualPart (PROCEDURE DoToObject (filteredObj: TObject));

PROCEDURE DoToObject (filteredObj: TObject) is a procedure that filters filteredObj before filteredObj is displayed. DoToObject cannot be a method.

EachVirtualPart is used in implementing a filter. A *virtual part* is a selectable part of the set of objects. Every virtual part must be an object. DoToObject acts on the object it is given. When you Perform the paste command with cmdPhase equal to doPhase copy the contents of the clipboard into a temporary list. When drawing the view, use EachVirtualPart to draw the original document as well as your copy of the just pasted list. When you Commit the pasteCommand, copy your copy of the clipboard contents into your application's set of objects.

INTERNAL METHOD:

Perform (cmdPhase: TCmdPhase);

cmdPhase is either doPhase, undoPhase, or redoPhase.

TPasteCommand.Perform calls pastecommand.DoPaste to execute the paste command. The first time Perform is used, the cmdPhase should be doPhase. After that, undoPhase and redoPhase alternate.

FAILURE CONDITIONS: none

NOTES:

1. See TCommand for general information on command implementation.
2. TPasteCommand is used to paste the contents of the clipboard into the document. The clipboard contents are inserted at the place indicated by the selection.
3. You must override this class to use it. At a minimum, you will have to implement DoPaste. Note that building blocks often implement this for you.
4. The clipboard can be loaded by a cut or copy operation by your own application or by another ToolKit application. If the clipboard is loaded by another application, the ToolKit converts the data to a better form for your application before pasting the data into your document. (The conversion is done by TCutCopyCommand.Perform.) Every object that was of a class your application does not have is converted to an object of an ancestor class that exists in your application. This transformation process can lose information.
5. Before you copy the data, you can use InClass to test the class of clipSelection. As you copy the data, it is a good idea to examine each component that might have been an object of another class in another

application. You can sometimes use **InClass** to help transform objects from the clipboard into objects of classes that your application is prepared to process. This is demonstrated in the examples in the ToolKit Segments.

6. See the ToolKit Segments for examples of using **TPasteCommand** and further explanations.

CLASS: TPrintManager

SUPERCLASS: TObject

DEFINED SUBCLASS: TStdPrintManager (defined in the Dialog Building Block)

INHERITED DATA FIELDS: none

DATA FIELDS:

view: TView;	The view whose printing is managed by this printManager.
pageView: TView;	The view that draws the headings in the margins of the page. The size of this view is the size of one page.
breaks: ARRAY[vhSelect] OF TArray {of LONGINT};	The vertical and horizontal pagebreaks that divide the view into pieces that fit onto individual pages. breaks[v] holds information for page breaks that, when drawn on the screen, are represented by vertical lines; breaks[h] holds information for page breaks that, when drawn on the screen, are represented by horizontal lines. The absolute value of each array element gives the location of the break in the view; the sign tells whether the break is an automatic one (nonnegative) or a manual (user-set) one (negative).
pageMargins: LRect;	The page margins to use when fitting view pieces onto pages: top and left are positive and bottom and right are negative. These numbers are in view coordinates, which is why the values are LRects even though the actual numbers involved are small.
headings: TList {of THeading};	The headings to be printed on the pages.
canEditPages: BOOLEAN;	FALSE by default; the stdPrintManager sets this TRUE, as headings and margins can be edited with the page.
layoutDialogBox: TDialogBox;	NIL by default; the stdPrintManager places a reference to a dialog box it creates here.

frameBody: BOOLEAN;	FALSE by default; you can set it to TRUE, in which case the page body is separated from the page margins on the printed page by a box. Mostly for debugging purposes.
paperLRect: LRect;	The physical bounds of a single page on the current printer, in view coordinates.
printableLRect: LRect;	The printable area of a single page on the current printer, in view coordinates.
contentLRect: LRect;	The area on a page into which the body (that is, sections of the main view) will be placed, obtained by taking the paperLRect and allowing room for the margins specified in the pageMargins..
printerMetrics: TPrinterMetrics;	The data characterizing the physical properties of the printer currently formatted for; this is a RECORD whose fields are: paperRect, printableRect (the counterparts of paperLRect and printableLRect, but in grafPort coordinates), and res (the resolutions of the device, packaged into a Point).
pageRiseDirection: vhSelect;	Whether page numbers should be assigned left-to-right (h) or top-to-bottom (v).

METHODS YOUR APPLICATION MAY OVERRIDE:

CREATE (object: TObject; heap: THeap): TPrintManager;

TPrintManager.CREATE creates an object of type TPrintManager.

DrawPage;

Draw the page with the page-number indicated by theMarginPad.pageNumber.

EnterPageEditing;

Enter into whatever method the printManager has of allowing the user to edit page headings and margins.
TPrintManager.EnterPageEditing does nothing.

GetPageLimits (pageNumber: LONGINT; VAR viewLRect: LRect);

pageNumber is the number of one of this application's pages.
viewLRect is the part of the application's view that makes up the body of the given page.

Given the page number, returns the bounds of the part of the view that makes up the body of that page. Your application can call this method.

NewPaginatedView (object: TObject): TPaginatedView;

The return value is a paginatedView.

TPrintManager.NewPaginatedView creates a new paginatedView. By default, the ToolKit's standard Page-preview mode is invoked.

NewPageView (object: TObject): TView;

The return value is a pageView.

TPrintManager.NewPageView creates a new pageView. By default, an object of the ToolKit's standard TPageView class is launched.

PageWith (VAR lPtInView: LPoint; VAR strip: Point):LONGINT;

lPtInView is a point in the application's view.
strip is the vertical and horizontal strip of pages that contain lPtInView.
The return value is the number of the page that contains lPtInView.

Given a point in the view, PageWith returns the page number into which that point falls, as well as returning the vertical and horizontal strip of pages containing the lPtInView. Your application can call this method.

Print;

TPrintManager.Print orders the actual printing. You might override Print in order to do something special before or after printing. In that case, call SUPERSELF.Print for the actual printing. You can override Print entirely and handle printing yourself, but you must then have direct access to the underlying Lisa Printing software.

ReactToPrinterChange;

TPrintManager.ReactToPrinterChange reacts to a change in printer specification. TPrintManager.ReactToPrinterChange gets fresh printer metrics, recomputes the printableLRect, paperLRect, and contentLRect, resizes the view if necessary, recomputes the bounds of the pageView, and calls RedoBreaks. This method is called by TView.ReactToPrinterChange. Your application can call this method.

RedoBreaks;

TPrintManager.RedoBreaks recomputes page-breaks. Your application can call this method.

SetDfltHeadings;

SetDfltHeadings in intended to set the default headings. It is called by TPrintManager.Init so that the printManager can create desired default headings and install them in its list of headings. TPrintManager.SetDfltHeadings does nothing.

SkipPage (pageNumber: LONGINT);

TPrintManager.SkipPage does nothing. You can override this routine to do whatever you want to do when that the indicated page-number, at printing time, is not going to be printed, but larger page-numbers will be. This allows an application to cycle through its data structures to the start of the next page, if it needs to.

METHOD YOUR APPLICATION MAY CALL:

ChangeMargins (newMargins: LRect);

Install a new set of margins into the printManager.

NOTES:

1. A printManager must be associated with every printable view.
2. You associate a printManager with a view by handing a reference to a printManager to the method that creates the view. The printManager-reference is then placed in the view.printManager field. If the view is not printable, hand NIL to the method that creates your view (usually panel.NewView).
3. You can use an instance of TPrintManager for your printManager. In that case, simply call TPrintManager.CREATE, and install the resulting object in view.printManager. The view will be printable, but you will not be able to specify headings or margins.
4. Alternately, you can use the Dialog Building Block class TStdPrintManager. Call TStdPrintManager.CREATE, and install the resulting object in view.printManager. The view will be printable, and you will be able to interactively create and edit headings and margins through the use of the Headings and Margins dialog. In this case, you must USE the Dialog Building Block.
5. As a third alternative, you can create your own subclass of TPrintManager or TStdPrintMangeger, and give an instance of that to the method that creates your view.

6. Once a printManger is installed in your view, the same drawing methods that draw in your application's window on the screen can draw on the printed page. The ToolKit takes care of converting the output so that it appears correctly on the printer.

***CLASS: TProcess

SUPERCLASS: TObject

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: none

DATA FIELDS: None. However, the global variables are important to the process. See Chapter 6.

METHODS YOUR APPLICATION MUST OVERRIDE:

CREATE (object: TObject; heap: THeap): Tprocess;

TProcess.CREATE creates a process object.

NewDocManager (volumePrefix: TFilePath; openAsTool: BOOLEAN): TDocManager;

volumePrefix, a name of the form -VOLUME-{documentnumber}, is set by the ToolKit. Your application simply passes the value along. All files that comprise the document begin with the string contained in volumePrefix.
openAsTool defines whether or not the Tool icon was opened by the user. The calculator and clock are examples of applications where this is true.

NewDocManager creates a docManager object for the process. The method you write should call the CREATE method for your descendant of TDocManager. If you do not want your application to create a docManager, return NIL. You can do that when the user tries to open the tool and the tool cannot be opened, or when the application cannot handle another document. See the ToolKit Segments for detailed descriptions of how your application is initialized.

METHODS YOUR APPLICATION WILL CALL:

Commence (phraseVersion: INTEGER);

phraseVersion is a version number used to check that the proper phrase file is on the disk. No version check is done if phraseversion<0. A mismatched phraseVersion is only a warning. The ToolKit does nothing.

TProcess.Commence initializes a process.

Complete (allIsWell: BOOLEAN);

allIsWell indicates whether or not the process completed successfully. The application program normally passes TRUE. The ToolKit passes FALSE when there has been an error.

TProcess.Complete signals that the process has finished. It terminates the OS process, and never returns to its caller.

Run;

TProcess.Run starts the main event loop running.

METHODS YOUR APPLICATION MIGHT OVERRIDE:

Commence (phraseVersion: INTEGER);

See above. You might override this method in order to initialize some of your own global variables. In all cases, you should call TProcess.Commence before initializing your own variables.

Complete (allIsWell: BOOLEAN);

See above. You might override this method in order to clean-up after your process. Call TProcess.Complete after finishing your own cleanup.

OTHER METHODS:

ChangeCursor (cursorNumber: TCursorNumber);

cursorNumber is the reference number for an application or ToolKit defined cursor.

ChangeCursor changes the cursor to the given cursor shape. Applications may call this routine. ChangeCursor calls DoCursorChange.

DoCursorChange (cursorNumber: TCursorNumber);

cursorNumber is the reference number for an application or ToolKit defined cursor.

TProcess.DoCursorChange handles standard cursor numbers. Applications can reimplement DoCursorChange in order to test cursorNumber for one of the application's special cursor shapes. If the cursorNumber is defined as one of your application's special cursors, DoCursorChange should call QuickDraw's SetCursor routine. If the cursorNumber is not one of your special cursors, your DoCursorChange should call SUPERSELF.DoCursorChange. Application cursor numbers start at 100.

TrackCursor;

TrackCursor tracks the cursor while the process is in its idle loop. It eventually calls **view.CursorAt** where you get a chance to set up your own cursor.

ArgAlert (whichArg: TAlertArg; argText: S255);

whichArg is a number from 1 to 5, corresponding to the ^1 to ^5 placeholders in the phrase file. (^0, in the phrase file, is replaced by the tool name given when you run the Install program.)
argText is the text you want to replace the given placeholder.

ArgAlert replaces the **whichArg** placeholder with **argText**. The text is included the next time an alert with that placeholder is displayed.

Ask (phraseNumber: INTEGER): INTEGER;

phraseNumber is the phrase file reference number for an ask alert. The return value is the number of the button the user pressed.

Ask displays an ask alert, which is an alert that presents the user with choices.

BeginWait (phraseNumber: INTEGER);

phraseNumber is the phrase file reference number for a dialog.

BeginWait displays a wait alert.

Caution (phraseNumber: INTEGER): BOOLEAN;

phraseNumber is the phrase file reference number for a dialog.

Caution displays a caution alert using the text referred to by **phraseNumber**.

CountAlert (whichCtr: TAlertCounter; counter: INTEGER);

whichCtr is 7 to 9.
counter is the number you want displayed in **whichCtr**.

CountAlert changes the number displayed in a wait alert.

DrawAlert (phraseNumber: INTEGER; marginLRect: LRect);

phraseNumber is the identification number of the alert in the phrase file.
marginLRect is the size of the text area of an alert box. Only the top, left, and right values matter. The ToolKit extends the bottom of the alert to allow room for the text associated with **phraseNumber**.

DrawAlert draws an alert box.

EndWait;

EndWait ends a wait alert.

GetAlert (phraseNumber: INTEGER; VAR theText: S255);

phraseNumber is the phrase file reference number for a dialog. theText is the text of the alert referred to by phraseNumber.

GetAlert obtains the text string of a given alert from the phrase file.

Note (phraseNumber: INTEGER);

phraseNumber is the phrase file reference number for a dialog.

Note displays a note alert.

RememberCommand (cmdNumber: TCmdNumber);

cmdNumber is the number of an application or ToolKit command.

RememberCommand remembers the last command given for use in an alert. The command is used to replace the ^C and ^K placeholders in the phrasefile. ^K displays the command exactly as it appears in the menu. ^C converts the command to all lower case. This method also resets the counter for *staged* alerts. A staged alert is one where the alert response changes if the user repeats the action. For example, when you type on the Lisa when that is not allowed, the Lisa beeps on the first two keystrokes. An alert is displayed after the third keystroke, to point the error out. RememberCommand resets that counter, on the assumption that, since the user executed a command, the errors were not continuous.

Phrase (error: INTEGER): INTEGER;

error is an error number.

Phrase should return a phrase number for a given error. If the return value is phUnknown, refer to SuLib(OSERRS.ERR) in the Internals document to get the text of the error message.

Stop (phraseNumber: INTEGER);

phraseNumber is the phrase file reference number for a dialog.

TProcess.Stop usually displays a stop altert using the text referred to by phraseNumber. If there is no active document, it halts the process and initiates a dialog with the user.

AbortRequest: BOOLEAN;

If aborts are allowed, the return value indicates whether or not the apple/period command was typed. TRUE indicates that command was typed.

AbortRequest indicates whether the user has requested a command abort. You can call this periodically during a long command to tell if the user is trying to abort the command.

AbortXferSequential (whichWay: xReadWrite; pFirst: Ptr; numBytes, chunkSize: LONGINT; fs: TFileScanner);

whichWay defines whether the operation is a read or write.
pFirst is a pointer to the first byte to transfer.
numBytes defines how many bytes are transferred
chunkSize is the number of bytes to transfor before testing for an abort command.
fs is an active file scanner.

AbortXferSequential acts the same as XferSequential, except that it checks AbortRequest after each chunkSize bytes of data are transferred.

ObeyEvents (FUNCTION StopCondition: BOOLEAN);

StopCondition is a ToolKit function that returns TRUE if some condition requires suspension of the process.

ObeyEvents is the main event loop for the process. It normally returns only if amDying is TRUE, which indicates that the application is terminated or StopCondition returns TRUE. StopCondition is checked only when no events are available.

ObeyFilerEvent;

ObeyFilerEvent is called by the ToolKit when a filer event is received for a document owned by this process.
TProcess.ObeyFilerEvent calls an appropriate routine, usually a method of TDocManager, to handle the particular event.

ObeyTheEvent;

ObeyTheEvent is called by the ToolKit when an event is received for the process. TProcess.ObeyTheEvent then calls an appropriate event handling routine, which may be a ToolKit or an application routine.

HandlePrivateEvent (typeOfEvent: INTEGER; fromProcess: LONGINT; when: LONGINT; otherData: LONGINT); DEFAULT;

typeOfEvent is an event type number. This is generally 100, but other numbers may be defined in building blocks.
fromProcess is the OS process ID for the process that sent the event. You can pass in the process ID by using the OS procedure Info_Process.
when is the clock time when the message was sent.
otherData is some data sent by the process.

HandlePrivateEvent is called by the ToolKit when an event is received from another process. If you want your application to process such events, you must implement this method. TProcess.HandlePrivateEvent does nothing.

SendEvent (typeOfEvent: INTEGER; targetProcess: LONGINT; otherData: LONGINT);

typeOfEvent is an event type number. The possible numbers are 100 and above.
targetProcess is the OS process ID for the process that is to receive the event. You can pass in the process ID by using the OS procedure Info_Process.
otherData is any information you want.

SendEvent sends a message (an event) to another process.

BindCurrentDocument;

BindCurrentDocument puts the data segments for the document currently being used by this process into memory. You should not need to call this method; the ToolKit binds your data segments for you.

FAILURE CONDITIONS: none

NOTES:

1. **TProcess** is the default **process** class. You always create a subclass so that your application's type of **process** object is created. In the subclass you must define a **NewWindow** method that creates your kind of **window** object.
2. A ToolKit **process** is not the same as an OS process. An OS process is a particular instance of a program that may be executing. A ToolKit **process**, the only kind discussed in this manual, is the control object in a ToolKit program. There is, at most, one ToolKit **process** in an OS process.
3. A program must create an instance of a descendant of **TProcess** in order to run under the ToolKit.
4. No program may use more than one instance of **TProcess** or a descendant of **TProcess**. (Note, however, that a second OS process can be started by a ToolKit **process**.)
5. Normally, a main program creates an instance of its **TProcess** descendant, and then initializes, commences, and runs the **process**. When the **process.Run** method call is given, the **process** waits for events from the program user. The application's implementation unit and the ToolKit contain methods that respond to events, and carry out the actual work of the application. If the process completes successfully, **process.Run** completes, and the main program uses **process.Complete(TRUE)** to signal successful completion. If the **process** fails with an error, **process.Complete(FALSE)** is called by the ToolKit. Normally, a ToolKit **process** does not complete until the office system is shut down.

6. TProcess is normally the only class mentioned in a main program. It is not normally mentioned in any unit, except for in its own definition.
7. For more information on alerts, and how to set them up, see the Phrase File document.

EXAMPLE:

The following is a sample main program. After that is the section from the application's interface unit showing the definition for the class TSamProcess. The third program fragment is a section of the application's implementation unit showing the implementation of TSamProcess. Note that, other than CREATE and NewDocManager, all methods of TSamProcess are inherited from TProcess without being overridden.

```
PROGRAM Sample;          {This is an example of a main program}

USES
        {NOTE: Do not omit any unit that defines classes.  These units are
        always given in this order.}
        {$U UObject       } UObject,
        {$U QuickDraw     } QuickDraw,
        {$U UDraw         } UDraw,
        {$U UABC          } UABC,
        {$U USample       } USample; {This is the application's unit.
                                      Applications can have several units
                                      here.}
CONST
        phraseVersion = 1;
BEGIN
                                process := TSamProcess.CREATE; {NOTE: You
                                             must define your own subclass of
                                             TProcess.}
        process.Commence(phraseVersion);
        process.Run;
        process.Complete(TRUE);

END.
```

The following subclass definition appears in unit Usample:

```
TSamProcess = SUBCLASS OF TProcess
FUNCTION  TSamProcess.CREATE: TSamProcess;
FUNCTION  TSamProcess.NewDocManager(volumePrefix: TFilePath):
        TDocManager;
END;
```

The following method definitions appears in file Usample2, the implementation part of Usample.

```
FUNCTION  {TSamProcess.}CREATE;
BEGIN
      SELF := TSamProcess(TProcess.CREATE(object, heap));
END;

FUNCTION  {TSamProcess.} NewDocManager (volumePrefix: TFilePath):
                                                TDocManager;
BEGIN
      NewDocManager := TSamDocManager.CREATE (mainheap,
                              volumePrefix);
END;
```

CLASS: TScanner

SUPERCLASS: TObject

DEFINED SUBCLASSES: TArrayScanner
TFileScannerP (subclass of TStringScanner)
TListScanner
TStringScanner

INHERITED DATA FIELDS: none

DATA FIELDS:

R	collection: TCollection;	The collection being scanned.
R	position: LONGINT;	The current position of the scanner. The scanner position is always between, before, or after members: 0-before first, size+1-after last.
R	increment: INTEGER;	The change in position for every scan: 1 if scanning forward, -1 if scanning backward.
	scanDone: BOOLEAN;	When this is TRUE, the next Scan call returns FALSE, signalling the end of the scan. The VAR parameter of the Scan method, which normally contains the next object in the collection, is unchanged after the Scan call.
R	atEnd: BOOLEAN;	TRUE if the end of the collection is eminent, so that the next Scan call will return FALSE.

METHODS:

CREATE (object: TObject; itsCollection: TCollection; itsInitialPosition: LONGINT; itsScanDirection: TScanDirection): TScanner;

itsCollection is a collection object.
itsInitialPosition is the initial ordinal position in the collection.
itsScanDirection is scanForward or scanBackward.

TScanner.CREATE creates an object of type TScanner, which is used to access itsCollection. itsCollection must exist before a scanner can be created for it; however, you can use a TCollection.CREATE call in your TScanner.CREATE call, which will create the necessary collection object. You never call this method directly.

Advance (PROCEDURE DoToCurrect(anotherMember: BOOLEAN):
BOOLEAN;

DoToCurrent is a procedure that takes a member of the collection an an argument.
The return value tells whether or not the end of the collection has been reached.

Advance begins at the beginning of the collection and, each time it is called, returns the next member. The returned member is referred to as the *current* member. The value of Advance is TRUE until Advance is called after the last member in the collection is reached, or until scanner.Done has been called. When Done is called or the scanner reaches the end of the collection, the scanner is freed. You never call this method directly.

Allocate (slack: LONGINT); DEFAULT;

slack is the amount of space added to the collection, in bytes.

Allocate adds empty space to the collection, so that subsequence write operations work more quickly. It acts similarly to SELF.collection.StartEdit(slack).

Append (member: "TMember"); CONCEPTUAL;

member is a new member of the collection.

Append adds member to the collection immediately after the current position. position is adjusted by adding 1, so that the next Scan returns the member after the new member.

Close; DEFAULT;

Close is intended for use with disk-based files. It closes the collection scanned by this scanner object. The collection object and the scanner object are not deallocated; you can use the scanner.Open method to reopen the collection.

Compact; DEFAULT;

Compact rewrites the collection so that it uses the minimum amount of space necessary.

Delete; CONCEPTUAL;

Delete deletes the member at the current position. position is adjusted by adding the scanner.Increment.

DeleteRest; CONCEPTUAL;

DeleteRest deletes all members after the current position, if scanner.Increment is +1, or after the current position, if scanner.Increment is -1. position is unchanged. The next call to scanner.Scan returns FALSE.

Done; DEFAULT;

Done signals that you are finished using this scanner. The next call to scanner.Scan returns FALSE, as if the end of the collection had been reached. Unlike when the end of the collection is reached, however, the returned collection member in Scan remains unchanged.

Obtain: "TMember"; CONCEPTUAL;

The return value is a member of the collection.

Obtain returns the current character.

Open; DEFAULT;

Open is intended mainly for use with disk-based files. It opens a collection previously closed by scanner.Close. The collection must have been closed by this scanner.

Replace (member: "TMember"); CONCEPTUAL;

member is a member that replaces the current collection member.

Replace removes the current character from the collection and replaces it with the given character. The new character is now the current character.

Reverse; DEFAULT;

Reverse changes the direction of the scan.

Scan (VAR member: "TMember"): BOOLEAN; CONCEPTUAL;

member is the next member of the collection.
The return value is FALSE if the end of the collection has been reached or scanner.Done has been called.

Scan begins at the beginning of the collection and, each time it is called, returns the next member. The returned member is referred to as the *current* member. The value of Scan is TRUE until Scan is called after the last member in the collection is reached, or until scanner.Done has been called. If the end of the collection was reached, and Done was not called, the value of member is NIL. If Done was called, member keeps the last value it had. When method Done is called or when the scanner reaches the end of the collection, the scanner is freed.

Seek (newPosition: LONGINT); DEFAULT;

newPosition is the location to which you wish the scanner to move. newPosition = 0 is the beginning of the collection.

Seek moves the current scan position to newPosition. It transfers no data.

Skip (deltaPos: LONGINT); DEFAULT;

deltaPos is the number of bytes you wish the scan position to move within the collection. A positive value moves the position toward the end of the collection; a negative value moves the position toward the beginning of the collection.

Skip moves the scan position deltaPos bytes forward or backward within the collection. It transfers no data.

NOTES:

1. A scanner is used to access the various types of collections.
2. The ToolKit-defined collection types are files, strings, lists, and arrays. There is a subclass of TScanner for scanning each of those classes. See TFileScanner, TStringScanner, TArrayScanner, and TListScanner for more information.
3. In general, before you can use a scanner to access a collection, you must create a collection object of the right type. See TArray, TList, TFile, and TString for more information.
4. CONCEPTUAL methods are not implemented, or even defined in the interface. They are here as models only; every subclass can define its interface to these methods in its own way.

CLASS: TScrollBar

SUPERCLASS: TObject

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: none

DATA FIELDS:

R firstBox: TScroller;	The first scroller for this scroll bar. Other scrollers are found through a field of this scroller.
R isVisible: BOOLEAN;	TRUE if this scroll bar should be drawn.

METHODS: None that are important for applications.

NOTES:

1. Applications never subclass or otherwise deal directly with this class.

CLASS: TScroller

SUPERCLASS: TObject

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: none

DATA FIELDS:

R	scrollBar: TScrollBar;	The scroll bar that contains this scroller.
R	band: TBand;	The band affected by this scroller.
R	sBoxID: TSBoxID;	The scroll bar library representation of the entire scroll bar. If there is more than one scroller in the scroll bar containing this scroller, the ToolKit finds the others through this field.

METHODS: None that are important for applications.

NOTES:

1. Applications never subclass or otherwise deal directly with this class.
2. This call defines and controls the mechanisms that scroll bands.

***CLASS: TSelection

SUPERCLASS: TObject

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS: none

DATA FIELDS:

R	window: TWindow;	The window object in which the selection was made.
R	panel: TPanel;	The panel object in which the selection was made.
R	view: TView;	The view object of panel object in which the selection was made.
RW	kind: INTEGER;	The kind of selection. 0 means no selected object (nothingKind). The rest of the codes are defined by the view object.
RW	anchorLPt: LPoint;	The place the mouse went down (view-relative).
RW	currLPt: LPoint;	The place the mouse was last tracked.
R	boundLRect: LRect;	The bounding box of the selction.
RW	coSelection: TSelection;	If this selection does not implement certain commands, the ToolKit sends the commands on to coSelection, unless coSelection is NIL.
R	canCrossPanels: BOOLEAN;	TRUE if the selection can be in two panels. Used for cross-panel dragging.

METHODS YOUR APPLICATION WILL CALL:

CREATE (object: TObject; heap: THeap; itsView: TView; itsKind: INTEGER; itsAnchorLPt: LPoint): TSelection;

heap is the heap used for this document.
itsView references the view that displays the selected object.
itsKind is the type of the selection. 0 indicates there is no selected object. Other values are defined by the application.
itsAnchorLPoint is the mouse-down point. Used solely by the application.

TSelection.CREATE creates an object of type TSelection.

CantDoIt;

TSelection.CantDoIt is called by other routines when a command cannot be done. A warning message from the phrase file is displayed.

FreedAndReplacedBy (selection: TSelection): TSelection;

selection is the new selection (usually a CREATE method call to TSelection or some descendant of TSelection.)
The return value is the new selection.

TSelection.FreedAndReplacedBy replaces the old selection object with a new selection, keeping the same handle for the new selection as the old one had. You should always use this method when changing selections. This makes certain that handles in your application that pointed to the old selection to point to the new selection.

METHODS YOUR APPLICATION OFTEN OVERRIDES:

NewCommand (cmdNumber: TCmdNumber): TCommand; DEFAULT;

cmdNumber indicates the menu command chosen.
The return value is a command object, or NIL.

TSelection.NewCommand calls window.NewCommand if there is no coSelection. You must write your own selection.NewCommand method in order to do selection-specific command processing. If you do so, construct the NewCommand procedure around a CASE statement with a case for each of your commands. End it with OTHERWISE NewCommand:=TSelection.NewCommand. Return NIL or a new command object. NIL means the document was not changed signficantly. In that case, NewCommand must carry out the action of the command itself. Return an instance of TCommand when the command cannot be undone. In that case, first call window.CommitLast to Commit the last command object, then carry out the action of the command, and then return an instance of TCommand. In all other cases, return NIL or an instance of a descendant of TCommand. The descendant of TCommand has methods that perform the action of the command. See the ToolKit Segments for more information.

Highlight (highTransit: THighTransit); DEFAULT;

highTransit is the change in highlighting of the displayed object. The choices are: hNone, hOffToDim, hOffToOn, hDimToOn, hDimToOff, hOnToOff, and hOnToDim. Dim highlighting is normally used when the window is inactive.

Highlight changes the appearance of an object when it is selected and deselected, and when the window is activated or deactivated. Your Highlight method should not try to keep track of the state of highlighting. The ToolKit always calls this method with the proper hightransit value. Highlighting is normally done using XOR. Call SetPenToHighlight(highTransit) and then FrameLRect, PaintLRect, or some other appropriate drawing routine.

MouseMove (mouseLPt: LPoint); DEFAULT;

mouseLPt is the view-relative point where the pointer is located.

You may write a **MouseMove** routine that takes some application-dependent action, such as drawing something, when the mouse button is still down. **TSelection.MouseMove** does nothing.

MouseRelease; DEFAULT;

You may write a **MouseRelease** routine that takes some application-dependent action, such as creating a command object, when the mouse button is released. **TSelection.MouseRelease** does nothing.

METHODS YOUR APPLICATION MIGHT OVERRIDE:

MousePress (mouseLPt: LPoint); DEFAULT;

mouseLPt is the view-relative point where the pointer was located when mouse was pressed.

You may write a **MousePress** routine that takes some application-dependent action when the mouse is pressed. **TSelection.MousePress** does nothing.

INTERNAL METHODS:

DoKey (ascii: CHAR; keyCap: BYTE; shiftKey, appleKey, optionKey: BOOLEAN);

ascii is the character typed at the keyboard.
keyCap defines the actual key pressed on the keyboard. See the HWINT documentation in the Internals document.
shiftkey tells whether or not the shift key was pressed along with the character.
appleKey tells whether or not the Apple key was pressed along with the character.
optionKey tells whether or not the option key was pressed along with the character.

TSelection.DoKey determines what action is to be taken when a key is pressed. If **appleKey** is **TRUE**, the key combination is checked to see if it is one of the menu commands. If so, **window.DoCommand** is called. Otherwise, the method checks if there is a selected object. If not, **process.Stop** is called and a warning message is displayed. If there is a selected object, **selection.DoKey** checks which editing or character key was pressed. Depending on the key, one of the key methods listed under the following section, **INTERNAL NO-OP METHODS**, is invoked. When the ToolKit is used without those overridden (often by a building block), all produce an alert to the

user. All of these routines are implemented in one or several building blocks. You can implement these routines yourself, if you wish. You would probably override DoKey if you were writing a terminal emulator.

INTERNAL NO-OP METHODS:

All of these routines are implemented (or will be implemented) in a number of building blocks. In particular, the Text Building Block implements these in a way appropriate for text. You can override any of these routines.

KeyBack(fWord: BOOLEAN); DEFAULT;

KeyChar(ch: CHAR); DEFAULT;

KeyClear; DEFAULT;

KeyEnter(dh, dv: INTEGER); DEFAULT;

KeyForward(fWord: BOOLEAN); DEFAULT;

KeyPause; DEFAULT;

KeyReturn; DEFAULT;

KeyTab(fBackward: BOOLEAN); DEFAULT;

SelectedParagraphs; DEFAULT;

This group of methods handle keyboard entry. They are usually called by the ToolKit.

OTHER METHODS:

Clone (heap: THeap): TObject;

heap is the document heap.

TSelection.Clone creates a copy of the coSelection, as well as of the selection. You can override this method.

GetHysteresis (VAR hysterPt: Point); DEFAULT;

hysterPt is the current hysteresis, expressed with a vertical (hysterPt.v) and horizontal (hysterPt.h) component.

GetHysteresis gets the current mouse hysteresis. You can override this method.

HaveView (view: TView);

view is the view in which the selection is seen.

This method is called to change the view pointed to by the selection. Do not call this method. You might want to call TPanel.HaveView, however.

MarkChanged; DEFAULT;

MarkChanged marks the document as having been changed.

CanDoCommand (cmdNumber: TCmdNumber; VAR checkIt: BOOLEAN):
BOOLEAN; DEFAULT;

cmdNumber is the reference number for an application or ToolKit command.
checkIt tells whether or not the correspoding menu item should have a checkmark next to it.
The return value indicates whether or not the command can be done.

CanDoCommand reports whether the command referred to by cmdNumber can be done. You often need to override this method. When you do so, if the command is not one of your application's commands, call CanDoCommand:=SELF.window.CanDoCommand. See the ToolKit segments for a more complete description.

IdleBegin (centiSeconds: LONGINT); DEFAULT;

centiSeconds the amount of time, in hundredths of a second, since idle time started.

TSelection.IdleBegin calls coSelection.IdleBegin, if there is a coSelection, or TWindow.IdleBegin, if there is not. TWindow.IdleBegin lets other processes run. You may want to override this method to do idle time tasks that are very time critical.

IdleContinue (centiSeconds: LONGINT); DEFAULT;

centiSeconds the amount of time, in hundredths of a second, since idle time started.

TSelection.IdleContinue calls coSelection.IdleContinue, if there is a coSelection, or window.IdleContinue, if there is not. TWindow.IdleContinue calls process.TrackCursor, if the process is active. If this process is not active, TWindow.IdleContinue lets other processes run. You may want to override this method to do idle time tasks that should be repeated and can be put off until this time.

IdleEnd (centiSeconds: LONGINT); DEFAULT;

centiSeconds the amount of time, in hundredths of a second, since idle time started.

TSelection.IdleEnd calls coSelection.IdleEnd, if there is a coSelection, or window.IdleEnd, if there is not. TWindow.IdleEnd does nothing. You may want ot override this method to stop processing of your idle time tasks.

Deselect; DEFAULT;

TSelection.Deselect turns off highlighting of the selection and replace it with noSelection.

MoveBackToAnchor; DEFAULT;

MoveBackToAnchor is called by the ToolKit when a cross-panel drag is attempted, and it is not successful for some reason. In that case, MoveBackToAnchor should restore the selected object to its position before the drag attempt was begun. If you want that to happen, you must implement this method. TSelection.MoveBackToAnchor does nothing.

Restore; DEFAULT;

Restore replaces the current selection with the last selection saved with selection.Save, which is normally the selection before the last command. You may want to override this method.

Save; DEFAULT;

Save stores the current selection so that it can be restored if the user requests an undo of the last command. You may want to override this method.

Reveal (asMuchAsPossible: BOOLEAN); DEFAULT;

asMuchAsPossible indicates whether or not the selection should be shown to the greatest extent possible.

Reveal controls the displaying of the selection. If asMuchAsPossible is TRUE, the panel is scrolled whenever the user does anything that affects the selection so that the largest part of the selected objects are displayed as can be shown. If asMuchAsPossible is FALSE, the display is not changed to show more of the selection.

DrawGhost;

DrawGhost can be implemented to draw an XORed ghost image during cross panel dragging. It is called by the ToolKit, between each call to MousePress and the first call to MouseMove. If you imlement this method, you should also implement MouseMove so that it moves the ghost image. TSelection.MouseMove does nothing.

FAILURE CONDITIONS: none

NOTES:

1. This class is usually subclassed for use. However, when no object is selected, the null selection object can be an instance of TSelection itself with the kind field set to nothingKind.
2. The descendant of TSelection usually contains a handle on the selected object, or a list of handles on the selected objects.

3. The selected object is one of the objects displayed by the **view** object associated with the **panel** object containing this **selection**. Commands often apply only to a selected object. There can be many selected objects in each **panel**, or no selected objects. Each **panel** object, however, always has a **selection** (an instance of **TSelection** or a descendant of **TSelection**), even when there are no selected objects.
4. Successive **selection** objects in a particular **panel** object use the same object handle. Use the method **selection.FreedAndReplacedBy** to free the old **selection** and give its handle to the new **selection**.
5. If there is more than one **selection**, the one in the active **window's selectPanel** is active and receives **commands**.
6. If there is a selected object in the **selectPanel**, it is always highlighted in some way, while **selections** in other panels may or may not be highlighted.
7. **coSelection** is used when you want to take advantage of a building block other other pre-defined unit's action on most edit key events, while implementing different action on some edit key events.
8. To use the **coSelection** feature, create your own descendant of **TSelection**, and put the building block's **selection** in the **coSelection** field of your application's **selection** object. The ToolKit's default edit key routines, which are inherited by your descendant of **TSelection**, all check to see if there is a **coSelection**, and, if there is one, send the key event to the **coSelection**.
9. For the key events that you want handled differently from the way the building block handles them, reimplement that edit key routine in your descendant of **TSelection**. Since your routine presumably would not check for a **coSelection**, the **coSelection** is ignored. You can always send some keys to the **coSelection** and handle others yourself.
10. Successive **coSelections** of a particular **selection** object may use the same object handle. Use **FreedAndReplacedBy** to change **coSelections**. See note 4.
11. See the ToolKit Segments for examples and more details of use of **selections**.

CLASS: TString

SUPERCLASS: TCollection

DEFINED SUBCLASSES: none

INHERITED DATA FIELDS:

size: LONGINT;	The number of real elements in this list, not counting the hole. This is a LONGINT for the benefit of huge collections, such as remote data bases. It is always in the INTEGER range for instances of TString.
dynStart: INTEGER;	The number of bytes from the class pointer to the dynamic data area.
holeStart: INTEGER;	0 means hole at the beginning; value of size means hole at the end.
holeSize: INTEGER;	The initial size of the hole, measured in the number of members that can fit in the hole.
holeStd: INTEGER;	How much to grow the collection by if the holeSize goes to 0.

DATA FIELDS: none

METHODS:

CREATE (object: TObject; itsHeap: THeap; initialSlack: INTEGER): TString;

initialSlack is the size of the initial hole, in member-sized units.

TString.CREATE creates an empty string on itsHeap. initialSlack is a hole for string members. Because the string is created with a hole, the insert methods can be used to initialize the list, without allocating any space.

At (i: LONGINT): CHAR; DEFAULT;

i is an ordinal position in the string.
The return value is the character at the given position.

At returns the character at the given position in the string.

DelAll;

DelAll deletes all the characters of a string. The string itself is not deleted.

DelAt (i: LONGINT); DEFAULT;

i is an ordinal position in the string.

DelAt deletes the character at position i; characters after i are then moved to fill in the empty space.

DelFirst;

DelFirst deletes the character at the beginning of the string.

DelLast;

DelLast deletes the character at the end of the string.

DelManyAt (i, howMany: LONGINT);

i is an ordinal position in the string.
howMany indicates the number of string characters to be deleted.

DelManyAt deletes a number of characters from the string. The first deleted character is at position i.

Draw (i: LONGINT; howMany: INTEGER);

i is an ordinal position in the string.
howMany indicates the number of string characters to be drawn.

Draw calls QuickDraw to write the given part of the string at the current pen position.

Each (PROCEDURE DoToCharacter(character: CHAR));

DoToObject is PROCEDURE that takes a character as its argument.

Each applies the given PROCEDURE to each character in the string.

First: CHAR;

The return value is a character.

First returns the first character in the string.

InsAt (i: LONGINT; character: CHAR);

i is an ordinal position in the string.
character is a character.

InsAt inserts an character in the string. The newly inserted character occupies position i.

InsFirst (character: CHAR);

character is a character.

InsFirst inserts an character at the beginning of the string. The newly inserted character occupies the first position in the string.

InsPStrAt (i: LONGINT; pStr: TPString);

i is an ordinal position in the string.
pStr is a pointer to a string.

InsPStrAt inserts a string into this string at position i.

InsLast (character: CHAR);

character is a character.

InsLast inserts an character at the end of the string. The newly inserted character occupies the last position in the string.

Last: CHAR;

The return value is a character.

Last returns the character at the end of the string.

ManyAt (i, howMany: LONGINT): TString;

i is an ordinal position in the string.
howMany is a number of characters.

ManyAt returns a string with the characters from the original string beginning at position i and continuing through howMany characters.

MemberBytes: INTEGER; OVERRIDE;

MemberBytes returns the size of a member of the string, which is always 1. (This method is more useful for other types of collections.)

Pos (after: LONGINT; character: CHAR): LONGINT;

after is the index number of a character in the string.
character is a character.

Pos finds the positions of the first instance of character after position after.

PutAt (i: LONGINT; character: CHAR);

i is an ordinal position in the string.
character is a character.

PutAt deletes the character at position i, and replaces it with character.

Scanner: TStringScanner;

The return value is a **stringScanner** for this **string**.

Scanner returns an object of class **TStringScanner**, allowing use of **stringScanner** methods. This is equivalent to

```
string.ScannerFrom (1, scanForward);
```

ScannerFrom (firstToScan: LONGINT; scanDirection: TScanDirection): TStringScanner; DEFAULT;

firstToScan is a position in the **string**.
scanDirection **scanForward** or **scanBackward**.
The return value is a new **stringScanner**.

ScannerFrom returns an object of class **TStringScanner**, with **stringScanner.Position** equal to **firstToScan** minus one (or plus one, if **scanDirection** is **scanBackward**), so that the first call to **stringScanner.Scan** returns the character at **firstToScan**.

StartEdit (withSlack: INTEGER);

withSlack is the new value for **holeStd**.

StartEdit changes the value of **holeStd** to **withSlack**, so the next edit call creates a hole of size **withSlack**, so that subsequent edit calls act more quickly.

StopEdit;

StopEdit removes the hole from the array and sets **holeStd** to 0 (zero). Any subsequent edit call removes any hole it forms.

ToPStr (pStr: TPString);

pStr is a pointer to a **string**.

ToPStr transfers this **string** into the **string** at **pStr**.

ToPStrAt (i, howMany: LONGINT; pStr: TPString);

i is an ordinal position in the **string**.
howMany is a number of characters.
pStr is a pointer to a **string**.

ToPStrAt transfers part of this **string** into the **string** at **pStr**.

Width (i: LONGINT; howMany: INTEGER);

i is an ordinal position in the **string**.
howMany is a number of characters.

Width calls the QuickDraw routine, **TextWidth**.

FAILURE CONDITIONS: Heap can't grow.
Array > 32K bytes.
Deleting from empty **string**.
Subscript out of range.

NOTES:

1. A **string** is a space-optimized list of characters, similar to an **array**, except that TArray requires even **memberBytes**, and TString has a **memberByte** value of 1.
2. **strings** have three modes: *create mode*, *edit mode*, and *static mode*.
3. When you first create a **string**, it is in create mode. You define an **initialSlack** value in the CREATE call. The **string** is given enough empty space to hold **initialSlack** characters. That space is the hole. As you add characters, the space in the hole is used for the new characters. The amount of space allocated to the **string** does not change until you fill up the hole with characters. You can also call **string.StopEdit**, which removes the hole from the **string**.
4. When the hole is filled, the **string** enters static mode. In static mode, no hole is ever maintained. If you add a characters, the **string** is copied into a space large enough to hold the additional characters. If you delete a character, the extra space is freed.
5. Enter edit mode by calling **string.StartEdit(withSlack)**. In edit mode, a hole big enough for **withSlack** characters is initially created. As you add characters, the size of the hole decreases. When the hole is entirely filled, the space allocated to the **string** is increased, so there is a new hole big enough for **withSlack** characters. If you delete characters, the extra space is added to the hole. Call **string.StopEdit** to stop editing. Any space taken up by the hole is then freed.

CLASS: TStringScanner

SUPERCLASS: TScanner

DEFINED SUBCLASS: TFileScanner

INHERITED DATA FIELDS:

R	collection: TCollection;	The string being scanned.
R	position: LONGINT;	The current position of the scanner. The scanner position is always between, before, or after members: 0=before first, size+1=after last.
R	increment: INTEGER;	The change in position for every scan: 1 if scanning forward, -1 if scanning backward.
	scanDone: BOOLEAN;	When this is TRUE, the next Scan call returns FALSE, signalling the end of the scan. The VAR parameter of the Scan method, which normally contains the next object in the string, is unchanged after the Scan call.
R	atEnd: BOOLEAN;	TRUE if the end of the list is eminent, so that the next Scan call will return FALSE.

DATA FIELDS: R actual: LONGINT; The number of bytes transferred in the last transfer operation.

METHODS:

CREATE (object: TObject; itsString: TString; itsInitialPosition: LONGINT; itsScanDirection: TScanDirection): TStringScanner;

itsInitialPosition is the initial ordinal position in the string. itsScanDirection is scanForward or scanBackward.

TStringScanner.CREATE creates an object of type TStringScanner, which is used to access itsString. itsString must exist before a stringScanner can be created for it; however, you can use a TString.CREATE call in your TStringScanner.CREATE call, which will create the necessary string object.

Append (character: CHAR); DEFAULT;

character is a new member of the string.

Append adds character to the string immediately after the current position. position is adjusted by adding 1, so that the next Scan returns the character after the new character.

Delete; DEFAULT;

Delete deletes the character at the current position. position is adjusted by subtracting 1.

DeleteRest; DEFAULT;

Delete deletes all characters after the current position. position is unchanged. The next call to stringScanner.Scan returns FALSE.

Done; DEFAULT;

Done signals that you are finished using this stringScanner. The next call to stringScanner.Scan returns FALSE, as if the end of the string had been reached. Unlike when the end of the string is reached, however, the returned character in Scan remains unchanged.

Free; OVERRIDE;

Free deallocates this stringScanner object.

Obtain: CHAR; DEFAULT;

The return value is a character from the string.

Obtain returns the current character.

ReadArray (heap: THeap; bytesPerRecord: INTEGER): TArray;

heap is a heap on which to allocate a new TArray.
bytesPerRecord is the number of bytes per record in the desired TArray.
The return value is an array containing data from the string.

readArray first reads a 2-byte count of the records in the array and then transfers that number of records times bytesPerRecord bytes from the string into the array that it allocates.

ReadNumber (numBytes: SizeOfNumber): LONGINT;

numBytes is between 1 and 4.
The return value is a LONGINT read from the current position in the string.

ReadNumber reads a number of length numBytes. If numBytes is even, the number is signed. stringScanner.Position is increased by numBytes. The number is not read as ASCII.

ReadObject (heap: THeap): TObject;

heap is a heap.
The return value is a new object.

ReadObject reads the next four bytes from the string and takes that as a class pointer. It then creates a new object on heap using that class pointer.

Replace (character: CHAR); DEFAULT;

character is a character that replaces the current character.

Replace removes the current character from the string and replaces it with the given character. The new character is now the current character.

Scan (VAR nextChar: CHAR): BOOLEAN; DEFAULT;

nextChar is the next character from the string.
The return value is FALSE if the end of the string has been reached or stringScanner.Done has been called.

Scan begins at the beginning of the string and, each time it is called, returns the next character. The returned character is referred to as the *current* character. The value of Scan is TRUE until Scan is called after the last character in the string is reached, or until stringScanner.Done has been called. If the end of the string was reached, and Done was not called, the value of nextChar is NIL. If Done was called, nextChar keeps the last value it had. When method Done is called or when the scanner reaches the end of the string, the scanner is freed.

Seek (stringPos: LONGINT);

stringPos is the location to which you wish the scanner to move. stringPos = 0 is the beginning of the string.

Seek moves the current scan position to stringPos. It transfers no data.

Skip (deltaPos: LONGINT);

deltaPos is the number of bytes you wish the scan position to move within the string. A positive value moves the position toward the end of the string; a negative value moves the position toward the beginning of the string.

Skip moves the scan position deltaPos bytes forward or backward within the string. It transfers no data.

WriteArray (a: TArray);

a is the TArray you wish copied into the string.

WriteArray copies a into the string, preceded by a 2-byte count of the number of records.

WriteNumber (value: LONGINT; numBytes: SizeOfNumber);

value is the number you want written.
numBytes is the number of bytes you want the number to occupy in the string (1 to 4).

WriteNumber writes value into numBytes bytes of the string, dropping high-order bits if numBytes < 4.

WriteObject (heap: THeap): TObject;

heap is a heap.
The return value is an object.

WriteObject is the inverse of ReadObject.

XferContiguous (whichWay: xReadWrite; collection: TCollection);

whichWay is either xRead, xWrite, or xSkip.
collection is the TCollection you are reading or writing.

XferContiguous reads or writes collection; when whichWay = xRead, it resizes collection and appends the contents of the string to the end of collection.

XferFields (whichWay: xReadWrite; object: TObject);

whichWay is either xRead, xWrite or xSkip.
object is the object from which you are reading or to which you are writing.

XferFields reads or writes the contents of the fields of object. The class pointer is not read or written.

XferRandom (whichWay: xReadWrite; pFirst: Ptr; numBytes: LONGINT;
mode: TIOMode; offset: LONGINT); DEFAULT;

whichWay is either xRead, xWrite or xSkip.
pFirst points at the first byte to read data into or write data out of. It is ignored with xSkip.
numBytes is the number of bytes read, written or skipped.
mode is mAbsolute, mRelative or mSequential.
offset is a byte position from the beginning of the string when mode is mAbsolute, or from the current string position if mRelative.

XferRandom reads, writes or skips numBytes bytes of the string, starting at the indicated position. All the other I/O methods call this method. XferRandom calls the Lisa OS directly. Reading

and writing are always done left to right (from the beginning towards the end) regardless of the **scanDirection**.

XferSequential (**whichWay**: xReadWrite; **pFirst**: Ptr; **numBytes**: INTEGER); DEFAULT;

whichWay is either **xRead**, **xWrite** or **xSkip**.
pFirst points at the first byte to read data into or write data out of. It is ignored when **whichWay** equals **xSkip**.
numBytes is the number of bytes read, written or skipped.

XferSequential reads, writes or skips the next **numBytes** bytes of the **string**. Reading and writing are always done left to right (from the beginning towards the end) regardless of the **scanDirection**.

XferPString (**whichWay**: xReadWrite; **pStr**: TPString);

whichWay is either **xRead**, **xWrite** or **xSkip**.
pStr is a pointer to a string to read or write.

XferString reads a string **pStr^** from a **string** or writes it to a **string**. If **whichWay** is **xRead**, the destination string must be long enough. The amount transferred depends on the length of **pStr^**.

NOTES:

1. A **stringScanner** is used to access a stored data **string**, to scan it, and to manipulate its individual elements. Use **TListScanner** and its subclasses to scan **lists**.
2. Before you can use a **stringScanner** to access a **string**, the **string** must be opened by **TStringScanner.CREATE**. **stringScanner.Free** closes the **string**.
3. The **Actual** variable contains the number of bytes of data most recently transferred to or from the **string**. It is the same as the number requested unless error > 0.
4. The Xfer methods are preferred, because the same procedure may then be able to transfer data to or from a **string**.
5. You may have more than one **stringScanner** open on the same **string** at the same time.
6. **WARNING:** No checking is done that sufficient space has been allotted for data reads at **pFirst^** or **pStr^**.

******CLASS: TView**

SUPERCLASS: TImage

DEFINED SUBCLASSES: TPaginatedView
TPageView

INHERITED DATA FIELDS:

R	extentLRect: LRect;	The size of the entire view.
R	view: TView;	Always contains SELF.
	allowMouseOutside: BOOLEAN;	FALSE by default. When FALSE, and the mouse point is outside the view, the point is converted to the closest point within the view. If TRUE, the point is not converted. In that case, mouse points can be outside extentLRect and, particularly, can be negative. This feature exists primarily for use with sidebands. Note that your program must be prepared to handle mousepoints outside extentLRect.

DATA FIELDS:

R	panel: TPanel;	The panel that looks on this view.
R	clickLPt: LPoint;	The last place the user clicked the mouse button.
R	printManager: TPrintManager;	The printManager for this view.
R	res: Point;	The resolution of this view, in terms of spots per inch. A spot is the smallest unit that can be displayed, equivalent to a pixel. The resolution has two dimensions: a vertical (res.v) and a horizontal (res.h) resolution.
R	fitPagesPerfectly: BOOLEAN;	Whether or not the ToolKit should alter the view size automatically so that there is always a whole number of pages.
R	isPrintable: BOOLEAN;	Whether or not this view can be printed.

R	isMainView: BOOLEAN;	Whether or not this is a main view. If FALSE, this is an auxiliary view such as a page view or paginated view.
R	stdScroll: LPoint;	The standard amount of scrolling from one click on the scroll arrow. There is a standard scroll value for two dimensions: a vertical (stdScrollV) and a horizontal (stdScrollH) scroll distance.
RW	scrollPastEnd: Point;	How much the ToolKit should scroll past the end of the view.

METHOD YOUR APPLICATION MUST OVERRIDE:

CREATE (object: TObject; heap: THeap; itsPanel: TPanel; itsExtent: LRect; itsPrintManager: TPrintManager; itsDfltMargins: LRect; itsFitPagesPerfectly:BOOLEAN; itsRes: Point; isMainView: BOOLEAN): TView;

itsPanel is the panel looking on this view.
itsExtent is the size of the view.
itsPrintManager is the printManager to be used for printing this view, or NIL, if this view cannot be printed.
itsDfltMargins is a rectangle which specifies, in view coordinates, the page margins used when printing.
itsFitPagesPerfectly indicates whether or not the ToolKit should format the view so that, when it is printed, it occupies a whole number of pages.
itsRes is the resolution of this view, in dots per inch.
isMainView is TRUE if this is the primary representation of the data, and not a paginatedView or a pageView.

TView.CREATE creates an object of type TView. Normally, your application does not call this method directly; instead, you call panel.NewView or panel.NewStatusView. Those methods call CREATE.

Draw;

You should implement a Draw method in each descendant of TView to draw the part of the view visible in the pane or on the page. Your Draw method should assume that thePad is set up to draw in one pane. (See TPad in Chapter 3 for an explanation of thePad.)

METHODS YOUR APPLICATION OFTEN OVERRIDES:

MousePress (mouseLPt: LPoint);

mouseLPt is the view-relative point where the pointer was located when the mouse button was pressed.

You normally write a MousePress routine in each descendant of

TView that takes some application-dependent action when the mouse is pressed. TView.MousePress calls TSelection.MousePress.

EachActualPart (PROCEDURE DoToObject (obj: TObject));

PROCEDURE DoToObject is any procedure that takes a TObject object-reference variable as an argument. DoToObject cannot be a method.

EachActualPart is intended to apply DoToObject to each object in the view's set of objects. If you store your application's objects in the window, you should implement window.EachActualPart rather than view.EachActualPart. TView.EachActualPart actually calls TWindow.EachActualPart, which returns an error message indicating that EachActualPart was not implemented.

METHODS YOUR APPLICATION MIGHT OVERRIDE:

MouseMove (mouseLPt: LPoint);

mouseLPt is the window-relative point where the pointer is located.

TView.MouseMove calls TSelection.MouseMove. You may write a routine that takes some application-dependent action when the mouse button is still down. More often, your MousePress method creates a selection that handles the mouse move itself. View.MouseMove is always called at least once before view.MouseRelease is called.

MouseRelease;

TView.MouseRelease calls TSelection.MouseRelease. You may write a routine that takes some application-dependent action when the mouse button is released. More often, your MousePress method creates a selection that handles the mouse release itself. When the mouse button is released, the ToolKit calls view.MouseMove one last time with the final pointer position, and then calls view.MouseRelease.

OkToDrawIn (lRectInView: LRect): BOOLEAN;

lRectInView is a view-relative rectangle.

TView.OkToDrawIn always returns FALSE. You may reimplement it to improve the performance of your application. It is called when the application or a building block wants to draw or erase from command or mouse code. Panel.InvalLRect is normally called to cause the ToolKit to tell the view to draw the next time window.Update is called. If, for speed, you want the application or building block to draw directly, you must call panel.OkToDrawIn to ask for permission. The panel object may deny permission if page breaks are in the way, or may call the view.OkToDrawIn method to ask the view object for permission. If the application is certain that only one object's image is in lRectInView (as opposed to several layered images), it can return TRUE. Note that it is permissible to

draw in the view using XOR, as is sometimes done in Highlight methods, without asking permission.

SetMinViewSize (VAR minLRect: lRect);

minLRect is the minimum view size acceptable to the application.

SetMinViewSize is called by the ToolKit to obtain the minimum view size. It returns the views current extentLRect by default. You can reimplement this method if you want to return some other value. The actual size of the view set by the ToolKit can be larger than the value given by SetMinViewSize, depending on the value of view.fitPagesPerfectly.

OTHER METHODS:

AddStripOfPages (vhs: VHSelect); DEFAULT;

vhs tells whether a vertical or horizontal strip is to be added.

AddStripOfPages increases the view size by one vertical or horizontal strip of pages.

BeInPanel (panel: TPanel);

panel is one of the application's panels.

BeInPanel installs this view in the given panel.

CursorAt (mouseLPt: LPoint): TCursorNumber; DEFAULT;

mouseLPt is the view-relative point where the pointer is located. The return value is the type of cursor to be set, or noCursor.

CursorAt returns the position of the mouse pointer. The return value indicates the kind of cursor that is to be displayed. If you do not care, you can set the value to noCursor. In that case, the ToolKit displays the default cursor, usually the arrow cursor. Applications often override this method.

DoReceive (selection: TSelection; lPtInView: LPoint): BOOLEAN;

selection is the selection object that points to objects that the user is dragging.
lPtInView is the point where the objects were left by the user.
The return value is whether or not this operation can be done.

DoReceive is used to implement cross-panel dragging.
TView.DoReceive always returns FALSE. If you want to implement cross-panel dragging, your view should pick up the selection and do something appropriate with it, and return TRUE if the operation is successful. Applications might override this method.

ForceBreakAt (vhs: VHSelect; precedingLocation: LONGINT;
proposedLocation: LONGINT): LONGINT;

vhs indicates either or vertical or a horizontal page break.
precedingLocation is the location of the preceding page break.
proposedLocation is the proposed page break location.
The return value is where the page break is actually placed.

The ToolKit calls ForceBreakAt before setting each page break. You can override the default implementation to examine the location of the page break and decide where you want the break to actually be. Return that value. The return value must be greater than precedingLocation and less than or equal to proposedLocation.

GetStdScroll (VAR deltaLStd: LPoint);

deltaLStd is the preset standard scroll. deltaLStd.v is the vertical scroll; deltaLStd.h is the horizontal scroll. Both are expressed in dots, which are the units of the view's resolution.

GetStdScroll finds out the standard scroll values. The values indicate the distance scrolled for each click on a scroll arrow.

MaxPageToPrint: LONGINT;

The return value is a page number.

MaxPageToPrint returns the last page that will print when this view is printed. By default, that page number is derived by multiplying the number of columns of pages times the number of rows of pages. You can override this method if you want so that some of the highest-numbered pages are not printed. For example, you might want to do that if your application is a spreadsheet, and only the first few cells have data in them.

NoSelection: TSelection;

The return value is a null selection.

TView.NoSelection returns a null selection for use when the view has no selected object. You can override this routine so that it returns a selection object of your own selection class.

RedoBreaks; DEFAULT;

RedoBreaks orders the ToolKit to recalculate page breaks. You can override this method, if you wish.

RemapManualBreaks (FUNCTION NewBreakLocation(vhs: VHSelect;
oldBreak: LONGINT): LONGINT);

NewBreakLocation is a function that calculates new page breaks. vhs indicates whether the page breaks are vertical or horzontal. oldBreak is the old page break location.

RemapManualBreaks changes the locations of all page breaks. You never override it, but you might call it. Call this method if the locations of manual page breaks in your view are dependent on printer metrics. If that is the case, when printer metrics change, you need to recompute the locations of the manual breaks. Call RemapManualBreaks from your view.ReactToPrinterChange method. To do so, you must first create a FUNCTION that takes the old manual break location and returns the corresponding new location.

When TView.RemapManualBreaks is called, it first clears all automatic page breaks. RemapManualBreaks then hands your NewBreakLocation function each manual page break, one at a time. RemapManualBreaks replaces each old manual page break with the corresponding new page break returned by NewBreakLocation. Finally, after all the manual page breaks have been processed, RemapManualBreaks calls RedoBreaks, which recomputes the automatic page breaks.

Resize (newExtent: LRect); DEFAULT;

newExtent is the new view size.

Resize sets the size of the view to the indicated new size, and removes any superfluous page breaks.

SetFunctionValue (keyword: S255; VAR itsValue: S255);

keyword is the name of a variable whose value you want. itsValue is the value of the variable.

SetFunctionValue finds the value of the given variable, and returns it as a text string. TView.SetFunctionValue works only for {PAGE} and {TITLE}. If you want to use this to get the values of other variables, you must re-implement it.

FAILURE CONDITIONS: none

NOTES:

1. You normally subclass TView and then use the subclass to create the application's view, or one of the application's views. The subclass often contains data fields for the information being viewed. For example, if the information displayed in a view is stored as a list of objects, then the subclass of TView adds one field of class TList. As another example, the window object could contain fields referencing document objects, and the view could access them through the view.panel.window field.

2. Usually only a portion of the view is displayed on the screen at any one time.
3. The parts of the view that are displayed are displayed by **pane** and **bodyPad** objects.
4. Every panel contains one or more panes, each of which may display a different portion of the **view** object.
5. A window may contain a number of panels, each of which looks on a different **view** object.
6. Invalidated screen areas are updated between events, and when the application requests. Also, automatic scrolling may occur while the mouse button is down, depending on the position of the mouse pointer.

****CLASS: TWindow

SUPERCLASS: TArea

DEFINED SUBCLASS: TDialogBox

INHERITED DATA FIELDS:

R	innerRect: Rect;	Contains the size of the window, excluding the entire frame.
R	outerRect: Rect;	Contains the size of the window, including the frame.
	parentBranch: TBranchArea;	Not used in TWindow.

DATA FIELDS:

R	panels: TList {OF TPanel};	The panels in the window. There is always at least one. See class TPanel.
	panelTree: TArea;	When the window contains no panels (when it is first created) contains NIL. When the window contains one panel, contains the panel. When the window contains more than one panel, contains a TBranchArea that points to the panels.
R	dialogBox: TDialogBox;	References the dialogbox for this window. NIL if SELF is a dialog box window, or if there is no dialog box for this window.
R	selectPanel: TPanel;	The panel with the active selection.
R	undoSelPanel: TPanel;	The selectPanel during the last command.
	clickPanel: TPanel;	The panel where the mouse button was last clicked.
	undoClickPanel: TPanel;	The clickPanel during the last command.
R	selectWindow: TWindow;	The window with the active selection. Either SELF or its dialogBox.
	undoSelWindow: TWindow;	The window with the active selection during the last command.
	wmgrID: TWindowID	ORD of the pointer to the Window Manager's grafPort.
R	isResizable: BOOLEAN;	Tells whether or not there is a Resize Box.

	believeWmgr: BOOLEAN;	If TRUE, the Toolkit should believe the window manager's idea of the size of the window; this is FALSE (for example) when the application creates the window object before the window is put on the screen.
	maxInnerSize: Point;	The window size the user explicitly set by using the grow icon.
	changes: LONGINT;	Shows the number of changes since the last save.
R	lastCmd: TCommand;	The last command object that can be undone.
	printerMetrics: TPrinterMetrics;	Properties of the printer.
	pgSzOK: BOOLEAN;	Whether to allow user-defined page-sizes in Format For Printer dialog.
	pgRgOK: BOOLEAN;	Whether page-range dialog should be enabled in the dialog that appears in response to the Print... menu command. This is normally TRUE.
	panelToPrint: TPanel;	The panel to print when the user asks for printing. Note: If there is more than one printable panel in the window, choice should be made by providing separate menu items.
	objectToFree: TObject;	This field is used to hold a reference to an object that should be freed at end of the event loop.

METHOD YOUR APPLICATION MUST OVERRIDE:

BlankStationery; DEFAULT;

BlankStationery creates a panel, a view, and an initial selection. It is called when the user tears off a piece of stationery. The application is completely responsible for implementing this method. See the ToolKit Segments for a complete discussion of this method.

CREATE (object: TObject; itsHeap: THeap; itsWmgrID: TWindowID; itsResizability: BOOLEAN): TWindow;

itsHeap is the heap used for this document.
itsWmgrID is the internal identification number for the window manager.
itsResizability tells whether or not the window should have a size control box.

TWindow.CREATE creates an object of type TWindow. You need to

implement CREATE so that it creates a window of your application's subclass of TWindow. See the ToolKit Segments for a complete discussion of this method.

METHOD YOUR APPLICATION WILL CALL:

CREATE (object: TObject; itsHeap: THeap; itsWmgrID: TWindowID; itsResizability: BOOLEAN): TWindow;

itsWmgrID is the internal identification number for the window manager.
itsResizability tells whether or not the window should have a size control box.

TWindow.CREATE creates an object of type TWindow.

METHODS YOUR APPLICATION MIGHT CALL:

CommitLast; DEFAULT;

Commits the last command object, and sets window.lastCmd to NIL.

ChildWithPt (pt: Point; childList: TList; VAR nearestPt: Point): TArea;

pt is a point in window-relative coordinates.
childList is a list of areas contained within this window.
nearestPt is the point in the returned area closest to pt.
The return value is one of the areas from childList.

ChildWithPt first finds the point in window.innerRect closest to pt. It then the compares the new point to the outerRects of all the areas in childList. When it finds the area that contains the point, it finds the point in that area's innerRect that is closest to the point. That value is returned as nearestPt. The area containing nearestPt is the return value.

DoCommand (cmdNumber: TCmdNumber); DEFAULT;

TWindow.DoCommand handles command dispatch. If you override this method for some reason, you should end your routine by calling TWindow.DoCommand. In general, however, you should override TWindow.NewCommand rather than this method. You might call this method if there is an action that acts like a menu command, such as a button in the main window.

EachActualPart (PROCEDURE DoToObject (filteredObj: TObject));

See below.

EachVirtualPart (PROCEDURE DoToObject (filteredObj: TObject));

PROCEDURE DoToObject is any procedure that takes a TObject object-reference variable as an argument. DoToObject cannot be a method.

TWindow.EachVirtualPart calls TWindow.FilterDispatch which checks the last command and, if commandDoing = TRUE, calls command.EachVirtualPart to apply DoToObject to each object in the window's set of objects. If commandDoing = FALSE, TWindow.EachVirtualPart calls image.EachActualPart. If you store your application's objects in the view, you should call view.EachVirtualPart rather than window.EachVirtualPart.

ResizeTo (newSize: Point);

point is the new lower right corner of the innerRect.

ResizeTo changes the size of window.innerRect. It performs higher-level operations on the window, such as resizing areas contained within this window, and then calls SELF.REsize.

ToggleFlag (VAR flag: BOOLEAN); DEFAULT;

flag is any boolean variable.

ToggleFlag switches the value of flag, from TRUE to FALSE, or from FALSE to TRUE.

Update (doHilite: BOOLEAN); DEFAULT;

doHilite tells whether or not selections should be highlighted.

TWindow.Update updates the display within the window to fill in newly uncovered areas or areas changed by the application. Only invalidated areas are redrawn. The ToolKit calls this after every event, which includs commands, keystrokes, and calls on MousePress, MouseMove, and MouseRelease. See note 4.

METHOD YOUR APPLICATION MIGHT OVERRIDE:

EachActualPart (PROCEDURE DoToObject (filteredObj: TObject));

PROCEDURE DoToObject is any procedure that takes a TObject object-reference variable as an argument.

EachActualPart is intended to apply DoToObject to each object in the window's set of objects. If you store your application's objects in the view, you should implement view.EachActualPart rather than window.EachActualPart. TWindow.EachActualPart actually returns an error message indicating that EachActualPart was not implemented.

The following methods are listed in groups according to function.

WINDOW ATTRIBUTES METHODS:

GetTitle (VAR title: S255); DEFAULT;

title is the string in the window's title bar.

GetTitle obtains the string in the window's title bar.

IsActive: BOOLEAN; DEFAULT;

The return value indicates whether or not the window is action.

IsActive returns TRUE if this window is presently the active window.

SetWmgrId (itsWmgrId: TWindowID); DEFAULT;

itsWmgrId is a window manager ID.

SetWmgrId sets the ID for this window. It also sets the port fields of the panes in this window. Your application should not call this method.

MOUSE BUTTON METHODS:

DownEventAt (mousePt: Point); DEFAULT;

mousePt is the point where the mouse button was last pressed.

DownEventAt is called when the mouse button is pressed. This method computes double and triple mouse clicks. It also calls window.DownAt.

DownAt (mousePt: Point): BOOLEAN;

mousePt is the point where the mouse button was last pressed.

DownAt figures out where the mouse was located when the button was pressed by checking the resize box and calling panel.DownAt until it finds the right panel.

DIALOG BOX HANDLING METHODS:

PutUpDialogBox (dialogBox: TDialogBox); DEFAULT;

dialogBox is a dialogBox-reference.

PutUpDialogBox displays the given dialog box.

TakeDownDialogBox; DEFAULT;

TakeDownDialogBox removes the displayed dialog box.

DISPLAY HANDLING METHODS:

Focus;

Focus focuses the grafPort on the window.

Frame;

Frame draws the frame of the window, including the grow box. The title bar is drawn by the window manager; the scroll bars are drawn by TPanel.Frame.

Highlight (highTransit: THighTransit); DEFAULT;

highTransit is the change in highlighting of the displayed object. The choices are: hNone, hOffToDim, hOffToOn, hDimToOn, hDimToOff, hOnToOff, and hOnToDim.

Highlight changes the state of highlighting within the window. TWindow.Highlight calls panel.Highlight for every panel in the window.

Refresh (rActions: TActions; highTransit: THighTransit);

rActions is one of the set (rErase, rFrame, sBackground, rDraw). highTransit is the change in highlighting of the displayed object. The choices are: hNone, hOffToDim, hOffToOn, hDimToOn, hDimToOff, hOnToOff, and hOnToDim.

Refresh refreshes the window's display. It calls panel.Refresh for each panel in the window.

RESIZING METHODS:

DownInSizeBox (mousePt: Point); DEFAULT;

mousePt is the point where the mouse button was pressed.

DownInSizeBoxP is called by the ToolKit when the user presses the mouse button is the resize box.

Resize (moving: BOOLEAN); DEFAULT;

moving indicates whether the window is being moved (TRUE) or resized (FALSE).

Resize resets size from portRect size (with adjustments).

ResizeTo (newSize: Point); DEFAULT;

newSize sets the new lower right corner of the window.

ResizeTo changes the size of the window. Your application can call this method.

COMMAND AND MENUS METHODS:

CanDoCommand (cmdNumber: TCmdNumber; VAR checkIt: BOOLEAN): BOOLEAN; DEFAULT;

cmdNumber is the reference number for an application command. checkIt tells whether or not a check mark should appear next to the command in the menu.

CanDoCommand tells whether or not the command can be done at this time. You can override this method to handle your commands; call SUPERSELF.CanDoCommand in your method.

CanDoStdCommand (cmdNumber: TCmdNumber; VAR checkIt: BOOLEAN): BOOLEAN; DEFAULT;

cmdNumber is the reference number for a standard command. checkIt tells whether or not a check mark should appear next to the command in the menu.

CanDoStdCommand tells whether or not the command can be done at this time. You should not need to override this method.

CommitLast; DEFAULT;

CommitLast commits the last command.

LoadMenuBar; DEFAULT;

LoadMenuBar inserts the application's menus into the menu bar.

MenuEventAt (mousePt: Point); DEFAULT;

mousePt is the point where the mouse button was pressed.

MenuEventAt gives the point where the mouse button was last pressed when the pointer was in a menu. You should not call or override this method.

NewCommand (cmdNumber: TCmdNumber): TCommand; DEFAULT;

cmdNumber indicates the menu command chosen.
The return value is a TCommand object, or NIL.

You must write your own window.NewCommand method in order to process commands that apply to the entire window. If you do so, construct the NewCommand procedure around a CASE statement with a case for each of the commands you implement. End it with OTHERWISE newCommand:=TWindow.NewCommand. Return NIL or a new TCommand object. NIL means the document was not changed signficantly. In that case, NewCommand must carry out the action of the command itself. Return an instance of TCommand when the command cannot be undone. In that case, first call window.CommitLast to Commit the last command object, then carry out the action of the command, and then return an instance of

TCommand. In all other cases, return an instance of a descendant of TCommand. The descendant of TCommand has methods that perform the action of the command.

NewStdCommand (cmdNumber: TCmdNumber): TCommand;

cmdNumber is the reference number for a standard command. The return value is a command object.

NewStdCommand returns a command object for the given standard command. This method is used for standard commands that affect the entire window. You should not need to override this method.

PerformCommand (newCommand: TCommand);

newCommand is a command object.

PerformCommand performs the given command. It first calls lastCmd.Commit and then calls SELF.SaveCommand. Finally, it calls SELF.PerformLast.

PerformLast (cmdPhase: TCmdPhase);

cmdPhase is either doPhase, undoPhase, or redoPhase.

PerformLast calls Perform for the last command. The first time the command is performed, the cmdPhase is doPhase. After that, undoPhase and redoPhase alternate.

SaveCommand (newCommand: TCommand);

newCommand is a command object.

SaveCommand saves the given command as lastCommand. When referring to the command after using this method, use window.lastCmd instead of newCommand.

SetupMenus;

SetUpMenus is called when the user clicks in the menu bar or types an apple key combination. The method initiates the process of checking if the command can be done. In general, you do not override this method. Instead, implement CanDoCommand for each of you descendants of TSelection.

UndoLast;

UndoLast calls window.lastCmd.Perform. If window.lastCmd.Perform was last called with cmdPhase=doPhase or redoPhase, window.lastCmd.Perform is now called with cmdPhase=undoPhase. If window.lastCmd.Perform was last called with cmdPhase=undoPhase, window.lastCmd.Perform is now called with cmdPhase=redoPhase.

WantMenu (menuID: INTEGER; inClipboard: BOOLEAN): BOOLEAN;

menuID is a menu ID number.
inClipboard tells whether or not the clipboard is the active document.
The return value indicates whether or not this menu should be displayed in the menu bar.

WantMenu is called by the ToolKit each time the menu bar is displayed, which happens whenever the window is activated. LoadMenu calls WantMenu once for each menu in the phrase file with numbers from 1-89 in non-debug versions of the ToolKit, and with numbers from 1-99 for debug versions of the ToolKit. When inClipboard is TRUE, however, this is only called for menu 1000. WantMenu always returns TRUE by default. You can override it so that it returns FALSE for certain menus.

SELECTIONS DURING COMMANDS METHODS:

RestoreSelection;

RestoreSelection is called when undoing a command.

RevealSelection (asMuchAsPossible, doHilite: BOOLEAN);

asMuchAsPossible tells whether or not the document should be scrolled to reveal as much as possible of the selection.
doHilite tells whether or not the selection should be highlighted if it is displayed. This value is passed on to Update.

RevealSelection determines how the selection is displayed when a command is given.

SaveSelection;

SaveSelection saves the current selection in all panels. This is the reverse of RestoreSelection.

DESKTOP METHODS:

Activate;

TWindow.Activate activates the window. This method assumes that the grafPort is focused on this window. Be very careful if you override this method.

Deactivate;

TWindow.Deactivate deactivates the window. This method assumes that the grafPort is focused on this window. Be very careful if you override this method.

StashPicture (highTransit: THighTransit);

highTransit is the change in highlighting of the displayed object. The standard choices are: hNone, hOffToDim, hOffToOn, hDimToOn, hDimToOff, hOnToOff, and hOnToDim.

StashPicture saves a QuickDraw picture of the window when the window in inactive.

CURSOR METHODS:

CursorFeedback: TCursorNumber;

The return value is the current cursor type number.

CursorFeedback returns the cursor type number for the cursor that is currently displayed. Do not override this method; you can override TView.CursorAt instead. TView.CursorAt gets called as a result of this method.

PickStdCursor;

PickStdCursor displays the standard arrow cursor. Do not override this method; you can override TView.CursorAt instead.

PRINTING METHODS:

AcceptNewPrintingInfo (document: TDocManager; prReserve: TPrReserve);

document the docManager for the affected document.
prReserve holds the printer information.

This method is called when the printer formatting information changes.

ChkPrMismatch;

This method checks to see if the printer information has changed, and if it is valid.

GetPrinterMetrics;

This method finds out the characteristics of the printer for which this document is formatted.

Print (panel: TPanel; nixPgRange: BOOLEAN; nixWholeDialog: BOOLEAN);

panel is a panel in the window.
nixPgRange indicates whether or not all pages should be printed. If TRUE, the user is not given a dialog to choose a page range for printing.
nixWholeDialog, if TRUE, suppresses the entire dialog with the user.

Print prints the contents of the view displayed in the given panel.

FAILURE CONDITIONS: none

NOTES:

1. TWindow is always subclassed for use.
2. The methods of TWindow create and manage the window seen on the display. The default window object provides resizing, and activation and deactivation capabilities.
3. To use TWindow, subclass it and define your own window.BlankStationery routine. Other functions, except for application commands, are taken care of automatically.
4. When an area of the screen is changed, either by a window resize or scroll, or by a command, that area is invalidated using panel.InvalLRect or thePad.InvalLRect. Window.Update is automatically called in the process.Run control loop. All invalidated areas are then redrawn. (You can call Draw methods directly to redraw the display, but first you must seek permission from panel.OkToDrawIn.) If you want the program to redraw the display before returning to the control loop, which is rarely necessary, call window.Update after invalidating the regions to be redrawn. The easiest way to invalidate an area is with panel.InvalLRect. thePad.InvalLRect must be called through panel.OnAllPadsDo. panel.InvalLRect does that for you.