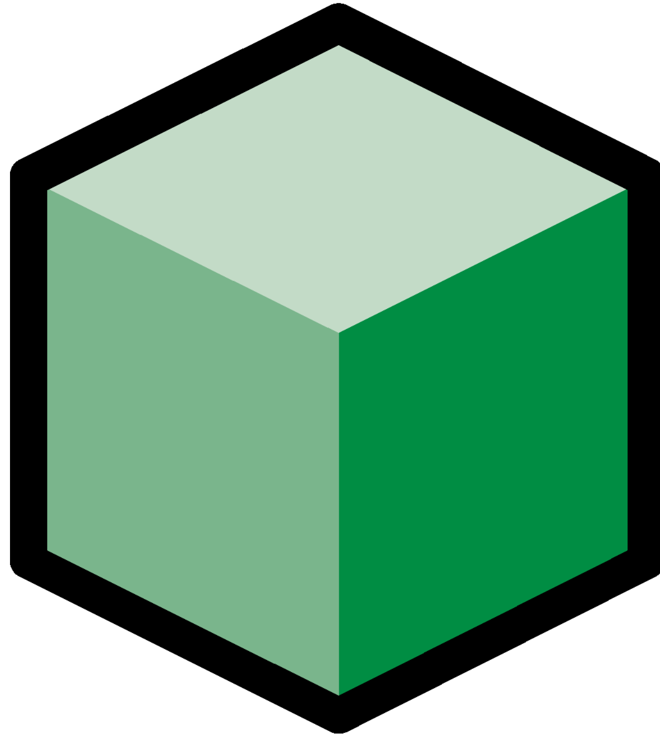
REALbasic®



DEVELOPER'S GUIDE

REALbasic Developer's Guide

Documentation by Geoff Perlman and David Brandt.
© 1999-2000 by REAL Software, Inc. All rights reserved.
Printed in U.S.A.

Mailing Address	REAL Software, Inc. 3300 Bee Caves Road Suite 650-220 Austin, TX 78746
Web Site	http://www.realsoftware.com
ftp Site	ftp://ftp.realsoftware.com
Support	support@realsoftware.com
Bugs/ Feature Requests	bugs@realsoftware.com
Sales	sales@realsoftware.com
Phone	512-263-1233
Fax	512-263-1441

V2.1, April, 2000

Contents

CHAPTER 1 **Introduction** 9

Contents 9
Welcome to REALbasic. 10
Installing REALbasic 11
Where to Begin 12
Documentation Conventions 12
Using the On-Line Help 14
Other Helpful Resources. 17
Contacting REAL Software 20

CHAPTER 2 **Getting Started with REALbasic** 23

Concepts 24
The Development Environment 26
Working with Projects 31

CHAPTER 3 **Building a User Interface** . . . 35

Working with Windows 36
Interacting with the User Through Controls 43
Object Binding 110
Adding Menus and Menu Items. 118
Apple's Macintosh User Interface Guidelines. 125

CHAPTER 4 **BASIC Programming Concepts** 127

Contents 127
BASIC versus REALbasic 128
Storing Values in Properties and Variables 129
Executing Instructions with Methods . . . 142
Comparison Operators. 147
Executing Instructions Repeatedly with Loops 149
Making Decisions with Branching. 156

CHAPTER 5 **Programming with Events and Objects** 161

Contents 162
Understanding Event-Driven Programming 162
Using The Code Editor 163
Printing Your Code. 179
Importing and Exporting Your Classes, Menus, Modules, and Windows 180
Responding To User Actions with Event Handlers 183

CHAPTER 6 **Adding Global Functionality with Modules** 217

Contents 218
Understanding Modules. 218
Adding A New Module 219
Adding Methods to Modules 220
Adding Properties to Modules 220
Adding Constants to Modules. 221
Importing and Exporting Modules 227

CHAPTER 7

Working With Text and Graphics 229

Contents 229
Working With Fonts 230
Working with the Selected Text 233
Creating a Password Field 234
Handling Styled Text 234
Formatting Numbers, Dates, and Times. . 239
Adding Pictures and Drawing Graphics . . 245
Working With Color 259
Printing Text and Graphics. 264
Transferring Text and Graphics with the Clipboard 268
Creating Animation with Sprites 272

CHAPTER 8

Working With Files 277

Contents 277
Understanding File Types 278
Understanding FolderItems 282
How Are Aliases Handled?. 283
Getting a File at a Specific Location. . . . 284
Getting The Selected Folder From An Open Folder Dialog Box 291
Using the Save As Dialog Box 293
Working With Text Files 296
Working With Styled Text Files 299
Working With Picture Files 300
Working With Sound Files 302
Working With QuickTime Movie Files. . . 303
Working With Binary Files 304
Working With Macintosh Resources . . . 308
Files Opened From the Desktop 316

CHAPTER 9

Creating Reusable Objects with Classes 319

Contents 319
The Benefits of Classes. 320
Understanding Subclasses 321
Referring To A Class's Properties and Methods From Within the Class 325
Constructors 326
Overloading 327
Modifying Classes 328
Managing Menus within Classes 333
Using Classes in Your Projects 334
The Application Class 337
Creating Custom Controls with Classes . . 339
Virtual Methods 342
Interface Inheritance. 343
Custom Object Bindings 346
Importing Classes From Other Projects . . 351
Exporting Classes For Use In Other Projects 352
Deleting Classes From a Project 353

CHAPTER 10

Creating Databases with REALbasic 355

Contents 355
REALbasic's Database Architecture 356
Structured Query Language 357
REALbasic's Database Tools 363
Creating and Modifying Databases from the Project Window 365
Using Object Binding 369
Creating a Database Front End Programmatically 371

CHAPTER 11 **Debugging Your Code** 377

Contents 377
What is Debugging? 378
The Debugger 380
Following the Execution of Methods . . . 382
Watching Your Values 386
Starting and Stopping Your Project 388
Runtime Exception Errors 388

CHAPTER 12 **Communicating With The Outside World** 395

Contents 395
Communicating With Serial Devices . . . 396
TCP/IP Communications with the Socket Control 400

CHAPTER 13 **Extending the Capabilities of REALbasic** 407

Contents 407
Using XCMDs and XFCNs. 408
Making Toolbox Calls 411
Calling AppleScripts 412
Communicating with AppleEvents 416
Using and Writing REALbasic Plug-ins . . 418
Using PowerPC Shared Libraries. 420

CHAPTER 14 **Building Stand-Alone Applications** 425

Contents 425
Building Your Application 426
Project Window Items 436

Assigning Custom Icons 437
Registering Your Creator Code 438
The Thread Manager. 438
Appendix: Region Codes. 439

CHAPTER 15 **Converting Visual Basic Projects to REALbasic**. 441

Contents 441
Importing Forms and Code 442
Making The Conversion Easier 442
What about VBX and ActiveX controls? . 443
What Are My Database Options? 444

Index 445

CHAPTER 1

Introduction

Before you get started developing applications with REALbasic, there are a few things you should know. Reading this chapter will help you understand how to install REALbasic and how to get answers to your questions.

Contents

- Welcome to REALbasic
- Installing REALbasic
- Documentation Conventions
- Using the On-Line Reference
- Other Helpful Resources
- Contacting REAL Software

Welcome to REALbasic

REALbasic makes it easy to build powerful applications quickly. If you are new to programming, you will find REALbasic's programming language easy to learn. If you are an experienced programmer, you will find the language to be powerful. In either case, you will find you can accomplish quite a bit in a short period of time.

REALbasic has a visual graphical user interface ("GUI") builder that lets you build your applications user interface without any (or very little) programming. If you know how to drag and drop, you can build an interface. REALbasic provides a rich set of interface controls and you can create your own controls as well.

REALbasic's programming language is an object-oriented version of the BASIC programming language. BASIC is an acronym that stands for Beginners All-Purpose Symbolic Code. It was originally designed to be used for teaching programming. Consequently, its syntax is less cryptic and easier to understand than most languages. REALbasic supports most of BASIC's commands. However, that is where the similarities between BASIC and REALbasic end.

Most forms of BASIC are interpreted. This means that they include a translator that has to constantly translate BASIC code into the code that the computer can actually understand. REALbasic has no interpreter. REALbasic compiles your code when you run your application. In fact, REALbasic has a dynamic recompiler. The recompiler compiles only what needs to be recompiled each time you run your project. That means that a small change to your code doesn't always require the entire project to be recompiled before it can be run.

REALbasic's form of the BASIC language is also "object-oriented." This means that it uses a modern architecture that most popular programming languages (like C++ and Java) are

using today. Object-oriented programming languages make it easier to write and debug because the code is written as individual objects that are similar to objects in the real world. In fact, in many ways REALbasic is more object-oriented than languages like C++ and certainly easier to learn and program.

REALbasic also makes application development faster and easier than traditional languages by removing the need to learn how to access the programming interface for the operating system. This application programming interface (or "API" for short) consists of 8,000 commands in the Mac OS, not one of which you ever need to learn to build applications in REALbasic.

Installing REALbasic

The REALbasic application, electronic documentation, and examples are installed by dragging files from the CD-ROM to your hard disk. To run REALbasic you must have the following:

- A Macintosh with a 68020 or greater processor or a Macintosh with any PowerPC processor.
- MacOS System 7.6.1 with the Thread Manager and Drag and Drop extensions installed. MacOS System 7.6.1 is the minimum recommended version. If you are running Mac OS 7.5 or greater, the Thread Manager and Drag and Drop are built-in to the Mac OS.
- At least 4.5 megabytes of available memory (5.5 megabytes preferred) with virtual memory on. Increase memory by approximately 2MB if virtual memory is turned off.
- A hard disk with at least 6.5 megabytes of free space available to install REALbasic, 13 megabytes available for the electronic

documentation, and 51 megabytes of space available for all the examples.

To install REALbasic from the CD ROM, drag the REALbasic application from the CD-ROM to your hard disk.

To install the documentation and examples, drag them from the REALbasic CD-ROM to your hard disk.

Where to Begin

After installing REALbasic, you should begin by going through the Tutorial. This will give you a good overview of REALbasic and introduce you to the programming language. Next, read the Developer's Guide. This guide will provide you with detailed information on the language and the various components that make up REALbasic. When you need details about a specific control or command in the language, consult the Language Reference.

Documentation Conventions

This documentation uses the following typographical conventions:

Initial References

The first time a new phrase or term is used, it will appear in *italics* for *emphasis*.

Menu References

When you are told to select a menu item, the menu name is listed first, following by an arrow, then the item name and command key shortcut. For example File ▶ Quit (⌘-Q) means "choose Quit from the File menu".

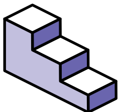
Code Examples

Code examples are all this font:

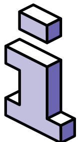
```
Dim i, x as Integer
x=0
For i = 1 to 100
 x = x + i
Next
```

Icons

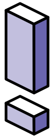
There are three icons used to call your attention to steps, and important notes:



This icon means that there are numbered steps for you to follow.



This icon means that the text to the right of it is supplemental information that clarifies a point or is relevant only to some REALbasic users.



This icon means that the text to the right of it is important information that should not be overlooked.

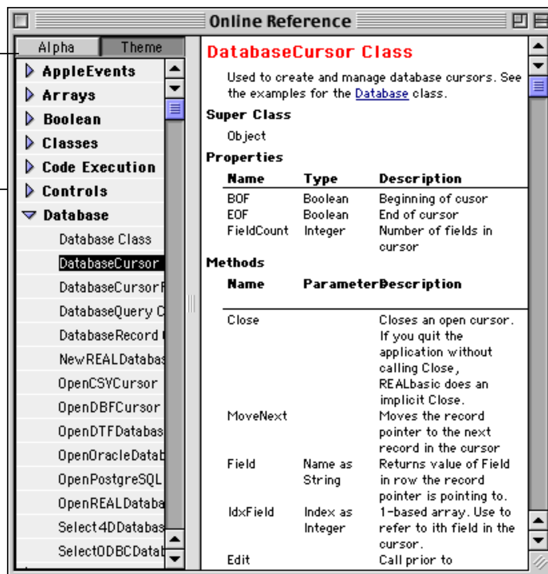
Using the On-Line Help

An electronic version of the REALbasic Language Reference is built-in to REALbasic. To access this language reference, choose Window ▶ Reference.

FIGURE 1. **The On-Line Reference**

Click to list commands by theme or alphabetically

Expand a theme to view commands



The main entries in the Language Reference are organized by theme or alphabetically. The default organization is by theme. You can change the organization to alphabetical by clicking the Alpha column heading.

The terms that are underlined and printed in blue are hypertext links within the online reference. You can click any hypertext link to read more about that term.

Each code example is surrounded by a dotted rectangle. You can drag any code example as a block to the REALbasic code editor.

Context-Sensitive Help

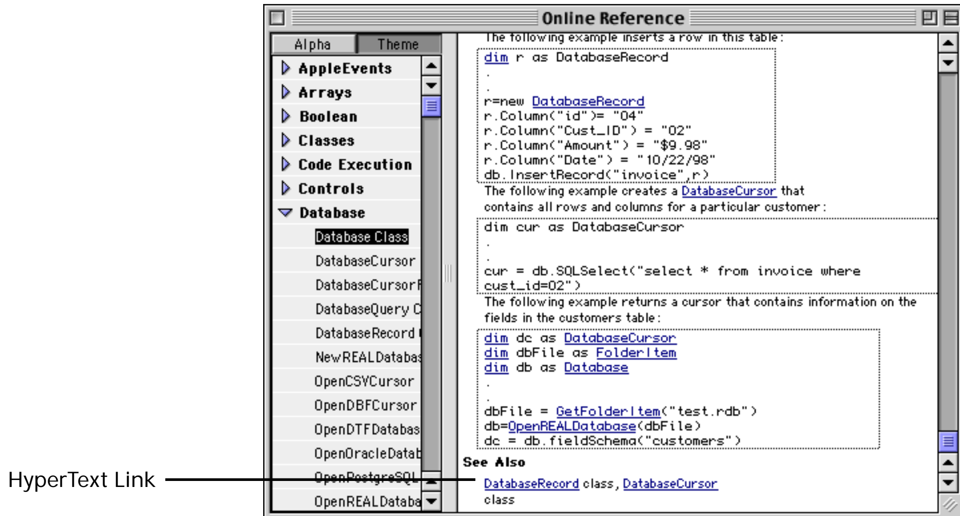
The On-Line Reference is context-sensitive. If you select an interface control in a window then open the On-Line Reference, the Reference will open to information about the selected control.

If you highlight a command in the Code Editor and open the On-Line Reference, the reference will open to information about the selected command.

Using the HyperText Links in the On-Line Help

Any text that appears in the blue, underline style in the On-Line Reference is a *hypertext link*. Clicking on the text will switch the Reference to a page about the topic you clicked on.

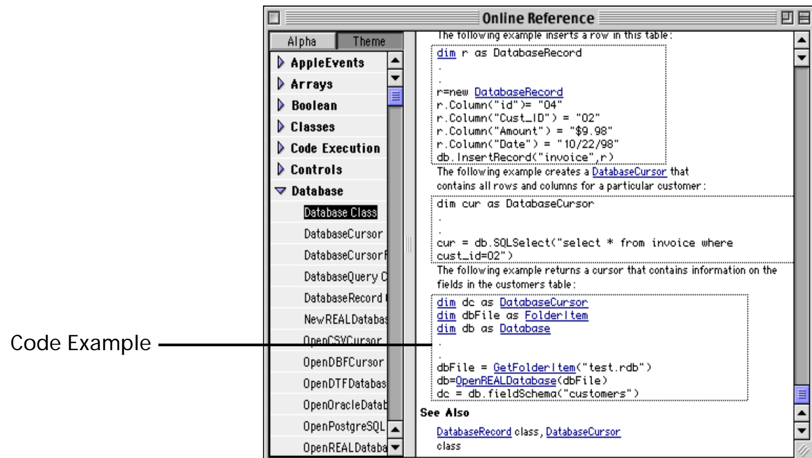
FIGURE 2. **A hypertext link in the on-line reference.**



Using the Code Examples

The On-Line Reference contains many code examples that you can use in your projects. The code examples appear in Courier font and are surrounded by a grey rectangle. Figure 3 shows an example of this. You can use these code examples in your project by dragging the grey rectangle from the On-Line Reference to your Code Editor window.

FIGURE 3. **A draggable code example in the on-line reference.**



Other Helpful Resources

There are many sources of helpful information to make learning and building powerful application easier.

Electronic Documentation

All of the REALbasic documentation is available on the REALbasic CD and at our web (http://www.realsoftware.com/release. html) and ftp (ftp://ftp.realsoftware.com) sites. These documents are available in PDF (Adobe Acrobat) and eDoc forms. The PDF version is especially easy to search and allows searching across all the Tutorial, Developer's Guide, and Language Reference. The PDF version of the Language Reference also includes the same hypertext links that are in the on-line reference.

You can purchase printed copies of the Tutorial, Developer's Guide, and Language Reference from us for an extra charge.

Our Support Web Page

Our support page is located http://www.realsoftware.com/support.html. This page is the place to check for information on REALbasic. You'll find tips, information about user groups and more.

Our FTP Site

Our ftp site is located at ftp://ftp.realsoftware.com. This site contains everything on the REALbasic CD. It also includes dozens of examples that may not be on the CD created by REAL Software and other users. You will always find the latest released version of REALbasic and the latest developer release as well.

End User Web Sites

There are dozens of web sites created by other users dedicated to REALbasic. Check our support page at www.realsoftware.com/support.html for links to these sites.

The Definitive Guide

An excellent third-party book, *REALbasic: The Definitive Guide*, by Matt Neuburg is available from O'Reilly (ISBN 1-56592-657-9). It contains in-depth discussions of the theory of event-driven, object-oriented programming, the REALbasic visual interface builder, and helpful programming tips and techniques.

The REALbasic CD

The REALbasic CD contains the latest version of REALbasic, lots of examples, and documentation. We update the CD from time to time, adding updated documentation, examples and other useful information. Most of the files on the CD are available at our web site. However, if you don't have an Internet connection or you just don't want to download hundreds of megabytes of files, you can always purchase an updated CD for a nominal fee. You can find the CD revision number of your CD in the Read Me file on the CD. You can check the purchase page of our web site to see if your CD is the latest version.

Our Internet Mailing Lists

We sponsor several Internet Mailing lists that give you the opportunity to ask questions, and share information with other REALbasic users via email. For more information on the available Internet Mailing Lists, see our support page at www.realsoftware.com/support.html.

Technical Support from REAL Software

As a registered user of REALbasic, you get one year of free technical support via electronic mail. Each time you upgrade to the latest release of REALbasic, your free technical support is extended for another year. Send your questions to support@realsoftware.com.

We also have phone support programs available at an extra charge. See the support page at our web site (http://www.realsoftware.com/support.html) for more information or call us at 512-263-1233.

Contacting REAL Software

If you need to contact REAL Software, we can be reached in the following ways:

Phone	512 263-1233 from 9AM to 6PM Central Time, Monday through Friday
Fax	512 263-1441
email	support@realsoftware.com
Mail	3300 Bee Caves Road Suite 650-220 Austin, Texas 78746 USA

Reporting Bugs and Making Feature Requests

If you think you have found a bug in REALbasic or have a feature request, please let us know about it. The best way to report bugs or make feature requests is using the REAL Bugs application available on the REALbasic CD and at our web site. This application was designed to gather all the necessary information that helps us track down bugs and implement feature requests. For each bug or feature request reported, you will receive a confirmation message via email with a tracking number you can use to check on the status of your bug report or feature request. Once we close the issue, we will email you with the reason the issue was closed (e.g., the bug has been fixed for the next release, the feature will be implemented in the next release, it's not a bug after all, etc.).

If you can't use our REAL Bugs application for some reason, please email your bug reports and feature requests to bugs@realsoftware.com.

If you don't have an email account, you can send us your bug reports and feature requests via regular mail to our mailing address or fax them to us.

Accessing The Latest Developer Release

As a registered REALbasic user, you have the opportunity to take part in the development of REALbasic. The REALbasic Developer Release is the version of REALbasic currently in development. Registered users can use this version for free while it's in development to access features as they are developed. This gives you an opportunity to give us feedback on the next release and make suggestions for features you would like to see. Keep in mind that this is a developer release so it may not be as stable as the current commercial release.

A new commercial release of REALbasic will be available approximately every six months.

The latest developer release is available at our web and ftp sites.

Web Site: http://www.realsoftware.com/release. html
Ftp Site: ftp://ftp.realsoftware.com/developer_release

CHAPTER 2

Getting Started with REALbasic

Building an application with REALbasic can take just a few minutes. First, you create your user interface which consists of menus and windows filled with interface controls. Once you have created the interface, you use REALbasic's programming language to make the interface do what you want it to do when you want it to do it!

This chapter will give you an overview of the important concepts you need to understand, the REALbasic development environment and how to work with projects.

Contents

- Concepts
- The Development Environment
- Working with Projects

Concepts

There are a few important concepts you will need to understand in order to develop applications with REALbasic. You should also be very comfortable with the graphical user interface your computer uses. If you are not, it would be a good idea to spend some time getting familiar with it before you begin using REALbasic. Otherwise, you may find many of the references in this documentation confusing.

Applications are Driven by Events

Before computers used graphical user interfaces, applications ran by simply executing a series of programming code statements starting with the first statement and ending with the last. Interfaces were all character-based. A menu was just a numbered list of commands that the user selects from to instruct the application to do a task. Most of the time, the application was just sitting there waiting for the user to make up his mind. When the user finally chose a command (perhaps by selecting the number next to the menu item and pressing the Enter key) the application would take whatever action was associated with the chosen command. When the user pressed the Enter key, an *event* occurred. In other words, something happened to which the application can respond.

Now that desktop computers use a graphical user interface, users have a far more intuitive way to interact with applications. However, one thing hasn't changed: applications are still driven by events. The difference is that back in the old days there were very few events the application had to worry about responding to. The old-fashioned application was always in a modal state: It only had to respond to the limited number of choices it presented to the user. With a graphical user interface, many more choices and ways of interacting with the computer are available. The user

might choose a menu item, click on a button, or type in a field. Also, the applications themselves may cause events to occur that were not directly caused by the user. For example, when a window opens, an event occurs (the window opened). When a window is moved or resized, an event occurs.

Fortunately, REALbasic makes it easy to deal with all of these different events. You can easily find out which events each part of your application's interface can respond to. Making your application respond to an event is as easy as locating the object that will receive the event, selecting the event, and entering the instructions (using REALbasic's programming language) you want the object to follow when the event occurs. Later on, you will learn about events in more detail. For now, it's just important to understand the concept of event-driven programming.

Developing Software with REALbasic

If you have written computer programs using traditional programming languages, you already know that the process of development is three steps: write some code, compile the code (turning the code into something the computer can really understand), and test your application. When you find a problem in your application, you start the process over again. Developing software applications with REALbasic isn't much different than that. The big difference is how often you go through this process. Compilers for traditional languages can take several minutes or more to compile an application before you can begin testing. Consequently, you spend a lot of time writing code before compiling to avoid waiting for the compiler. REALbasic's compiler is so fast that you will find you can make a small change to your code and immediately run it to make sure the change you made works as expected.

Like traditional programming language compilers, REALbasic's compiler will stop if it finds a syntactical error in your code and inform you what the error is so you can fix it. But unlike traditional compilers that require you to track down the line of code where the error occurred, REALbasic's compiler takes you right to the point in your source code where the error occurred. It then displays the error message just below the line of code that caused the error. It puts you right where you need to be to fix the problem.

If you have used traditional programming languages, you will find developing applications with REALbasic to be easier, faster and more fun.

The Development Environment

REALbasic is an *Integrated Development Environment* (IDE) which means that it contains everything you need to build an application. An interface builder, code editor, compiler and debugger are all integrated into one package. In traditional programming languages, these items would each be a separate application. REALbasic's IDE is made up of the following items:

The Menus

The menu bar provides menus for:

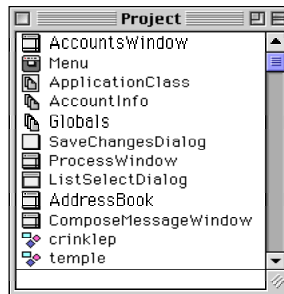
- Managing your projects
- Turning your projects into stand-alone, double-clickable applications
- Creating new windows

- Setting fonts, styles, and sizes of the objects that make up your interface
- Arranging the objects in your interface
- Testing and debugging your projects
- Getting more information about REALbasic from the on-line references

The Project Window

A *project* is the collection of items that make up a particular application you are developing. An example Project window is shown in Figure 4.

FIGURE 4. **The Project Window**

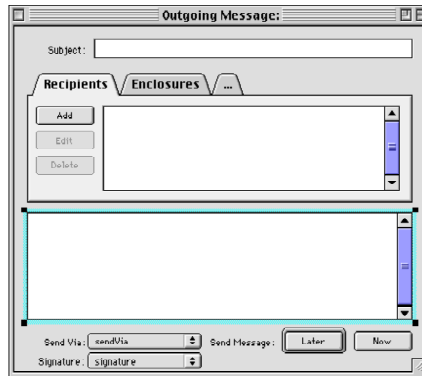


The Project window displays a list of these elements to give you easy access to them. For example, each of the windows that make up your application will be listed in the Project window. Some of the other items that might be listed in the Project window are pictures, sounds, REAL databases, QuickTime movies, as well as several others. You will learn more about projects in the next chapter.

The Window Editor

This window is used to design the user interface for a window in your project. A window created in a Window Editor is shown in Figure 5.

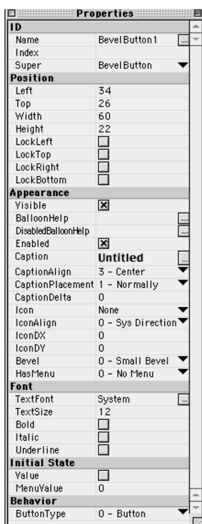
FIGURE 5. **An example window displayed in its Window Editor**



Double-clicking on one of the windows listed in the Project window displays a Window Editor. You can use the Window Editor to add all kinds of interface controls (like those in the example in Figure 5) to a window, arrange, edit, and delete them. The Window Editor is also used to access the programming code associated with the controls in your windows.

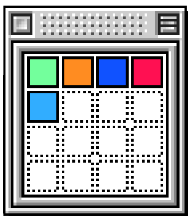
The Tools Window

This window is used to add controls to the windows you design with the Window Designer. To add a control to a window, you simply double-click on the window's name in the Project window to open it and drag the icon that represents the control you want to add from the Tools window to the window you are designing.



The Properties Window

Properties are values that are part of a particular control, such as a button or a menu item. For example, PushButtons have a Caption property that holds the button's caption. Buttons also have Left, Top, Width and Height properties which store the button's position and size. The Properties window displays all of the properties that *can be modified in the Design environment* for the currently selected item. This is an important point because some objects have properties that can be modified only by your programming code. An object may also have properties that cannot be modified or can be modified only from the Design environment. The appearance of the Properties window depends on which object is selected.



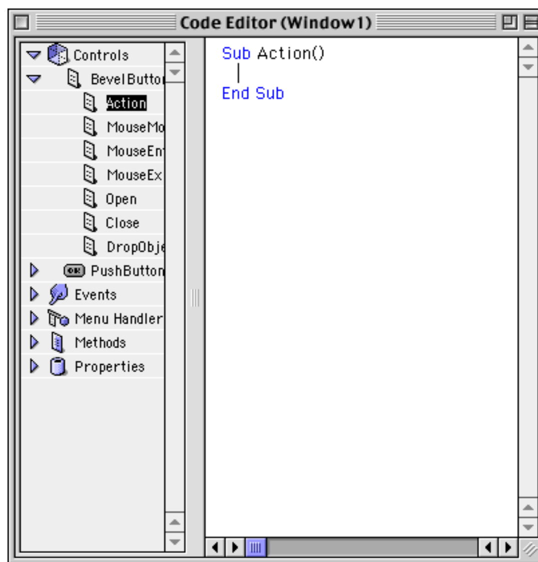
The Colors Window

Colors are actually stored as three numbers, each between 0 and 255. The Colors window makes it easy to keep track of colors you are using in your project by storing up to 16 colors. Clicking on a square in the Colors window presents the Macintosh Color Picker. After you choose a color and close the Color Picker dialog box, a small "swatch" of that color will be displayed in the square you clicked on in the Colors window. You can then change various color properties of controls by dragging a color swatch and dropping it on a color property in the Properties window.

The Code Editor Window

This window is used to edit the programming code you have added to objects in your project, such as buttons and windows. The Code Editor window has a browser that makes it easy to locate the object and view all of the events the object can receive. The Code Editor is shown in Figure 6.

FIGURE 6. **The Code Editor**

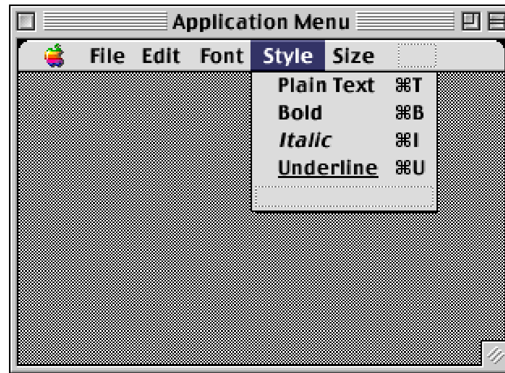


For a detailed description of the Code Editor, see the section "Using The Code Editor" on page 163.

The Menu Editor

This window is used to set up the menus and menu items that will be displayed when your application executes. The Menu Editor is shown in Figure 7 on page 31.

FIGURE 7. The Menu Editor



You can assign Macintosh keyboard shortcuts to menu items[1] and even create sub-menus (a menu item that is actually just another menu). REALbasic adds the Apple, File, and Edit menus for you by default.

Working with Projects

All of the windows, menus, pictures, sounds, QuickTime movies, plug-ins, and programming code that make up a single application are stored in a Project document. Projects simply give you a convenient way to organize the objects that make up your application.

Projects can contain any of the following items:

- Windows

1. You can add Windows keyboard shortcuts (“accelerators”) using modules. For more information, see “Using Constants to Add Windows Keyboard Shortcuts to Menus and Menu Items” on page 224.

- A Menu bar
- Classes
- Modules
- Pictures
- Sounds
- QuickTime™ movies
- Databases
- AppleEvent Templates
- PPC Shared Libraries
- XCMDs and XFCNs

If some of these items are not familiar to you, don't worry. You will learn more about them in later chapters.

Double-clicking on an item in the Project window will either display the item in its editor or a viewer for the item, if REALbasic has no editor for that type of item.

Creating A New Project

When you open REALbasic by double-clicking on the REALbasic application icon, a new project is created for you automatically. If you have a project open and wish to begin a new one, simply choose New from the File menu. If you have made modifications to your project, you will be given the opportunity to save the project before creating a new one.

Adding and Removing Items to Your Project

The method you use to add items to a project depends on the type of item you wish to add. For example, new windows are added by choosing New Window from the File menu. If you have a picture, sound, movie, or REAL database you wish to use in your project, you can add it by dragging the file from the desktop and dropping it into the Project window. You will learn in later chapters how to add each type of item that can appear in the Project window.

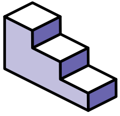
You can remove items from a Project by clicking once on the item in the Project window to select it, then pressing the Delete key, or by highlighting the item and choosing Edit ▶ Clear.

Saving Your Project

When you want to save the changes you have made to your project, choose Save from the File menu. If you are making lots of changes, save your project often just as you would if you were editing a document in a word processor. If you aren't sure whether you want to keep the changes you have made, you can choose not to save your project or choose Save As from the File menu and save the project under another name. This will keep your original project intact.

Creating Project Templates

If you have several items you commonly use in every project, you can save them in a project file and make the project file a stationery pad. When opened, a stationery pad creates a new, untitled document that is an exact copy of the stationery document. The stationery pad remains unchanged. This lets you create project templates without worrying about modifying the template itself.



To create a stationery pad, do this:

1. At the desktop, locate a project file you wish to change into a stationery document.
2. Click on the project file once to select it.
3. Choose File ▶ Get Info.
4. Place a checkmark in the stationery Pad checkbox.

When you open the stationery pad document, REALbasic creates a copy of it and names it Untitled so that you don't accidentally modify your template. If you want to modify the stationery pad itself, open the Get Info dialog box for that document and remove the checkmark in the stationery Pad checkbox.

CHAPTER 3

Building a User Interface

Your application's user interface is probably the most important part any application. The old saying "You don't get a second chance to make a first impression" couldn't be more true when it comes to your application's user interface. If the interface is unintuitive and sloppy, the user will react the same way they might react to someone who has poor communication skills and cares little for his appearance. Using your application will be frustrating at best and, at worst, the user will give up and look for another solution to his problem. This leaves you with whatever goals you had for your application unfulfilled.

Fortunately, REALbasic makes building your application's user interface so fast and easy that you can spend the time you need to get the interface just right. REALbasic's built-in Interface Assistant™ actually helps you build a proper, clean interface.

In this chapter you will learn just about everything you need to know about creating all of the elements that make up your application's user interface. You will learn some guidelines to

follow when creating your interface and how to build windows and menus.

Contents

- Working with Windows
- Interacting with the User Through Controls
- Adding Menus
- User Interface Guidelines

Working with Windows

Typically, most of an application's user interface will be in the application's windows. This, of course, is highly application-specific. Some applications have no windows at all, relying completely on menus to provide the user interface. REALbasic makes it easy to create new windows of just about any type. You create your user interface by creating its windows and adding interface controls such as PushButtons and CheckBoxes. You can also drag a picture directly into a window; the picture will be used as the "backdrop" picture for the window.

This section reviews the seven types of windows supported by REALbasic.

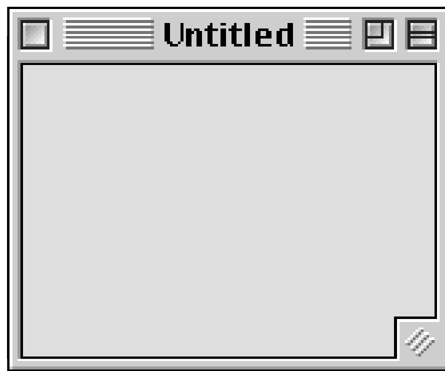
Window Types

REALbasic supports seven different types of windows. The type you choose for a particular window depends mostly on how the window will be used.

Document

The Document window is the most common type of window. They are most often used when the window should stay open until the user dismisses it by clicking its close box (if it has one) or clicking a button programmed to close the window. The user can click on other windows to bring them to the foreground, moving the document window behind the others. Figure 8 on page 37 shows an example of a small, blank document window.

FIGURE 8. A Document window



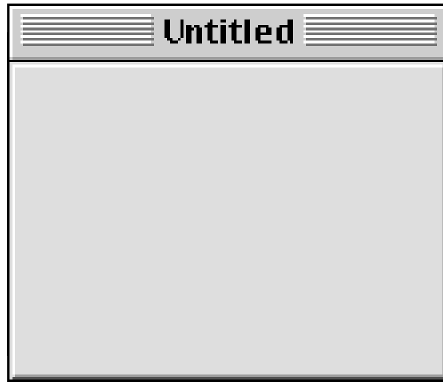
Document windows can have a close box, a zoom box, and a grow handle (making them user-resizable).

Movable Modal

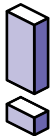
This type of window stays in front of the application's other open windows until it is closed. Use a Movable Modal window when you need to briefly communicate with the user without the user having access to the rest of the application. Because the window is movable, the user will be able to drag the window to another location in case they need to see information in other windows in order to finish what they are doing in the Movable Modal

window. Figure 9 on page 38 shows an example of a blank Movable Modal window.

FIGURE 9. **A Movable Modal window**



Movable Modal windows cannot have a close box, so you need to include a button that the user can click to dismiss the window unless the window will dismiss itself after the application finishes a particular task. Also, they are not resizable by the user and cannot have zoom box. This means you will have to consider the amount of available screen space the user will have in determining the size you will make a Movable Modal window.



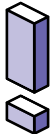
Note: There is one exception to the rule regarding Movable Modal windows being in front of all other windows. If a Movable Modal window or one of its controls executes code that opens a Floating window, the Floating window will be in front of the Movable Modal window. However, it is poor interface design for a Movable Modal window to open another window because Movable Modal windows are mostly used in situations where the interaction with the user will be brief.

Modal Dialog

These windows are very similar to Movable Modal windows. The only difference is that Modal Dialog windows have no titlebar, so

they cannot be moved. The Page Setup dialog box is an example of a Modal Dialog window.

FIGURE 10. **A Modal Dialog window**



Note: Because Modal Dialog windows and Movable Modal windows are both modal, the same exception applies regarding floating windows opening in front of Modal windows. See the note for Movable Modal windows on page 38.

Floating

Like Movable Modal and Modal Dialog windows, a Floating window (also known as a *Windoid*) stays in front of all other windows. The difference is that the user can still click on other windows to access them. If you have more than one Floating window open, clicking on another Floating window will bring that window to the front, but all open Floating windows will be in front of all non-floating windows. Because they are always in front of other types of windows, their size should be kept to a minimum or they will quickly get in the user's way. This type of window is most commonly used to provide tools the user will frequently access.

A Global Floating Window is a Floating window that can float in front of a particular application's window or all applications' windows.

FIGURE 11. A Floating window

Like Document windows, Floating windows can have a close box and can be user-resizable. However, they cannot have a zoom box.

Plain Box

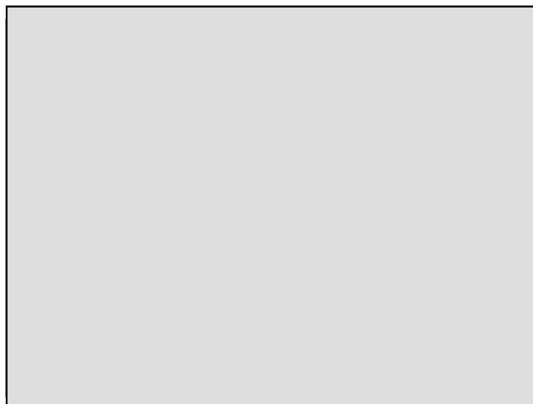
These windows function as Modal Dialog windows. The only real difference is their appearance, as you can see in Figure 12 on page 41. Plain Box windows are commonly used for About Box windows and for applications that need to hide the desktop.

FIGURE 12. **A Plain Box**

Shadowed Box

Like Plain Box windows, Shadowed Box windows function as Modal Dialog windows. The only difference is their appearance, as you can see in Figure 13 on page 41. Shadowed Box windows are commonly used for About Box windows.

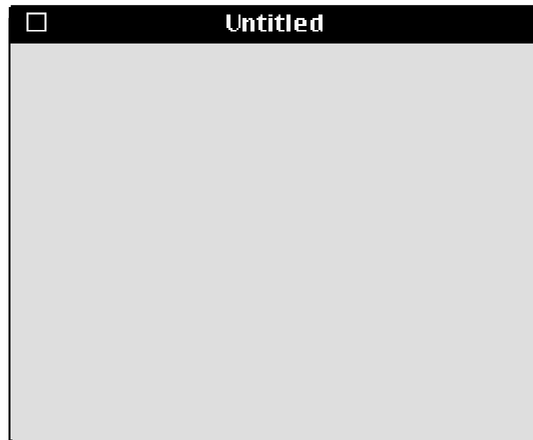
FIGURE 13. **A Shadowed Box**

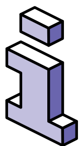


Rounded

Rounded windows act like Document windows. The only differences are appearance (as you can see in Figure 14 on page 42) and the fact that Rounded windows cannot have a zoom box or be resizable. They are not commonly used anymore and there is really no reason to use them instead of Document windows.

FIGURE 14. **A Rounded window**





Note: The Window class has a property, MacProcID, that allows you to create custom window types. This property give you more options than descried in this section. However, these custom types are supported only on Macintosh. All of the window types described in this section are fully cross-platform. For more information, see the discussion of the MacProcID property of the Window class in the *Language Reference*.

Creating Windows

When you create a new project, REALbasic adds a window named “Window1” to your project automatically. To add additional windows, choose File ▶ New Window. The windows you create act as templates. When your application opens one of

these windows, it's really opening a copy of the window. This means that your application can open several copies of the same window at the same time. It's important to understand this when creating your user interface because there is no need to go to the extra trouble of duplicating a window in the Design environment if your application needs to open two of them at the same time.

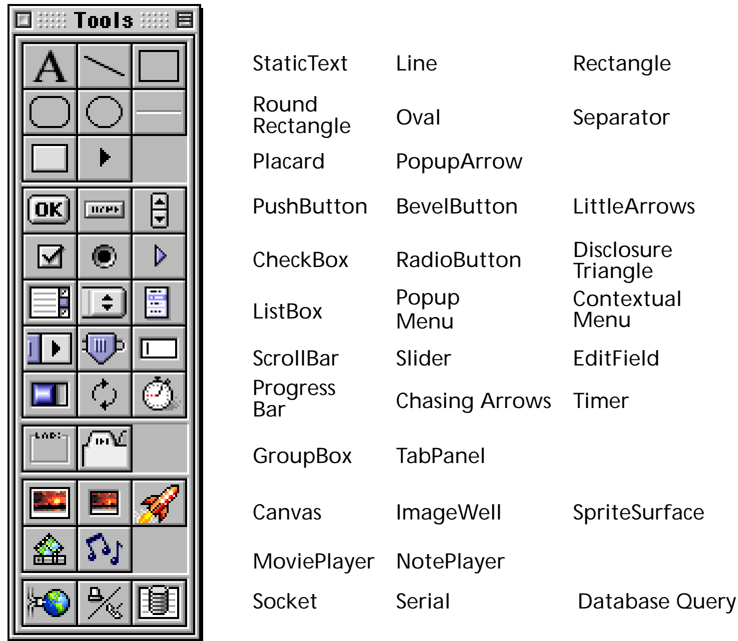
Removing Windows

To remove a window from your project, simply click on it once in the Project window to select it and press the Delete key or choose Edit ▶ Clear. You can undo many actions in REALbasic. For example, if you delete a window by mistake, choose Edit ▶ Undo (⌘-Z).

Interacting with the User Through Controls

Users provide information to your application through user interface controls. REALbasic provides a tremendous amount of flexibility in this area. Not only are there many built-in controls, but you can even create your own controls (you will learn more about this later). REALbasic's built-in controls are added to windows using the Tools Window, shown in Figure 15.

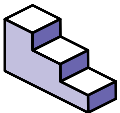
FIGURE 15. **The Tools Window.**



Adding, Changing, and Removing Controls

REALbasic makes adding, changing, and removing controls easy.

Adding Controls



To add a control to a window in your project, do this:

1. Bring the window to the front. If it's not open, double-click on it in the Project window to open it.
2. Drag the desired control from the Tools window and drop it on the window.

Selecting Controls

Controls can be selected in one of two ways: using the mouse button or the Tab key. If you click on a control, it will be selected. When a control is selected, REALbasic draws a border around the control using the highlight color selected in the Appearance Control panel on your Macintosh.

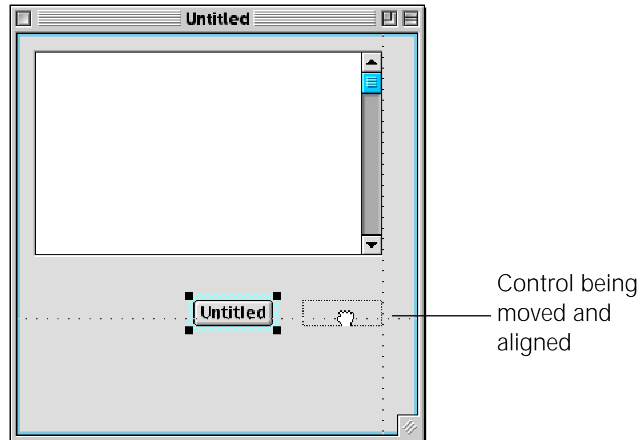
You can also move through the controls in a window by pressing the Tab key. Each time you press the Tab key, REALbasic will move from one control to another. This is also the order the user will move through the controls when using the Tab key. For more information, see "Changing The Tab (Control) Order" on page 115. Holding down the Shift key while pressing the Tab key selects controls in reverse Tab order. If only one control is selected, REALbasic draws resize squares at each corner of the control. You can select several controls by holding down the Shift key as you click on the controls.

Changing a Control's Position

A control's position can be changed by dragging the control using the mouse, by using the arrow keys (to move it one pixel at a time in the horizontal or vertical directions) and by changing the Position properties in the Properties window.

When you drag a control, you can align it with other objects in the window by taking advantage of built-in horizontal and vertical alignment guides. When the object you are dragging is near the horizontal and/or vertical side of another object, alignment guides temporarily appear, allowing you to position the object precisely. Figure 16 on page 46 illustrates the process of aligning a PushButton with the baseline of another PushButton and the right side of a ListBox control.

FIGURE 16. **Alignment guides help you position objects.**



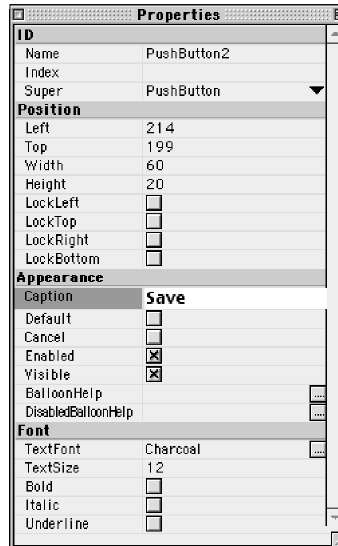
Changing a Control's Properties with the Properties window

Some changes to a control must be made with the Properties window. For example, controls can be rearranged by simply dragging them from one place to another inside the window. However, most of the changes you make to controls will be made using the Properties window.

The Properties window displays the properties of the currently selected control *that can be changed from the Design environment*. If more than one control is selected, the Properties window displays only those properties common to all of the selected controls.

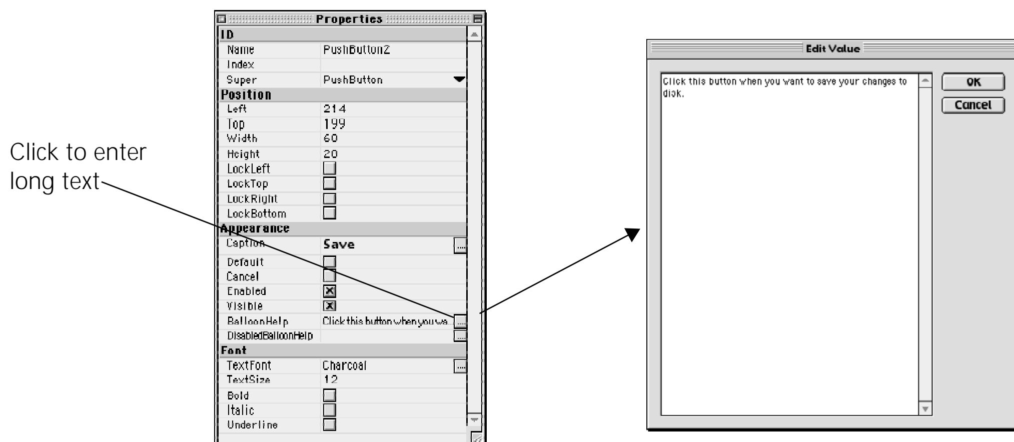
Some properties are entered by typing, while others with on/off-type values are represented by a checkbox. If the property is set by typing, you can use either the Enter or the Return key to commit the new value.

FIGURE 17. **Changing the Caption property of a PushButton.**



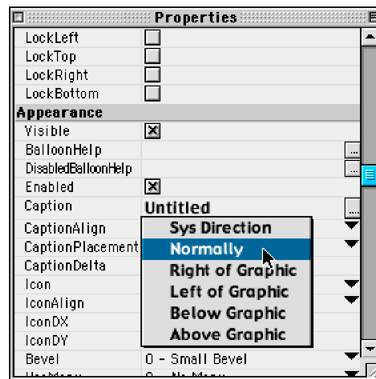
If the value you want to enter is long, you can click the text icon to bring up a modal window into which you can enter more lengthy text. A typical Properties window is shown in Figure 18.

FIGURE 18. **Entering text for Balloon help in a separate window.**



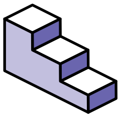
Some properties that require you to choose a value from a fixed list are displayed as pop-up menus. Such properties have a downward-pointing arrow to the right. Simply choose the desired value from the pop-up menu. In Figure 19, the "Right of Graphic" value is being assigned to the CaptionPlacement property of a BevelButton control.

FIGURE 19. **Choosing a value from a Property pop-up menu.**



Color properties display the selected color. These colors can be changed by clicking on the color and using the Color Picker to choose a color or by dragging a color from the Colors window and dropping on a color property.

Removing Controls



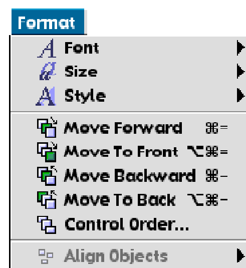
To remove a control from a window, do this:

1. Bring the window that contains the control to the front. If it's not open, double-click on it in the Project window to open it.
2. Click on the control to select it.
3. Choose Edit ▶ Cut (⌘-X), Edit ▶ Clear, or press the Delete key.

Understanding Control Layers

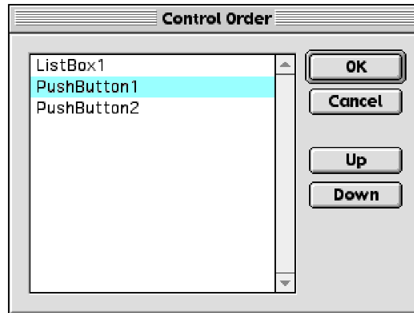
Each control in a window has its own layer. This layer is like a clear sheet of plastic and determines whether one control is in front of the other. The Format menu provides commands for moving a control forward one layer, to the front, backwards one layer, and to the very back of the layers.

FIGURE 20. **The Format Menu.**



These layers will usually only be important when controls overlap. For example, when you place controls on top of a GroupBox control or a TabPanel control, the GroupBox or TabPanel must be in back of the other controls. Otherwise, the GroupBox or TabPanel will be in front of one or more of the controls, obscuring them from view. Control layers also determine the order that your application selects the controls as the user presses the Tab key. However, you don't have to rearrange the layers of controls in order to determine their tab order. Instead, you can use the Control Order dialog box to determine the tab order.

FIGURE 21. **The Control Order dialog box.**

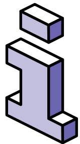


see "Changing The Tab (Control) Order" on page 115 for more details.

Understanding The Focus

The focus is a visual cue that tells the user which control receives keystrokes. Only EditFields and ListBoxes can receive the focus (PushButtons and Checkboxes can receive the focus on Windows). EditFields display the focus by showing a blinking cursor. When a ListBox has focus, REALbasic draws a border around the ListBox. If the user is running System 7, this border is a black rectangle. If the user is running MacOS 8 (or above), the border is drawn in the Accent Color chosen in the user's Appearance control panel. When a ListBox has the focus, it automatically responds to the arrow keys. It also receives any other keys the user types. This allows you to provide *type selection* functionality where typing automatically selects the item that matches the characters being typed. An example of type selection is provided with REALbasic.

FIGURE 22. A ListBox with the focus



Note: A ListBox will not receive the focus if it is the only item in the window that can receive the focus.

Duplicating Controls

You can duplicate the selected control or controls by choosing Edit ▶ Duplicate (⌘-D) or by holding down the Option key and dragging the selected control.

The Appearance of Controls

The part of the Mac OS that handles how menus, windows, and controls appear is called the Appearance Manager. If you are running Mac OS 8 (or greater), you have probably used the Appearance Control panel to select a highlight color and perhaps an accent color. MacOS 8.5 added new feature called “themes” to the Appearance Manager. Themes provide several “looks” that allow the user to subtly or radically change the appearance of menus, windows, and controls. This in no way changes the functionality of the interface. This is simply a way of allowing the user to customize their computing environment one step further.

FIGURE 23. **Selecting a theme from the Appearance Manager dialog box.**

REALbasic supports the Appearance Manager. This means that REALbasic itself will appear differently based on your Appearance Control Panel settings. It also means that if you or your users select a custom theme, REALbasic's interface will change based on the selected theme. The applications you create with REALbasic also support the Appearance Manager automatically.

There is also a shareware Control Panel that provides the most of the functionality of Apple's themes. It's called Kaleidoscope and it runs under System 7 and above. You can download it from www.download.com or www.kaleidoscope.net. A set of Kaleidoscope specifications is called a *scheme* instead of a *theme*. You choose a scheme via the Kaleidoscope Control panel rather than the Appearance Manager.

There are, at present, thousands of shareware and freeware Kaleidoscope schemes that range in design from simple to outlandish. With Kaleidoscope, you can preview alternative schemes and make any necessary changes to your interface if you wish;

however, you cannot anticipate the effects of each and every scheme. Figure 24 shows a standard pushbutton as it appears with different (relatively restrained) Kaleidoscope schemes.

FIGURE 24. **A standard PushButton displayed in three different Kaleidoscope schemes.**

Apple Computer updated the look for many interface elements in Mac OS8. Part of this change gives controls a more "3D" look. REALbasic, by default, draws controls with this 3D look regardless of whether the user is running System 7, System 7 with the Kaleidoscope extension, or MacOS8 or 9.

Button Controls for Performing Actions

There are four controls that are commonly used to perform actions when clicked: the CheckBox, the PushButton, the BevelButton, and the RadioButton.

PushButton

When clicked, a PushButton appears to depress giving the user feedback that they have clicked it. Pushbuttons are typically used to take an immediate and obvious action when pressed, like printing a report or closing a window.

FIGURE 25. A PushButton pressed and unpressed

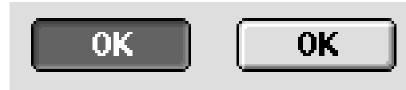


TABLE 1. PushButton properties

Name	Description
Super	The class of object the PushButton is based on.
Name	The internal name of the PushButton used to identify it in programming code.
Index	The PushButton's position in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the PushButton.
Top	The distance (in pixels) between the top edge of the window and the top edge of the PushButton.
Width	The width (in pixels) of the PushButton.
Height	The height (in pixels) of the PushButton.
LockLeft	Keeps the distance between the left side of the window and the left side of the PushButton from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the PushButton from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the PushButton from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the PushButton from changing when the window is resized.
Visible	The PushButton will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the PushButton.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.

TABLE 1. **PushButton properties (Continued)**

Name	Description
Caption	The text that appears on the PushButton.
Default	Adds the standard default ring to the PushButton and associates the Return and Enter keys with the it.
Cancel	Associates the Escape key and Command-Period key combination with the PushButton.
Enabled	The PushButton will be initially enabled.
TextFont	The font used to display the PushButton's caption.
TextSize	The font size used to display the PushButton's caption.
Bold	Adds the bold style to the PushButton's caption.
Italic	Adds the italic style to the PushButton's caption.
Underline	Adds the underline style to the PushButton's caption.

BevelButton

The BevelButton control provides very similar functionality of the PushButton and adds several additional powerful features. You can, for example:

- Add a PICT image to the control,
- Control the alignment of the button's text and/or the positioning of the text with respect to the graphic,
- Add a popup menu to the control,
- Control the feedback the user receives when the BevelButton is clicked.

The usage of a BevelButton control as a pop-up menu is described in the section "BevelButton" on page 80. Note that you can combine a PICT image with a pop-up menu.

Here are several examples of BevelButton options:

FIGURE 26. Icon, Text, and 'combo' BevelButtons

FIGURE 27. Bevel Sizes

TABLE 2. Bevelbutton Properties

Name	Description
Super	The class of object the BevelButton is based on.
Name	The internal name of the BevelButton used to identify it in programming code.
Index	The BevelButton's position in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the BevelButton.

TABLE 2. Bevelbutton Properties

Name	Description
Top	The distance (in pixels) between the top edge of the window and the top edge of the BevelButton.
Width	The width (in pixels) of the BevelButton.
Height	The height (in pixels) of the BevelButton.
LockBottom	Determines whether the bottom edge of the control should stay at a set distance from the bottom edge of the owning window.
LockLeft	Determines whether the left edge of the control should stay at a set distance from the left edge of the owning window. LockLeft has no effect unless LockRight is True.
LockRight	Determines whether the right edge of the control should stay at a set distance from the right edge of the owning window.
LockTop	Determines whether the top edge of the control should stay at a set distance from the top edge of the owning window. LockTop has no effect unless LockBottom is True.
Visible	The BevelButton will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the BevelButton.
DisabledBalloon-Help	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
Enabled	The BevelButton will be initially enabled.
Caption	The button's text.
CaptionAlign	0—Flush left 1—Flush right 2—Sys direction 3—Center

TABLE 2. **Bevelbutton Properties**

Name	Description
CaptionPlacement	0—Sys Direction 1—Normally 2—Right of graphic 3—Left of graphic 4—Below graphic 5—Above graphic
CaptionDelta	Distance in pixels of the caption from the left of the button.
Icon	Name of the PICT image to use as icon. Drag the image to the Project window or import it using File ▶ Import.
IconAlign	0—Sys Direction 1—Center 2—Left 3—Right 4—Top 5—Bottom 6—Top left 7—Bottom left 8—Top right 9—Bottom right
IconDx	Distance in pixels from ‘flush’ left or right, depending on alignment. If center is chosen, IconDx does nothing.
IconDy	Distance in pixels from ‘flush’ top or bottom, depending on alignment. If center is chosen, IconDy does nothing.
Bevel	0—Small bevel 1—Normal bevel 2—Large bevel
HasMenu	0—No menu 1—Normal menu 2—Menu on right
TextFont	Name of the font used to display the button caption.
TextSize	Size of the font used to display the button caption.
Bold	Applies the bold style to the button caption.

TABLE 2. **Bevelbutton Properties**

Name	Description
Italic	Applies the italic style to the button caption.
Underline	Applies the underline style to the button caption.
Value	If True, the button initially appears as if it is pressed.
MenuValue	The number of the menu item the user selects. A separator cannot be selected, but "counts" as a menu value.
ButtonType	0—Button. Remains in 'down' position until mouse is released. 1—Toggles. Remains in 'down' position until clicked again. 2—Sticky. Remains in 'down' position when clicked.

CheckBox

Checkboxes are used to let the user state a preference that has only two possible choices, where one of the choices can be selected by default. Checkboxes should not cause an immediate and obvious action to occur except perhaps to enable or disable other controls.

FIGURE 28. **A CheckBox checked and unchecked**

If space permits, consider using two RadioButton controls instead of a single CheckBox control as it will make the user's choice more obvious especially to the new computer user.

TABLE 3. **CheckBox properties**

Name	Description
Super	The class of object the CheckBox is based on.
Name	The internal name of the CheckBox used to identify it in programming code.
Index	The position of the CheckBox in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the CheckBox.
Top	The distance (in pixels) between the top edge of the window and the top edge of the CheckBox.
Width	The width (in pixels) of the CheckBox.
Height	The height (in pixels) of the CheckBox.
LockLeft	Keeps the distance between the left side of the window and the left side of the CheckBox from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the CheckBox from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the CheckBox from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the CheckBox from changing when the window is resized.
Caption	The text that appears on the CheckBox.
Enabled	The CheckBox will be initially enabled.
Visible	The CheckBox will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the CheckBox.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.

TABLE 3. CheckBox properties (Continued)

Name	Description
TextFont	The font used to display the CheckBox caption.
TextSize	The font size used to display the CheckBox caption.
Bold	Adds the bold style to the CheckBox caption.
Italic	Adds the italic style to the CheckBox caption.
Underline	Adds the underline style to the CheckBox caption.
Value	The default value of the CheckBox.

RadioButton

RadioButtons are used to present the user with two or more choices, where one of the choices can be selected by default. Selecting one RadioButton causes the RadioButton that is currently selected to become unselected. They are called RadioButtons because they act just like the row of buttons for changing radio stations on car radios. Pushing one button deselects the current radio station and selects another station. RadioButtons should always be displayed in groups of at least two.

FIGURE 29. A group of RadioButtons with one selected

If you are creating a window that will have two or more independent sets of RadioButtons, you will need to use a

GroupBox control to make your RadioButton groups respond independently. see “GroupBox” on page 82.

TABLE 4. **RadioButton properties**

Name	Description
Super	The class of object the RadioButton is based on.
Name	The internal name of the RadioButton used to identify it in programming code.
Index	The position of the RadioButton in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the RadioButton.
Top	The distance (in pixels) between the top edge of the window and the top edge of the RadioButton.
Width	The width (in pixels) of the RadioButton.
Height	The height (in pixels) of the RadioButton.
LockLeft	Keeps the distance between the left side of the window and the left side of the RadioButton from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the RadioButton from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the RadioButton from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the RadioButton from changing when the window is resized.
Caption	The text that appears on the RadioButton.
Enabled	The RadioButton will be initially enabled.
Visible	The RadioButton will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the RadioButton.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
TextFont	The font used to display the RadioButton caption.

TABLE 4. **RadioButton properties (Continued)**

Name	Description
TextSize	The font size used to display the RadioButton caption.
Bold	Adds the bold style to the RadioButton caption.
Italic	Adds the italic style to the RadioButton caption.
Underline	Adds the underline style to the RadioButton caption.
Value	The default value of the RadioButton.

Controls for Displaying and Entering Text

REALbasic provides controls that let you display text the user can't select, display text the user can select but not edit, and display text the user can both select and edit.

StaticText

Used to display text that the user cannot select. StaticText controls are most commonly used to label other controls (like PopupMenus) or provide titles for groups of controls.

FIGURE 30. **A StaticText control used to label a PopupMenu control**

TABLE 5. **StaticText properties**

Name	Description
Super	The class of object the StaticText is based on.
Name	The internal name of the StaticText used to identify it in programming code.
Index	The position of the StaticText in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the StaticText.

TABLE 5. **StaticText properties (Continued)**

Name	Description
Top	The distance (in pixels) between the top edge of the window and the top edge of the StaticText.
Width	The width (in pixels) of the StaticText.
Height	The height (in pixels) of the StaticText.
LockLeft	Keeps the distance between the left side of the window and the left side of the StaticText from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the StaticText from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the StaticText from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the StaticText from changing when the window is resized.
Text	The text that appears in the window.
TextAlign	The alignment of the text within its area (left, middle, right).
TextColor	The color of the text.
MultiLine	Causes the text to start at the top of its area rather than being centered vertically within it.
Visible	The StaticText will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the StaticText.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
TextFont	The font used to display the StaticText caption.
TextSize	The font size used to display the StaticText caption.
Bold	Adds the bold style to the StaticText caption.
Italic	Adds the italic style to the StaticText caption.
Underline	Adds the underline style to the StaticText caption.

EditField

EditFields can be used to allow the user to enter text or to display text that can copied to the Clipboard but not changed in the EditField. They can be configured to allow multiple lines of text, display a scrollbar if necessary, and display text in multiple fonts, styles, and sizes.

FIGURE 31. A Empty EditField

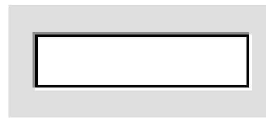


FIGURE 32. An EditField configured for multiple lines of text

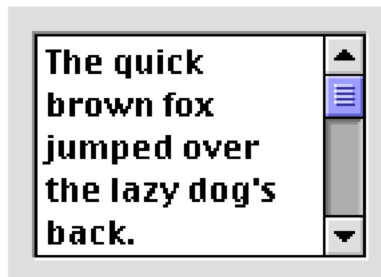


FIGURE 33. An Editfield with multiple fonts, styles and sizes

TABLE 6. EditField properties

Name	Description
Super	The class of object the EditField is based on.
Name	The internal name of the EditField used to identify it in programming code.

TABLE 6. **EditField properties (Continued)**

Name	Description
Index	The position of the EditField in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the EditField.
Top	The distance (in pixels) between the top edge of the window and the top edge of the EditField.
Width	The width (in pixels) of the EditField.
Height	The height (in pixels) of the EditField.
LockLeft	Keeps the distance between the left side of the window and the left side of the EditField from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the EditField from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the EditField from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the EditField from changing when the window is resized.
Border	Draws a border around the EditField.
MultiLine	Causes the text to start at the top of its area rather than being centered vertically within it.
ScrollBar	Displays a scrollbar if MultiLine property is checked.
Styled	Allows EditField to contain styled (multiple fonts, styles and sizes) text.
Password	Every character entered is replaced with a bullet character. The actual characters typed are stored in the Text property.
UseFocusRing	If True, the object indicates that it has the focus with a ring around its border; if False, the appearance of the object does not change when it has the focus.
TextColor	The color of the text entered into the EditField
BackColor	The color of the background in the Editfield
Enabled	The EditField will be enabled when the window opens.

TABLE 6. **EditField properties (Continued)**

Name	Description
Visible	The EditField will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the EditField.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
TextFont	The font used to display the EditField caption.
TextSize	The font size used to display the EditField caption.
Bold	Adds the bold style to the EditField caption.
Italic	Adds the italic style to the EditField caption.
Underline	Adds the underline style to the EditField caption.
Text	The default value of the EditField.
ReadOnly	Allows copying of text to the clipboard but no editing.
LimitText	The maximum number of characters allowed (0=no limit).
AcceptTabs	If True, pressing the Tab key will enter a tab into the EditField instead of moving the focus to the next item in the tab order.

Controls for Displaying and Entering Numeric Values

REALbasic provides controls that can be used to let the user choose a numeric value from a range or to display a numeric value from a range. In some cases, these controls can also be used to control the display of another control. For example, a ScrollBar control might be used to determine which portion of a picture in a Canvas control is displayed (in other words, act as the Canvas control's scrollbar).

ScrollBar

ScrollBars can be presented vertically or horizontally. By default, they are horizontal. To make a vertical ScrollBar, simply resize the Scrollbar object so that the height is greater than the width. Although you can resize a ScrollBar in the direction that the thumb travels, ScrollBars should always be 16 pixels thick.

FIGURE 34. **Horizontal and vertical ScrollBars**

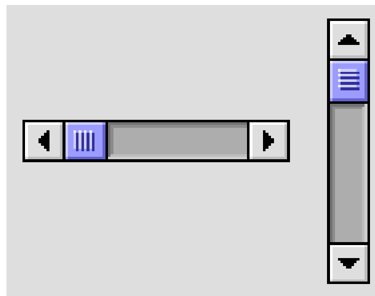


TABLE 7. **ScrollBar Properties**

Name	Description
Super	The class of object the ScrollBar is based on.
Name	The internal name of the ScrollBar used to identify it in programming code.
Index	The position of the ScrollBar in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the ScrollBar.
Top	The distance (in pixels) between the top edge of the window and the top edge of the ScrollBar.
Width	The width (in pixels) of the ScrollBar.
Height	The height (in pixels) of the ScrollBar.
LockLeft	Keeps the distance between the left side of the window and the left side of the ScrollBar from changing when the window is resized.

TABLE 7. ScrollBar Properties (Continued)

Name	Description
LockTop	Keeps the distance between the top of the window and the top of the ScrollBar from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the ScrollBar from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the ScrollBar from changing when the window is resized.
Enabled	The ScrollBar will be initially enabled.
Visible	The ScrollBar will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the ScrollBar.
DisabledBal-loonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
Minimum	The value of the Value property when the scroll indicator is all the way left (for horizontal scrollbars) or at the very top (for vertical scrollbars).
Value	The current position of the scroll indicator.
Maximum	The value the Value Property will be set to when the scroll indicator is all the way to the right (for horizontal scrollbars) or at the bottom (for vertical scrollbars).
LineStep	The amount by which the Value property will change when the user clicks on one of the ScrollBar's arrows.
PageStep	The amount by which the Value property will change when the user clicks inside the ScrollBar on either side of the scroll indicator.
LiveScroll	If true, a ValueChanged event occurs as the user drags the thumbnail in the scrollbar. Otherwise, a single Val-ueChanged event occurs when the user stops dragging the thumbnail.

Slider

This control was added to the Macintosh user interface in MacOS 8. It has the same functionality as a ScrollBar control. However, ScrollBar controls have come to be associated with scrolling text or a picture and less with assigning numeric values. The Slider control provides an interface that is clearly for increasing or decreasing a numeric value. Like the ScrollBar, the Slider control can appear horizontally (which is the default) or vertically. You can create a vertical Slider by changing its height so that it's greater than its width. Unlike the ScrollBar control, the Slider control automatically maintains the correct proportions regardless of the dimensions you give it. Because the Slider was added in MacOS 8, it appears as a ScrollBar for System 7 users.

FIGURE 35. **Horizontal and vertical Slider controls**

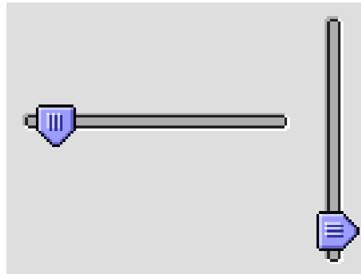


TABLE 8. **Slider properties**

Name	Description
Super	The class of object the Slider is based on.
Name	The internal name of the Slider used to identify it in programming code.
Index	The position of the Slider in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the Slider.
Top	The distance (in pixels) between the top edge of the window and the top edge of the Slider.

TABLE 8. **Slider properties (Continued)**

Name	Description
Width	The width (in pixels) of the Slider.
Height	The height (in pixels) of the Slider.
LockLeft	Keeps the distance between the left side of the window and the left side of the Slider from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the Slider from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the Slider from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the Slider from changing when the window is resized.
Enabled	The Slider will be initially enabled.
Visible	The Slider will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the Slider.
DisabledBal-loonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
Minimum	The value of the Value property when the indicator is all the way left (for horizontal Sliders) or at the very top (for vertical Sliders).
Value	The current position of the indicator.
Maximum	The value the Value Property will be set to when the indicator is all the way to the right (for horizontal Slid-ers) or at the bottom (for vertical Sliders).
LineStep	This property is only used when the user is running System 7 as the Slider appears as a Scrollbar. The amount by which the Value property will change when the user clicks on one of the ScrollBar's arrows.

TABLE 8. **Slider properties (Continued)**

Name	Description
PageStep	This property is only used when the user is running System 7 as the Slider appears as a Scrollbar. The amount by which the Value property will change when the user clicks inside the ScrollBar on either side of the scroll indicator.
LiveScroll	If True, a ValueChanged event occurs as the user drags the thumbnail in the scrollbar. Otherwise, a single ValueChanged event occurs when the user stops dragging the thumbnail.

ProgressBar

ProgressBars are designed to indicate that some function of your application is progressing (hence the name) towards its goal or to show capacity. Unlike ScrollBars and Sliders, ProgressBars are designed to display a value. They cannot be used for data entry. Also, they appear only in a horizontal orientation. When using a ProgressBar to show duration, the ProgressBar can be configured to show progress where the length is determinate or indeterminate. Indeterminate ProgressBars are sometimes referred to as " Barber Poles."

FIGURE 36. **Determinate and indeterminate ProgressBars**

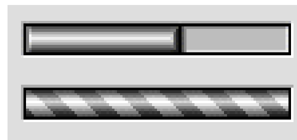


TABLE 9. **ProgressBar properties**

Name	Description
Super	The class of object the ProgressBar is based on.
Name	The internal name of the ProgressBar used to identify it in programming code.

TABLE 9. ProgressBar properties (Continued)

Name	Description
Index	The position of the ProgressBar in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the ProgressBar.
Top	The distance (in pixels) between the top edge of the window and the top edge of the ProgressBar.
Width	The width (in pixels) of the ProgressBar.
Height	The height (in pixels) of the ProgressBar.
LockLeft	Keeps the distance between the left side of the window and the left side of the ProgressBar from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the ProgressBar from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the ProgressBar from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the ProgressBar from changing when the window is resized.
Visible	The ProgressBar will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the ProgressBar.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
Value	The current position of the indicator.
Maximum	The value the Value Property will be set to when the indicator is all the way to the right.

Controls for Presenting a List of Choices

RadioButton and CheckBox controls can, of course, be used to provide the user with a limited list of choices. There are situations,

however, when using these controls is either an inefficient use of space or impossible. Some of these situations are:

- When the number of choice items is quite long, making it difficult or impossible to use RadioButton or CheckBox controls
- When the choices change dynamically based on the application's logic
- When the choice items need to display more than one column of information

If your situation doesn't match one of these cases, consider using RadioButton or CheckBox controls. They are easier for a new computer user to use because all of their choices will be right in front of them.

ContextualMenu

ContextualMenu controls display a list of choices in a menu when the user holds down the Control key and clicks on any control or window that receives a MouseDown event. One ContextualMenu control can actually display contextual menus for any number of other controls.

FIGURE 37. An example of a contextual menu

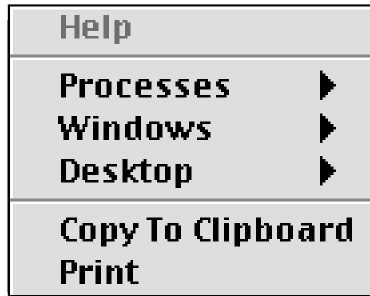


TABLE 10. Contextual menu properties

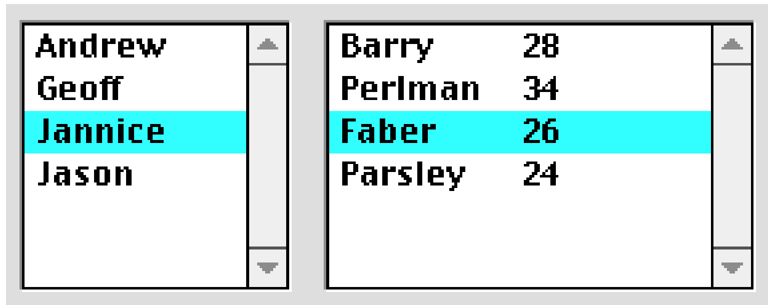
Name	Description
Super	The class of object the ContextualMenu is based on.
Name	The internal name of the ContextualMenu used to identify it in programming code.
Index	The position of the ContextualMenu in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the ContextualMenu.
Top	The distance (in pixels) between the top edge of the window and the top edge of the ContextualMenu.
UseCCM	If True, the Help item is displayed. If False, the Help item is omitted.

ListBox

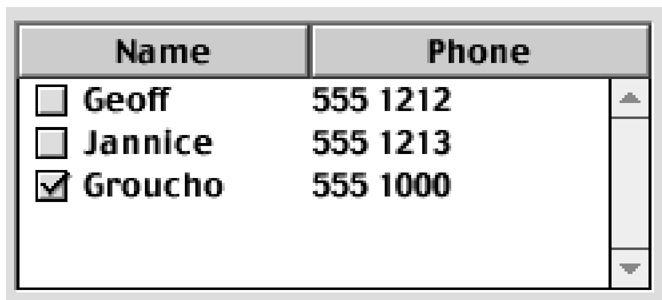
ListBox controls display a scrolling list of values. The user can use the mouse or the arrow keys to choose an item. ListBox controls can contain one or more columns of data, can be hierarchical, and can allow one row selection or multiple row selection. You can add a header with column labels to a ListBox or add a check box to each row in the ListBox; the user can sort the data in the ListBox by clicking on a column header.

FIGURE 38. Examples of ListBoxes

Single column and multiple column



Multiple column with headers and checkboxes



Two column hierarchical

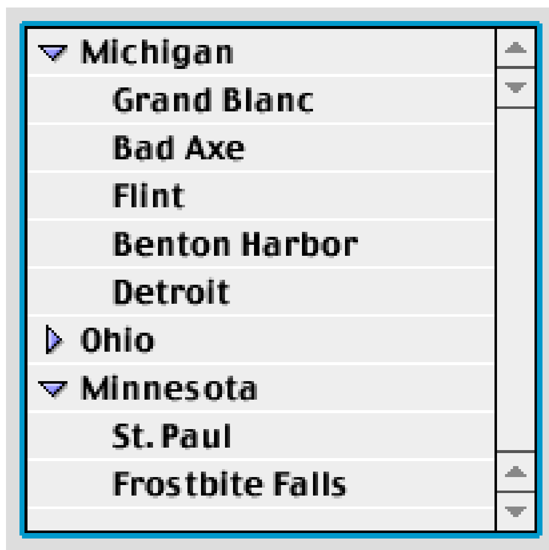


TABLE 11. **ListBox properties**

Name	Description
Super	The class of object the ListBox is based on.
Name	The internal name of the ListBox used to identify it in programming code.
Index	The position of the ListBox in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the ListBox.
Top	The distance (in pixels) between the top edge of the window and the top edge of the ListBox.
Width	The width (in pixels) of the ListBox.
Height	The height (in pixels) of the ListBox.
LockLeft	Keeps the distance between the left side of the window and the left side of the ListBox from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the ListBox from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the ListBox from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the ListBox from changing when the window is resized.
Enabled	The ListBox will be initially enabled.
ColumnCount	The number of columns the ListBox can display.
ColumnWidths	A list of comma-separated values, with each value controlling the width of the associated column. Each value can be express in pixels or as a percentage.
HasHeading	If True, a row of column headers is added to the ListBox. The user can sort the column by clicking the heading.
UseFocusRing	If True, the object indicates that it has the focus with a ring around its border; if False, the appearance of the object does not change when it has the focus.

TABLE 11. **ListBox properties (Continued)**

Name	Description
InitialValue	A list of the default items separated by carriage returns. This property is unreadable at runtime because the list is removed from memory once the control is created.
Visible	The ListBox will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the ListBox.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
TextFont	The font used to display the ListBox caption.
TextSize	The font size used to display the ListBox caption.
Bold	Adds the bold style to the ListBox caption.
Italic	Adds the italic style to the ListBox caption.
Underline	Adds the underline style to the ListBox caption.
Hierarchical	Allows rows to be added with disclosure triangles (using the AddFolder method) and draws ListBox with a grey background.
EnableDrag	Allows rows to be dragged from the listbox.
SelectionType	Determines whether the user can select (highlight) a single row or multiple rows. 0—Single 1—Multiple

PopupMenu

PopupMenu controls are useful when you have a single column of data to present in a limited amount of space.

FIGURE 39. A PopupMenu control

TABLE 12. PopupMenu properties

Name	Description
Super	The class of object the PopupMenu is based on.
Name	The internal name of the PopupMenu used to identify it in programming code.
Index	The position of the PopupMenu in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the PopupMenu.
Top	The distance (in pixels) between the top edge of the window and the top edge of the PopupMenu.
Width	The width (in pixels) of the PopupMenu.
Height	The height (in pixels) of the PopupMenu.
LockLeft	Keeps the distance between the left side of the window and the left side of the PopupMenu from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the PopupMenu from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the PopupMenu from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the PopupMenu from changing when the window is resized.
Enabled	The PopupMenu will be initially enabled.
InitialValue	A list of the default items separated by carriage returns. This property is unreadable at runtime because the list is removed from memory once the control is created.
Visible	The PopupMenu will initially be visible when the window opens.

TABLE 12. **PopupMenu properties (Continued)**

Name	Description
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the PopupMenu.
DisabledBal-loonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
TextFont	The font used to display the PopupMenu caption.
TextSize	The font size used to display the PopupMenu caption.
Bold	Adds the bold style to the PopupMenu caption.
Italic	Adds the italic style to the PopupMenu caption.
Underline	Adds the underline style to the PopupMenu caption.

BevelButton

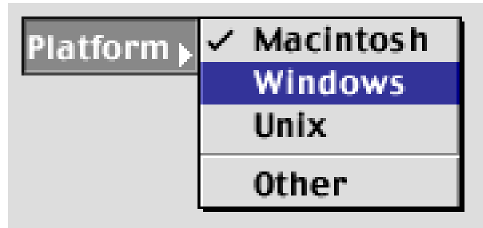
A BevelButton control can be configured to operate as a pop-up menu. Simply set the HasMenu property to 1 or 2 (Normal menu or Menu on Right). See the section "BevelButton" on page 55 for the list of a BevelButton's properties.

The BevelButton menu shown in Figure 40 was created with this code in the Open event of the Bevelbutton:

```
me.captionalign=0 //caption aligned flush left
me.hasMenu=2      //menu on right
me.caption="Platform"
me.addRow("Macintosh")
me.addRow("Windows")
me.addRow("Unix")
me.addseparator
me.addRow("Other")
```

You would use the MenuValue property to determine which menu item the user has selected.

FIGURE 40. **A BevelButton popup menu**



Controls for Visually Grouping Other Controls

If a window contains groups of controls in which each group of controls serves a different purpose, it can be confusing to the user to see all of these groups lumped together in a window. It often makes sense (and is sometimes necessary) to group related controls. Fortunately, REALbasic provides several built-in controls to make grouping controls simple.

Separator

The Separator control simply places a vertical or horizontal line in the window that you can use to help organize other objects.

FIGURE 41. **A Separator control**

TABLE 13. Separator Properties

Name	Description
Super	The class of object the Separator is based on.
Name	The internal name of the Separator used to identify it in programming code.
Index	The position of the Separator in a control array.
Left	The distance in pixels from the left of the window to the left of the Separator.
Top	The distance in pixels from the top of the window to the top of the Separator.
Width	The width of the Separator.
Height	The height of the Separator.
LockLeft	Determines whether the left edge of the control should stay at a set distance from the left edge of the owning window. LockLeft has no effect unless LockRight is True.
LockTop	Determines whether the top edge of the control should stay at a set distance from the top edge of the owning window. LockTop has no effect unless LockBottom is True.
LockRight	Determines whether the right edge of the control should stay at a set distance from the right edge of the owning window.
LockBottom	Determines whether the bottom edge of the control should stay at a set distance from the bottom edge of the owning window.

GroupBox

A GroupBox can be displayed with or without a caption. If a window has more than one group of RadioButton controls, one of the groups must be contained within a GroupBox control in order for the RadioButton groups to function independently.

FIGURE 42. A GroupBox control with and without a caption

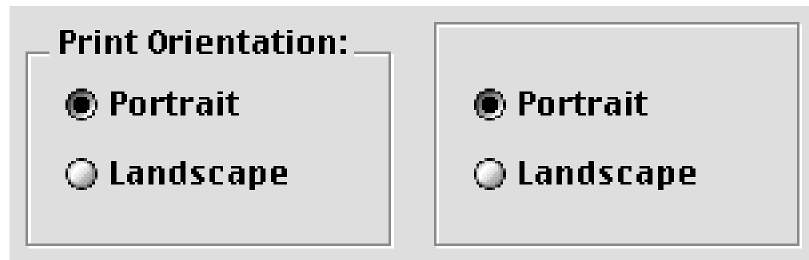


TABLE 14. GroupBox properties

Name	Description
Super	The class of object the GroupBox is based on.
Name	The internal name of the GroupBox used to identify it in programming code.
Index	The position of the GroupBox in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the GroupBox.
Top	The distance (in pixels) between the top edge of the window and the top edge of the GroupBox.
Width	The width (in pixels) of the GroupBox.
Height	The height (in pixels) of the GroupBox.
LockLeft	Keeps the distance between the left side of the window and the left side of the GroupBox from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the GroupBox from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the GroupBox from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the GroupBox from changing when the window is resized.
Caption	The text that appears on the GroupBox.
Enabled	The GroupBox will be initially enabled.

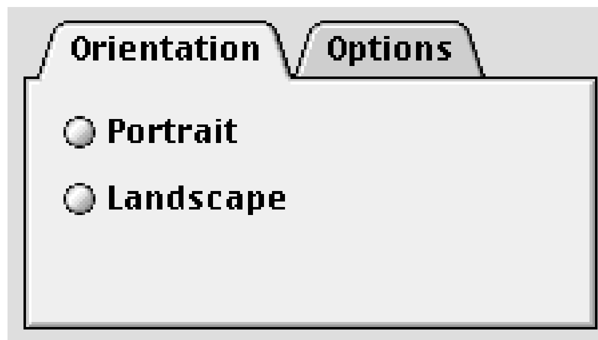
TABLE 14. **GroupBox properties (Continued)**

Name	Description
Visible	The GroupBox will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the GroupBox.
DisabledBal-loonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
TextFont	The font used to display the GroupBox caption.
TextSize	The font size used to display the GroupBox caption.
Bold	Adds the bold style to the GroupBox caption.
Italic	Adds the italic style to the GroupBox caption.
Underline	Adds the underline style to the GroupBox caption.

TabPanel

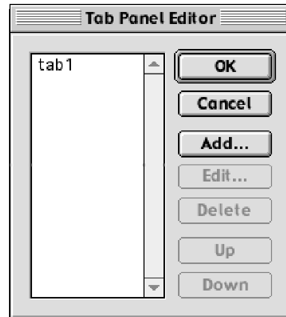
When you have several groups of controls and space is very limited, TabPanels are most appropriate. TabPanels presents each group of controls in a separate panel. When the user clicks on a tab in the TabPanel, REALbasic automatically hides the controls on the current panel and displays the controls on the panel the user selected.

FIGURE 43. **A two-panel TabPanel control**



You add, modify, rearrange, or delete tabs and tab labels using the Tab Panel Editor. Click the last tab, which is always labelled " ..." , to display the editor.

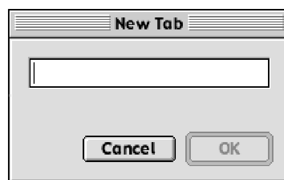
FIGURE 44. **The Tab Panel Editor.**



With the Tab Panel Editor, you can:

- Rename the first tab by highlighting it and clicking the Edit button.
- Add a new tab and its label by clicking the Add button. The following dialog box appears in front of the Tab Panel Editor:

FIGURE 45. **The New Tab Dialog Box**



- Delete a tab by highlighting it and clicking the Delete button.
- Rearrange the tabs by highlighting a tab and clicking the Up or Down buttons.

After you have added the desired tabs to the TabPanel, you can add other controls to each page. Select a tab and then drag the necessary controls to that page. Repeat the process for each page.

With the Facings property, you can place the tabs along any edge of the control. This property works only if the user is running MacOS 8.5 (or above).Figure 46 shows the use of the " South" setting of the Facings property:

FIGURE 46. **A Tab Panel that uses the "South" Facing.**

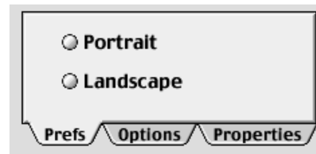


TABLE 15. **TabPanel properties**

Name	Description
Super	The class of object the TabPanel is based on.
Name	The internal name of the TabPanel used to identify it in programming code.
Index	The position of the TabPanel in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the TabPanel.
Top	The distance (in pixels) between the top edge of the window and the top edge of the TabPanel.
Width	The width (in pixels) of the TabPanel.
Height	The height (in pixels) of the TabPanel.
LockLeft	Keeps the distance between the left side of the window and the left side of the TabPanel from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the TabPanel from changing when the window is resized.

TABLE 15. **TabPanel properties (Continued)**

Name	Description
LockRight	Keeps the distance between the right side of the window and the right side of the TabPanel from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the TabPanel from changing when the window is resized.
Facing	Tabs can face North, South, East, or West (available on MacOS 8.5 or above only)
SmallTabs	If True, the tabs' height (North/South) or width (East/West) is reduced.
Visible	The TabPanel will initially be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the TabPanel.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
TextFont	The font used to display the TabPanel text.
TextSize	The font size used to display the TabPanel text.
Bold	Adds the bold style to the TabPanel text.
Italic	Adds the italic style to the TabPanel text.
Underline	Adds the underline style to the TabPanel text.
Enabled	The TabPanel will be initially enabled.

Controls for Displaying Graphics and Pictures

REALbasic is very flexible when it comes to displaying graphics and pictures. You can use the built-in graphic controls, display pictures from documents, or draw the graphics using REALbasic's programming language.

Line

Draws a line that can be of any length, width, color, and direction. By default, lines are 100 pixels in length, 1 pixel in width, black, and horizontal.

TABLE 16. Line properties

Name	Description
Super	The class of object the Line is based on.
Name	The internal name of the Line used to identify it in programming code.
Index	The position of the Line in a control array.
X1	The distance (on the horizontal axis) from the left side of the window to the end of the Line that is leftmost by default.
X2	The distance (on the horizontal axis) from the left side of the window to the end of the Line that is right most by default.
Y1	The distance (on the vertical axis) from the top of the window to the end of the Line that is leftmost by default.
Y2	The distance (on the vertical axis) from the top of the window to the end of the Line that is right most by default.
BorderWidth	The width (in pixels) of the Line.
LineColor	The color of the Line.
Visible	The Line will be visible when the window opens.

Rectangle

Draws a rectangle that can be of any length, width, border color, and fill color. By default, rectangles are 100 pixels in length and width, 1 pixel in width, have black borders and a white center. Because you can control the color of the left and top borders independently from the right and bottom borders, you can easily create rectangles that appear to be sunken or raised.

FIGURE 47. **A Rectangle with default, sunken and raised appearances**

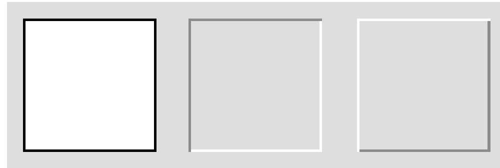


TABLE 17. **Rectangle properties**

Name	Description
Super	The class of object the Rectangle is based on.
Name	The internal name of the Rectangle used to identify it in programming code.
Index	The position of the Rectangle in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the Rectangle.
Top	The distance (in pixels) between the top edge of the window and the top edge of the Rectangle.
Width	The width (in pixels) of the Rectangle.
Height	The height (in pixels) of the Rectangle.
LockLeft	Keeps the distance between the left side of the window and the left side of the Rectangle from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the Rectangle from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the Rectangle from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the Rectangle from changing when the window is resized.
FillColor	The color that will fill the interior of the Rectangle.
BorderWidth	The width (in pixels) of the sides of the Rectangle.
TopLeftColor	The color of the lines that make up the top and left sides of the Rectangle.

TABLE 17. **Rectangle properties (Continued)**

Name	Description
BottomRightColor	The color of the lines that make up the right and bottom sides of the Rectangle.
Visible	The Rectangle will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the Rectangle.
DisabledBalloon-Help	The text that should appear when the user moves the mouse over the control while the control is dis-abled and BalloonHelp is on.

RoundRectangle

RoundRectangles are similar to regular Rectangle controls. The differences are that you don't have the independent color control for the border (because it is one continuous line) but you can control the width and height of the arcs that make up the round corners.

FIGURE 48. **A RoundRectangle control**

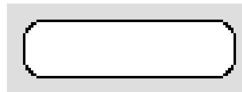


TABLE 18. **RoundRectangle Properties**

Name	Property
Super	The class of object the RoundRectangle is based on.
Name	The internal name of the RoundRectangle used to iden-tify it in programming code.
Index	The position of the RoundRectangle in a control array.
Left	The distance (in pixels) between the left edge of the win-dow and the left edge of the RoundRectangle.
Top	The distance (in pixels) between the top edge of the win-dow and the top edge of the RoundRectangle.

TABLE 18. **RoundRectangle Properties**

Name	Property
Width	The width (in pixels) of the RoundRectangle.
Height	The height (in pixels) of the RoundRectangle.
LockLeft	Keeps the distance between the left side of the window and the left side of the RoundRectangle from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the RoundRectangle from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the RoundRectangle from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the RoundRectangle from changing when the window is resized.
FillColor	The color that will fill the interior of the RoundRectangle.
BorderWidth	The width (in pixels) of the sides of the RoundRectangle.
OvalWidth	The width of the arcs that make up the corners.
OvalHeight	The height of the arcs that make up the corners
Visible	The RoundRectangle will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the RoundRectangle.
DisabledBal-loonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.

Oval

Draws an oval with a single pixel, black border, and filled with white. All of these properties can be modified. The “ovalness” of the Oval is controlled by its height and width. For example, an Oval with the same width and height is a perfect circle.

FIGURE 49. **An Oval control**

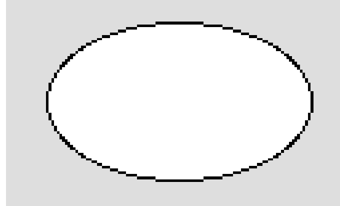


TABLE 19. **Oval properties**

Name	Description
Super	The class of object the Oval is based on.
Name	The internal name of the Oval used to identify it in programming code.
Index	The position of the Oval in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the Oval.
Top	The distance (in pixels) between the top edge of the window and the top edge of the Oval.
Width	The width (in pixels) of the Oval.
Height	The height (in pixels) of the Oval.
LockLeft	Keeps the distance between the left side of the window and the left side of the Oval from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the Oval from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the Oval from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the Oval from changing when the window is resized.
Visible	The Oval will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the Oval.

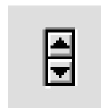
TABLE 19. **Oval properties (Continued)**

Name	Description
DisabledBal-loonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
FillColor	The color that will fill the interior of the Oval.
BorderWidth	The width (in pixels) of the sides of the Oval.
BorderColor	The color of the Oval's border.

Canvas

A Canvas control can be used to display a picture from a file or a picture drawn using REALbasic's programming language. If your application requires a type of control that is not built-in, you can use a Canvas control and REALbasic drawing commands to create the controls you need.

FIGURE 50. **A Canvas control used to create a "Little Arrows" control**



Canvas controls can be used to create extremely sophisticated controls. In Figure 51, a Canvas control is used to provide a table of data with rows that can be selected and columns that can be sorted by clicking on the column title.

FIGURE 51. A sophisticated control created using a Canvas control

Name	Age	City
Smith	20	London
Jones	10	Paris
Blake	30	Paris
Clark	20	London
Adams	30	Athens
Björn	22	Reykjavik

The "Little Arrow" and Table controls above were created by Björn Eiríksson.

TABLE 20. Canvas properties

Name	Description
Super	The class of object the Canvas is based on.
Name	The internal name of the Canvas used to identify it in programming code.
Index	The position of the Canvas in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the Canvas.
Top	The distance (in pixels) between the top edge of the window and the top edge of the Canvas.
Width	The width (in pixels) of the Canvas.
Height	The height (in pixels) of the Canvas.
LockLeft	Keeps the distance between the left side of the window and the left side of the Canvas from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the Canvas from changing when the window is resized.

TABLE 20. **Canvas properties (Continued)**

Name	Description
LockRight	Keeps the distance between the right side of the window and the right side of the Canvas from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the Canvas from changing when the window is resized.
Visible	The Canvas will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the Canvas.
DisabledBal-loonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
Backdrop	A picture from the Project window that will be displayed inside the Canvas control.
Enabled	The control will be initially enabled.

ImageWell

The ImageWell control provides an area in which you can display a PICT image. You can easily program the ImageWell control to accept a dragged picture.

FIGURE 52. **An ImageWell**

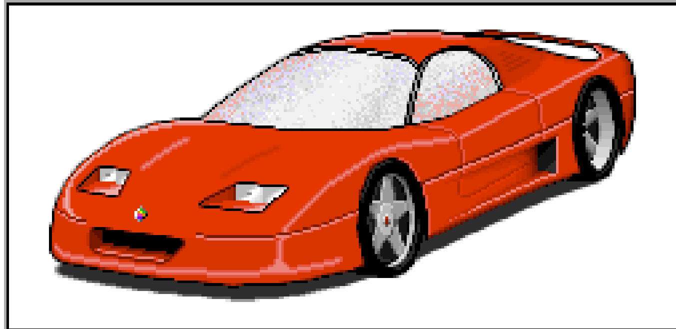


TABLE 21. **ImageWell Properties**

Name	Description
Super	The class of object the ImageWell is based on.
Name	The internal name of the ImageWell used to identify it in programming code.
Index	The position of the ImageWell in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the ImageWell.
Top	The distance (in pixels) between the top edge of the window and the top edge of the ImageWell.
Width	The width (in pixels) of the ImageWell.
Height	The height (in pixels) of the ImageWell.
LockLeft	Keeps the distance between the left side of the window and the left side of the ImageWell from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the ImageWell from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the ImageWell from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the ImageWell from changing when the window is resized.

TABLE 21. ImageWell Properties

Name	Description
Visible	The ImageWell will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the ImageWell.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
Image	The name of the PICT image to be displayed. Drag a picture to the Project Window to make it available.

Controls for Playing Movies, Music, and Animation

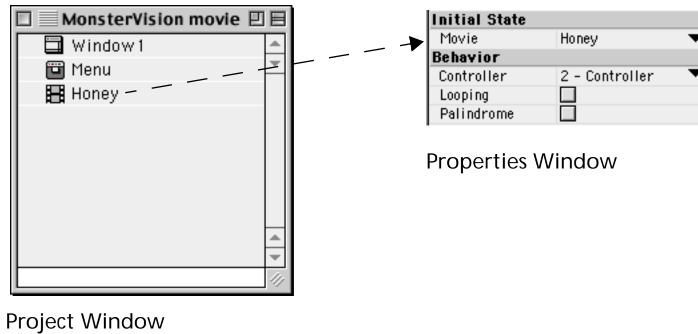
If the user has QuickTime™ installed, your application can play QuickTime movies and use QuickTime Musical Instruments to play music.

MoviePlayer

The MoviePlayer control displays the standard QuickTime movie controller. From the Design environment, you can select the QuickTime movie that will be associated with a particular MoviePlayer control. You can also determine the default appearance of the movie controller. Your choices are: the controller is displayed, a badge (a small icon that, when clicked, reveals the controller) is displayed, or no controls are displayed.

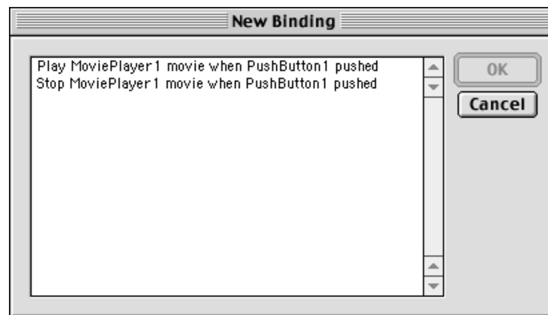
Assigning a QuickTime movie to a MoviePlayer control is amazingly easy. First, drag the QuickTime movie to the Project window. Then assign the movie to the Movie property of the control using the Properties Window. This is illustrated in Figure 53.

FIGURE 53. **Assigning a movie to a MoviePlayer control.**



You can then add Stop and Play pushbuttons to the window and assign them actions using object binding.

Add a PushButton to the window. Hold down the Shift and Command keys and draw a line from the PushButton to the MoviePlayer control. A New Binding dialog box appears, giving you a choice of automatic actions:



Choose Play MoviePlayer1 Movie when PushButton1 pushed. Next, add another PushButton to the window and assign the Stop MoviePlayer1 Movie binding to it. The result is a fully functional movieplayer application shown in Figure 54

FIGURE 54. **A simple movieplayer application.**

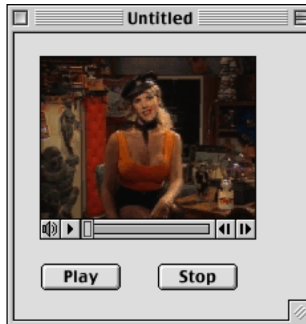


TABLE 22. **MoviePlayer properties**

Name	Description
Super	The class of object the MoviePlayer is based on.
Name	The internal name of the MoviePlayer used to identify it in programming code.
Index	The position of the MoviePlayer in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the MoviePlayer.
Top	The distance (in pixels) between the top edge of the window and the top edge of the MoviePlayer.
Width	The width (in pixels) of the MoviePlayer.
Height	The height (in pixels) of the MoviePlayer.
LockLeft	Keeps the distance between the left side of the window and the left side of the MoviePlayer from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the MoviePlayer from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the MoviePlayer from changing when the window is resized.

TABLE 22. **MoviePlayer properties (Continued)**

Name	Description
LockBottom	Keeps the distance between the bottom of the window and the bottom of the MoviePlayer from changing when the window is resized.
AutoResize	Changes the size of the movie area to fit the size of the movie.
Border	Draws a border around the MoviePlayer.
Speaker	Adds the volume slider to the MoviePlayer.
HasStep	Adds the previous and next frame buttons to the MoviePlayer.
Visible	The MoviePlayer will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the MoviePlayer.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
Movie	The movie to be played in the MoviePlayer.
Controller	Determines how the controller will appear at the bottom of the MoviePlayer (none, badge, or controller).
Looping	Plays the movie continuously once it has started.
Palindrome	Plays the movie backwards once it reaches its end.

NotePlayer

Although the NotePlayer control displays an icon when placed in a window in the Design environment, it has no interface. It is designed only for playing musical notes using QuickTime Musical Instruments. see “NotePlayer Control” on page 276 of the Language Reference for more details.

TABLE 23. **NotePlayer properties**

Name	Description
Super	The class of object the NotePlayer is based on.
Name	The internal name of the NotePlayer used to identify it in programming code.

TABLE 23. **NotePlayer properties (Continued)**

Name	Description
Index	The position of the NotePlayer in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the NotePlayer.
Top	The distance (in pixels) between the top edge of the window and the top edge of the NotePlayer.
Instrument	The number that represents the QuickTime Musical Instrument to be used to play notes. see "NotePlayer Control" on page 276 of the Language Reference for a list of instruments.

SpriteSurface

This control is used to create animation. When you call the SpriteSurface's Run method, the menu bar, all windows, and the desktop are hidden. This allows the animation to fill the screen. This animation is done using *Sprites*. A Sprite is simply an object with a picture that can be moved across the screen by your programming code. The SpriteSurface will automatically handle all of the drawing of the background and the sprites for you. The SpriteSurface will tell you when two sprites collide and give you the opportunity to test for keys pressed by the user to allow to you interact with them.

For a complete list of SpriteSurface properties, see the Language Reference.

Miscellaneous Controls

PopupArrow Control

In REALbasic, you can control both the direction and size of the popup arrow using one property, Typically, you would use a PopupArrow control as part of a custom control. The orientation

of the arrow indicates whether the custom control can display additional information or options. Figure 55 shows all possible directions.

FIGURE 55. **Examples of the PopupArrow Control**

TABLE 24. **PopupArrow Properties**

Name	Description
Super	The class of object the PopupArrow is based on.
Name	The internal name of the PopupArrow used to identify it in programming code.
Index	The position of the PopupArrow in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the PopupArrow.
Top	The distance (in pixels) between the top edge of the window and the top edge of the PopupArrow.
Width	The width (in pixels) of the PopupArrow.
Height	The height (in pixels) of the PopupArrow.
LockLeft	Keeps the distance between the left side of the window and the left side of the PopupArrow from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the PopupArrow from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the PopupArrow from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the PopupArrow from changing when the window is resized.

TABLE 24. PopupArrow Properties

Name	Description
Visible	The PopupArrow will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the PopupArrow.
DisabledBal-loonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.
Facing	Controls direction and size of a PopupArrow 0—East 1—West 2—North 3—South 4—Small East 5—Small West 6—Small North 7—Small South

DisclosureTriangle Control

A disclosure triangle control is used to display hierarchical lists, i.e., the List view of files and folders in a Finder window. In REALbasic, you can control the direction of the DisclosureTriangle (left or right) and whether it is in the 'disclosed' (down) state.

FIGURE 56. Disclosure Triangles

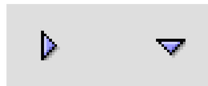


TABLE 25. **DisclosureTriangle Properties**

Name	Description
Super	The class of object the DisclosureTriangle is based on.
Name	The internal name of the DisclosureTriangle used to identify it in programming code.
Index	The position of the DisclosureTriangle in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the DisclosureTriangle.
Top	The distance (in pixels) between the top edge of the window and the top edge of the DisclosureTriangle.
Width	The width (in pixels) of the DisclosureTriangle.
Height	The height (in pixels) of the DisclosureTriangle.
LockLeft	Keeps the distance between the left side of the window and the left side of the DisclosureTriangle from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the DisclosureTriangle from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the DisclosureTriangle from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the DisclosureTriangle from changing when the window is resized.
Visible	The DisclosureTriangle will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the DisclosureTriangle.
DisabledBalloonHelp	The text that should appear when the user moves the mouse over the control while the control is disabled and BalloonHelp is on.

TABLE 25. DisclosureTriangle Properties

Name	Description
Facing	Direction in ‘undisclosed’ state. 0—Right facing 1—Left facing
Value	True corresponds to downward; False corresponds to either left or right depending on the value of *facing.*

LittleArrows Control

The LittleArrows control is commonly used as an interface for scrolling. You use two events, Up and Down, to determine whether the user has clicked an arrow.

FIGURE 57. LittleArrows Control

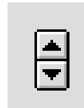


TABLE 26. LittleArrows Events

Name	Description
Up	The user has clicked the Up arrow.
Down	The user has clicked the Down arrow.

TABLE 27. LittleArrows Properties

Name	Description
Super	The class of object the LittleArrows is based on.
Name	The internal name of the LittleArrows used to identify it in programming code.
Index	The position of the LittleArrows in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the LittleArrows.

TABLE 27. **LittleArrows Properties**

Name	Description
Top	The distance (in pixels) between the top edge of the window and the top edge of the LittleArrows.
Width	The width (in pixels) of the LittleArrows.
Height	The height (in pixels) of the LittleArrows.
LockLeft	Keeps the distance between the left side of the window and the left side of the LittleArrows from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the LittleArrows from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the LittleArrows from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the LittleArrows from changing when the window is resized.
Visible	The LittleArrows will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the LittleArrows.
Disabled Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the LittleArrows when the control is disabled.
Enabled	If True, the LittleArrows control responds to mouse clicks.

ChasingArrows Control

The ChasingArrows control is often displayed to indicate that a time-consuming operation is in progress. The ChasingArrows control appears when its Visible property is set to True.

FIGURE 58. **The ChasingArrows Control**

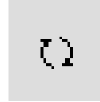


TABLE 28. **ChasingArrows Properties**

Property	Description
Super	The class of object the ChasingArrows is based on.
Name	The internal name of the ChasingArrows used to identify it in programming code.
Index	The position of the ChasingArrows in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the ChasingArrows.
Top	The distance (in pixels) between the top edge of the window and the top edge of the ChasingArrows.
Width	The width (in pixels) of the ChasingArrows.
Height	The height (in pixels) of the ChasingArrows.
LockLeft	Keeps the distance between the left side of the window and the left side of the ChasingArrows from changing when the window is resized.
LockTop	Keeps the distance between the top of the window and the top of the ChasingArrows from changing when the window is resized.
LockRight	Keeps the distance between the right side of the window and the right side of the ChasingArrows from changing when the window is resized.
LockBottom	Keeps the distance between the bottom of the window and the bottom of the ChasingArrows from changing when the window is resized.
Visible	The ChasingArrows will be visible when the window opens.
Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the ChasingArrows.
Disabled Balloon Help	The text that will appear if the user has Balloon Help on and moves the pointer over the ChasingArrows when the control is disabled.

Controls for Handling Communications

REALbasic provides controls that allow your application to communicate through the serial port (for communicating via a modem or through a serial cable to another device) and over a network to other computers using TCP/IP, the Internet's communication protocol.

Serial

Although the Serial control displays an icon when placed in a window in the Design environment, it has no interface. It is designed only for executing code to communicate via the serial port. see "Serial Control" on page 375 of the Language Reference for more details.

TABLE 29. **Serial control properties**

Name	Description
Super	The class of object the Serial control is based on.
Name	The internal name of the Serial control used to identify it in programming code.
Index	The position of the Serial control in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the Serial control.
Top	The distance (in pixels) between the top edge of the window and the top edge of the Serial control.
Port	Determines which port (serial or printer) port will be used to read and write data.
Baud	The speed at which data will be read or written through the chosen port.
Bits	Determines the number of data bits used during communications.
Parity	Determines the type of parity (no parity, odd parity, even parity).
Stop	Determines the number of stop bits used during communications.

TABLE 29. **Serial control properties (Continued)**

Name	Description
XON	Enables XON flow control.
CTS	Enables CTS flow control.
DTR	Enables DTR flow control.

The Serial control can be instantiated via code since it is not a subclass of Control. This allows you to easily write code that does communications without adding the control to a window.

Socket

Although the Socket control displays an icon when placed in a window in the Design environment, it has no interface. It is designed only for executing code to communicate with other computers on the Intranet or Internet using TCP/IP.

The Socket control can be instantiated via code since it is not a subclass of Control. This allows you to easily write code that does communications without adding the control to a window.

For more information, see “Socket Control” on page 384 of the Language Reference.

TABLE 30. **Socket control properties**

Name	Description
Super	The class of object the Socket control is based on.
Name	The internal name of the Socket control used to identify it in programming code.
Index	The position of the Socket control in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the Socket control.
Top	The distance (in pixels) between the top edge of the window and the top edge of the Socket control.

TABLE 30. **Socket control properties (Continued)**

Name	Description
Address	The IP address to send data to.
Port	The TCP/IP port to transmit/receive data on.

The Timer Control

The Timer control executes some code once or repeatedly after a period of time has passed. Although the Timer control displays an icon when placed in a window in the Design environment, it has no interface. see “Timer Control” on page 432 of the Language Reference for more details.

TABLE 31. **Timer control properties**

Name	Description
Super	The class of object the Timer control is based on.
Name	The internal name of the Timer control used to identify it in programming code.
Index	The position of the Timer control in a control array.
Left	The distance (in pixels) between the left edge of the window and the left edge of the Timer control.
Top	The distance (in pixels) between the top edge of the window and the top edge of the Timer control.
Mode	Determines the number of times the Timer will execute (0 - off, 1- single, 2- multiple).
Period	The time in milliseconds between executions.

Object Binding

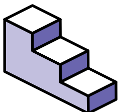
Once you have added the application’s controls to a window, you can use the Code Editor to add any desired functionality. For example, you can use the Code Editor to specify the behavior of a

PushButton when the user clicks it by adding some code to the PushButton's Action event.

In some cases, you can actually add functionality without writing any code whatsoever. To do this you use a feature called "object binding."

With object binding, you specify an action when some aspect of one control changes without code. A simple example of object binding was presented in the section on the MoviePlayer control on page 97. Two PushButtons, a "Play" and a "Stop" pushbutton were bound to the MoviePlayer control. The bindings themselves specify the functionality; there is no code being written "behind the scenes."

In the case of the MoviePlayer example, the binding has a directional characteristic. One control is referred to as the "source" control—the control that the user interacts with—and the other control is the "destination" control—the control that does something when the user invokes the binding



To establish an object binding, do this:

1. Hold down the Shift and Command keys and drag from the "source" control to the "target" control.
2. An Object Binding dialog appears, listing the possible binding actions that are available. The built-in bindings are shown in Table 32 on page 112. Custom bindings can be added.
3. Choose the desired action and click OK.

A line connecting the two controls appears in the window. When you run your application, the binding works just as if you had entered equivalent code for the action in the Code Editor. For example, the object bindings in the MoviePlayer example are equivalent to the lines `MoviePlayer1.Play` and

`MoviePlayer1.Stop` that could have been inserted into the Action events of the Play and Stop buttons, respectively.

If you forget what the binding specifies, you can select the line and its Properties window will display the currently selected binding. If you need to modify the binding, select the line, press the Delete key, and establish a different bind.

Here are the binds that are included with REALbasic.

TABLE 32. **Built-in Object Binds in REALbasic**

Source Object	Target Object	Binds
PushButton or BevelButton	MoviePlayer	Play MoviePlayer when Button is clicked Stop MoviePlayer when Button is clicked.
PushButton or BevelButton	DataBaseQuery	Requery DatabaseQuery when button is clicked.
RadioButton	DatabaseQuery	Requery DatabaseQuery when RadioButton is selected. Requery DatabaseQuery when RadioButton is deselected.
Database-Query	PopupMenu	Bind PopupMenu with Database-Query results. Bind DatabaseQuery parameter with PopupMenu. Bind DatabaseQuery parameter with PopupMenu RowTag. (See the section “Using Object Binding” on page 369 for an example of these bindings.)

TABLE 32. **Built-in Object Binds in REALbasic (Continued)**

Source Object	Target Object	Binds
Database-Query	ListBox	Bind Listbox with DatabaseQuery results. Bind DatabaseQuery parameter with ListBox. Requery DatabaseQuery when ListBox gains focus. Requery DatabaseQuery when ListBox loses focus.
PushButton or BevelButton	ListBox	Enable Button when ListBox has a selection.
Checkbox	ListBox	Enable Listbox when Checkbox is checked Enable Checkbox when Listbox has a selection.
RadioButton	Listbox	Enable RadioButton when Listbox has a selection.
CheckBox	EditField	Enable EditField when CheckBox is checked.
Checkbox	MoviePlayer	Play MoviePlayer movie when Check-box is checked. Stop MoviePlayer movie when Check-box is checked. Play MoviePlayer movie when Check-box is unchecked. Stop MoviePlayer movie when Check-box is unchecked. Enable MoviePlayer when Checkbox is checked.
Checkbox	DatabaseQuery	Requery DatabaseQuery when Check-box is checked. Requery DatabaseQuery when Check-box is unchecked.
Checkbox	Contextual Menu	Enable ContextualMenu when Check-box is checked.

TABLE 32. **Built-in Object Binds in REALbasic (Continued)**

Source Object	Target Object	Binds
CheckBox	PopupMenu	Enable PopupMenu when Checkbox is checked.
CheckBox	PopupArrow	Enable PopupArrow when Checkbox is checked.
CheckBox	DisclosureTrian-gle	Enable DisclosureTriangle when Checkbox is checked.
CheckBox	SpriteSurface	Enable SpriteSurface when CheckBox is checked.
CheckBox	NotePlayer	Enable NotePlayer when CheckBox is checked.
CheckBox	ImageWell	Enable ImageWell when CheckBox is checked.
CheckBox	TabPanel	Enable TabPanel when CheckBox is checked.
CheckBox	ChasingArrows	Enable ChasingArrows when CheckBox is checked.
RadioButton	MoviePlayer	Play MoviePlayer movie when RadioButton is selected. Stop MoviePlayer movie when RadioButton is selected. Play MoviePlayer movie when RadioButton is deselected. Stop MoviePlayer movie when RadioButton is deselected.
ListBox	LittleArrows	Enable LittleArrows when Listbox has a selection.
ListBox	DisclosureTrian-gle	Enable DisclosureTriangle when List-Box has a selection.
ListBox	PopupMenu	Enable PopupMenu when ListBox has a selection.
ListBox	PopupArrow	Enable PopupArrow when ListBox has a selection.
ListBox	Contextual-Menu	Enable ContextualMenu when ListBox has a selection.

TABLE 32. **Built-in Object Binds in REALbasic (Continued)**

Source Object	Target Object	Binds
ListBox	ScrollBar	Enable ScrollBar when ListBox has a selection.
ListBox	Slider	Enable Slider when ListBox has a selection.
EditField	ListBox	Bind EditField with ListBox (places selected row in ListBox in EditField). Enable EditField when ListBox has a selection.
EditField	PopupMenu	Bind EditField with PopupMenu (places selected menu item in EditField). Bind EditField with PopupMenu RowTag.

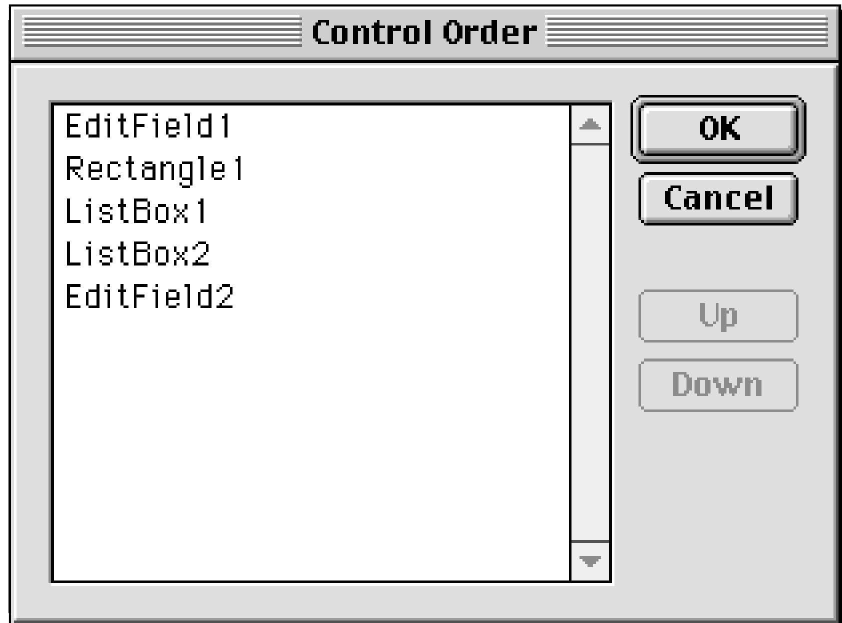
In addition, it is possible to define custom object bindings using the language. For information on custom bindings, see the section “Custom Object Bindings” on page 346.

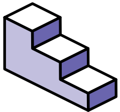
Changing The Tab (Control) Order

The order in which the user moves through controls that receive the focus (EditFields and ListBoxes) when he presses the Tab key is called the *Control Order* (also known as the *Tab Order*). The Control Order is actually controlled by the control layers. When a window opens, REALbasic places the focus in the control that is farthest back that can also receive the focus. You could change the control order by using the Format menu to move controls through the control layers.

Instead, the Control Order dialog box makes the job much easier.

FIGURE 59. The Control Order dialog box





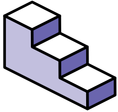
To change the Control order, do this:

1. Choose Format ▶ Control Order.
2. Select the control in the list, whose tab order you wish to change.
3. Use the Up button to move the control up one position in the tab order or the Down key to move the control down one position in the tab order.

Aligning Controls with Other Controls

REALbasic's Interface Assistant makes it easy to align a particular control with another control. Simply drag the control until it is close to being aligned with the other control. When you get close to aligning the two controls, horizontal and/or vertical alignment rules will appear. When you release the mouse button, REALbasic will snap the control you are dragging into place.

Note: If this feature is getting in your way, you can turn it off temporarily by holding down the ⌘ key while dragging a control.



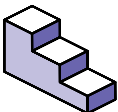
If you need to align several controls, do this:

1. Click on the control whose position is already correct to select it.
2. Choose Format ▶ Move to Back to insure that the selected control remains in place while the other controls move to align with it.
3. While holding down the Shift key, select each of the controls you wish to align together.
4. Depending on which edges you wish to align, choose Format ▶ Align Objects then choose Align Left Edges, Align Right Edges, Align Top Edges, or Align Bottom Edges from the Align Objects submenu.

FIGURE 60. **The Align Objects submenu.**

Spacing Controls Evenly

REALbasic provides an easy way to reposition controls to evenly distribute empty space between them.



To distribute the controls evenly, do this:

1. Click on a control to select it.
2. Hold down the Shift key and select at least two other controls.
3. To distribute the controls horizontally, choose Format ▶ Align Objects then choose Space Horizontally from the Align Objects sub-menu.
4. To distribute the controls vertically, choose Format ▶ Align Objects then choose Space Vertically from the Align Objects submenu.

Adding Menus and Menu Items

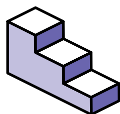
REALbasic has a built-in Menu editor that makes adding menus and menu items to your project easy. The menus displayed in the Menu Editor will be displayed when you choose Debug ▶ Run (⌘-R) in the Design environment or when in a stand-alone version of your application.

There are four steps for implementing a menu and its items:

- Adding the menu using the Menu Editor,
- Adding the menu's items using the Menu Editor,
- Enabling the menu item. By default, a menu item is disabled until the user attempts to pull down a menu. At that moment, you can decide if conditions are right for the menu item to be enabled. For example, a Save menu item would need to check that the document have never been saved or that changes have been made to the document since the last save operation. For information about enabling menu items, see the section, "Enabling Menu Items" on page 212.
- Adding a menu handler that tells REALbasic what to do when the user chooses the (enabled) menu item. The menu handler is where you place the code that is run when the user actually selects the menu item. For information about creating a menu handler, see the section, "Adding Code To a Menu Item" on page 211.

Adding Menus

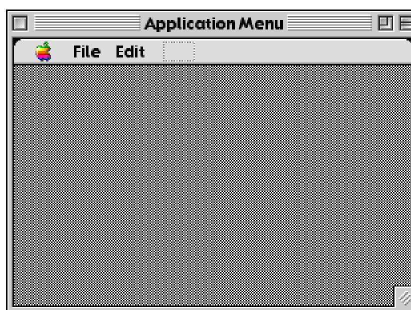
REALbasic adds File and Edit menus to your project automatically. Every application should have at least a File menu with a Quit menu item. You can remove the Edit menu only if your application has no controls that could be edited by the Edit menu items.



To add a menu to your project, do this:

1. Bring the Project window to the front by clicking on it. If it is obscured by other windows, Choose Window ▶ Project (⌘-0).
2. Double-click on the Menu object to open the Menu Editor. The Menu Editor window appears.

FIGURE 61. **The Menu Editor.**



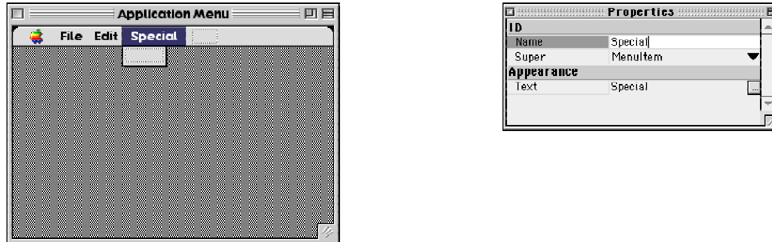
3. Click on the dotted rectangle in the Menu Editor's menu bar to select it.
4. In the Properties Window, enter the Name of the menu and the Text that will appear in the menu bar.

TABLE 33. **Menu properties**

Name	Description
Name	The internal name of the Menu used to identify it in programming code.
Super	The class of object the Menu control is based on.
Text	The text that will appear in the Menu bar.

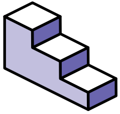
Figure 62 shows a new menu and its properties in the Properties window.

FIGURE 62. **A new menu in the Menu Editor.**



Adding a Help Menu

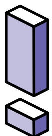
Most Macintosh applications have a Help menu that is the right-most menu in the application. At a minimum, this menu has an About Balloon Help menu item and a Show Balloons menu item. The Help menu may also contain menu items that give the user access to Apple Guide files or an application specific help system. You can add a Help (complete with About Balloon Help and Show Balloons menu items) menu to your project.



To add a Help menu, do this:

1. Add a menu to the end of your menu bar.
2. Set the Text property of the menu to **Help**.

Any menu items you add to the Help menu will be displayed after the About Balloon Help and Show Balloons menu items.



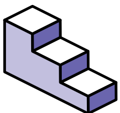
Note: If you are planning on including Apple Guide files with your application, there are two handy applications for generating Guide files. AG Author is from Lakewood Software and you can download a demo version from their web site at www.lakewoodsoftware.com. The other is called Guide Composer and you can download a demo version from www.downloads.com.

Adding Menu Items

The Menu Editor makes it easy to add menu items to your menus. You can assign keyboard shortcuts to menu items but remember that the Macintosh looks for a shortcut starting from the left most menu. That means that if you assign the same keyboard shortcut to two different menu items, one of them won't work. There are also several specific keyboard shortcuts that are reserved for specific functions. According to Apple's Macintosh Human Interface Guidelines, these are:

TABLE 34. **Reserved keyboard shortcuts**

Menu	Keys	Command
File	⌘-N	New
File	⌘-O	Open…
File	⌘-W	Close
File	⌘-S	Save
File	⌘-P	Print…
File	⌘-Q	Quit
Edit	⌘-Z	Undo
Edit	⌘-X	Cut
Edit	⌘-C	Copy
Edit	⌘-V	Paste
Edit	⌘-A	Select All
Edit	⌘-period	Terminate an operation



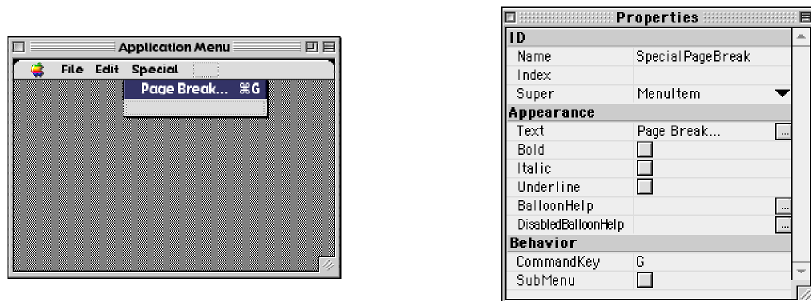
To add a menu item to a menu, do this:

1. In the Menu Editor, select the menu you wish to add a menu item to by clicking on it.
2. Click on the dotted rectangle at the bottom of the menu to select it.
3. In the Properties window, enter the Text for the menu item and press Return. The Menu Editor will supply a default Name.

4. If desired, add a keyboard shortcut by assigning a letter to the CommandKey property.
5. (Optional) Select the Bold, Italic, or Underline properties or add Help text.

Figure 63 on page 122 shows the Page Break menu item added to the Special menu.

FIGURE 63. **A new menu item in the Special menu.**

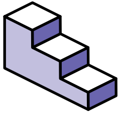


If you plan to deploy your application on Windows and want to add keyboard shortcuts for that platform ("accelerators"), you must create such menus and menu items using the Constants system. This process is described in the section "Using Constants to Add Windows Keyboard Shortcuts to Menus and Menu Items" on page 224.

Note: Although the Menu Editor allows you to use lowercase characters as keyboard shortcuts, only uppercase characters should be used.

Adding a Submenu

Submenus are menu items that, when selected, display an additional menu to their right. The menu item itself is not selectable. It acts only as a title for the submenu.



To add a submenu to an existing menu item, do this:

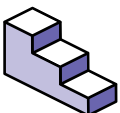
1. Click on the menu item in the Menu Editor to select it.
2. In the Properties window, place a checkmark in the Submenu property. A new submenu item appears in the Menu Editor.
3. In the Menu Editor, click on the dotted rectangle that appears in the new submenu item just to the right of the menu item you selected.
4. In the Properties window, enter the Name and Text for the submenu Item.

Submenus can give the user fast access to a group of menu items. However, they can be difficult to navigate for the new computer user. They also hide menu items from view. If a user scans through the menu items looking for a particular menu item, he may not look at the submenus. Consider the audience for your application before using submenus. If many of your users will be new computer users, consider displaying a dialog box to choose the functions you could put in a submenu.

Submenu items themselves can be submenus. Seriously consider your audience when choosing to have multiple level submenus. Many of your users may find navigating multiple level submenus difficult.

Moving Menus and Menu Items

A menu item can be moved to a new position by dragging the menu item. You can only move a menu item to another position on the same menu. If you need to move the menu item to another menu, you have to delete it and recreate it on the other menu.



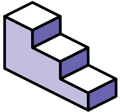
To move a menu item, do this:

1. Click on the menu item you want to move to select it.

2. Drag the menu item towards the position on the menu where you want it. A bold line appears above the menu item.
3. When the bold line is in the position where you want to move the menu item to, release the mouse button.

You can also move menus within the menu bar. In a similar fashion, drag a menu in the menu bar and move it to the left or right. As you drag, a vertical bar in the menu bar indicates the position of the menu.

Removing Menu Items



To remove a menu item from a menu, do this:

1. In the Menu Editor, click on the menu item to select it.
2. Press the Delete key or choose Edit ▶ Clear.

Adding A Menu Item Separator

Menu item separators are lines that appear in between menu items to logically group items together. To add a menu item separator, simply create a new menu item and type a dash ("-") in the menu item's Text property.

Menu Item Arrays

In certain cases, you cannot specify the menu items that belong in a menu in advance. They may change depending on the context in which the application is used or on the computer environment in which the application is running.

A common example of this is the Font menu that is normally included in any application that supports styled text. The programmer has no way of knowing in advance which fonts happen to be installed on the user's computer.

You begin by creating the Menu normally. You then create a menu item, name it, and set its Index property to 0 to start the menu item array. You then must write code to populate the array with the names of the fonts installed on the user's computer. If we assume that fonts won't be added or deleted while the application is running, we can build the font list when the application starts up.

To do this, place the code in the Application class's Open event handler. This runs when the application starts up. Create a new class named **App** and make its Super Class **Application**.

The following code, for example, populates the Font menu when the application opens. It uses the built-in Font function which returns the name of the ith font on the user's computer.

```
Dim m as MenuItem
Dim i, nFonts as Integer

nFonts=FontCount-1
//build the font menu
FontFontName(0).text=Font(0)
For i=1 to nFonts
 m=New FontFontName //create new menu item
 m.Text=Font(i)//obtain name of ith font
Next
```

Apple's Macintosh User Interface Guidelines

The quality of your application's interface will determine how successful your user will be in using it. It's absolutely critical that

your users find the interface intuitive. Studies have shown that if a user can't accomplish something within the first 15 minutes of using an application, he will give up in frustration. Beyond simply being intuitive, the more polished an application's interface is, the more professional it will appear to the user. Remember that without realizing it, your users will be comparing your application's interface to all of the other applications they have used.

REALbasic's Interface Assistant™ helps you create a nice interface by making it easy to align controls with other controls. But there is more to a professional, polished interface than simply aligning controls. We all think we know how to create a nice interface because we have used lots of applications. But using an interface is a lot different from designing one. If you haven't done so already, read Apple's *Macintosh Human Interface Guidelines*. This comprehensive guide will teach you what you need to know to give your application a professional interface. You will also learn the reasons behind the implementation of many of the features of the Macintosh user interface. Apple's Macintosh Human Interface Guidelines is part of the *Inside Macintosh* series published by Addison-Wesley and can be purchased through most bookstores. You can also download it for free through Apple's Developer World web site at www.devworld.apple.com.

CHAPTER 4

BASIC Programming Concepts

Programming is all about getting the computer to do what you want it to do. The key is knowing how give the computer instructions in a way it will understand. That's where programming languages come in. There are many different programming languages that are designed to make the communication easier in different situations.

In this chapter you will learn about the BASIC programming language, how it is different in REALbasic, and the fundamentals of programming.

Contents

- Data Types
- Storing Values in Properties and Variables
- Executing Instructions with Methods

- Executing Instructions Repeatedly with Loops
- Decision Making

BASIC versus REALbasic

The BASIC language was created in the 1960's for the purpose of teaching people programming. Most of what made other languages difficult to master was removed from BASIC to make learning it easier. In fact, BASIC is an acronym that stands for Beginners All-Purpose Symbolic Instruction Code.

For a long time BASIC was considered less powerful than other languages, but this was mostly due to the way it was implemented rather than the language itself. Spoken languages wouldn't be considered to be very powerful if you could only speak one word every 10 minutes. Computers actually only understand two things, 1 and 0. That's it. That's all they know. The rest of what a computer does all breaks down to that fundamental concept. These 1's and 0's that computers understand are referred to as machine language. Most versions of BASIC used an interpreter program to execute the code. This means that each time a program ran, the BASIC interpreter had to turn the BASIC code into machine language. Other languages had compilers which are special programs that translate the programing language into machine language all at once. This makes programs execute faster because the real-time interpretation is removed.

REALbasic has a compiler built-in to it. That means your code runs as fast as possible. BASIC is a traditional programming language that starts with the first line programming code and continues until the last line. REALbasic is a modern, object-oriented version of BASIC. If you are new to programming that might not

mean much now but it will. REALbasic takes the simplicity of the BASIC language and adds the power of modern programming through its object-oriented implementation and compiler. Also, most programming languages require you to know quite a bit about how to communicate with the computer's operating system. REALbasic abstracts you from all of that making it easier for you to learn and easier to run your application on computers running operating systems that are different from the one you created your application on.

Storing Values in Properties and Variables

When you need to store information so you can access it again even after you have shut off your computer, you tell your computer to store the information in a document. When a computer needs to store information temporarily, it's stored in the computer's memory. The computer's memory is like a series of organized boxes. Each box has a location in memory with an address that is used to locate it. These locations are given names to make them easier to work with. Depending on how these memory locations are used, they are called Variables and Properties.

What are Properties?

The values that make up the description of an object like a window are called *Properties*. The title of a window is a property. The width of the window is a property. When a window is opened, these properties are copied into memory. You can access them using their names. You can get values from them and you can store new values in them. For example, if you wanted the title of a window to change when the user clicks a button, you would

set the title property of the window to the new value. Each property can hold a certain type of data. Some properties store text (like a window title) while others store numbers (like the window's width property). Later in this chapter, you will learn how to assign values to properties and how to get the values that are stored in properties.

Variables

Sometimes you will need to store a value that isn't related directly to an object like a window or a button. In this case you use a variable. A variable is just like a property but it isn't directly related to any particular object. Later in this chapter, you will learn how to create variables, assign values to them and get values from them.

Data Types

To make programming code execute faster and to provide powerful commands that save you time when programming, computers have to be able to make certain assumptions about the information you give them. For example, when you give a computer a piece of information, the computer needs to know if it's a number, a string of characters, a date, etc. If you didn't tell the computer what kind of data you are giving it, it wouldn't know whether you meant 1 plus 1 to be 2 or 11. In this example, telling the computer that you are giving it numbers will result in 2. Telling it you are giving it simply a string of typed characters will result in 11. There are many data types that REALbasic understand but there are five data types that are by far the most common. They are String, Integer, Single, Double, and Boolean.

String

A String is just series (or string) of characters. Basically any kind of information can be stored as a string. "Jeannie", "3/17/98",

"45.90" are all examples of strings. You might be thinking "Hey, those last two don't look like strings" but they are. When you place quotes around information in your code, you are telling REALbasic to look at the data as just a string of characters and nothing more. The maximum length of a string is based only on available memory.

You can concatenate two strings with the addition symbol (+). For example, the statement "Big" + "Dog" results in the string "BigDog". That is really the extent of the "mathematics" you can perform on strings. However, REALbasic has many built-in functions that make processing strings easy.

Integer

An Integer is a whole number between approximately -2 billion and +2 billion. In other programming languages, REALbasic's Integer type is called a Long Integer or just a Long. Because integers are numbers, you can perform mathematical calculations on them. Unlike strings, integers do not have quotes around them in your code. An Integer value uses 4 bytes of memory.

Single

A Single is a number that can contain a decimal value. There is no practical limit to the value of a Single. In other languages, REAL-basic's Single may be referred to as a single precision real number. Because Singles are numbers, you can perform mathematical calculations on them. Single numbers use 4 bytes of memory.

Double

A Double is a number that can contain a decimal value. Unlike Integers, Doubles have no limit to the range of numbers they can hold. In other languages, REALbasic's Double may be referred to as a double precision real number. Because Doubles are numbers,

you can perform mathematical calculations on them. Doubles use 8 bytes of memory. Since Doubles use more decimal places to represent a number, you would want to use Doubles rather than Singles if the extra precision is important in your calculations. However, calculations on Singles is faster than Doubles on both the Macintosh and Windows platforms.

Table 35 gives the upper and lower limits of integers, singles, and doubles:

TABLE 35. **Upper and lower limits of Integers, Singles, and Doubles**

Data Type	Smallest Value	Largest Value
Integer	-2147483648	2147483647
Single	1.175494 e-38	3.402823 e+38
Double	2.2250738585072013 e-308	1.7976931348623157 e+308

Boolean

Boolean means True or False. Boolean values are False by default but can be set to True using REALbasic's True function and back to False using the False function. Some of the properties of objects in REALbasic are boolean values. For example, most of the controls have an Enabled property that is boolean.

Other Data Types

There are many other data types. You will learn about these in the next chapter.

Changing a Value From One Data Type to Another

There may be times when you need to change a value from one data type to another. This is usually because you want to use the value with something that is designed to work with a different data type. For example, you might want to include a number in

the title of a window. The title of a window is a string, not a number. Consequently, if you try to assign a number to the title of a window, REALbasic will display an error message when you run your application. The error will tell you that the two data types are not compatible (they are different). Since the window title is a string, you will need to change the number into a string before you can assign it to the window title.

Fortunately, REALbasic has a built-in function called Str (which stands for String) that can change a number into a string. see "Str Function" on page 405 of the Language Reference for more information. There is also a built-in function called Val (which is short for Value) that changes strings into numbers. see "Val Function" on page 441 of the Language Reference for more information.

Assigning Values to Properties

The basic syntax for assigning a value is:

```
objectName.propertyName=value
```

For example, if you have a pushbutton called pushbutton1, and you want to set its caption property to "OK", you would use the following code:

```
pushbutton1.caption="OK"
```

You can read this as *change pushbutton1's caption property to "OK".* This syntax is used when you want a control in a window to change a property of a control in the same window. If you want a control to change a property of a control in another open window, you must include the target window's name (not title) in the syntax. For example, say you have two open windows whose names are window1 and window2 respectively. You want a

pushbutton on window1 to set the value of pushbutton1's caption on window2 to "OK". The syntax looks like this:

```
window2.pushbutton1.caption="OK"
```

If you didn't specify the window, REALbasic would implicitly assume you meant the control called pushbutton1 in the window that contains the object executing the code. If you specify a window that is not open, REALbasic will open the window and make the change. If you have more than one copy of the window open that contains the control you are trying to change, this syntax won't work because you won't be able to tell REALbasic which copy of the window you are referring to. You will learn how to deal with this issue in the next chapter.

If a control is going to change a property of its own window, the window name is not required. The window name is implicit. For example, if you wanted a pushbutton to change its window's title property to "Hello World" when the user clicks it, you would use this syntax:

```
title="Hello World"
```

Getting Values From Properties

You can get a value from a property in almost the same way you store values in properties. The only difference is that the target of the value (where you want the value of the property stored) goes on the left side of the equals sign and the object and property names go on the right. For example, if you had a variable named X and you wanted to assign pushbutton1's caption to it, the syntax would be:

```
x=pushbutton1.caption
```

And just as in setting properties, you can get the property of a control in another window by including the window's name. For example, if you want to assign the variable x to window2's pushbutton1 caption property, you would use this syntax:

```
x=window2.pushbutton1.caption
```

And just like setting properties, if you include only the property name, REALbasic assumes you are referring to a property of the window that contains the control that is executing the code. For example, if you have a pushbutton called pushbutton1 and you want it to assign the window title to the variable x when it is clicked, you would use this syntax:

```
x=title
```

Getting and Setting Values in Variables

When you need to store a value that is not associated with an object (the way a property is associated with a control or window), you use a variable. A variable is nothing more than a location in memory that stores a value. Variables have names just like properties do. The name you give a variable should describe the purpose of the variable. Suppose you want to calculate the age of a person in days from the year he was born. You might have a variable called " Days" to keep track of that information. Variable names can be any length but must begin with a letter and can contain only alphanumeric characters (A-Z, a-z, 0-9). Variable names are case-insensitive so REALbasic sees x and X as the same variable.

You can put values in variables and get values from variables in the same way you do with properties. To get a value from a variable, it must be on the right side of the assignment operator (=). Say for example, you wanted to set the caption of a

pushbutton to the value in a variable called “buttonTitle”. The example below accomplishes that:

```
Pushbutton1.Caption=buttonTitle
```

Conversely, if you wanted to store the value of a property (like the pushbutton’s caption in the last example) in a variable, you would simply reverse the syntax:

```
buttontitle=pushButton1.Caption
```

Like properties, variables have data types. Before you can use a variable, its data type must be made known using the Dim statement. Dim is short for Dimension which means to make space for the variable. In the example below, the variable i is dimensioned (or dimed) as an integer:

```
Dim i as integer
```

If you have several variables of the same type, you can declare them all with one Dim statement:

```
Dim i,j,k as integer
```

You already know about the data types Integer, String, Boolean, Single and Double. But variables can also be declared as specific object types. For example, REALbasic has an object type called a *FolderItem*. A FolderItem can represent any item that can exist in a folder on the desktop (file, application, or another folder). To store a FolderItem object, you must first declare a variable of type FolderItem as in this example:

```
Dim f as FolderItem
```

In this case, f is now an object with properties. One of the properties of a FolderItem is its name which is the name of the file, application, or folder that the FolderItem represents. The variable f's name property could then be assigned to say, variable n like this:

```
n=f.name
```

The Dim statement creates the variable but when does the variable get erased from memory? You will find out the answer to that question in the next chapter.

Just like properties, you can only assign values to variables that are compatible with the variable's data type. The last line of the following example generates an error because the types don't match:

```
Dim x,z as integer
Dim y as string
x=1
y="Hello"
z=x+y
```

In the example above x is a number and y is a string. An error is generated because you can't add different data types together.

Using Arrays

The Dim statement also lets you create and type arrays. An array is simply a variable that can contain multiple values of the same data type. You create an array by specifying the number of elements (values) of the array in parentheses. The number of values that you specify in the Dim statement is actually one less than the actual number of array elements because REALbasic

arrays always have an element zero. Such an array is sometimes referred to as a *zero-based array*. For example, the statement:

```
Dim aNames (10) as string
```

creates a string array with eleven elements.

You can create multi-dimensional arrays in REALbasic. You do so by indicating the size of each dimension. For example, the statement:

```
Dim aNames (2,10) as string
```

creates a two-dimensional array with 3 rows and 11 columns.

You refer to an element of an array by placing the desired element in parentheses. For example, the statement:

```
aNames(1,1)="Frank"
```

places the string "Frank" in array element (1,1)

A one-dimensional array may be passed as a parameter in a call to a subroutine or function. To specify that a parameter is an array, put empty parentheses after its name in the declaration. For example,

```
Names () as String
```

can be used in the declaration when you want to pass an array of strings to the subroutine. Since you do not need to specify the number of elements in the array to be passed, you can pass a different number of elements at different places in your code.

When you pass an array to the subroutine, omit the parentheses. For example, use

```
PrintLabels (Names)
```

where PrintLabels is the name of the method that accepts the string array as its parameter.

Mathematical Operators

Performing mathematical calculations is a very common task in programming. REALbasic supports all of the common mathematical operations.

Operation Performed	Operator	Example
Addition	+	2 + 3 = 5
Subtraction	-	3 - 2 = 1
Multiplication	*	3 * 2 = 6
Floating Point Division	/	6 / 4 = 1.5
Integer Division	\	6 \ 4 = 1
Modulo	Mod	6 Mod 3 = 0 6 Mod 4 = 2

There are also many built-in mathematical functions. See the Language Reference for more information.

REALbasic supports standard mathematical precedence. This means that equations surrounded by parenthesis are handled first. REALbasic will begin with the set of parenthesis that is embedded inside the most other sets of parenthesis. Next any multiplication or division from left to right is performed. Finally any addition or subtraction is performed. In the example below,

the three expressions return different results because of the placement of parentheses:

Expression	Result
2+3*(5+3)	26
(2+3)*(5+3)	40
2+(3*5)+3	20

Constants

You can create constants in REALbasic at either the local or global level. A local constant is assigned its value within a method and can be referred to anywhere within that method. A global constant can be created only within a module and can be referred to anywhere in your application. Global constants are described in the section "Adding Constants to Modules" on page 221.

Global constants can make it easier to maintain your code because you can adopt the convention of defining your constants at one central place in the application. Whenever you need to modify a constant, you know where to find its definition and you can be sure that the change will take effect throughout the application.

Global constants in REALbasic also provide a very handy way to manage multiple language versions of your application. This feature is discussed in the section "Using Constants to Localize Your Application" on page 222.

To define a local constant, use the keyword **Const** within a method, followed by an assignment statement. That is,

```
Const <constname> = <value>
```

You do not have to type the constant using a Dim statement. For example, the following code is acceptable:

```
Const Accept="OK"
bevelbutton1.caption=Accept
```

This code sets the caption of bevelbutton1 to “OK”.

Reserved Words

The following words should not be used as variable names because they are used as part of the REALbasic language itself:

TABLE 36. **Reserved Words**

And	Mod
Array	New
As	Next
Boolean	Nil
Case	Not
Color	Of
Dim	Or
Do	Raise
Double	Redim
Downto	Rem
Else	Return
ElseIf	Select
End	Self
Exception	Single
Exit	Step
False	String
For	Sub
Function	Then
GoTo	To
If	True
Integer	Until
Isa	Var

TABLE 36. **Reserved Words (Continued)**

Loop	Wend
Me	While

Executing Instructions with Methods

A *method* is one or more instructions that are performed to accomplish a specific task. REALbasic has many built-in methods. For example, the Quit method causes your application to exit to the Finder. Some object types (classes) have built-in methods. For example, the ListBox class has a method called AddRow for adding rows to a ListBox (as the name implies). You can also create your own custom methods. Just like variables, methods are given names to describe them and the same rules apply: the name can be any length, but must start with a letter and can contain only alphanumeric values (a-z, A-Z, 0-9).

You can also write your own methods and use them in your code. The following is an example of a simple method that calculates how many days old a person is in 1998 who was born in 1960:

```
Dim yearBorn, thisYear, daysOld as Integer
yearBorn=1960
thisYear=1998
daysOld=(thisYear-yearBorn)*365
```

Methods can, of course, be far more complex and longer than this example. There are three different places you can put your code. You will learn about these in the next chapter.

Documenting (Commenting) Your Code

Documenting your code is important because while it might make sense at the time you write it, it may not make sense days or weeks later. Also, if someone else has to understand your methods, documentation will make their job a whole lot easier. Comments can be added to your code as separate lines or to the right of any code on an existing line. Comments are ignored by REALbasic when it runs your application and have no impact on performance. In order for the REALbasic compiler to ignore your comments, you must start the comment with a single-quote (‘), two forward slashes (//) or the word REM (short for reminder). The example below shows how the previous example could be commented:

```
//Create the necessary variables
Dim yearBorn, thisYear, daysOld as Integer
yearBorn=1960  //set the year they were born
thisYear=1998  //store the current year
//Now calculate the number of days old
daysOld=(thisYear-yearBorn)*365
```

Comments in your code appear in red automatically.

If you have several consecutive lines that you want to convert to comments, highlight the lines and press Command-' (up-down quote). You also use this keystroke to convert the lines of comments back to executable code. This technique is especially useful if you want to temporarily convert several lines of programming statements to comments for testing purposes.

Passing Values to Methods

Some of REALbasic’s built-in methods require one or more pieces of information to perform their function. These pieces of infor-

mation are called *parameters*. Parameters are passed to a method placing them to the right of the method name in your code. In the example below, the AddRow method of a ListBox called ListBox1 is being called. AddRow requires one parameter which is the text that should be displayed in the new row:

```
ListBox1.AddRow "January"
```

If a method requires more than one parameter, commas are used to separate them. The ListBox class has a method called InsertRow which is used to insert new rows into a ListBox at any position. The InsertRow method requires two values: the row number where the new row should appear and the text value that should be displayed in the new row. Because more than one parameter is required, the parameters are separated by commas:

```
ListBox1.InsertRow 3, "January"
```

Parameters can also be variables. If a variable is passed as a parameter, it is the current value of the variable that is passed. In the example below, a variable is assigned a value and then passed as a parameter:

```
Month="January"
ListBox1.InsertRow 3, Month
```

In the next chapter, you will learn how to define parameters for your own custom methods.

Returning Values from Methods

Some methods return values. This means that a value is passed back from the method to the line of code that called the method. For example, REALbasic's built-in method, Ticks, returns the number of ticks (60th's of a second) that have passed since you

turned on your computer. You can assign the value returned by a method the same way you assign a value. In the example below, the value returned by Ticks is assigned to the variable x:

```
x=Ticks
```

Some methods require parameters and return a value. For example, the Chr function returns the character whose ASCII code is passed to it. When you pass parameters to a method that returns a value, the parameters must be enclosed in parenthesis. In the example below, the Chr function is passed 13 (the ASCII code for a carriage return) and returns a carriage return to the variable x:

```
x=Chr(13)
```

The parentheses are required because the value returned might be passed as a parameter to yet another method. Without the parentheses, it would be difficult to distinguish which parameters were being passed to which method. In the example below, the numeric value returned by the Len function (which returns the number of characters in the string passed to it) is then passed to the Str function (which converts a numeric value to a string). The string returned by the Str function is then passed as a parameter to the InsertRow method of a ListBox:

```
ListBox1.InsertRow 3, Str(Len("Hello"))
```

Methods that return a value are referred to as *functions*. In the REALbasic Language Reference, the names of methods that return a value are followed by the word *function*. In the next chapter, you will learn how to return values from your own custom functions.

Passing Parameters by Value and by Reference

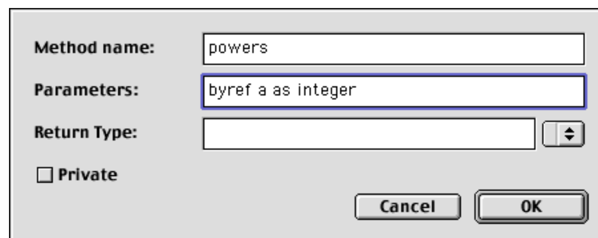
By default, you pass values to a method by value. When you do so, the method receives a copy of the data in the object that you pass. Your method receives the data and can perform operations on it.

When you write your own methods, you have the option of passing information by reference. When you pass information by reference, you actually pass a pointer to the object containing the information. The practical advantage of this technique is that the method can *change the values* of each parameter. When you pass parameters by value, you can't do this because the parameter only represents a copy of the data itself.

Passing parameters by reference is especially valuable when your method must return several values. When you pass parameters by value, the method can return only one value. You do this by making the method a function and obtaining the value as the result of the function.

You use the keywords ByVal or ByRef to specify the type of parameter passing. To pass a parameter by reference, use the ByRef keyword in the method declaration. For example, Figure 64 on page 146 shows a parameter that is declared ByRef. The method can then replace the parameter with computed values.

FIGURE 64. **Declaring a parameter ByRef**



Suppose the code that calls this method is:

```
dim a as integer
a=3
powers a
editField1.text=str(a)
```

and the method is simply:

```
a=a*a
```

The EditField will display the number 9.

When you want to use parameter passing by value, you do *not* need to use the ByVal keyword explicitly. Parameter passing by value is the default and is used unless overridden by use of ByRef.

Comparison Operators

There are many times when you need to compare two values to determine whether or not a particular condition exists. When making a comparison, what you are really doing is making a statement that will either be True or False. For example, the statement "My dog is a cat" evaluates to False. However, the statement "My dog weighs more than my cat" may evaluate to True. The table below shows examples of the comparison operators that are available:

Description	Symbol	Numeric Example	Evaluates To
Equality	=	5=5	True
Inequality	<>	5<>5	False
Greater Than	>	6>5	True

Description	Symbol	Numeric Example	Evaluates To
Less Than	<	6<5	False
Greater Than or Equal To	>=	6>=5	True
Less Than or Equal To	<=	6<=5	False

String and boolean values can also be used for comparisons. String comparisons are case insensitive and alphabetical. This means that “Jeannie” and “jeannie” are equal. But “Jason” is less than “Jeannie” because “Jason” falls alphabetically before “Jeannie”. If you need to make case sensitive or lexicographic comparisons, see “StrComp Function” on page 406 of the Language Reference.

Testing Multiple Comparisons

You can test more than one comparison at a time using the And and Or operators.

And Operator

Use this operator when you need to know if all comparisons evaluate to True. In the example below, if the variable x is 5 then the expression evaluates to False:

```
x>1 And x<5
```

Or Operator

Use this operator when you need to know if any of the comparisons evaluate to True. In the example below, if the variable x is 5 then the expression evaluates to True:

```
x>1 Or x<5
```

Executing Instructions Repeatedly with Loops

There may be times when one or more lines of code need to be executed more than once. If you know how many times the code should execute, you could simply repeat the code that many times. For example, if you wanted a pushbutton to beep three times when clicked, you could simply put the Beep method in your code three times like this:

```
Beep
Beep
Beep
```

But say you need it to beep fifty times or perhaps until a certain condition is met? Simply repeating the code over and over in these cases will either be just tedious or not possible. How do you solve this problem? The answer is a *loop*.

Loops execute one or more lines of code over and over again.

While...Wend

A While loop executes one or more lines of code between the While and the Wend (While End) statements. The code between these statements is executed repeatedly, provided that the condition passed to the While statement continues to evaluate to True. Consider the following example:

```
Dim n As Integer
While n<10
   n=n+1
   Beep
Wend
```

The variable "n" will be zero by default when it is created by the Dim statement. Because zero is less than ten, execution will move inside the While...Wend loop. The variable n is incremented by one. The Beep method plays the alert sound. REALbasic checks to see if the condition is still True and if it is, then the code inside the loop executes again. This continues until the condition is no longer True. If the variable n was not less than ten in the first place, execution would continue at the line of code after the Wend statement.

Do...Loop

Do loops are similar to While loops but a bit more flexible. Do loops continue to execute all lines of code between the Do and Loop statements *until* a particular condition is True. While loops on the other hand execute as long as the condition remains True. Do loops provide more flexibility than While loops because they allow you to test the condition at the beginning or end of the loop. The example below shows two loops; one testing the condition at the beginning and the other testing it at the end:

```
Do Until n=10
  n=n+1
  Beep
Loop

Do
  n=n+1
  Beep
Loop Until n=10
```

The difference between these two loops is this. In the first case, the loop will not execute if the variable n is already equal to ten. The second loop executes at least once regardless of the value of n because the condition is not tested until the end of the loop.

It is possible to create a Do loop that does not test for any condition. Consider this loop:

```
Do
  n=n+1
  Beep
Loop
```

Because there is no test, this loop will run endlessly. You can call the Exit method to force a loop to exit without testing for a condition. However, this is poor design because you have to read through the code to figure out what will cause the loop to end.

Endless Loops

Make sure that the code inside your While and Do loops eventually causes the condition to be satisfied. Otherwise, you will end up with an endless loop that runs forever. Should you do this accidently, you can attempt to switch back to the Design environment by clicking on one of the Design environment's windows. Then you can choose Debug ▶ Kill (⌘-K) to stop the loop. If this doesn't work, you will need to force REALbasic to quit by pressing ⌘-Option-Escape.

Lengthy Loops

When a loop starts running, its process 'takes over' and doesn't allow the user to interact with interface elements such as menus, buttons, and scroll bars. On modern computers and reasonably short loops, this isn't a problem because the loop executes faster than the user can think of another button to push or menu item to select. If this is not true, there are a couple of things you can do:

- If the user should wait until the loop is finished before doing anything else (e.g., if a user action might invalidate the results

of the loop), you can signal that a lengthy operation is in progress by changing the mouse cursor to a watch cursor until the loop ends. Keep in mind that, with this solution, the user can't even stop the loop prematurely because the loop has 'taken over' the application.
See the section on the MouseCursor class in the Language Reference for more information.

- If the user is permitted to do other tasks while the loop is running, you should place the code for the loop in a separate thread. A thread runs as a background task, allowing user operations in the foreground.

For...Next

While and Do loops are perfect when the number of times the loop should execute cannot be determined because it's based on a condition. A For loop is for cases in which you can determine the number of times to execute the loop. For example, suppose you want to add the numbers one through ten to a ListBox. Since you know exactly how many times the code should execute, a For loop is the right choice. For loops also differ from While and Do loops because For loops have a loop counter variable, a starting value for that variable and an ending value. The basic construction of a For loop is:

```
Dim counter,startingvalue,endingvalue As Integer
.
.
For counter=startingValue to endingValue
   [your code goes here]
Next
```

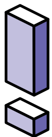
Notice the Dim statement is declaring counter as an Integer. This is because the counter variable in a For loop must be an integer.

The first time through the loop, the counter variable will be set to startingValue. When the loop reaches the Next statement, the counter variable will be incremented by one. When the Next statement is reached and the counter variable is equal to endingValue, the counter will be incremented and the loop will end.

Let's take a look at the example mentioned earlier. You want to add the numbers one through ten to a ListBox. The following example accomplishes that:

```
Dim i As Integer
For i=1 to 10
  ListBox1.AddRow Str(i)
Next
```

The counter variable (i in this case) is passed to the Str function to be converted to a string so that it can be passed to the AddRow method of ListBox1.



Note: The letter "i" is commonly used as the loop counter for historical reasons. In FORTRAN, the letters I to N are typed as integers by default. Therefore, FORTRAN programmers began the practice of using those letters as counters, and in the order they appear in the alphabet. That is, if a FORTRAN programmer needed to nest one loop in another (as is described on page 155), he would use j as the counter for the inner loop. This convention made it easy for FORTRAN programmers to follow the logic of code that processed multi-dimensional arrays.

By default, For loops increment the counter by one. You can specify another increment value using the *Step* statement. In this example, the Step statement is added to increment the counter variable by 5 instead of 1:

```
Dim i As Integer
For i=5 to 100 Step 5
   ListBox1.AddRow Str(i)
```

```
Next
```

In this example, the For loop starts the counter at 100 and decrements by 5:

```
Dim i As Integer
For i=100 DownTo 1 Step 5
 ListBox1.AddRow Str(i)
Next
```

So far, we have looked at cases where *StartingValue* and *EndingValue* are integer numbers. If either *StartingValue* or *EndingValue* are expressions that must be evaluated to integers, the For loop will perform the evaluation each time it increments the counter — even if the expression always evaluates to the same integer.

Therefore, it is advisable to perform any evaluations before entering the loop. For example, consider a loop that needs to process all the fonts that are installed on the user's computer. This number cannot be known in advance but there is a built-in function in REALbasic, FontCount, that you can use to obtain the total number of fonts. If you use it in the For statement to compute *EndingValue* (like so):

```
For i=0 to FontCount-1
.
.
Next
```

The loop will run more slowly than if you calculate the value only once:

```
Dim nFonts as integer
nFonts=FontCount-1
For i=0 to nFonts
```

```
.
.
Next
```

However, the difference in speed may be of no practical value unless it is a very lengthy loop. On the Macintosh used to write this manual, the number of installed fonts is 120 and the difference in speed between these two loops is approximately 1/250 of a second—not enough to lose sleep over.

A For loop (as well as any other kind of loop) can have another loop inside it. In the case of a For loop, the only thing you will have to watch out for is making sure that the counter variables are different so that the loops won't confuse each other. The example below uses a For loop embedded inside another For loop to go through all the cells of a multi-column ListBox counting the number of items the word "Hello" appears:

```
Dim row, column, count As Integer
For row=0 to listBox1.ListCount-1
  For column=0 to listBox1.ColumnCount-1
    if listbox1.cell(row,column)="hello" then
    count=count+1
    End if
  Next
Next
MsgBox Str(count)
```

Another way to keep this straight is to use the naming convention started by FORTRAN programmers of using the letters of the alphabet beginning with "i" as the counters. In that way, you'll always know which loop in inside another loop without trying to figure out what the loops are supposed to be doing.

For loops are generally more efficient than Do and While loops because the compiled code generated is more efficient.

Making Decisions with Branching

The methods you write execute one line at a time from top to bottom, left to right. There will be times when you want your application to execute some of its code based on certain conditions. When your application's logic needs to make decisions it's called *branching*. This allows you to control what code gets executed and when. REALbasic provides two branching statements: If…Then and Select…Case.

If…Then…End If

The If…Then statement is used when your code needs to test a single boolean (True or False) condition and then execute code based on that condition. If the condition you are testing is True, then the lines of code you place between the If...Then line and the End If line are executed.

```
If condition Then
 [Your code goes here]
End If
```

Say you want to test the integer variable month and if its value is 1, execute some code:

```
If month=1 Then
 [Your code goes here]
End If
```

month=1 is a boolean expression; it's either True or False. The variable month is either 1 or it's not 1.

Suppose you have a pushbutton that performs an additional task if a particular checkbox is checked. The value property of a checkbox is boolean so you can test it in an If statement easily:

```
If checkbox1.value Then
 [Your code goes here]
End If
```

If...Then...Else...End If

In some cases, you need to perform one action if the boolean condition is True and another if it is False. In these cases, you can use the optional Else clause of an If statement. The Else clause allows you to divide the code to be executed into two sections: the code that is executed when the condition is True and the code that is executed when it's False. In this example, one message is displayed if the condition is True while another is displayed if it's False:

```
If month=1 Then
  MsgBox "It’s January."
Else
  MsgBox "It’s not January."
End If
```

If...Then...ElseIf...End If

In some cases, you need to perform an additional test when the initial condition is False. Use the optional ElseIf statement. In the example below, if the variable month is not 1, then the ElseIf statement performs an additional test:

```
If month=1 Then
  MsgBox "It's January."
ElseIf month<4 Then
  MsgBox "It's still Winter."
End If
```

You could, of course, use an additional If…Then…EndIf statement inside the Else portion of the first If statement to perform another test. However, this adds another EndIf and needlessly complicates your code. You can use as many ElseIf statements as you need.

In this example, another ElseIf has been added to perform an additional test:

```
If month=1 Then
  MsgBox "It's January."
ElseIf month<4 Then
  MsgBox "It's still Winter."
ElseIf month<6 Then
  MsgBox "It must be Spring."
End If
```

If the initial condition is False, REALbasic continues to test the ElseIf conditions until it finds one that is True. It then executes the code associated with that ElseIf statement and continues executing the lines of code that follow the End If statement.

Select...Case

When you need to test a property or variable for one of many possible values and then take action based on that value, use a Select…Case statement. Consider the following example that tests a variable (dayNumber) and displays a message to the user to tell him which day of the week it is:

```
If dayNumber=2 Then
  MsgBox "It’s Monday."
ElseIf dayNumber=3 Then
  MsgBox "It’s Tuesday."
ElseIf dayNumber=4 Then
  MsgBox "It’s Wednesday."
ElseIf dayNumber=5 Then
  MsgBox "It’s Thursday."
ElseIf dayNumber=6 Then
  MsgBox "It’s Friday."
Else
  MsgBox "It’s the weekend."
End If
```

No two of these conditions can be True at the same time. While this method of writing the code works, it’s not that easy to read. In this example, the same code is presented in a Select...Case statement, making it far easier to read:

```
Select Case dayNumber
Case 2
  MsgBox "It’s Monday."
Case 3
  MsgBox "It’s Tuesday."
Case 4
  MsgBox "It’s Wednesday."
Case 5
  MsgBox "It’s Thursday."
Case 6
  MsgBox "It’s Friday."
Else
  MsgBox "It’s the weekend."
End Select
```

The Select…Case statement compares the variable or property passed in the first line to each case value. Once a match is found, the code between that case and the next is executed. Select…Case statements can contain an Else statement to handle all other values not explicitly handled by a case.

The Select…Case statement supports string and integer comparisons only. If you need to compare boolean, single or double values, or if you need to use a comparison operator other than the equality operator (=), use an If statement.

CHAPTER 5

Programming with Events and Objects

Most of your code will execute in response to something the user does, such as selecting a menu item, clicking on a button, or typing in an EditField. This kind of programming is called *event-driven programming* because events cause the programming code to execute. Understanding how events work and which user actions cause which events to occur will take you a long way towards getting your application to do what you want it to do.

In this chapter you will learn about event-driven programming, how to use the Code Editor, and how to get your application to respond when the user clicks on interface objects or types on the keyboard.

Contents

- Understanding Event-Driven Programming
- Using the Code Editor
- Printing and Exporting Your Code
- Responding to User Actions with Event Handlers

Understanding Event-Driven Programming

Your users will interact with your applications by clicking the mouse and typing on the keyboard. Each time the user clicks the mouse on a part of your application's interface or types something in an EditField, an event occurs. The event is simply the action the user took (the mouse click or the key press) and where it took place (on this button, on that menu item or in this EditField). Some events can indirectly cause other events. For example, when the user selects a menu item (causing an event) that opens a window, it causes another event — the opening of the window).

Each object you create in REALbasic can include, as part of itself, the code you write that executes in response to the various events that can occur for that type of object. For example, a PushButton can include the code you wish to execute when the PushButton is pushed. An object can even respond to events you might not have thought it could — such as responding as the user moves the pointer over a button. When the user causes an event, REALbasic checks to see if the object the event was directed towards has any code that needs to execute in response

to that event. If the object has code for the event, REALbasic executes that code and then waits for the user to cause another event to occur. This continues until something causes the application to quit (usually the user's choosing Quit from the File menu).

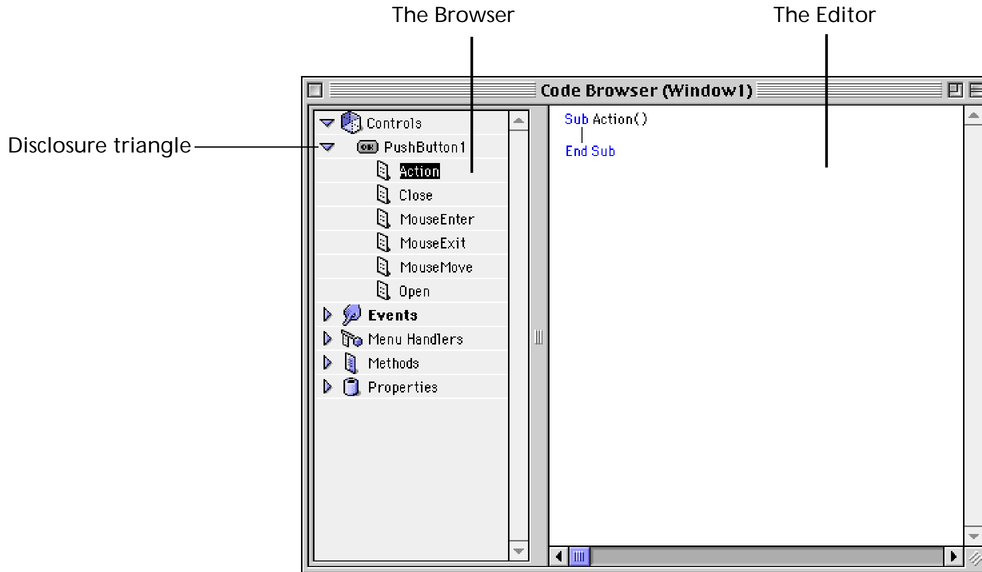
As mentioned earlier, the user can also indirectly cause events to occur. Buttons, for example, have an event called Action which occurs when the user clicks the button. The code that handles the response to an event is called (appropriately enough) an *event handler*. Suppose the button's Action event handler has code that opens another window. When the user clicks the button, the Action event handler opens a window and REALbasic sends an Open event to the window. This is not an event the user caused directly. The user caused this event indirectly by clicking the button whose code opened the new window.

There are many events that can occur to each object in your application. The good news is that you don't have to learn about all of them. You simply need to know where to look for them so that, if you want to respond to an event, you can find out if the object is able to respond to that event. Later in this chapter, you will learn about many of the common events you will need to be aware of in order to create your applications.

Using The Code Editor

The Code Editor is used to enter the code for the various events that can occur for the objects that make up your application's interface. It's also used to add properties and methods to objects. The Code Editor has two sections: the Browser and the Editor itself.

FIGURE 65. **The Code Editor**



The Browser is a hierarchical list of the programming-related components that make up a particular window. The Browser lists the window's:

- Controls
- Events
- Menu Handlers
- Methods
- Properties

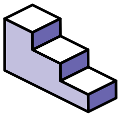
You will learn more about each of these items later in this chapter.

Opening The Code Editor

The Code Editor is used to edit the code for controls, windows, classes, and modules. You will learn more about classes and modules in later chapters. There are two ways to open the Code Editor for a specific window.

To open the Code Editor when the window is already open in the interface builder, double-click anywhere in the window (but not on a control) or press Option-Tab.

To open the Code Editor from the Project window without opening the window in the interface builder, click on the window whose code you wish to view then press ⌘-Tab.



To open the Code Editor for a specific control, do this:

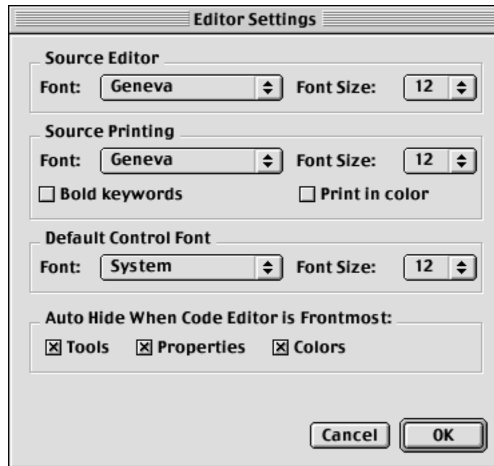
1. Open the window in the interface builder that contains the control.
2. Double-click on the control or single-click and then press the Return key.

This will open the Code Editor for the control's parent window. REALbasic will then automatically expand the control's category, expand the control you double-clicked on, and select either the default event handler (e.g., the Action event handler for a pushbutton) or the first event handler for the control.

Configuring the Code Editor

You can specify various preferences for the Code Editor. Choose Edit ▶ Editor Settings to display the Editor Settings dialog box, shown in Figure 66 on page 166.

FIGURE 66. The Editor Settings Dialog box



You can specify the font and font size separately for screen display and printing. For printing, you can elect to retain bold keywords and the colors used on-screen. The default Control Font is used as the default text font for controls that use text, such as PushButtons and Tab panels. By default, REALbasic hides the Tools, Properties, and Colors windows when the Code Editor window is active. You can selectively turn this set of options off.

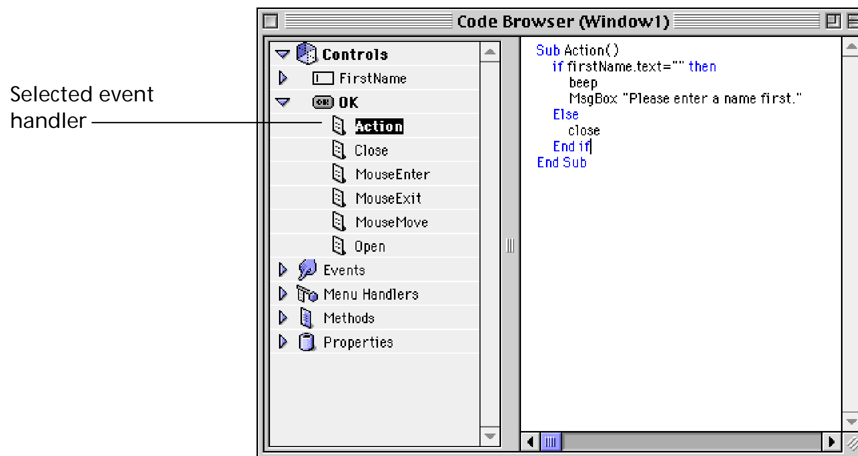
The Browser

To view the items in each category, click the disclosure triangle to the left of the category name. For example, to view all of the controls for the window, click the disclosure triangle next to the Control's category name. When you do this, the list of controls will appear below and to the right. Each of controls can then be expanded in the same way to display a list of the event handlers for that control. For example, in Figure 65 on page 164 you can see that Window1 has a pushbutton named PushButton1. You can also see that pushbuttons have the following event handlers:

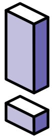
- Action
- Close
- MouseEnter
- MouseExit
- MouseMove
- Open

You will learn more about these event handlers later in this chapter. Clicking on a control's event handler in the Browser list displays the code associated with that event handler in the Code Editor.

FIGURE 67. **Some code associated with a control's event handler**



Items in the Browser that have code associated with them appear in bold. For example, if one of a control's event handlers has code in it, the event handler's name, the control's name, and the Controls category will all appear in bold. When you are trying to find some code, the bold style acts as a visual cue to let you know if there is any code you might need to look at.



Note: When you add new controls to a window, REALbasic gives them default names. For example, the first pushbutton you add to a window will be named "PushButton1" by default. A name like that describes the type of object but not what it does. Fortunately, the Browser displays icons next to each control to make the control type clear. This allows you to give the controls names that describe their function rather than their type. Figure 67 on page 167 shows an example of this. The pushbutton is named "OK" rather than "pushbutton1" and the EditField is named "FirstName" instead of "EditField1."

Understanding Methods in the Code Editor

Event handlers, menu handlers, and methods are all, in fact, methods. Event and menu handlers are simply methods that are called when certain events occur or menu items are selected. When you select a method, its code appears in the Editor. Methods are made up of three parts: The parameter line, your lines of code, and the End method line.

FIGURE 68. **The three parts of a method**

Parameter line

```
Sub Action()
  if firstName.text="" then
    beep
    MsgBox "Please enter a name first."
  Else
    close
  End if
End Sub
```

Your Code

End Method

The Parameter Line

The parameter line displays Sub (short for subroutine) if the method does not return any values, followed by the name of the method or event handler, and then any parameters surrounded by parens. The example in Figure 69 on page 169 shows the MouseMove event handler. This event handler is called anytime

the mouse is moved inside the control. It is passed two parameters that can be used to determine the current mouse location.

FIGURE 69. The parts of the parameter line

For more information on parameter passing, see "Passing Values to Methods" on page 143 of chapter 4.

If the method returns a value, it's called a *function*. A function's parameter line begins with *Function* instead of *Sub* and has an additional parameter; the parameter for the value that will be returned by the function. The declaration of the value returned by the function follows the parameters. Figure 70 shows the parameter line for an EditField's KeyDown event handler. This event handler is called when the user types a key in an EditField. It is passed the key that was pressed in the parameter *key*. The value returned is a boolean. If you return True from the function, the event is discarded as if it never happened at all and the key that was pressed will not appear in the EditField.

FIGURE 70. The parameter line of a function

For more information on functions, see "Returning Values from Methods" on page 144 of chapter 4.

Entering Your Code in the Code Editor

As you enter your code, REALbasic does a few things for you automatically. First, it indents your If…Then, Select…Case, and loops as you type them to make it easier to see which lines of code fall inside a particular statement. Figure 68 on page 168 shows an example of this indentation.

As you type, REALbasic also attempts to guess what you are typing and makes a suggestion to complete the typing for you. Suppose you have a ListBox control called "Listbox1." As you type the first few characters of the control's name in the Code Editor, REALbasic will guess you mean ListBox1. It will then display the rest of the name in light grey. Figure 71 shows an example of this. If you want REALbasic to complete the entry for you, simply press the Tab key. If REALbasic has guessed incorrectly, simply continue typing the rest of the name.

FIGURE 71. REALbasic's auto-code completion feature in action

Before	List|Box1
After	ListBox1

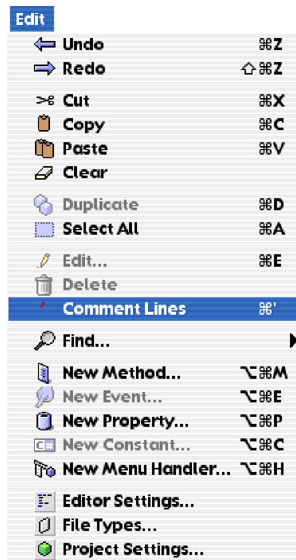
Auto-code completion also works for method and property names.

Using the Edit Menu

While you are entering code, the standard Edit menu Cut, Copy, and Paste commands are available. The Code Editor also supports both Undo and Redo (Shift-⌘-Z). The Comment Lines command (⌘-') is especially useful. When applied to lines of code, it comments them out; when applied to comments, it converts them back to code.

Figure 72 shows the Edit menu items during code entry.

FIGURE 72. **The Edit Menu.**



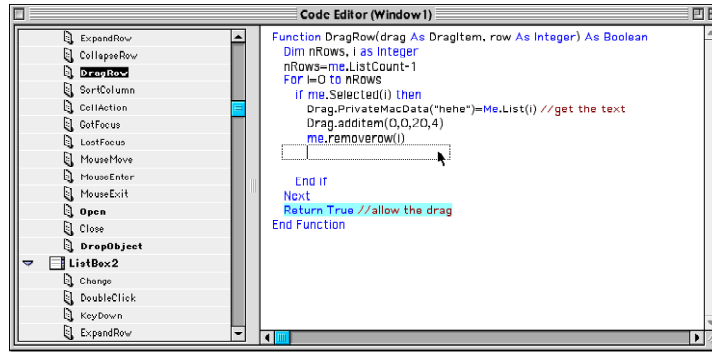
Using Drag and Drop

The Code Editor fully supports drag and drop. Within a method, you can:

- **Move text:** Highlight the text to be moved and drag it to the desired location, as indicated by the insertion point.
- **Copy text:** Highlight the text to be copied, hold down the Option key, and drag it to the desired location.

Figure 73 on page 172 shows a line of text being dropped:

FIGURE 73. **Dropping a line of code in a method.**

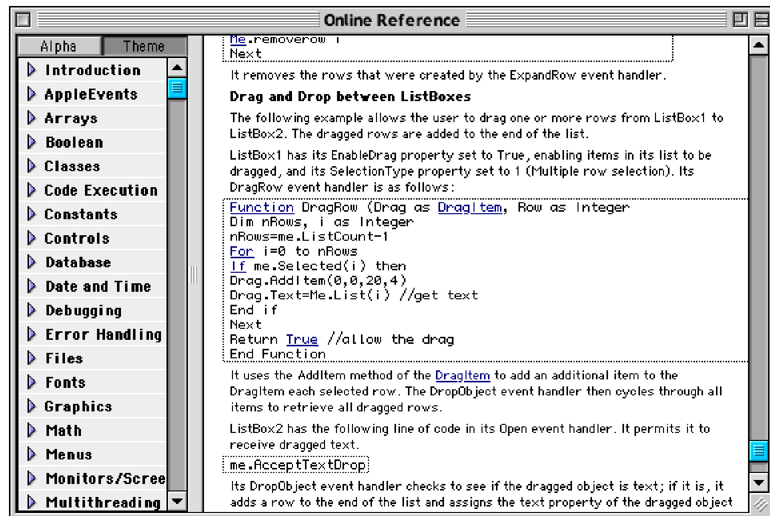


You can also drag and drop selected code to and from any word or text processor that supports drag and drop, such as BBEdit.

You can also drag a text clipping from the Finder into the Code Editor or drag selected text from the Code Editor to the Finder to create a text clipping.

The Online Reference (⌘-1) contains numerous examples of REALbasic code. Each example is enclosed in a dotted rectangle. Any code rectangle can be dragged and dropped into the Code Editor. However, you can’t select individual lines within a rectangle for drag and drop. Figure 74 on page 173 shows two draggable examples from the Online Reference.

FIGURE 74. **Draggable examples from the Online Reference**

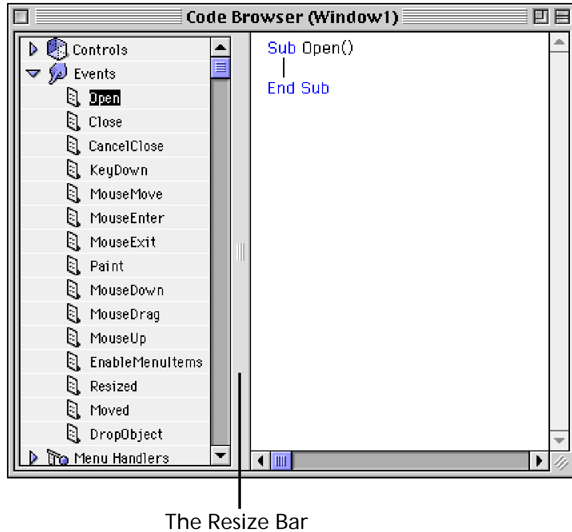


Getting More Usable Space in the Code Editor

There may be times when you need more vertical or horizontal space in the Code Editor. You can, of course, resize the Code Editor window to get more space, but this isn't always an option. One way to get more space is to use a smaller font. You can set the font and font size for the Code Editor by choosing Editor Settings from the Edit menu and selecting the Code Editor font and font size.

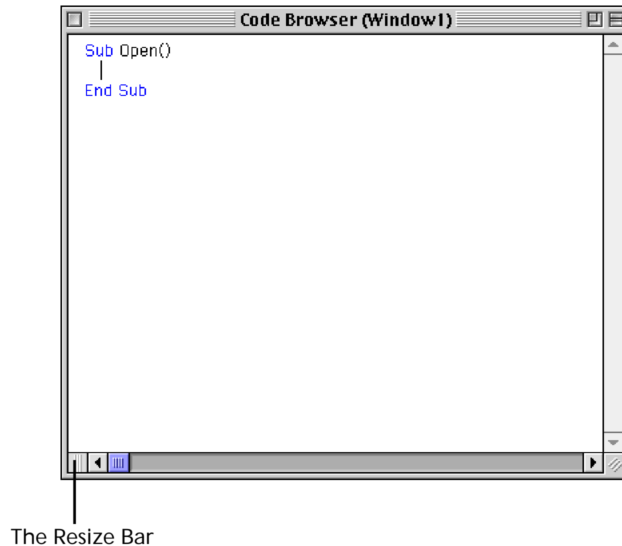
You can also hide the Browser when you don't need it. You can hide the Browser by dragging the resize bar (the space between the Browser and the Editor) all the way to the left side of the Code Editor window.

FIGURE 75. **The Code Editor's Resize Bar.**



When you do this, the Browser is hidden and the resize bar is reduced to a small square in the bottom left corner of the Code Editor.

FIGURE 76. **The Code Editor with the Browser hidden.**

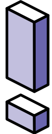


As you can see in Figure 76, this gives you quite a bit of horizontal space to work with in the Code Editor. You can show the Browser again by dragging the Resize Bar towards the right side of the Code Editor window.

If you prefer to use the keyboard, all of this dragging might seem tedious. The good news is there is a keyboard shortcut for hiding and showing the Browser. After you have hidden the Browser by dragging, press Shift-Tab to show the Browser. This will also place the focus on the Browser, allowing you to use the arrow keys to move between items. You can then hide the Browser again by pressing Shift-Tab.

You might notice that Shift-Tab doesn't appear to always hide and show the Browser. It will always work if you are using the keyboard to move between items in the Browser. If you use the mouse to click on an item in the Browser, Shift-Tab will simply

move the focus between the Browser and the Code Editor. The assumption here is that if you are using the mouse to select items in the Browser, you don't want the focus to move to the Browser when you click in it. If it did, you would then have to click in the Editor to give it the focus to continue typing your code.

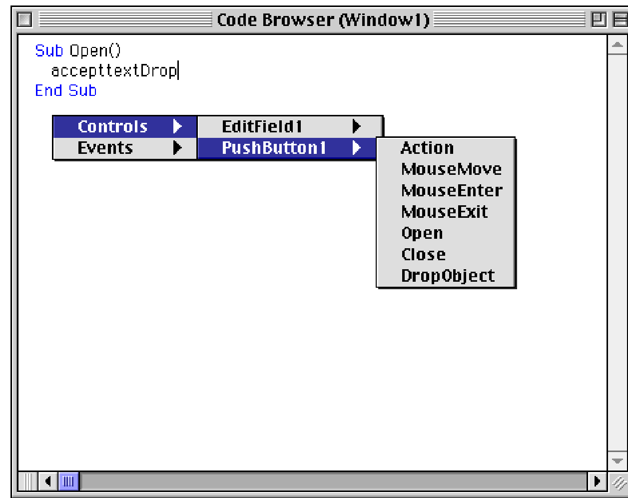


Note: The Browser will expand and collapse the categories (Controls, Events, Menu Handlers, Properties) when they are selected by pressing ⌘-Left Arrow (to collapse) and ⌘-Right Arrow (to expand) just like the Macintosh Finder.

Using Contextual Menus

Another way to access the items in the Browser is with *contextual menus*. Contextual menus are context-sensitive pop-up menus that appear when you Control-click on an interface item. Control-clicking in the Code Editor displays a contextual menu with all of the items from the Browser. This is especially handy when you have the Browser hidden to provide more horizontal space in the Code Editor. Although contextual menus were added in MacOS 8, REALbasic's contextual menus work with System 7 as well. Figure 77 on page 177 shows a contextual menu.

FIGURE 77. The Code Editor's contextual menus

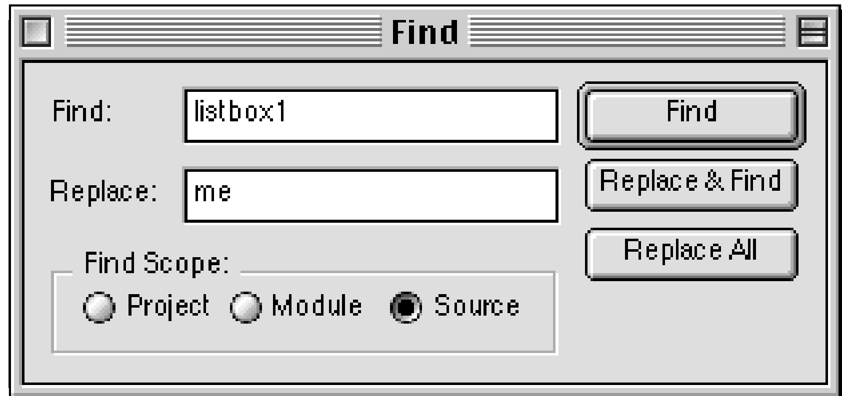


The contextual menu will only show categories that have items. For example, in Figure 77, Menu Handlers and Properties are not displayed because the window has no menu handlers or properties.

Find and Replace

Use the Find/Replace window (⌘-F) to find something in your code and perhaps replace it with something else. With this window you can find the next occurrence of the item you are searching for and then, optionally, replace it with something else.

FIGURE 78. The Find/Replace Window



You can also determine the scope of the find and replace. Table 37 describes the various scopes of find and replace.

TABLE 37. The Find/Replace Scope

Scope	Description
Source	Find and replace will affect only the currently displayed method.
Module	Find and replace will affect only the methods of the current window, module, or class.
Project	Find and replace will affect all code in the project.

The Find/Replace window's buttons give you the ability to find the next occurrence of the item you are searching for, replace the highlighted text in the Code Editor with the text in the Find

window's Replace field, and replace all occurrences within the chosen scope. The keyboard equivalents are shown in Table 38:

TABLE 38. **Find and Replace Keyboard Equivalents**

Command	Keyboard Equivalent
Find	⌘-F
Find Next	⌘-G
Replace and Find Next	⌘-L

Printing Your Code

When you need to print your source code, choose File ▶ Print (⌘-P). The Print Code dialog box lets you choose how much code you wish to print.

FIGURE 79. **The Print Code dialog box**

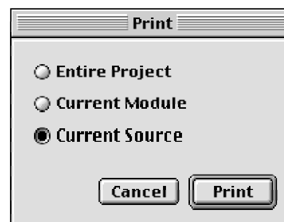


TABLE 39. **Print Code dialog box options**

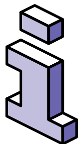
Option	Description
Current Source	Prints the currently displayed method.
Current Module	Prints all code for the currently displayed window, module, or class.
Entire Project	Prints all code for the entire project.

Importing and Exporting Your Classes, Menus, Modules, and Windows

REALbasic makes it easy to import and export the various objects you can create. You can also import files you wish to use in your project, such as REALbasic code, REALbasic windows, REALbasic menus, sounds, pictures, QuickTime movies, REAL databases, and resources.

Importing

To import a file you wish to use in your project, simply drag it from the desktop and drop it in your Project window. Or, if the file is not conveniently located on the desktop, choose File ▶ Import. An open-file dialog box appears, allowing you to navigate to and import the file.



To delete a file that has been added to the Project window, highlight it and press the Delete key on the keyboard or choose Edit ▶ Clear.

Some of the items you import are copied into your project. Some types of objects are not copied but instead an alias to the original file is stored inside your project. When you build a stand-alone version of your project, most of these files are then copied into the stand-alone application. Table 40 on page 181 shows how all of the different file types are handled.

TABLE 40. **How REALbasic handles imported files**

File Type	Copied Into Project?	Copied into stand alone applications?
Bitmap, PICT, JPEG, GIF	N	Y
Cursors	Y	Y
PowerPC Shared Libraries	N	N
QuickTime Movies	N	Y
REAL Databases	N	N
REALbasic Classes	Y	Y
REALbasic Menubars	Y	Y
REALbasic Modules	Y	Y
REALbasic Plug-ins	N	Y
REALbasic Windows	Y	Y
Resources	N	Y
Sounds	N	Y
XCMDs and XFCNs	N	Y

Because REALbasic stores aliases to your imported files, they can be renamed and even moved. If both the project file and the imported files are moved to another drive, REALbasic may have trouble locating the files. Should this happen, REALbasic will ask you to locate any files it can't find.

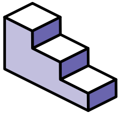
All file types, except PowerPC shared libraries and REAL databases are included in the stand-alone version of your application, so there is no need to include them with your application when you distribute it.

Exporting

The code for methods, events, constants, properties, and so forth can be dragged out of the Code Editor as text clippings. You can either select some code in the Code Editor and drag or select the

name of the object in the Browser and drag that object. In the latter case, all the code associated with the object will be included in the text clipping.

You can also export your source code to a text file or in REALbasic's native format using an Export… menu command. Which method you use depends on what you will be doing with the exported code. If you are going to be including code in some kind of documentation, drag the code to the other application or export your code to a text file. Choosing File ▶ Export Source will export all of the code in the project to a text file. This is the same as the drag and drop method.



If you want to export a window, module, class, or menu bar for use in another REALbasic project, do this:

1. Open the item so that it's displayed on the screen or select it in the Project window.
2. Choose File ▶ Export Window/Menu/Module/Class.
3. When the Save As dialog box appears, type a name and click the Save button.

Protecting Your Source Code

If you want to distribute a copy of a window, menu bar, module, or class for others to use but you do not want them to be able to view or edit your code, select the Protected option in the Save As dialog box when you export.

Responding To User Actions with Event Handlers

The applications you create with REALbasic are event-driven. This means that the user takes some action which results in something happening. For example, the user chooses Print from the File menu to print something or clicks a button to confirm a message in a dialog box. The user takes an action, and the application reacts to that action. The user's actions are called *events*. Earlier in this chapter, you learned that some events are caused directly by the user. For example, the Action event of a pushbutton occurs when the user clicks the pushbutton. Other events are indirectly caused by the user, such as the Open event of a window that occurs when the window opens.

The key to writing the code for your applications is to know what events (both direct and indirect) you can respond to.

Object-Oriented Programming

REALbasic's programming language is *object-oriented*. This means that the code that is executed in response to an event is actually part of the object itself. Code that handles an event is called (appropriately) an *event handler*.

Objects can also have their own methods. This allows you to associate code with an object even though it may not be executed in response to an event directed at that object. For example, suppose you have a window that displays the contents of a document and allows the user to edit it. It would make sense that the window would know how to save changes made to the document. You can add a method to the window that is called automatically when the user indicates that he wants to save changes to the document.

Because objects in your application are supposed to be just like objects in the real world, you want to associate code with the object that it truly belongs to. For example, if you want a window to change its size automatically when it opens based on certain conditions, it makes the most sense to put that code in the window's Open event handler. On the other hand, if you want a button to be enabled or disabled when the window opens that the button is a part of, you would put that code in the pushbutton's Open event handler because the code affects the button. The code works perfectly in both places, but it is more object-oriented to associate it with the pushbutton, since it affects the pushbutton. For example, in the real world, when the door to the room you are in suddenly opens, you probably turn to look at it to see why it opened. The door does not turn your head. You have that ability to react to the door opening (an event). You choose to handle that event by turning and looking in the direction of the door. That ability is part of you — just as the code to enable or disable the button when the window opens should be part of the button and not the window.

Another benefit of associating code with the appropriate object is that the code goes with the object when you use the object elsewhere. If the code is not associated with the object, you will have to look for it or rewrite it. When you go somewhere, you take your computer skills with you because they are part of you.

Windows

Events

Windows get many different events. Table 41 on page 185 describes these events in general. If you need specific information

about window events, see “Window Class” on page 446 of the Language Reference.

TABLE 41. Window events

Event	Description
Open	The window is about to open but hasn’t been displayed yet. Controls also receive Open events. A window receives its Open event after all of the controls have received their Open events.
Close	The window is about to close but hasn’t closed yet. Controls also receive Close events. A window receives its Close event after all of the controls have received their Close events.
CancelClose	The Quit method has been called so the application is about to quit. Returning True from this method will cancel the quit and the application will remain open.
Resized	The window has been resized by the user or by code that changes the window’s Width or Height properties.
Moved	The window has been moved by the user or by code that changes the window’s Left or Top properties.
Paint	Some portion of the window needs to be redrawn either because the window is opening or it’s been exposed when a window in front of it was moved or closed. This event handler receives a Graphics object as a parameter which represents the graphics that will be drawn in the window. Graphics objects have their own methods for drawing graphics. See “Graphics Class” on page 192 of the Language Reference for more information.
EnableMenuItems	While the window is front of all other windows, the user has clicked in the menu bar to select a menu item or pressed a menu item’s keyboard equivalent. This event handler gives you a place to decide which menu items should be enabled before the user can actually choose one.
DropObject	A file, piece of text, or a picture has been dropped on the window itself (not on a control in the window). This event handler is passed a parameter that gives you access to the item dropped.

TABLE 41. Window events (Continued)

Event	Description
KeyDown	A key has been pressed that has to be handled by the window. For example, the tab key is never sent to any control. It is instead handled by the window itself. If the window has no controls that can receive the focus, any keys that are pressed will generate KeyDown events for the window. This event handler is passed a parameter that tells you which key was pressed.
MouseDown	The mouse button has been pressed and has not yet been released. You can return False in this event handler to filter the event causing the window to act as if the mouse button was never clicked. This event handler receives parameters that indicate where the mouse was clicked in local window coordinates.
MouseUp	The mouse button has been released inside the window. This event will not occur unless you return True in the MouseDown event handler. The idea behind this is that if the mouse was never down, it can't be up. This event handler receives parameters that indicate where the mouse was released in local window coordinates.
MouseDrag	The user has moved the mouse inside the window (but not over a control) while the mouse button is held down. This event handler receives parameters that indicate where the mouse is in local window coordinates.
MouseMove	The user has moved the mouse inside the window. This event handler receives parameters that indicate where the mouse is in local window coordinates.
MouseEnter	The user has moved the mouse inside the window from a location outside the window.
MouseExit	The user have moved the mouse outside the window from a location inside the window.

Opening Windows

There are two different techniques you can use to open windows. The technique you use depends on what you are going to do with the window once it's open. If your application will never have more than one copy of a particular window open at a

time, you can open the window simply by making reference to any of the window's properties or by using the window's Show method.

The following example opens a window by accessing one of the window's properties (the window title in this case):

```
aboutBoxWindow.Title="About My Application"
```

If you don't need to change any properties of the window, you can simply call its Show method to open it, as in this example:

```
aboutBoxWindow.Show
```

This technique works when you will only have one copy of the window open at a time because the name of the window acts as a reference to the window. If you have two copies of the window open, REALbasic will access the window that is already open rather than opening a second copy of the window.

If your application may have more than one copy of a window open at a time, you need to use the New operator to explicitly create a new instance of the window. To use the New operator, you must have a local variable or a property defined as the window you are going to open. This variable or property is used to store a reference to the window once it has been created. You can then use this reference to access the window.

```
Dim w as aboutBoxWindow
w=New aboutBoxWindow
```

Because aboutBoxWindow is an object of type Window, you can also Dim the variable as a Window, as in this example:

```
Dim w as Window
```

```
w=New aboutBoxWindow
```

This is beneficial when your code may open many different windows and you can't be sure which window it will need to open, as in this example:

```
Dim w as Window
If theOptionKeyIsDown then
 w=New secretAboutBoxWindow
Else
 w=New aboutBoxWindow
End if
```

You could, of course, dimension two different variables; one as secretAboutBoxWindow and the other as aboutBoxWindow. But that might be a bit more confusing, especially if you had ten possible windows.

Because windows are objects, you can also dimension the variable as an object, as in this example:

```
Dim w as Object
w=New aboutBoxWindow
```

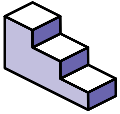
There is less of a need to dimension a window variable as type "Object" than there is to use type "Window." However, you might use this technique when you are creating new instances of controls on the fly. With controls, you can have a variable storing a reference to many different kinds of controls. See "Creating New Instances of Controls On The Fly" on page 198 for more information. See "Accessing Controls, Methods, and Properties of Other Windows" on page 193 for more information on how to use window references.

Adding Properties to Windows

Properties of an object are simply pieces of information that help define the object. Windows have many pre-defined properties such as their title, width, height, etc. You can also add your own properties to windows that allow you to store information that is specific to the instance of the window. For example, if you have a window that displays the contents of a document, you might need to keep track of whether the user has modified the data to determine if he should be given a chance to save changes when he quits your application. Where do you keep track of this? Since the window is effectively a representation of the document, you can add a boolean property called *Changed* to the window. When the user makes a change in the window that affects the document, your code can change the value of the *Changed* property from False to True. Later, when the user closes the window, the code in the window's Close event handler can check the Changed property to determine if the user needs to be given the opportunity to save his changes. The syntax for accessing the properties you add to windows is the same as the syntax you use to access a window's pre-defined properties. For example to set the Changed property of a window called "myDocumentWindow" to True, you use the following syntax:

```
myDocumentWindow.Changed=True
```

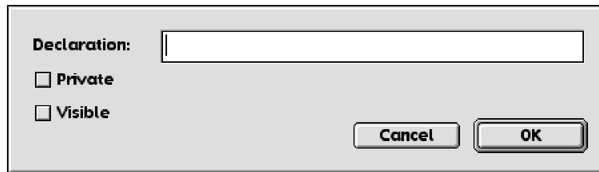
The *Changed* property should be not changed (no pun intended) from anywhere but the window. It wouldn't make sense for another window to be changing this property. However, six months after you add a property to a window, you might have forgotten this fact and add some code to another window that changes the *Changed* property. To avoid this problem, you can make the *Changed* property *private*. Properties that are marked as private can be accessed only by the window they are a part of.



To add a property to a window, do this:

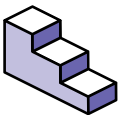
1. Open the Code Editor for the window.
2. Choose Edit ▶ New Property.
3. Enter the name of the property and define its type. For example, to the Changed property would be entered as *Changed as Boolean*.

FIGURE 80. **The Property Declaration dialog box**



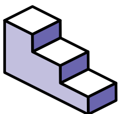
The Private and Visible checkboxes add the following features:

- Making a property Private means that the property can be accessed only by the event handlers and methods of the window.
- Making a property Visible means that, if you drag an instance onto a window, the property can be assigned a value from the Properties window (rather than only with code). This option is available only for non-control classes.



To Edit a property you've added to a window, do this:

1. Open the Code Editor for the window that contains the property.
2. In the Browser, expand the Properties category to display the list of properties for the window.
3. Double-click on the property or choose Edit ▶ Edit (⌘-E) to edit it.



To Delete a property from a window, do this:

1. Open the Code Editor for the window that contains the property.
2. In the Browser, expand the Properties category to display the list of properties for the window.

1. Click on the property you want to delete to select it.
2. Choose Edit ▶ Delete.

The properties of a window can be accessed from any code within the window itself or any of its controls, using the property name alone. The window name is not required as in this example that changes the window's title:

```
Title="My New Window"
```

In the absence of the window name, the current window is assumed.

Adding Methods to Windows

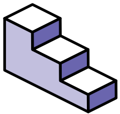
Like properties, windows can also have their own methods. The benefit of associating a method with a window is that you can keep code that will be used only with a particular window with that window. For example, suppose you have a window that displays the contents of a document. If the user can save changes to the document in the window, you will need some code that handles saving those changes. Since the window is handling the document, it makes sense that the window should know how to save changes to the document. Therefore, you might want to add a method called *SaveChanges* to the window that handles this. Later, should you decide to use this window for another project, it will have the *SaveChanges* method.

You can pass parameters to methods you add to windows and they can return a value, if necessary. Parameters are defined the same way that properties are (e.g., " Age as Integer"). If the method requires multiple parameters, the parameter definitions should be separated by commas. The Return Type is the data type of the value to be returned if your method will be returning a value. The pop-up menu to the right of the Return Type field has

a list of common data types but any type can be defined in the Return Type field.

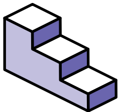
Like properties, methods can be made private so that they can only be called from within the window and not from other windows.

FIGURE 81. **The Method Declaration dialog box**



To add a method to a window, do this:

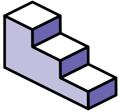
1. Open the Code Editor for the window.
2. Choose Edit ▶ New Method.
3. Enter the name of the method.
4. If the method will be passed parameters, define the parameters as you would properties, with multiple parameters separated by commas (example: "Age as Boolean, Name as String").
5. If the method will return a value (making it a function), enter the type of data it will return.
6. If this method should not be accessible by code in other windows, check the Private checkbox.



To Edit the name, parameters, or return type of a method you have added to a window, do this:

1. Open the Code Editor for the window that contains the method.
2. In the Browser, expand the Methods category to display the list of methods for the window.

3. Double-click on the method or highlight it and choose Edit ▶ Edit (⌘-E) to edit it.



To Delete a method you've added to a window, do this:

1. Open the Code Editor for the window that contains the method.
2. In the Browser, expand the Methods category to display the list of methods for the window.
3. Click on the method you want to delete to select it.
4. Choose Edit ▶ Delete.

Accessing Controls, Methods, and Properties of Other Windows

Items in other windows can be accessed using the window name followed by the control, method, or property name. In the case of controls, the control name can then be followed by one of its property names. For example, suppose a button in window1 will, when clicked, place the text "Hello World" in the text property of a control called StaticText1 in window2. The syntax is:

```
Window2.StaticText1.Text="Hello World"
```

Methods can be called using the same syntax. For example a button in Window1, when clicked, passes the value "Hello" to the "Find" method of Window2. The syntax is:

```
Window2.Find "Hello"
```

The properties of other windows can also be accessed using this syntax. For example, if a button in Window1 should, when clicked, change the title of Window2 to "Hello World", the syntax is:

```
Window2.Title="Hello World"
```

The syntax in the previous examples works provided there is only one instance of the target window open. If there are two instances of Window2 open, the code in the previous examples would affect only the first instance of Window2 that was opened.

If there can be more than one instance of the target window open, you need to store a reference to that window somewhere so your code will know which instance of the window you are referring to. Where you store this reference depends on how your application works. Suppose you have many instances of a window named "DocWindow" open that displays the contents of a text document. A button in this window opens a Find window that lets the user enter a value he wishes to search for in that instance of DocWindow. Since there can be many DocWindows open, you will need to store a reference to the specific instance of the DocWindow that opens the Find window in a property of the Find window. You do this by adding a property (let's call it "Target") to the Find window of type DocWindow. When the Find button in an instance of the DocWindow opens the Find window, it can store a reference to the DocWindow in that property. Assuming your application only allows one Find window to be open at a time (perhaps by making the Find window modal), the syntax looks like this:

```
FindWindow.target=Self
```

The Self function returns a reference to the instance of a window (or class) that calls the Self function. In this case, the target property of the FindWindow is being set to a reference to the specific instance of the DocWindow that executed this code. Later, when the user clicks the Find button in the FindWindow, the FindWindow can use the Target property to reference the instance of the DocWindow that opened the FindWindow in the first place.

FIGURE 82. An Example Find window

In Figure 82 the FindWindow has an EditField named "Find-Value" where the user types what he wishes to find. Let's also assume that the DocWindow has a method called "Find" that, when passed a value, locates that value (if it exists) in an EditField in the DocWindow and highlights the value found. When the user clicks the Find button in the FindWindow, the Find button's Action event handler calls the Find method of the instance of the DocWindow that opened the FindWindow. It does this using the FindWindow's target property and the following syntax:

```
Target.Find FindValue.Text
```

The Target property contains a reference to the DocWindow, so its Find method can be called. In this example, the Find method is being passed the value of the Text property of the FindValue EditField.

The Target property can also be used to change properties of controls in the target window. For example, if you want to disable the Find button in the DocWindow from the FindWindow, you can do so using the following syntax:

```
Target.FindButton.Enabled=False
```

In this example, the Target property of the FindWindow is defined as being of type DocWindow. However, if the FindWindow needs to reference more than one window class, you would define the Target property as type Window to be more generic. This allows the Target property to store a reference to an instance of any kind of window rather than just an instance of DocWindow. However, it also makes the code less readable because it is not clear which windows the FindWindow meant to work with. For this reason, use the generic Window type only when necessary.

Controls

Controls are items that appear inside a window that can have their own code to respond to events directed to them. Unlike windows, you cannot add methods or properties to the controls you drag to the window from the Tools window. However, you can create controls that have custom properties, methods, and even menu handlers by creating new classes based on the controls. See the chapter, “Creating Reusable Objects with Classes” on page 319 for more information.

Events

Controls, like windows, receive events and have event handlers to respond to the events they receive. For every event a control receives that you can respond to, there is a corresponding event handler.

TABLE 42. **The standard events that all controls receive**

Name	Description
Open	The window containing the control is about to open. This event handler is a great place to doing anything to the control you need to do before the window is displayed.

TABLE 42. **The standard events that all controls receive (Continued)**

Name	Description
Close	The window containing the control is about to close. This event handler is a great place to do any cleanup related to the control before the window closes.
DropObject	Something has been dropped on the control. For more information on handling drag and drop, see "Drag and Drop" on page 201.

All of the visible controls have several standard mouse events they can receive as well.

TABLE 43. **The standard mouse events for visible controls**

Name	Description
MouseEnter	The mouse has moved from a point outside the control to a point inside the control.
MouseMove	The mouse has moved from a point inside the control to another point inside the control.
MouseExit	The mouse has moved from a point inside the control to a point outside the control.

The button controls (pushbuttons, radiobuttons, bevelbuttons, and checkboxes) all have an Action event handler that is executed when the button is clicked.

ListBoxes and PopupMenus both have a *Change* event handler that is executed when the user changes the selected item or items. ListBoxes have additional event handlers because they can be hierarchical, can receive the focus, and can be draggable.

TABLE 44. **Additional ListBox event handlers**

Name	Description
DoubleClick	The user has double-clicked on an item.
KeyDown	The user has pressed a key while the listbox has the focus.

TABLE 44. **Additional ListBox event handlers (Continued)**

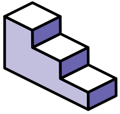
Name	Description
ExpandRow	The user has clicked on a row's disclosure triangle to expand it. In order for a disclosure triangle to appear, the Hierarchical property of the Listbox must be set to True and the row must be added using the AddFolder method.
CollapseRow	The user has clicked on a row's disclosure triangle to collapse it.
DragRow	The user has dragged a row from the Listbox. In order for a user to drag a row, the EnableDrag property of the Listbox must be set to True.

Because Sliders and Scrollbars operate the same way, they both have a *ValueChanged* event handler that is executed when the user scrolls the Scrollbar or drags the Slider.

The Serial and Socket controls both have a *DataAvailable* event handler that is executed when the control receives data.

Creating New Instances of Controls On The Fly

There may be situations where you can't build the entire interface ahead of time and need to create some or all of the interface elements on the fly. This can be done in REALbasic provided that the window already contains a control of the type you wish to create. The existing control is used as a template. For example, if you wish to create a pushbutton via code, there must already be a pushbutton in the window that you can "clone." Remember that controls can be made invisible, so there is no need for your template control to appear in the window. Once you have created a new instance of the control, you can then change any of its properties.



To create a new control on the fly via code, do this:

1. Dimension a local variable of the type of the control you will be using as a template. For example, if the template control is a pushbutton, dimension your variable as a pushbutton.
2. Assign the variable a reference to a new control using the New operator and pass it the name of the template control.

This example shows a new pushbutton being created using the existing Pushbutton1 as a template. Because the new control will have the same properties and code as the template, once the new control is created, the control is then moved to the right of the template control:

```
Dim b as PushButton
b= new Pushbutton1
b.Left=me.Left+me.Width+10
```

Since any new control you create shares the same code as the template control, you may need to be able to differentiate between them from the code. You can use the index property of the control to identify which control was clicked, but in order for this to work, the template must have an index value. This effectively makes all of the controls of a particular type act as a control array. For more information on control arrays, see "Sharing Code Among An Array of Controls" on page 200.

If your code needs to create different kinds of controls and store the reference to the new control in one variable, you can dimension the variable as being of the type of object that all the possible controls you might be creating have in common. For example, if a variable can contain a reference to a new radiobutton or a new checkbox, the variable can be dimensioned as a RectControl because both radiobuttons and checkboxes are RectControls. Keep in mind, however, since the variable is a RectControl, the properties specific to a radiobutton or checkbox

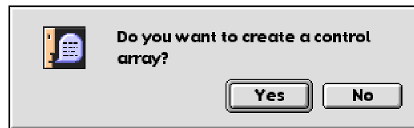
will not be accessible. If you need to see which classes of control are common to different controls, see " "The Class Hierarchy" on page 3 of the Language Reference.

Sharing Code Among An Array of Controls

When you have several controls of the same type that all have essentially the same code, the best solution is a control array. A control array allows two or more controls to share the same code. You create a control array by assigning all of the controls the same name and using the Index property to identify the elements of the array of controls.

The first time you give a control the same name as another control (that's not already part of a control array), REALbasic will ask you if you wish to create a control array, as shown in Figure 83.

FIGURE 83. **Creating an array of controls with a dialog box.**



If you click Yes, REALbasic will assign the first control's Index property the value 0. The control you are renaming will then have its Index property set to 1. After that, any controls in the same window with the same name will be assigned the next number in the sequence automatically.

For example, you have a checkbox named "Option". If you create a second checkbox and rename it "Option", REALbasic will ask you if you wish to create a control array. When you click Yes, REALbasic will assign the Index property to 0 for the first Option checkbox and 1 for the second.

The other way to create an array of controls is to enter zero as the value of the Index property of the first control and then assign the first control's name to the second control. REALbasic will then automatically assign 1 to the Index property of the second control, and so on. This procedure bypasses the dialog box shown in Figure 83 but achieves the same effect.

In the Code Editor, rather than seeing several controls with the same name, the control will appear only once followed by parens to let you know it's a control array. All of the controls in the control array share one set of events. Each event in a control array is automatically passed an Index parameter which tells you which control in the control array actually receives the event.

Drag and Drop

Drag and drop is a very important part of the interface of many applications. It extends the concept of the mouse's being an extension of the user's hand. Fortunately, drag and drop is easy to implement in REALbasic. Dragging and dropping of text, pictures, documents, and specified data types is supported.

When something is dragged, a DragItem object is created. DragItems have a Text property that is used to hold text being dragged, a Picture property for holding images being dragged, a MacData property for holding a specified data type, and a FolderItem property that can contain a FolderItem that references a document, folder, or application being dragged. In some cases, you need to populate these properties with data you wish dragged, while in others, the appropriate property will be populated automatically.

A DragItem object can actually contain more than one item and the items don't necessarily have to be of the same type. When you allow the user to drag multiple items, you need to create

additional items within the DragItem object and, when the DragItem is dropped, cycle through all items in the DragItem.

DragItems that are dragged to the Desktop or to other applications will act just as you would expect them to. For example, dragging text to the Desktop creates a text clipping file. A DragItem containing a picture that is dragged to the Desktop creates a picture clipping file.

Dragging Text From EditFields

Only text in EditFields, rows in ListBoxes, portions of Canvas controls, images in ImageWells, and Windows can be dragged. If you have never implemented drag and drop before, this may sound like a limitation, but in fact, it isn't. These controls are the only types of objects that can be dragged in other applications that support drag and drop.

The text in an EditField can be dragged automatically without any coding necessary, provided that the MultiLine property of the EditField is True. A DragItem object is automatically created and the text the user is dragging is placed in the Text property of the DragItem.

Dragging A Row From a ListBox

In order for the user to be able to drag a row from a ListBox, the EnableDrag property of the ListBox must be set to True. When the user attempts to drag a row, the DragRow event handler of the ListBox executes and is passed the DragItem that was created and the row number of the row being dragged. You then have to populate the Text property of the DragItem passed. Finally, since the DragRow event handler is actually a function, your code must return True to allow the drag to occur. Returning False or returning nothing at all prevents the drag. This example code

from the DragRow event handler of a Listbox handles dragging a row from the ListBox:

```
Function DragRow(Drag as DragItem, Row as Integer)
 Drag.Text=Me.List(Row)+chr(13) //get the text
 Return True //allow the drag
End Function
```

Dragging from an ImageWell

Drag and drop to or from an ImageWell is simple. When dragging from an ImageWell control you must:

- Create a DragItem object using the NewDragItem method of the Control class,
- Load the data to be dragged into the new DragItem instance,
- Call the DragItem's Drag method to allow the drag to occur

The DragItem object is the container that holds the dragged image. NewDragItem is a method of the Control class and takes as its parameters the left, top, width, and height of the drag rectangle you want displayed when the user begins the drag. The Drag method is called when the data to be dragged is loaded in the DragItem. You place this code in the ImageWell's MouseDown event handler, since the user presses the mouse button to initiate the drag.

```
Function MouseDown(X as Integer,Y as Integer) As Boolean
 Dim d as DragItem
 d=Me.NewDragItem(Me.left,Me.top,Me.width,Me.height)
 d.picture=Me.image
 d.Drag //Allow the drag
End Function
```

Dragging from a Canvas Control

Dragging the backdrop image from a Control is the same as dragging the image from an ImageWell. You place this code in the Canvas control's MouseDown event handler, since the user presses the mouse button to initiate the drag.

```
Function MouseDown(X as Integer,Y as Integer) As Boolean
 Dim d as DragItem
 d=Me.NewDragItem(Me.left,Me.top,Me.width,Me.height)
 d.Picture=Me.backdrop //populate DragItem with data
 d.Drag //Allow the drag
End Function
```

When you program the Drop, you will assign the Picture property of the DragItem to a property of the control receiving the drag.

Dropping

In order for the user to be able to drop a DragItem on a control or window in your application, the control or window must have previously indicated that it will accept the kind of data the user wishes to drop on it. There are four methods that any control can call to indicate the type or types of data that can be dropped on that control. You can indicate that the control or window can accept more than one data type by using as many methods in Table 45 as appropriate.

TABLE 45. **Methods for indicating acceptable data**

Name	Description
AcceptTextDrop	Indicates that the control or window will accept text being dropped on it.
AcceptPictureDrop	Indicates that the control or window will accept a picture being dropped on it.

TABLE 45. **Methods for indicating acceptable data (Continued)**

Name	Description
AcceptFileDrop	Indicates that the control or window will accept files (of the type or types passed) being dropped on it. The file types must be defined as file types for this project in the File Types dialog box.
AcceptMacDataDrop (*Type*)	Indicates that the control or window will accept the data type specified by the four-character Type code, e.g., AcceptMacDataDrop ("mytp"). Use this to drag and drop data in special formats or to control where text strings can be dropped.

Typically, the control or window will call one or more of these methods in its Open event handler. However, if a control or window only accepts items dropped on it under certain conditions, these methods can be called once those conditions are met even after the window is opened.

In most cases, when something acceptable is dropped on a control or window, the target's DropObject event handler is executed. This event handler is passed a DragItem object that represents the item being dropped. If the target has indicated that only some kinds of data are acceptable, your code can get the data from the appropriate property of the DragItem. The properties are:

TABLE 46. **DragItem properties that contain data**

Name	Description
FolderItem	Represents an application, folder, or document that has been dropped.
Picture	The picture, if any, that has been dropped.
Text	The text, if any, that has been dropped.
MacData (Type)	Macintosh data of the Type indicated by the four-character type code. Data of the format indicated by the Type code is returned as a string.
PrivateMacData (Type)	Same as MacData, except that the data cannot be dropped outside the REALbasic application.

If more than one kind of data can be dropped, the code in the DropObject event handler needs to determine what kind of data has been dropped. This can be done using these functions of the DragItem:

TABLE 47. **DragItem functions that determine what has been dropped**

Name	Description
FolderItemAvailable	Returns True if one or more applications, folders, or documents have been dropped.
PictureAvailable	Returns True if a picture was dropped.
TextAvailable	Returns True if text was dropped.
MacDataAvailable (Type)	Returns True if Macintosh data of the type indicated by the four-character Type code was dropped.

Dropping Items On EditFields

Text dropped on a multiLine EditField is placed in the EditField at the insertion point automatically. The EditField's DropObject event handler is not called. Pictures and files dropped on a multiline EditField, however, cause the DropObject event handler to execute. For example, if you want to be able to drop a text file on an EditField and have the contents appear in the EditField, you need to get the FolderItem from the DragItem that is passed to the EditField's DropObject event handler and read the contents of the file.

In this example, an EditField has been set up to accept text files dropped on it. *Me* is the generic representation for the object that owns the event handler:

```
Sub DropObject(Obj as DragItem)
 If Obj.FolderItemAvailable then
  Obj.FolderItem.OpenStyledEditField Me
 End if
End Sub
```

Since more than one file can be dropped at a time, you need to use the NextItem function of the DragItem to determine if there is another file that has been dropped. The NextItem function also changes the FolderItem property of the DragItem to the next file. The last example, modified to handle more than one file dropped on it, looks like this:

```
Sub DropObject(Obj as DragItem)
 If Obj.FolderItemAvailable then
  Do
   Obj.FolderItem.OpenStyledEditField Me
  Loop Until Not Obj.NextItem
 End If
End Sub
```

Dropping Items on ListBoxes

If you want to drag text to a ListBox, you need to tell the ListBox to receive dragged items by placing the following line in its Open event handler (or another event handler that runs prior to the user's drag and drop):

```
Me.AcceptTextDrop
```

Next, you need to tell the ListBox what to do when dragged text is coming its way. You do that in its DropObject event handler:

The following DropObject event handler determines whether the dragged item has text; if it does, it creates a new row and assigns the dragged item's text property to the new row.

```
Sub DropObject (Obj as DragItem)
 If Obj.TextAvailable then
  Me.AddRow(obj.text)
 end if
End Sub
```

Dropping Items on ImageWells and Canvas controls

To allow the user to drop a picture or a PICT document that is being dragged from the desktop, add the following statements to another ImageWell or Canvas control's Open event handler:

```
Me.AcceptPictureDrop
Me.AcceptFileDrop("image/x-pict")
```

The second statement assumes that the PICT file type has previously been added in the File Types dialog box (see "Using The File Types Dialog Box" on page 279 for more information).

Next, you need to tell the ImageWell or Canvas what to do when the user drops either type of DragItem. You do this in the DropObject event handler. The following works for an ImageWell.

```
Sub DropObject (Obj as DragItem)
 If Obj.PictureAvailable then
  ImageWell1.Image=Obj.Picture
 Elseif Obj.FolderItemAvailable then
  me.image=Obj.FolderItem.OpenAsPicture
 End if
End Sub
```

It tests whether the DragItem is a picture or a file (FolderItem). If it's a picture, it assigns the Picture property of the DragItem to the Image property of the ImageWell; if it's a FolderItem, it opens the document as a PICT file and assigns that to the ImageWell's Image property.

If you want to assign the dropped object to a Canvas control's BackDrop property, the DropObject event handler is practically identical:

```
If Obj.PictureAvailable then
 Canvas1.Backdrop=Obj.Picture
Elseif Obj.FolderItemAvailable then
 Canvas1.Backdrop=Obj.FolderItem.OpenAsPicture
End if
```

You can also drag a text item to a Canvas control and use the DrawString method in the Graphics class to draw the text. To allow the drag, use the line:

```
me.AcceptTextDrop
```

in an event handler that runs prior to the user's drop. Next, test for text in the DropObject event handler. The following code accepts dragged text and writes it at a specified location:

```
If Obj.TextAvailable then
 Canvas1.Graphics.DrawString(obj.text,20,20,50)
End if
```

You will need to update the data using the Paint event handler if you want the data to persist.

MacData and PrivateMacData Properties

The MacData and PrivateMacData properties allow you to drag and drop other data types. Each property takes a four-character Type that corresponds to the four-character resource code of the format you wish to drag. For example, if you want to support dragging sound clippings, you would use 'snd ' as the four-character code.

Regardless of format, the data is stored in the DragItem as a string and the DropObject event handler would pass the data to a property that stores a string. The REALbasic control or window that receives the DragItem must be capable of working with the

format. In most cases, that means that the control is a custom control, uses toolbox calls, and/or uses a plug-in that manages the data.

For example, to allow the user to drop a 'snd ' resource on a control, you would write

```
acceptMacDataDrop("snd ")
```

in the control's Open event handler. The DropObject event handler would pass the data to a property that stores a string. To actually play the sound, the control would have to make toolbox calls or pass the data to a plug-in since REALbasic doesn't provide a way to pass sound data into a Sound class.

You can also use MacData or PrivateMacData to manage internal drag and drop. If you make up a format type that is different than the name of any resource type, you can use it to control which objects are dropped on a control. For example, the DragRow event handler of a ListBox

```
Function DragRow(drag as DragItem, row as Integer) as Boolean
 Drag.MacData("mytp")=me.List(row)
 Return True
End Function
```

uses the Type "mytp" to define the format. The DragItem is assigned the row in a ListBox that is being dragged. Although the row is an ordinary string, this DragItem cannot be dropped on a control unless it accepts dragged data in the "mytp" format. In this manner, for example, you can create a control that will only accept a DragItem from a specific control and reject dragged items from the Finder, other applications, and other controls.

The ListBox receiving the data would use the statement:

```
me.AcceptMacDataDrop("mytp")
```

to permit the data to be dropped. The DropObject event handler assigns the data to a new row:

```
If obj.MacDataAvailable("mytp") then
 me.AddRow(obj.MacData("mytp"))
end
```

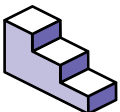
The PrivateMacData property works the same way, except that the DragItem cannot be dragged to the Finder or to any other application.

MacData and PrivateMacData are available only on Macintosh applications.

Menu Items

Menu items are handled in a way similar to controls and are just as object-oriented. This means that the handling of menus can occur at the application, window, or even control level. When the user selects a menu item or presses the menu item's command key equivalent, an event occurs much in the same way that an event occurs when the user clicks on a pushbutton. In this case, the event handlers are instead called *menu handlers*. For information on creating menus, see "Adding Menus" on page 118 of chapter 3.

Adding Code To a Menu Item



To add a menu handler to the current window or class, do this:

1. Open the Code Editor for the window or class.

2. Choose Edit ▶ New Menu Handler. The New Menu Handler dialog box appears.
3. Choose a menu item object from the Menu Item pop-up menu.
4. Click OK.
5. Enter the code that should execute when the user chooses the menu item.

FIGURE 84. The New Menu Handler dialog box

Enabling Menu Items

Menu items that you add with the Menu Editor are always disabled. When the user clicks on a menu to select a menu item or presses a keyboard equivalent, an EnableMenuItems event occurs. The purpose of this event is to give you the opportunity to determine whether the menu item being selected should be enabled or disabled based on conditions at the time. REALbasic first checks to see if the control that has the focus is capable of handing menus. If it is, it is sent an EnableMenuItems event. Then, assuming a window is open, the frontmost window is sent the EnableMenuItems event. Finally, the application object is sent the EnableMenuItems event.

Menu items are objects just like controls. Consequently they have an Enabled property that determines if the menu item is enabled or disabled. For example, this EnableMenuItems event handler is checking a property called *Changed* to determine if the Save menu item should be enabled:

```
Sub EnableMenuItems()
 If Me.Changed Then
  FileSave.Enabled=True
 End If
End Sub
```

Handling Menu Items From Individual Controls

If the control that has the focus is capable of handling menus, its EnableMenuItems event handler will be executed. If the menu item selected is then enabled and the user selects it, the control's menu handler for the selected menu item (if it has one) will be executed. In order for a control to be able to handle menu items, it must be able to receive the focus (it must be an EditField or ListBox) and it must be based on a class you have added to your project rather than created by dragging a control from the Tools window. See Chapter 9, "Creating Reusable Objects with Classes" on page 319 for more information on handling menu items from control classes.

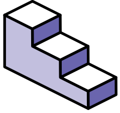
Handling Menu Items When a Window Is Open

You already know that when the user attempts to select a menu item, the frontmost window's EnableMenuItems event handler is executed followed by the application object's EnableMenuItems event handler. This gives you the opportunity to determine if conditions in the current window are right to permit the user to select various menu items. When the user selects the menu item, REALbasic executes the frontmost window's menu handler for the selected menu item (assuming one exists) followed by the application object's menu handler.

Handling Menu Items When No Windows Are Open

When there are no windows open, the EnableMenuItems event is sent to the Application object. Assuming the application object

enables the menu item and the user selects the menu item, the application object's menu handler for the selected menu item (if one exists) is executed.



To create an application object, do this:

1. Choose File ▶ New Class.
2. In the Properties window, choose Application from the Super pop-up menu.
3. Enter App in the Name property of the new class.

For more information on the application object, see "The Application Class" on page 337.

Creating New Menu Items On The Fly

This is handled in a way that is similar to how you create controls on the fly. A menu item that can act as a template must already exist. This menu item will effectively be "cloned." You can then change the clone's properties such as the Text, keyboard shortcut, etc. The difference is that the menu items must have an index value in their Index property in order to be used as a template. Assign a zero to the Index property of the menu item to create a menu item array. The menu handlers for the menu item will then be passed an Index parameter that allows you to determine which menu item was selected. If you don't assign an index value, you will have no way of knowing which menu item was passed. Once you have setup the template menu item, you can create new menu items on the fly using the New operator. This example creates a new menu item based on an existing menu item named "WindowItem."

```
Dim t as MenuItem
t=New WindowItem
```

Remember that once you have created a menu item array, you must refer to the items in that array as array elements. For example, to enable the first menu item (item zero from the WindowItem example), use the following syntax:

```
WindowItem(0).Enabled=True
```

If you wish to be able to programmatically remove menu items you have created dynamically, you need to store the reference that was returned when you created the menu item. You can then use this reference to remove the menu item by calling the Close method. For example, you are storing references to the menu items in a module property array called "WindowRefs." You can then remove a particular dynamically created menu item (the item stored in the fourth array element in this case) using this syntax:

```
WindowRefs(4).Close
```

Classes

Classes can be used to create custom controls that can also respond to the user. For more information on using classes to create custom controls, see Chapter 9, "Creating Reusable Objects with Classes" on page 319.

CHAPTER 6

Adding Global Functionality with Modules

Object-oriented programming can be very efficient but you may find occasions when you need to add methods, functions, and even properties that are not associated with any one object. For example, you might need to add some custom financial functions that will be called from many different places within your application. You may need to store a value that is associated with those functions. In most cases, when you need to add a method, function, or property that isn't associated with any particular object and needs to be accessible globally, a module is the perfect place to add it.

Constants can have two special purposes when added to modules. They can be used to localize the interface elements of an application and to provide Windows accelerators (keyboard shortcuts). These special uses are described in the sections "Using Constants to Localize Your Application" on page 222 and "Using Constants to Add Windows Keyboard Shortcuts to Menus and Menu Items" on page 224.

In this chapter, you will learn what modules are, when to use them, and how to add methods and properties to them.

Contents

- Understanding Modules
- Adding Methods
- Adding Constants
- Adding Properties

Understanding Modules

In REALbasic's object-oriented environment, methods, constants, and properties are usually part of another object. Methods, constants, and properties associated with objects are only accessible through those objects. However, the methods, constants, and properties associated with a module are accessible to all objects and code in your application at all times.

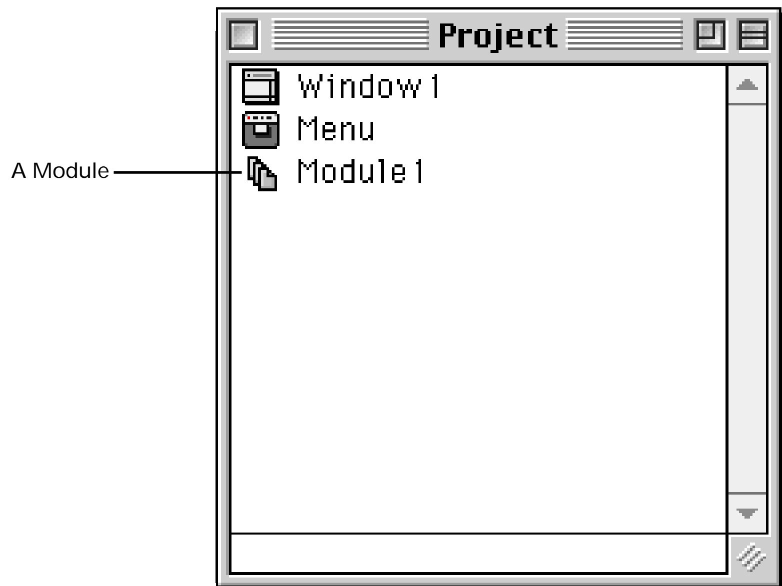
Modules are not objects. You don't instantiate modules in order to access them. Once you add a module to your project and then add methods, constants, or properties to it, those objects are immediately accessible. The only exceptions are private methods and properties. These methods and properties are accessible only from other methods in the same module.

Adding A New Module

You can add a new module to your project by choosing File ▶ New Module. The Code Editor for the module will be displayed automatically. The new module appears in your project window with a default name (the first module you add will be named "Module1," for example). You can then use the Properties window to rename the module to something more appropriate. If the module will contain your financial functions, you might name it "Financial."

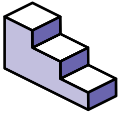
Modules can only contain methods, constants, and properties. The only way to modify them is through the Code Editor. To access the Code Editor for a module that is not already open, simply double-click on the module in the Project window. Modules can be identified by their special icon in the Project window.

FIGURE 85. **A module in the Project window**



Adding Methods to Modules

Adding methods to modules is done in the same way you add methods to a window.

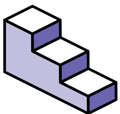


To add a method to a module, do this:

1. Double-click on the module in the Project window to open it. The Code Editor for the module appears.
2. Choose Edit ▶ New Method. The Method Declaration dialog box appears.
3. Enter the method name and parameters. If the method is going to be a function, choose the data type of the value the function will return. If you click the Private checkbox, the method will only be accessible to other methods in the same module.
4. Click OK.

Adding Properties to Modules

Module properties are global in scope. They are accessible to all code in the project unless you choose to make them private. Private properties are accessible only by methods in the same module as the property. Adding properties to modules is done in the same way you add properties to a window.



To add a property to a module, do this:

1. Double-click on the module in the Project window to open it. The Code Editor for the module appears.
2. Choose Edit ▶ New Property. The Property Declaration dialog box appears.
3. Enter the property name, “as,” and the data type. For example, a string property called “Name” would be entered as “Name as String” (without the quotations). If you click the Private checkbox,

the property will only be accessible to other methods in the same module.

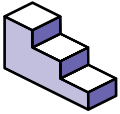
4. Click OK.

If you are creating a module for the sole purpose of adding properties to your application that will be global (accessible from everywhere in the application), consider creating a class based on the Application object and adding your global properties to the application object. They will still be global and this approach is more object-oriented since the properties are now associated with the application directly rather than with a module that happens to be part of the application. See the chapter “Creating Reusable Objects with Classes” on page 319 for more information on creating a class based on the application object.

Adding Constants to Modules

Like methods and properties, a constant added to a module is global in scope. It is recognized everywhere in your application. You can also add constants to individual methods (local constants), but adding all your constants to a module makes it easier to maintain your application. This point is discussed in the section “Constants” on page 140, which explains the process of creating local constants.

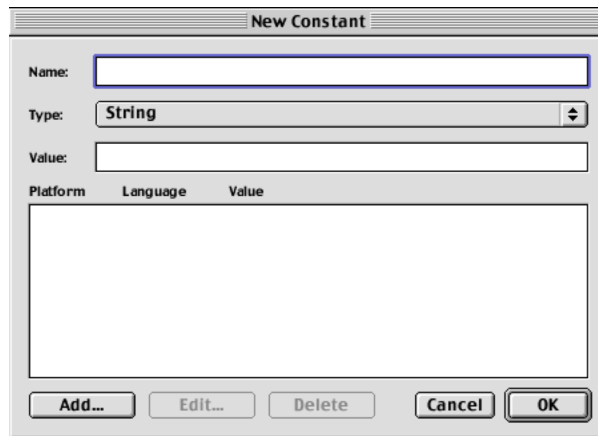
Global constants also provide a very convenient way to localize your application. If you use global constants for all the text that appears in your application’s interface, you can instantly localize the application simply by changing the Default Language setting in Project Settings and specifying the Default Language in the Build Application dialog box when you are ready to create a standalone application. For more information, see the section “Building Your Application” on page 426.



To add a constant to a module, do this:

1. Double-click on the module in the Project window to open it. The Code Editor for the module appears.
2. Choose Edit ▶ New Constant. The Constant Declaration dialog box appears. It is shown in Figure 86.

FIGURE 86. **The New Constant Dialog Box**



3. Enter the name of the constant, its data type, and its value.
4. Click OK.

The New Constant dialog box supports standard Macintosh Cut, Copy, and Paste operations.

Using Constants to Localize Your Application

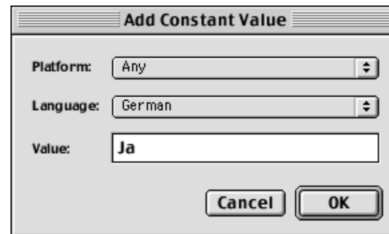
The lower section of the New Constant dialog box lets you assign different values to the constant depending on platform and default language. When you change the Default Language in Project Settings or the Build Application dialog box, the corresponding values for each constant take effect automatically.

The following illustrates how to set up a constant that will be used as the caption for a button control.

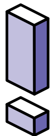
1. Using the New Constant dialog box, add new constant whose name is **OKButton**.
2. Define **OK** as the Value.
3. Click the Add button at the bottom of the dialog box and add a value of **Ja** for any platform and set the language to German.

This is shown in Figure 87.

FIGURE 87. **Localizing the OK Button for the German Version**



4. Click OK and then add a button to a window. Change the button caption to #**OKButton**.
5. Choose Edit ▶ Project Settings and change the default language to German.



It is mandatory that a value be provided for each Platform/ Language combination that you define. If you omit that value, REALbasic will have no idea what to use when you build the application for that platform and language.

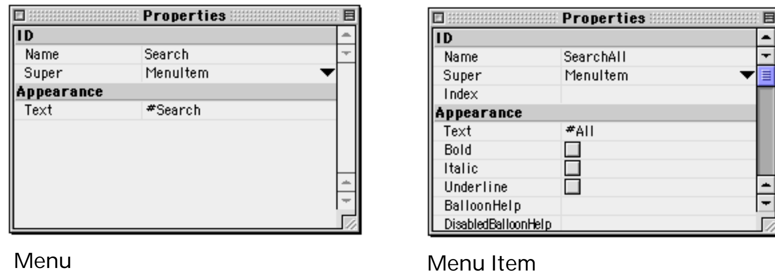
When you test your application, the button's caption will be "Ja" instead of OK.

You can localize menus and menu items in exactly the same way. Create a global constant for each text string that will be used as a menu and menu item. Then use a constant's name as the menu's

Text property, preceded by the number sign ("#"). Similarly, use another constant's name as each menu item's Text property.

A localized menu and menu item are shown in Figure 88 on page 224.

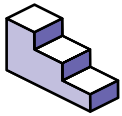
FIGURE 88. Localizing a Menu and a Menu item.



This technique works for all static text that appears in windows: bevel button menus, contextual menus, tab panel labels, etc.

Using Constants to Add Windows Keyboard Shortcuts to Menus and Menu Items

On the Windows platform, keyboard shortcuts for menus and menu items are denoted by an underlined character in the name of the menu or menu item. Although Macintosh keyboard shortcuts can be added as a property of the menu or menu item, this does not work for the Windows version of the application. Windows keyboard shortcuts can be added only via constants. You define the Windows keyboard shortcut using the Constants system and then assign the name of the constant to the Text property of the menu or menu item.

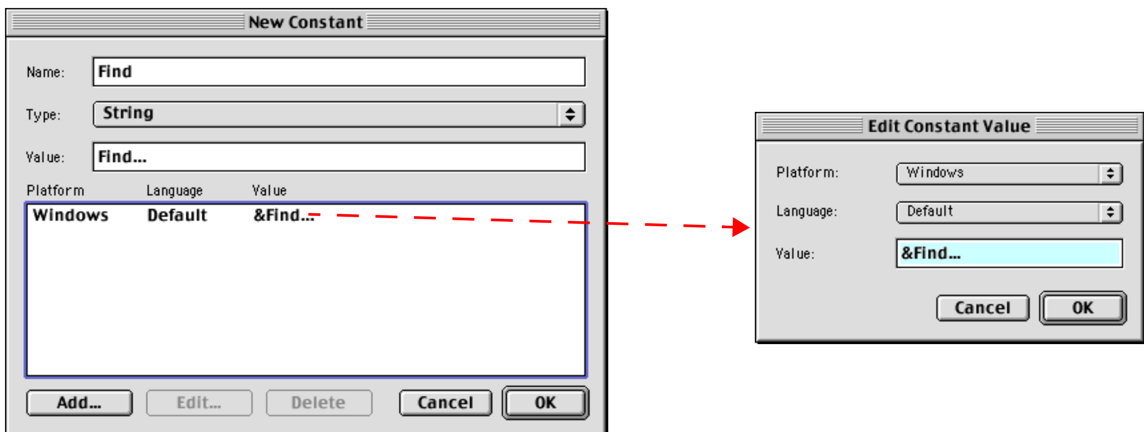


To add a Windows keyboard shortcut to a menu or menu item, do this:

1. Add a new constant to a module. Give it an appropriate name for the menu or menu item it will represent.
2. Assign the default value for the Macintosh platform in the Value field.
3. Click the Add button to add a platform-specific constant. Choose Windows as the platform name and enter the value in the Value field. Type an & just before the keyboard shortcut character.

This is illustrated in Figure 89.

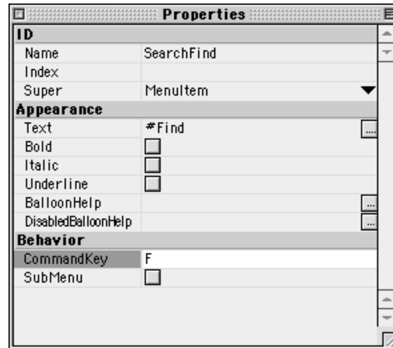
FIGURE 89. **Assigning "F" as the Windows keyboard equivalent for the Find menu item.**



4. Click OK to close this dialog box and click OK again to close the Constants editor.
5. Select the menu or menu item in the Menu Editor and enter # and the name of the constant as the Text property of the menu or menu item. If applicable, enter the Macintosh keyboard equivalent as the CommandKey property.

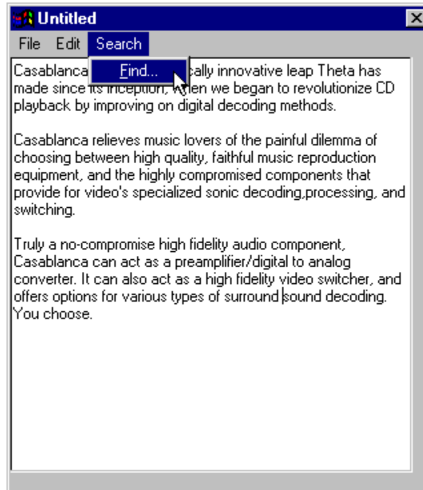
This is illustrated in Figure 90 on page 226.

FIGURE 90. **Assigning a constant and Macintosh keyboard equivalent to a menu item.**



When you deploy the application in Windows, the command key you assigned via the Constants system will appear. This is illustrated in Figure 91.

FIGURE 91. **The Windows keyboard shortcut denoted by an underlined character.**



Importing and Exporting Modules

Modules can be imported from other REALbasic projects. Modules that have been exported from other projects appear on the desktop with a cube icon.

FIGURE 92. **An exported module's desktop icon**

Importing

To import a module into your project, drag the module into your Project window. Or, choose File ▸ Import and locate the module to be imported using the open-file dialog box. If the module is protected, you won't be able to see or edit its methods or properties. To determine if a module is protected, double-click on it in the Project window after you import it. REALbasic will inform you when you attempt to open it in the Code Editor if it's protected.

Exporting

Modules can be exported for use in other REALbasic projects. You can export a module using two different procedures:

- Drag the module from the Project window to the desktop
- Click on the module in the Project window to select it and choose File ▸ Export Module. This method allows you to export a protected copy of the module.

Both procedures will export the module. The first procedure is easier if you can see the folder, the desktop, or the disk you wish

to copy the module to. If you need to save the exported module to a specific folder, use the second procedure. The second procedure also allows you to export a protected copy of your module that others can't edit. To export a protected copy, select the Protect option in the Save As dialog box when you choose File ▶ Export Module.

CHAPTER 7

Working With Text and Graphics

Almost every application manipulates text and graphics in some way. Fortunately, REALbasic provides a rich set of functions for creating, manipulating, displaying, and printing text and graphics. Should you wish to create your own custom control, you can use the Canvas control and its graphics methods to create it.

Contents

- Working With Fonts
- Working with the Selected Text
- Handling Styled Text
- Formatting Numbers, Dates and Times
- Understanding the Canvas Control and Graphics Object
- Drawing Pictures

- Working with Color
- Printing Text and Graphics
- Transferring Text and Graphics with the Clipboard
- Creating Animation with Sprites

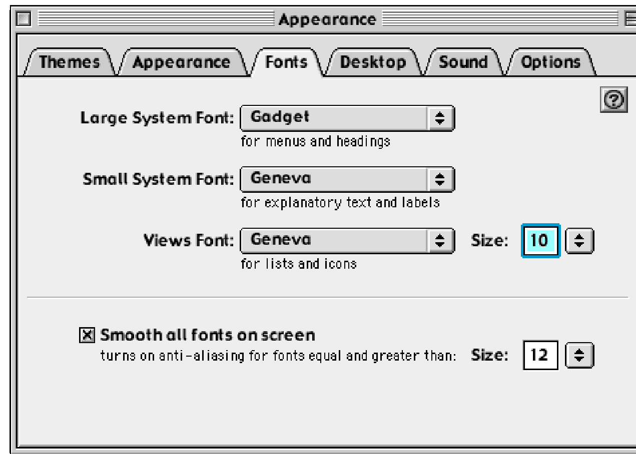
Working With Fonts

REALbasic gives you the ability to set the font, font size, and font style of many of the objects and controls in your application. EditFields support multiple fonts, styles, and sizes (collectively referred to as *styled text*) and ListBoxes support multiple styles. Controls that use a single font have a TextFont property that you can set by assigning it the name of the font you want used to display text for the control. EditFields have a TextFont property but they can also display multiple fonts. For information on styled text in EditFields, see "Handling Styled Text" on page 234.

The System Font

The System font is the font used by the system software as its default font. It's the font used for the menus as well. The System font can be changed. For example, in System 7, the System font is Chicago. If the user is running the Aaron extension, the system font may be Espi; if they are running Mac OS 8 or 9 the default is Charcoal. Under MacOS 9, REALbasic uses the font called "Large System Font" in the Appearance Control Panel as the System Font.

FIGURE 93. Apple's "Large System Font" is REALbasic's System Font.



Users who are running Kaleidoscope or are using alternate Apple themes can use any installed font as their System font.

If you want text to be displayed or printed in the user's System font, use the name "System" as the font when you assign it. This name doesn't appear in REALbasic's Font menu but you can enter it as the TextFont property in the Properties window.

What Fonts Are Available?

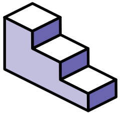
You may want to use fonts other than the System font. In this case you will need to determine if a particular font is installed on the user's computer. REALbasic has two global functions, FontCount and Font, that make determining available fonts easy. The following function, when passed a font name, will return True or False to inform you if the font passed is installed:

```
Function FontAvailable(FontName as String) As
Boolean
Dim i,nFonts as Integer
```

```
nFonts=FontCount-1
For i=0 to nFonts
  If Font(i)=FontName Then
    Return True
  End If
Next
Return False
End Function
```

The following code can be used in the Open event handler of a PopupMenu or ListBox to build a list of all available fonts:

```
Dim i,nFonts as Integer
nFonts=FontCount-1
For i=0 to nFonts
  Me.AddRow Font(i)
Next
```



To add a Font menu to your application, do this:

1. Add a menu with the name “Font” and set the Text property to “Font”.
2. Add an item to the Font menu set the Text property to “FontName”. REALbasic will automatically name the new item “FontFontName”.
3. Set the Index property of the menu item to 0 (zero).
4. If you don’t have a class based on “Application,” add a new class to your project, name the class “App” and set its Super property to “Application”.
5. Put the following code in the Open event handler of the App class:

```
Dim m as MenuItem
Dim i,nFonts as Integer
nFonts=FontCount-1
FontFontName(0).text=Font(0)
For i=1 to nFonts
  m=New FontFontName
```

```
  m.text=font(i)
next
```

All of these menu items will share one menu handler. This menu handler will be passed an Index parameter which will indicate which menu item as passed. This Index parameter can be used in conjunction with the Font function to determine which font was selected.

Working with the Selected Text

The "Selected Text" refers to text that is selected (or "highlighted") in the EditField that currently has the focus. EditFields have three properties that can be used to get and/or set the selected text.

TABLE 48. **EditField properties for getting and setting selected text**

Name	Description
SelLength	The number of characters currently selected. You can change the selected text by changing this number. Setting this value to 0 (zero) will position the cursor based on the value in the SelStart property rather than selecting any text.
SelStart	The number of the character just before the selected text. For example, if the fifth character in an EditField was selected, this property would be 4. Setting this value to 0 (zero) will start the selection at the beginning of the EditField.
SelText	A string containing all of the selected text. Changing this value will replace the selected text with the SelText value. If no text is selected, the SelText value will be inserted at the insertion point (the value in SelStart).

This code selects all the text in the EditField that currently has the focus:

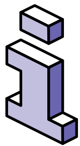
```
EditField1.SelStart=0
```

```
EditField1.SelLength=Len(EditField1.Text)
```

If you need to execute some code when the user moves the cursor or highlights some characters, place your code in the SelChange event handler of the EditField.

Creating a Password Field

EditFields have Password and LimitText properties that can be used to create password fields. When you set the Password property, bullet characters (Option-8) appear instead of the characters you type. However, the characters you enter are placed in the EditField's Text property. The LimitText property allows you to control the maximum number of characters the user can type in the EditField.



The Password property will function only if the MultiLine property is False (not checked).

Handling Styled Text

The term *styled text* means text that can have more than one font, font size, and/or font style. In order for an EditField to display styled text, its MultiLine property must be True (checked) and its Styled property must be True (checked). In order to print styled text, you must use the StyledTextPrinter class. See the section "Printing Styled Text" on page 267 for more information.

Determining the Font, Size, and Style of Text

EditFields have properties that make it easy to determine the font, font size, and font style of the selected text in an EditField. The SelTextFont property can be used to determine the font of the selected text. If the selected text has only one font, the SelTextFont property contains the name of that font. If the selected text uses more than one font, the SelTextFont property is empty.

This function returns the names of fonts for the selected text of the EditField passed:

```
Function Fonts(item as EditField) as String
  Dim fonts, theFont as String
  Dim i, Start, Length as Integer
  If Field.SelTextFont="" Then
    Start=Field.SelStart
    Length=Field.SelLength
    For i=Start to Start+Length
      Field.SelStart=i
      Field.SelLength=1
      If InStr(fonts,Field.SelTextFont)=0 Then
        If fonts="" Then
          fonts=Field.SelTextFont
        Else
          fonts=fonts+", "+Field.SelTextFont
        End if
      End if
    Next
    Return fonts
  Else
    Return Field.SelTextFont
  End If
End Function
```

The SelTextSize property is used to determine the font size of the selected text and works the same way as the SelTextFont property. If all characters of the selected text are the same font size, the SelTextSize property will contain that size. If different sizes are used, the SelTextSize property will be 0.

There are also boolean properties for determining if all of the characters in the selected text are the same font style. Since text can have multiple styles applied to it, these properties determine if all of the characters in the selected text have a particular font style applied to them. For example, if all of the characters in the selected text are bold but some are also italic, a test for bold returns True. On the other hand, a test for italic returns False since some of the selected text is not in the italic font style. For all of these properties, you test to see if the property is True or False. The test returns True, then all of the characters in the selected text have that font style. If it returns False, the selected text contains more than one font style. If you want to determine which styles are in use, you can programmatically select each character in the selected text and then test the style properties. This is an operation similar to the sample Fonts function that determines which fonts are in use in the selected text. The properties for testing the various available font styles are:

TABLE 49. **Font Style Properties**[a]

Property	**Style**
SelBold	Bold
SelItalic	Italic
SelUnderline	Underline
SelOutline	Outline
SelShadow	Shadow
SelCondense	Condensed
SelExtend	Extended

a. Outline, Shadow, Condense, and Extend are supported only on Macintosh.

In this example, if the selected text of the EditField is bold, then the Bold menu item is checked:

```
StyleBold.Checked=EditField1.SelBold
```

If all of the characters in the selected text are not bold then EditField1.SelBold returns False which will then be assigned to the Checked property of the StyleBold menu item.

Setting the Font, Size, and Style of Text

The properties used to check the font, font size, and font styles of the selected text are also used to set these values. For example, to set the font of the selected text to Helvetica, you do the following:

```
Editfield1.SelTextFont="Helvetica"
```

Keep in mind when setting fonts that the font must be installed on the user's computer or the assignment will have no effect. You can use the FontAvailable function mentioned earlier in this chapter to determine if a particular font is installed.

You can set the font size of the selected text using the SelTextSize property. For example, the following code sets the font size of Editfield1 to 12 point:

```
Editfield1.SelTextSize=12
```

To apply a particular font style to the selected text, set the appropriate style property to True. For example, the following code applies the Bold style to the selected text in Editfield1:

```
Editfield1.SelBold=True
```

Table 49 on page 236 lists all the font style properties of EditFields that can be used in this same way.

EditFields also have built-in methods for toggling the font styles on and off. “Toggling” in this case means applying the style if some of the selected text doesn’t have the style already applied or removing the style from any of the selected text that already has it applied. The following code toggles the bold style of the selected text in EditField1:

```
Editfield1.ToggleSelectionBold
```

The methods for toggling the styles of the selected text are shown in Table 50 on page 238.

TABLE 50. **EditField control methods for toggling selected text styles**

Method Name	Style[a]
ToggleSelectionBold	Bold
ToggleSelectionItalic	Italic
ToggleSelectionUnderline	Underline
ToggleSelectionOutline	Outline
ToggleSelectionShadow	Shadow
ToggleSelectionCondense	Condensed
ToggleSelectionExtend	Extended

a. Outline, Shadow, Condense, and Extend are supported only on Macintosh.

Formatting Numbers, Dates, and Times

REALbasic provides the ability to display and print numbers, dates, and times in many different formats.

Numbers

Numbers are stored unformatted. Fortunately, REALbasic provides a Format function that makes providing formatting to numbers easy. To use this function, pass it a format specification and the number you wish formatted. The Format function then returns a string that represents the number with the formatting applied to it. The syntax for the Format function is:

result=Format(*Number*, *FormatSpec*)

The FormatSpec is a string made up of one or more characters that control how the number will be formatted. For example, the format spec "$###,##0.00" applies the typical dollars and cents formatting used in the United States.

TABLE 51. **Formatting characters used with the Format function**

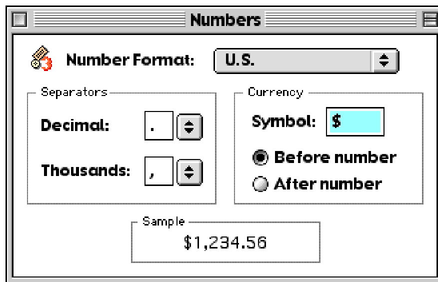
Character	Description
#	Placeholder that displays the digit from the value if it's present.
0	Placeholder that displays the digit from the value if it's present. If no digit is present, 0 (zero) is displayed in its place.
.	Placeholder for the position of the decimal point.
,	Placeholder that indicates that the number should be formatted with thousands separators.
%	Displays the number multiplied by 100.
(	Displays an open paren.

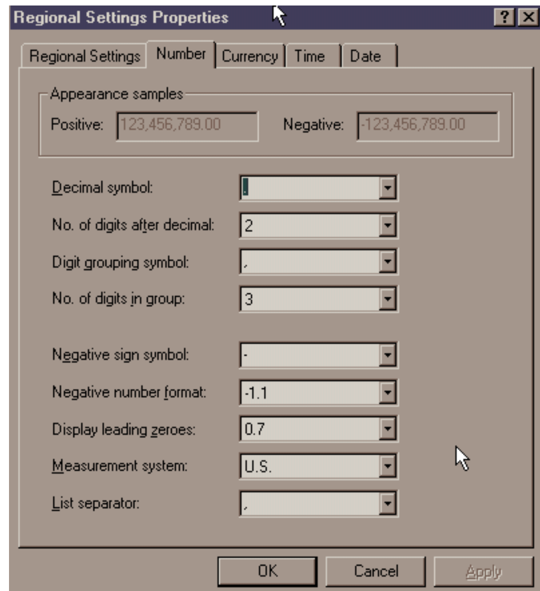
TABLE 51. **Formatting characters used with the Format function**

Character	Description
)	Displays a closing paren.
+	Displays a plus sign to the left of the number if the number is positive or a minus sign if the number is negative.
–	Displays a minus sign to the left of the number if the number is negative. There is no effect for positive numbers.
E or e	Displays the number is scientific notation.
\character	Displays the character that follows the backslash.

On Macintosh, the character that is actually used as the Decimal and Thousands separator is specified by the user in the Numbers Control Panel. On Windows, the task is handled in the Regional Settings Control Panel

FIGURE 94. **The Numbers and Regional Settings Control Panels**





By default, the FormatSpec applies to all numbers. If you want to specify different FormatSpecs for postive numbers, negative numbers, and zero, simply separate the formats with semi-colons within the FormatSpec. The last three examples in Table 52 on page 241 show this. It shows some examples of FormatSpecs:

TABLE 52. **Examples of FormatSpecs of the Format function**

Format Syntax	Result
Format(1.784, "#.##")	1.78
Format(1.3, "#.0000")	1.3000
Format(5, "0000")	0005
Format(.25, "#%")	25%
Format(145678.5, "#.##")	145,678.5
Format(145678.5, "#.##e+")	146e+5
Format(-3.7, "-#.##")	-3.7
Format(3.7, "+#.##")	+3.7
Format(3.7, "#.##; (#.##); \z\e\r\o")	3.7
Format(-3.7, "#.##; (#.##); \z\e\r\o")	(3.7)
Format(0, "#.##; (#.##); \z\e\r\o")	zero

Dates

Dates are objects and have properties that hold the date in various different formats. To get a date as a string formatted in a specific way, you simply access the appropriate property. Table 53 on page 241 lists the properties of date objects and an example of the format the property contains:

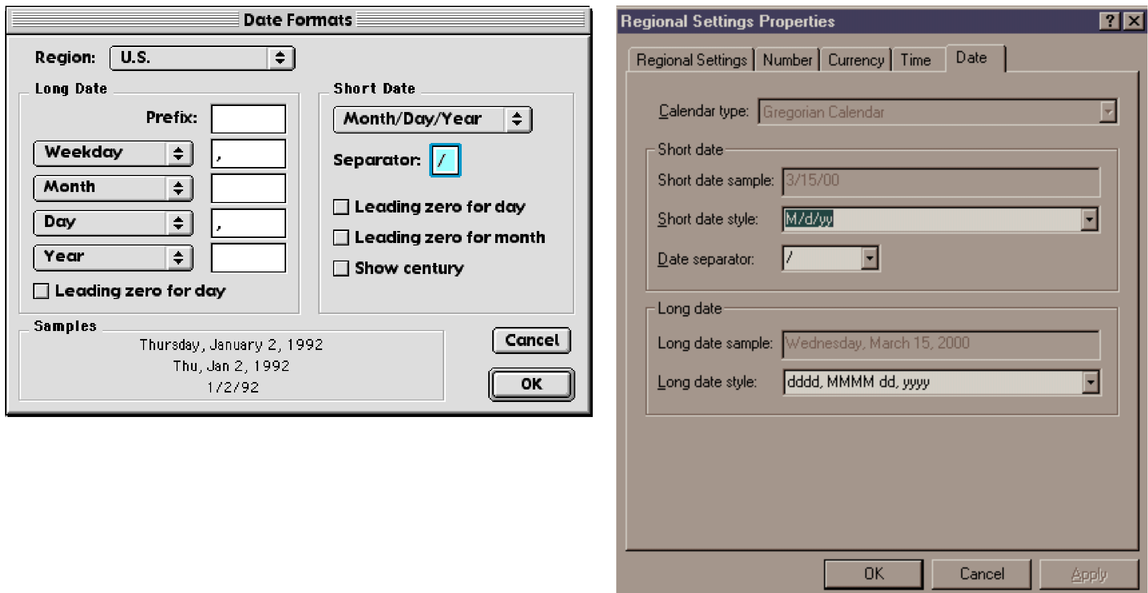
TABLE 53. **Date format properties**

Property	Example
ShortDate	12/31/97
LongDate	Wednesday, December 31, 1997
AbbreviatedDate	Wed, Dec 31, 1997

Several aspects of the Long and Short date formats are controlled by the user's Date Formats (Macintosh) or Date Properties (Windows) setttings. The Date Formats dialog is accessed on Macintosh from the Date and Time control panel and Date Properties is a screen in the Regional Settings control panel on Windows. Users can choose the order of the day, month, year, as well as the separators.

These screens are shown in Figure 95 on page 242.

FIGURE 95. **The Date Formats and Date Properties screens.**



To get the current date in any of these formats, simply create a date object and then access the appropriate property. In this example, the current date formatted as a long date, is assigned to a variable:

```
Dim today as Date
```

```
Dim theDate as String
today=new Date
theDate=today.LongDate
```

Times

Time values are stored as part of a date. Date objects have two properties that store time values in two different formats. Table 54 lists the two properties and shows examples of how the time is returned.

TABLE 54. **Time formats**

Property	Example
ShortTime	2:32 PM
LongTime	2:32:34 PM

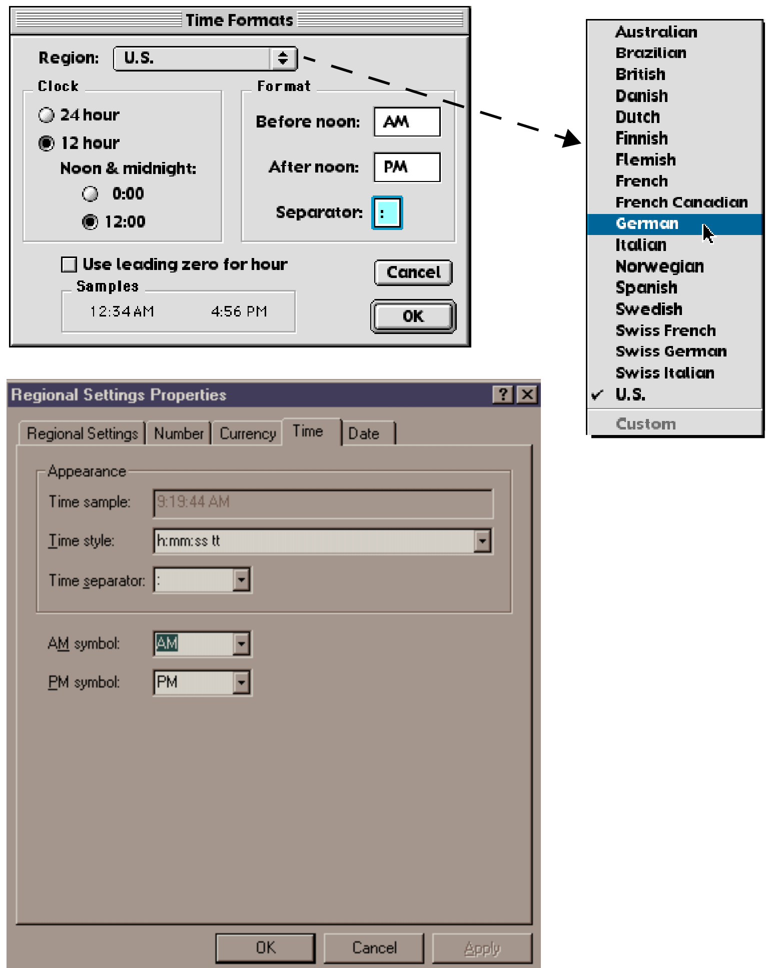
To get the current time in either of these formats, create a date object and then access the appropriate property. In this example, the current time formatted as a LongTime, is assigned to a variable:

```
Dim today as Date
Dim Now as String
today=new Date
Now=today.LongTime
```

As is the case with date formats, several aspects of the Short Time and Long Time formats are controlled by the user via the Time Formats dialog box (Macintosh) or the Time screen in the Regional Settings Control panel (Windows).

These screens are shown in Figure 96 on page 244.

FIGURE 96. The Time Formats and Time Properties screens.



Adding Pictures and Drawing Graphics

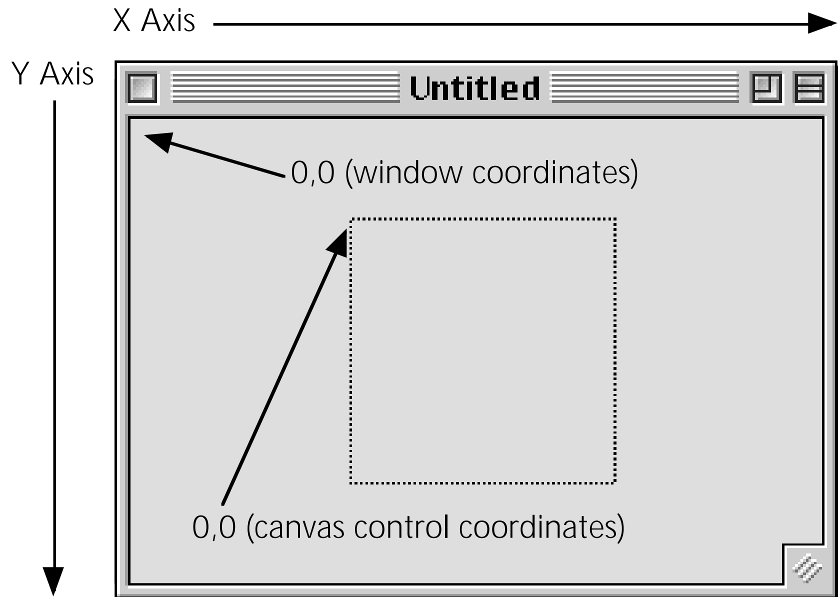
You can add pictures from documents or draw your own pictures in REALbasic. In some cases you can add the graphics you want without writing any code. When you do need to write code, REALbasic provides methods for creating all kinds of graphics.

Understanding the Coordinates System

Most of the graphics methods require you to indicate the location inside the window or within a Canvas control where you wish to begin drawing. This location is specified using the coordinates system. This system is a grid of invisible horizontal and vertical lines that are 1 pixel apart. If you have never done a computer drawing with a coordinates system, you might expect the origin (0,0) to be in the center of the window, but it's not. The origin is always in the upper-left corner of the area. For the entire screen, this is the upper-left corner of the screen. For a window, the origin is the upper-left corner of the window, and for a control, it's the upper-left corner of the control. The X axis (the horizontal axis) increases in value moving from left to right and the Y axis (the vertical axis) increases in value moving from top to bottom.

So, a point that at 10, 20 (within a window) is 10 pixels from the left side of the window and 20 pixels from the top of the window. If you are working within a Canvas control, the point 10, 20 is 10 pixels from the left edge of the control and 20 pixels down from the top edge of the control.

FIGURE 97. **The X,Y Coordinates System**



Displaying Pictures In a Window

There are different techniques you use to display pictures in a window. The technique you use depends on what you plan to do with the picture.

Using the Entire Window

If you want to use a window to display a picture, the window's Backdrop property is one way to do it. The Backdrop property is a picture that will be displayed behind any controls in the window. By default, the Backdrop is set to " None" meaning that no Backdrop picture will be displayed.

There are several ways to assign a picture to a window's BackDrop property. The most direct and intuitive way is to simply

drag a picture document from the Finder to the window in the Design environment. When you do so, several things happen: The picture appears immediately in the window in the Design environment, the name of the picture document is added to the Project window, and it is assigned to the BackDrop property in the window's Properties window.

You can also assign a picture to a window's Backdrop property by dragging a picture document into your Project window and then choosing it by name as the picture for the Backdrop property in the Properties window.

In addition, you can set the Backdrop property at runtime by assigning a picture in your Project to the Backdrop property, by loading a picture via code, or creating a new picture using the Graphics class drawing methods. This example presents the standard open file dialog box and lets the user choose a PICT, JPEG, or GIF file to be used as the backdrop of the current window[1]:

```
Dim f as FolderItem
f=GetOpenFolderItem("image/gif;image/
jpeg;image/x-pict")
If f<> Nil Then
  Backdrop=f.OpenAsPicture
End If
```

After you have assigned a picture to the BackDrop property, you can then resize the window to the size of the picture by setting the window's width and height properties to the backdrop's width and height properties:

```
width=Backdrop.width
```

1. The file types used as parameters in the GetFolderItem call must be defined in the File Types dialog box.

```
height=Backdrop.height
```

You don’t need to worry about redrawing the Backdrop. REALbasic will handle redrawing the Backdrop when necessary.

Using a Portion of the Window

If you only want to display the picture in an area in the window, you can use an ImageWell control. An ImageWell is similar to a Canvas control, except that it has no drawing tools: you can only display an image that has been created elsewhere.

To assign a picture to an ImageWell's Image property, simply drag it from the Finder to the Project window and then assign it to the ImageWell’s Image property using its Properties window.

You can also add a picture at runtime by loading an image using code. For example, the following code displays an open-file dialog box that allows the user to choose a PICT, JPEG, or GIF file and display it in the ImageWell. It is assumed that the file types used as parameters in GetOpenFolderItem have been assigned in the File Types dialog box.

```
dim f as FolderItem
f=GetOpenFolderItem("image/x-pict;image/
jpeg;image/gif")
if f <> Nil then
 ImageWell1.Image=f.OpenAsPicture
end if
```

You can also allow your users to drag a picture document from the Finder to the ImageWell rather than using the open-file dialog box. In the ImageWell's Open event handler, allow a file drop using the lines:

```
me.acceptfileDrop("image/x-pict")
```

```
me.acceptfileDrop("image/jpeg")
me.acceptfileDrop("image/gif")
```

In the ImageWell's DropObject event handler, use the code

```
Sub (DropObject (Obj as DragItem)
 If Obj.FolderItemAvailable then
  Me.Image=Obj.FolderItem.OpenAsPicture
 End if
End Sub
```

Note: The code will only support PICT files unless QuickTime is installed on the user's computer.

Your other option is to use a Canvas control to display a picture in a portion of the window. This type of control gives you a graphics area that can be drawn in and also receives events. You might use a Canvas control if you need to display a picture that the user will interact with. With the Canvas control's Backdrop property you can display an existing picture. This can be done manually in the Design environment by clicking on the Canvas control in a window to select it and then choosing a picture from your Project window from the Backdrop property's pop-up menu in the Properties window. A picture can also be assigned to the Backdrop property at runtime. This example displays an open-file dialog box when the user clicks on the Canvas control and then lets the user choose a picture to be displayed in the Canvas control:

```
Dim f as FolderItem
f=GetOpenFolderItem("image/x-pict")
If f<> Nil Then
 Me.Backdrop=f.OpenAsPicture
End If
```

You can also support drag and drop to a Canvas control in the same manner as for ImageWells. The advantage of using a Canvas control is that you can also customize the image using the methods of the Graphics class using the Canvas control's Paint event handler. This feature is described in the following section.

Creating Pictures

You can create pictures programmatically using the methods of the Graphics class. A Graphics object is simply an object in memory that holds an image. For example, Windows and Canvas controls have a Paint event. This event is executed any time the Window or Canvas control needs to be redrawn. For example, when a window opens, its Paint event is executed because the contents of the window needs to be drawn. Any Canvas controls in a window will also execute their Paint event when the window opens because the Canvas control needs to be drawn. These Paint events are also executed when a portion of the window and/or Canvas control that was previously hidden by another window is exposed.

The Paint event is passed a Graphics object. When the Paint event is finished executing, this graphics object will be drawn in the window or Canvas control. You draw in a window or Canvas control by calling the drawing methods of this graphics object.

Displaying Pictures

You can display a picture in a graphics object using the DrawPicture method of the Graphics class. This method is passed a picture and the coordinates that describe where you want the picture drawn within the graphics object. This example uses the Paint event to draw two pictures that have been dragged into the Project window (BartPict and LisaPict) side by side:

```
Sub Paint(g As Graphics)
  g.DrawPicture BartPict, 0,0
  g.DrawPicture LisaPict,BartPict.Width, 0
End Sub
```

Copying A Portion of a Picture

The DrawPicture method of the Graphics class can be used to copy a portion of a picture to a Graphics object. This is done using the optional parameters of the DrawPicture method. These parameters allow you to specify the portion of the picture you want to draw. You can specify the coordinates where you wish to begin copying from the picture as well as the amount (in width and height) you wish to copy.

This example draws a 20 pixel square portion of the source picture starting 10 pixels from the left and 10 pixels from the top of the source picture and drawing the picture 5 pixels from the left and 5 pixels from the top of the Canvas control or window background:

```
Sub Paint(g As Graphics)
  g.DrawPicture
Lisa,5,5,Lisa.width,Lisa.height,10,10,20,20
End Sub
```

Scaling Pictures

The DrawPicture method of the Graphics class can scale a picture when it is drawn. To do this, you must include all of the DrawPicture parameters. Scaling is done by specifying a destination width and/or height that is larger or smaller than the picture's original width and/or height. This example draws a picture at two times its original size:

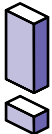
```
Sub Paint(g As Graphics)
```

```
  Dim w,l as integer
  w=lisa.width
  l=lisa.height
  g.DrawPicture lisa, 0,0,w*2,l*2,0,0,
lisa.width, lisa.height
End Sub
```

Drawing Standard Dialog Icons

REALbasic has a MsgBox method for displaying a standard message box with an note icon and an OK button. However, there may be times when this isn't appropriate. For example, the note icon is appropriate when you need to inform the user about something that isn't a warning. If the user is about to do something where data loss could occur (like quitting the application without saving a changed document), then the caution icon is more appropriate. If the user has started an operation that cannot be completed (such as saving a document to a locked volume), the stop icon is more appropriate.

FIGURE 98. **Note, Caution, and Stop icons**



Note: The look of these three icons may be affected by the end-user's choice of Appearance Manager schemes or Kaleidoscope themes.

The Graphics class provides the DrawNoteIcon, DrawCautionIcon, and DrawStopIcon methods that make it easy to display these icons in a Canvas control or a window background. The advantage of using these methods is that these icons change between different versions of the operating system and on different platforms.

Using these methods, you will also been displaying the appropriate icon. This example draws the note icon in a Canvas control:

```
Sub Paint(g As Graphics)
  g.DrawNoteIcon 0,0
End Sub
```

Drawing Pixels

You can get and set the color of individual pixels in a Graphics object using the Pixel property. You use this property by passing it X and Y coordinates and then setting the color of that pixel to a color object or getting its color.

This example draws pixels at randomly selected coordindates within a Graphics object using randomly selected colors until the user presses ⌘-Period:

```
Sub Paint(g As Graphics)
  Dim c as Color
  Do
    c=Rgb(Rnd*255,Rnd*255,Rnd*255)
    g.Pixel(Rnd*me.Width,Rnd*me.Height)=c
  Loop until UserCancelled
End Sub
```

This example gets the color of the pixel the mouse is over in a Canvas control and fills another Canvas control called PixelColor with that color:

```
Sub MouseMove(X As Integer, Y As Integer)
  Dim c as Color
  c=Me.Graphics.Pixel(X,Y)
  PixelColor.Graphics.ForeColor=c
  PixelColor.Graphics.FillRect 0,0,Pixel-
Color.Width,PixelColor.Height
```

```
End Sub
```

Drawing Lines

Lines are drawn using the DrawLine method of the Graphics class. The color of the line is the color stored in the ForeColor property of the Graphics object the line is being drawn in. To use the DrawLine method, you pass it starting coordinates and ending coordinates of the line.

This example uses the DrawLine method to draw a grid inside a Canvas control or window background. The size of each box in the grid is defined by the value of the boxSize variable:

```
Sub Paint(g as Graphics)
  Dim i, boxSize as Integer
  boxSize=10
  For i=boxSize to Me.Width Step boxSize
    g.DrawLine i,0,i,Me.Height
  Next
  For i=boxSize to Me.Height Step boxSize
    g.DrawLine 0,i,Me.Width,i
  Next
End Sub
```

The thickness of the line is controlled by the PenHeight and PenWidth properties of the Graphics object.

Drawing Ovals

Ovals are drawn with the DrawOval and FillOval methods of the Graphics class. Both require the same parameters: the X and Y coordinates where the oval starts and the width and height of the oval. Both draw ovals using the ForeColor property of the Graphics object. Both use the PenWidth and PenHeight properties of the Graphics object to determine the line thickness. The

difference between the two is that DrawOval draws only the border of the oval, leaving the interior blank. FillOval draws an oval with the interior filled with the ForeColor.

This example draws an oval in a Canvas control or Window background:

```
Sub Paint(g as Graphics)
  g.DrawOval 0,0,50,75
End Sub
```

Drawing Rectangles

Rectangles are drawn using the DrawRect, FillRect, DrawRoundRect, and FillRoundRect methods of the Graphics class. All of these methods use the ForeColor property of the Graphics object and the PenWidth and PenHeight properties to determine the line thickness. All of these methods require the X and Y coordinates of the upper-left corner of the rectangle, as well as the width and height of the rectangle. RoundRectangles are rectangles with rounded corners. Therefore, DrawRoundRect and FillRoundRect require two additional parameters: the width and height of the curve of the corners.

DrawRect and DrawRoundRect both draw empty rectangles. FillRect and FillRoundRect draw solid rectangles.

Drawing Polygons

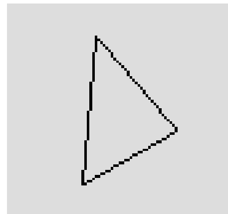
Polygons are drawn using the DrawPolygon and FillPolygon methods of the Graphics class. Polygons are drawn by passing the DrawPolygon or FillPolygon method an integer array that contains each point in the polygon. This is a 1-based array where odd numbered array elements contain X values and even numbered array elements contain Y coordinates. This means that element 1 contains the X coordinate of the first point in the

polygon and element 2 contains the Y coordinate of the first point in the polygon. Consider the following array values:

TABLE 55. **Array values for a polygon**

Element #	Value
1	10
2	5
3	40
4	40
5	5
6	60

When passed to the DrawPolygon or FillPolygon method, this array would draw a polygon by drawing a line starting at 10,5 and ending at 40,40 then drawing another line starting from 40,40 ending at 5,60 and finally a line from 5,60 back to 10,5 to complete the polygon. This polygon has only three sets of coordinates so it is a triangle.

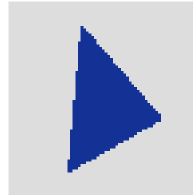


The code in the Canvas control or Window Paint event to draw this polygon, looks like this:

```
Sub Paint(g As Graphics)
  Dim points(6) as Integer
  points(1)=10
  points(2)=5
  points(3)=40
  points(4)=40
```

```
  points(5)=5
  points(6)=60
  g.DrawPolygon points
End Sub
```

FillPolygon draws the same polygon but with the interior filled with the ForeColor:



Creating Custom Controls with the Canvas Control

Visible controls (controls that have a graphical interface the user can interact with directly, like pushbuttons) are pictures that have code that controls how they are drawn. This means that a Canvas control can easily be used to create controls that are not built-in to REALbasic.

Suppose you wanted to create a simple custom control like a rectangle whose fill color toggles from black to white when clicked. First you would drag a Canvas control into a window. You want the rectangle to switch colors when the user clicks the mouse, so this code goes in the MouseDown event handler of the Canvas control. The code checks to see if the rectangle is white and, if it is, make fill it in black, otherwise fill it in white. You can check the color of any particular pixel using the Pixel property of the graphics property of the Canvas control. You can determine if a pixel is a particular color by comparing it to a color value returned by the Rgb function. Passing 0 (zero) to each of the parameters of the Rgb function returns the color white. Passing 255 to each parameter of the Rgb function returns the

color black. You will learn more about color later in this chapter. So, the code for the MouseDown event handler looks like this:

```
Function MouseDown(X As Integer, Y As Integer)
As Boolean
  If Me.Graphics.Pixel(X,Y)=Rgb(0,0,0) Then
    Me.Graphics.ForeColor=Rgb(255,255,255)
  Else
    Me.Graphics.ForeColor=Rgb(0,0,0)
  End If
  Me.Graphics.FillRect
     Me.Left,Me.Top,Me.Width,Me.Height
End Function
```

This code checks to see if the pixel the user clicked on is white and, if it is, the ForeColor property of the graphics object of the Canvas control (generically represented here using the Me function) is set to black, else it's set to white. Next, the FillRect method of the Graphics property of the Canvas control is called to fill the rectangle with the color stored in the ForeColor property.

There's one more step before our custom control is complete. If the Canvas control needs to be redrawn for some reason (such as when the window first opens or the user moves another window in front of the one with the Canvas control), REALbasic calls the Canvas control's Paint event handler to redraw the Canvas control. If there is no code in the Paint event handler, REALbasic won't draw the rectangle and, to the user it will seem to appear and disappear at different times, which will be confusing. To solve this problem, you need to put a slightly altered version of the code you have in the MouseDown event handler in the Paint event handler:

```
Sub Paint(g As Graphics)
  If g.Pixel(0,0)=Rgb(0,0,0) Then
```

```
    g.ForeColor=Rgb(255,255,255)
  Else
    g.ForeColor=Rgb(0,0,0)
  End If
  g.FillRect Me.Left,Me.Top,Me.Width,Me.Height
End Sub
```

Since the Paint event handler is passed a reference to the Graphics object of the Canvas (the g parameter), you can make the code a bit more generic and use "g" instead of "me.graphics". Also, since the user isn't clicking anywhere, you need to choose a pixel whose color you check. In this example we chose the pixel at 0,0.

This is an example of a very simple custom control. More complex and generic controls can be created using classes. See "Creating Custom Controls with Classes" on page 339 for more information.

Working With Color

Color in REALbasic is an object. A color can be specified using either the RGB, HSV, or CMY models. It has three properties which depend on the model you use. The RGB function, for example, specifies the amounts of red, green, and blue that make up the color. These values range from 0 to 255. The RGB function returns a Color object when passed values for the amount of red, green, and blue. Several classes have Color properties. For example, the ForeColor property of the Graphics class is a Color object.

If you need to store a Color, you can create a property or variable of type Color and then use the RGB, HSV, or CMY function. In

this example, a new variable of type Color is created and the values for the white are assigned using the Rgb function:

```
Dim c as Color
c=Rgb(255,255,255)
```

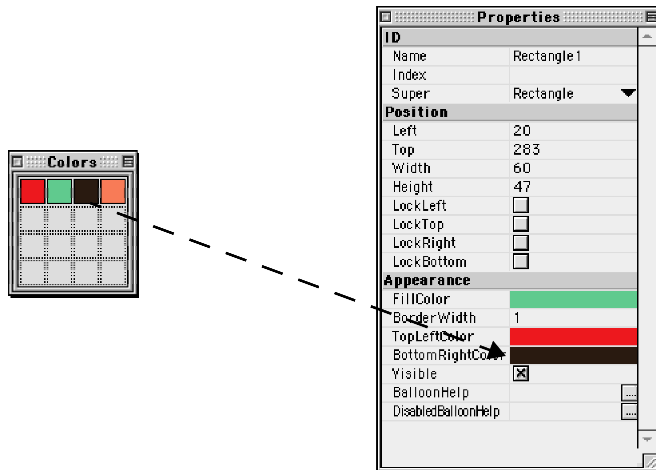
In this example, the ForeColor property of a Graphics object is set to blue so the text drawn will be in that color:

```
Sub Paint(g as Graphics)
  g.ForeColor=Rgb(9,13,80)
  g.DrawString "Hello World",50,50
End Sub
```

In the Design environment, you can create and assign colors using the Colors and Properties windows. The Colors window consists of a palette of up to 16 colors. When you first display the Colors window, no colors are assigned to it. To assign color to a palette element, click it to display the Apple Color Picker and make your selection. An advantage of this method is that you can make your choice visually, without having to know the RGB, HSV, or CMY values.

Once you have assigned colors to the Colors window, you can assign a color to an object property that accepts a color simply by dragging a color from the Colors window to the desired property in the Properties window. This is shown in Figure 99 on page 261.

FIGURE 99. **Assigning a color to an object property.**



Determining The RGB Values For A Color

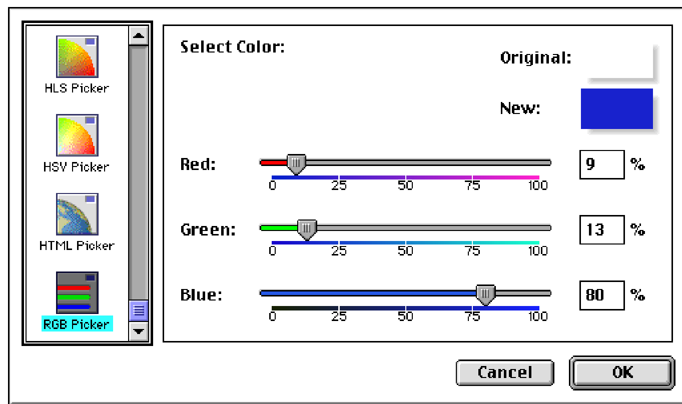
If you need to assign a color at runtime but aren't sure which RGB values to use to get a particular color, you can use the Mac OS Color Picker. The following code displays the Color Picker:

```
Dim c as Color
Dim b as Boolean
b=SelectColor(c,"Choose a color")
if b then //user chose a color
// do something with selected color here
end if
```

If the user cancelled out of the Color Picker dialog box, the boolean variable, b, is False; otherwise, the selected color is returned in the color object, c, and is available for assignment to a color property of an object.

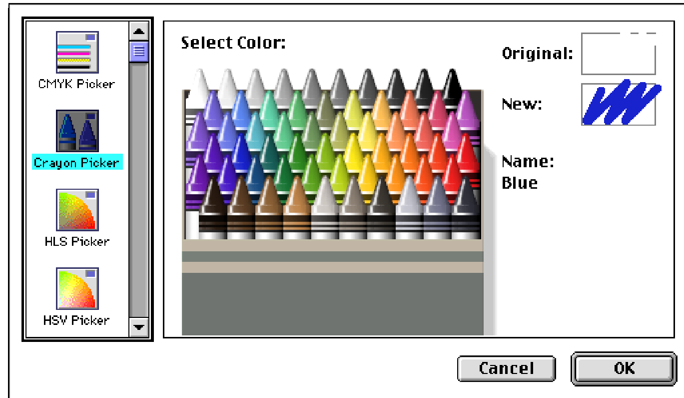
You may have already used the Color Picker to assign a color to a control's property in the Interface Builder. If you haven't, the Color Picker displays color and allows you to click on one to pick it (hence the name). Figure 100 on page 262 shows one panel in the Mac OS 8/9 Color Picker. If you are running System 7, the Color Picker looks a little different.

FIGURE 100. **The MacOS8 Color Picker.**



The System 7 Color Picker displays a large circle of color you can click on. The Color Picker included in Mac OS 8 and Mac OS 9 supports several color models. Figure 100 shows the RGB Color Picker panel that displays the percentage of red, green and blue for the selected color. Since Red, Green and Blue properties of a Color object in REALbasic are values between 0 and 255, you can convert the values from the RGB Color Picker into values you can use in REALbasic by multiplying 255 by the percentage shown. For example, in Figure 100, the red percentage is 9 so 255 x .09 equals 22.95, which rounds to 23. If you need a simpler way to choose a color than dragging the sliders back and forth, scroll up in the ListBox on the left and click on the Crayon Picker.

FIGURE 101. The Crayon Color Picker in MacOS8.



The Crayon Color Picker displays a box of crayons of commonly used colors. You can click on a crayon to select a color, then go back to the RGB Picker to look at the percentages of red, green, and blue and convert them to values between 0 and 255.

The Pixel Property of Graphics Objects

The Pixel property of a Graphics object lets you get and set the color of the pixel you specify. This property is an example of a property whose data type is Color. In this example, the Paint event handler is setting a pixel to black if it is white and white if it is black:

```
Sub Paint(g As Graphics)
  If g.Pixel(10,20)=Rgb(0,0,0) Then
    g.Pixel(10,20)=Rgb(255,255,255)
  Else
    g.Pixel(10,20)=Rgb(0,0,0)
End Sub
```

You can see that the code to check the color of a pixel and set the color of a pixel is basically the same.

Printing Text and Graphics

REALbasic provides a lot of flexibility when it comes to printing. You can display the Page Setup dialog box and store the settings the user chooses. You can choose to display the Print dialog box before printing.

Printing is almost exactly the same as drawing text and graphics into a Canvas control or the graphics property of a Window. When you call the OpenPrinter or OpenPrinterDialog function, a Graphics object is returned. To print, you simply draw your text and graphics into this Graphics object. To cause the page to print, you call the NextPage method of the Graphics object. This method forces the Graphics object to be printed, then clears it so you can use it again to draw the next page.

Working with the Page Setup Dialog Box

The PrinterSetup class lets you create an object that can be used to display the Page Setup dialog box, get and set the individual Page Setup settings, as well as store and restore these settings. To display the Page Setup dialog box, call the PageSetupDialog method of the PrinterSetup object you have instantiated. This method returns True if the user clicks the OK button in the Page Setup dialog box and False if he clicks the Cancel button. The PrinterSetup class has properties for accessing all of the settings in the Page Setup dialog box (page orientation, scale, etc.). For a list of PrinterSetup properties, see "PrinterSetup Class" on page 312 of the *Language Reference*. However, in most cases you won't have to deal with these properties because a compos-

ite version of these settings is stored in the SetupString property. The SetupString property is read/write and is used to get all of the PrinterSetup settings as string so you can store them and to restore that string later on. For example, in a document-based application, a string property could be added to the document window that stores the SetupString value. When the user chooses to display the Page Setup dialog box (in most applications by choosing Page Setup from the File menu), a PrinterSetup object is created and its SetupString property is assigned the value in the window property storing these settings. Then the Page Setup dialog box is displayed showing these settings. In this example, the window property is called "Settings":

```
Dim ps as PrinterSetup
ps=New PrinterSetup
ps.SetupString=Settings
If ps.PageSetupDialog Then
  Settings=ps.SetupString
End if
```

If the user clicks OK in the Page Setup dialog box, the window's Settings property is assigned the value of the SetupString because settings in the Page Setup dialog box may have been changed by the user.

PrinterSetup class objects can be optionally passed as a parameter to the OpenPrinter and OpenPrinterDialog functions so that the Page Setup settings can be used during printing.

If you wish to store the PrinterSetup's SetupString property with the document when the user saves the document (assuming you provide this capability), you will probably need to store it in a string resource in the resource fork of the document. See "Working With Macintosh Resources" on page 308 for more information on the resource fork.

Printing With The Print Dialog Box

You use the OpenPrinterDialog function to display the Print dialog box and print. If the user clicks the OK button in the Print dialog box, a Graphics class object is returned. If the user clicks the Cancel button, the Graphics object returned will be Nil. To create the first page to be printed, you utilize the Graphics object returned, calling the various Graphics class methods such as DrawString, DrawLine, DrawOval, DrawPicture, etc. Once you have created the page, you can send the page to the printer by calling the NextPage method of the Graphics class. This method will both send the page to the printer for printing and clear the Graphics object so you can begin creating the next page.

This example displays the Print dialog box then prints "Hello" on the first page and "World" on the second page:

```
Dim page as Graphics
page=OpenPrinterDialog()
If page<> nil Then
  page.DrawString "Hello", 50, 50
  page.NextPage
  page.DrawString "World", 50, 50
  page.NextPage
End
```

The Print dialog box page range is automatically supported. In the last example, if the user chooses to print pages 2 through 2, he will get only page 2.

If you are storing the SetupSting property of the a PrinterSetup class object, you can optionally pass this string to the Open-PrinterDialog function if you want it to consider the settings stored in the SetupString. This example assumes that the Setup-String is stored in a window property called "Settings" and

passes it to the OpenPrintertDialog function for consideration during printing:

```
Dim page as Graphics
Dim ps as PrinterSetup
ps=New PrinterSetup
If Settings <> "" Then
  ps.SetupString=Settings
End If
page=OpenPrinterDialog(ps)
If page <> nil Then
  page.DrawString "Hello", 50, 50
  page.NextPage
  page.DrawString "World", 50, 50
  page.NextPage
End If
```

For more information on the OpenPrinterDialog function, see “OpenPrinterDialog Function” on page 294 of the Language Reference.

Printing Without The Print Dialog Box

To print without displaying the Print dialog box, call the OpenPrinter function. This function is identical to the OpenPrintDialog function except that it doesn’t display the Print dialog box before printing. For information on printing, see “Printing With The Print Dialog Box” on page 266. For more information on the OpenPrinter function, see “OpenPrinter Function” on page 293 of the Language Reference.

Printing Styled Text

Because EditFields are capable of displaying styled text and multiple font sizes, you will usually want to retain the styled text in

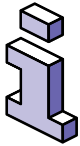
your reports. The StyledTextPrinter class supports this capability. It uses the DrawBlock method (rather than the DrawString method) to accomplish this. Here is a simple example that prints the contents of an EditField as styled text (The StyledTextPrinter method returns Nil unless the EditField's MultiLine property is set to True. Of course, the user cannot enter styled text into an EditField unless the MultiLine and Styled properties are True.).

```
dim stp as styledTextPrinter
dim g as graphics
g=openPrinterDialog()
if g <> nil then
 stp=EditField1.StyledTextPrinter(g,72*7.5)
 stp.drawBlock 0,0,72*9
end if
```

The parameters of DrawBlock are the top-left x, y coordinates on the page and the height of the block. This example starts at the top-left corner. See the description of the "StyledTextPrinter Class" on page 407 in the Language Reference for more information.

Transferring Text and Graphics with the Clipboard

The Clipboard is a class of object in REALbasic with properties and methods. The properties and methods let you determine what kind of data is available on the Clipboard, get data from the Clipboard, and send data to the Clipboard. The Clipboard class supports three kinds of data: text, picture, and binary. Binary data is represented in string form and is marked with a type you specify so you can tell what the binary data represents.



For EditFields, REALbasic handles the Cut, Copy, and Paste operations of the Edit menu automatically. However, for other controls that contain data such as Canvas and ListBox controls, this is not the case.

To access the Clipboard for any reason, you must first create a new object of type Clipboard:

```
Dim c as Clipboard
c=New Clipboard
```

In the event handler that opened the Clipboard, you must call the Clipboard object's Close method or an error may occur.

Testing The Clipboard For Specific Data Types

You can test the Clipboard using the following methods and properties all of which return True or False: TextAvailable, PictureAvailable, and MacDataAvailable. MacDataAvailable is used to determine if a specific kind of binary data (usually data put there by your application) is available. To use the MacDataAvailable method, you must pass it the MacType string that represents the type of data. This string was passed when the binary data was passed when the data was put on the Clipboard.

Getting Data From The Clipboard

Once you know what kind of data is available on the Clipboard, you can get the data using the Text, Picture, and MacData properties. In this example, if text is available, the text is placed in a variable called "Cliptext."

```
Dim c as Clipboard
Dim ClipText as String
c=New Clipboard
If c.TextAvailable Then
```

```
  ClipText=c.Text
End If
C.Close
```

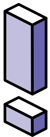
If a picture is available, the picture is placed in a variable called "ClipPict."

```
Dim c as Clipboard
Dim ClipPict as Picture
c=New Clipboard
If c.PictureAvailable Then
  ClipPict=c.Picture
End If
C.Close
```

In this example, rows from a ListBox that have been copied to the Clipboard are added to a ListBox:

```
dim theRows as string
dim c as clipboard
c=New Clipboard
If c.MacDataAvailable("rows") Then
  theRows=c.MacData("rows")
  Do
    Listbox1.AddRow
        Left(theRows,InStr(theRows,Chr(13))-1)

theRows=Mid(theRows,InStr(theRows,Chr(13))+1)
  Loop until theRows=""
End If
C.Close
```



Remember, you must call the Clipboard object's Close method in the event handler that opened the Clipboard or an error may occur.

Putting Data On The Clipboard

You can put text, picture, or binary data (in the form of a string) on the Clipboard. To do this, you create a new Clipboard object then use the appropriate method or property based on the type of data you wish to put on the Clipboard.

TABLE 56. **Methods or properties for putting data on the Clipboard**

Data Type	Method or Property
Text	SetText method
Picture	Picture property
Binary Data	AddMacData method

In this example, text is added to the Clipboard:

```
Dim c as clipboard
c=New Clipboard
c.SetText "Hello World"
c.Close
```

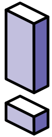
In this example, a picture from Canvas1 is copied to the Clipboard:

```
Dim c as Clipboard
c=New Clipboard
c.Picture=Canvas1.Backdrop
c.Close
```

In this example, rows from a ListBox are copied to the Clipboard. They are copied using the AddMacData method so they don't appear as text on the Clipboard:

```
Dim i as Integer
Dim c as Clipboard
Dim rows as String
```

```
c=New Clipboard
For i=0 to ListCount
  If Listbox1.Selected(i) Then
    rows=rows+Listbox1.List(i)+Chr(13)
  End If
Next
c.AddMacData rows,"rows"
c.Close
```



Remember, you must call the Clipboard object's Close method in the event handler that opened the Clipboard or an error may occur.

Creating Animation with Sprites

The SpriteSurface control is used to create animation where pictures can be moved around the screen with all redrawing handled automatically by the SpriteSurface control. Each picture is a Sprite object. Sprite objects have x and y properties that determine their current location on the screen when the SpriteSurface is running the sprite animation.

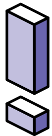
Causing Sprites to Move and Change Images

The NextFrame event handler is called each time the SpriteSurface is ready to draw the next frame of animation. If you want a Sprite to change position in the next frame, change its X and/or Y properties in the NextFrame event handler. If you want the Sprite's picture to change in the next frame of animation, change its Image property in the NextFrame event handler. To remove a Sprite from the animation, call the Sprite's Close method.

Frame Redrawing

The speed at which frames are redrawn is based on the FrameSpeed property. This property determines the number of times the monitor will refresh each second. This also determines the number of times per second that the NextFrame event handler will execute.

The FrameSpeed parameter is defined as the number of vertical retraces per frame. Zero is the fastest the computer can redraw. Compute FrameSpeed by dividing 60 by the number of frames per second you want and round to the next integer. Each frame will cause the NextFrame event to execute.



The definition of FrameSpeed has changed under version 2.x of REALbasic. Please update version 1 applications accordingly.

Starting and Stopping the Animation

To begin or continue the animation, call the SpriteSurface's Run method. To stop the animation, call the SpriteSurface's Close method. Once the Run method is called, NextFrame events will continue to be called until the user clicks the mouse button or the Close method of the SpriteSurface is called. If you want to prevent the SpriteSurface from closing when the user clicks the mouse button, set the CloseOnClick property of the SpriteSurface to False.

Sprite Surface Area

When the SpriteSurface's Run method is executed, the screen turns black, hiding the menu bar and all windows. While the screen is completely black, the total sprite area available is 640 by 480, by default, although you can change this with SpriteSurface properties. The SpriteSurface Backdrop property can hold an

image that is displayed when the animation begins. Because this is a picture, you can update it while the animation is running. However, you should avoid drawing into the active animation area. This area is controlled by the SurfaceWidth, SurfaceHeight, SurfaceLeft, and SurfaceTop properties.

Responding To The User During Sprite Animation

In the NextFrame event handler, you can use the SpriteSurface's KeyTest method to determine if the user is pressing a particular key. To use this method, you pass it a key code and the KeyTest method returns True if that key is being pressed and False if it's not. Key codes are not ASCII codes because some keys don't have ASCII codes (like the Shift, Command, and Option keys). Instead, key codes are special codes assigned to each key on the keyboard and they can vary for different keyboard configurations (i.e., between the English keyboard and the French keyboard). Figure 102 shows the English and French keyboards.

FIGURE 102. **Keycodes for use with the KeyTest method.**

CHAPTER 8

Working With Files

Many applications read from and/or write to files. Some create files that have their own special formats. Often this process starts with the user's selecting a file with the Open File dialog box or saving a file with the Save As dialog box. REALbasic makes it easy to use the Open and Save dialog boxes, as well as to read from and write to many different types of files.

Contents

- Understanding File Types
- Understanding FolderItems
- Accessing Files
- Working with Text and Binary Files
- Working with Pictures, Sounds, and QuickTime Files

- Reading and Writing to the Resource Fork
- Handling Files Double-Clicked At the Desktop

Understanding File Types

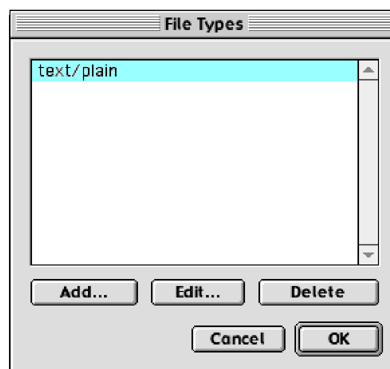
There are many different file types. The type of a file defines a unique type of data stored in that file. For example, a text file stores text while a PICT file stores pictures. Every file on your Macintosh computer has a four letter file type code and a four letter file creator code stored with it. For Windows users, files have a three letter suffix that defines the file type. The file type makes it easy for an application to know if it is prepared to deal with a particular file. For example, any application that can open text files expects the file type of any text file it shall open is "TEXT". This file type tells the application that this is a standard text file. PICT files are so named because "PICT" is the file type of a PICT file. Applications are also files but all applications have a file type of "APPL" which tells the Mac OS that this file is executable and not just data.

Rather than writing code that deals directly with all of these file types, creator codes, and file suffixes, REALbasic abstracts you and your code from them with *file types*. A file type in REALbasic is an item stored with your project that represents a specific file type, creator, and one or more suffixes. Each file type has a name that is used in your code when opening and creating files. This allows you to work with names you can choose and easily remember instead of cryptic codes. It also abstracts your code from the Mac OS, making it easier for you to create versions of your application for other operating systems when compilers for them are added to REALbasic.

Using The File Types Dialog Box

The File Types dialog box is used to create the items that will represent the different kinds of files you want your application to be able to open or create. You can access the File Types dialog box by choosing Edit ▶ File Types.

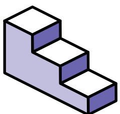
FIGURE 103. **The File Types dialog box**



This list displays any file types stored with the project. REALbasic creates a default file type called "Text" which is used when accessing text files. The File Types dialog box makes it easy to add, edit, and delete file types in your project.

Adding a File Type

REALbasic provides many File Type templates you can choose from. There's a good chance the file type you need to add to your project is already available in the File Type Templates pop-up menu.

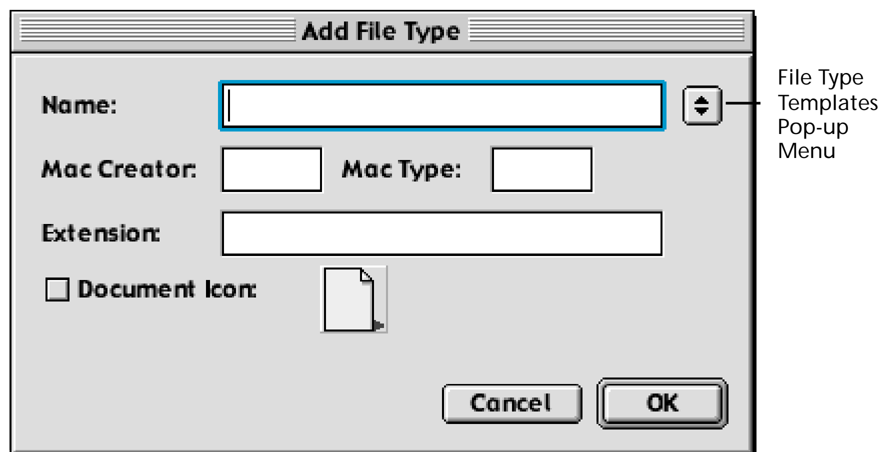


To add a file type, do this:

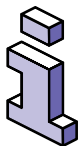
1. If the File Types dialog box is not already open, choose Edit ▶ File Types.
2. Click the Add button.

The Add File Type dialog box appears.

FIGURE 104. **The Add File Type dialog box**



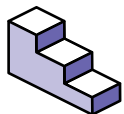
3. Choose a File Type template from the File Type Templates pop-up menu or enter the Name, Mac Creator, Mac Type, and any extensions.
4. If you are finished with the File Types dialog box, click the OK button to save any changes you have made.



Multiple extensions can be entered separated by semicolons.

Editing a File Type

Making changes to file types is easy.

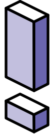


To edit a file type, do this:

1. If the File Types dialog box is not already open, choose Edit ▶ File Types.
2. Click on the file type you wish to edit to select it.
3. Click the Edit button.

   The Edit File Type dialog box appears.
4. Make any changes you wish and click the OK button.

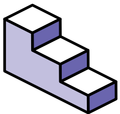
5. If you are finished with the File Types dialog box, click the OK button to save any changes you have made.



If you change the name of the file type, make sure you update any code that uses this file type. You can replace any occurrences of the old file type name with the new one easily using the Find/Change dialog box.

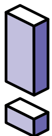
Deleting a File Type

Deleting a file type is simple.



To delete a file type, do this:

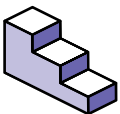
1. If the File Types dialog box is not already open, choose Edit ▶ File Types.
2. Click on the file type you wish to delete to select it.
3. Click the Delete button.
4. If you are finished with the File Types dialog box, click the OK button to save any changes you have made.



If you delete a file type, make sure you update any code that uses this file type.

Creating Custom File Types for Your Application

Most applications create files and assign custom icons to them. These icons usually look similar to the application's custom icon. This makes it easier for the user to recognize that the file goes with the application that produced it. Any custom icons you add will appear only if you have assigned a creator code to your project and built a stand-alone application.

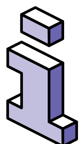


To add custom icons to any of the file types for your project, do this:

1. Choose Edit ▶ Project Settings.
2. Enter a four letter creator code that uniquely identifies your application.
3. Click the OK button.

4. Choose Edit ▶ File Types.
5. Add a new file type or edit an existing one.
6. Make sure the file type's Mac Creator exactly matches the one you assigned to your application in the Project Settings dialog box.
7. Make sure the Document Icon checkbox is selected.
8. Copy your custom icon to the Clipboard.
9. Click on the plain document icon in the File Types dialog box to select it.
10. Choose Edit ▶ Paste (⌘-V).
11. Click OK.

Once you have assigned custom icons and built a stand-alone application, you may need to rebuild the Finder's desktop before the Finder will display the icons. Rebuilding the desktop forces the Finder to update its icon database. You can rebuild the Finder's desktop by restarting your computer and holding down the ⌘ and Option keys until the computer asks you if you want to rebuild the desktop.



Creator codes are case sensitive and must be unique. You can register a unique creator code for your application with Apple Computer at their web site at http://developer.apple.com/dev/cftype/find.html.

Understanding FolderItems

To REALbasic, volumes, folders, applications, and documents are all considered to be *FolderItems*. A FolderItem is anything that can appear on the desktop. This doesn't mean that only items on the desktop are FolderItems. It means that if the item could be placed on the desktop, it's a FolderItem. For example, the Trash is a FolderItem because it appears on the desktop.

The FolderItem class is your first point of contact with any item on a disk you want to read from or write to. To read from a file, for example, you get a FolderItem that represents the file, then use various methods to read from the file via the FolderItem. There are many different ways to get a FolderItem object that represents a particular volume, folder, application, or document. You can present the user with an Open File or Save As dialog box, you can get the FolderItem at a specific path, or you can even get a FolderItem from another FolderItem.

FolderItems have properties that store the path to the item, the name of the item, the size of the item, its type. etc. FolderItems also have methods you can use to create files, open files, delete files, copy files, etc.

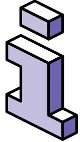
For detailed information on the properties and methods of the FolderItem Class, see "FolderItem Class" on page 141 of the *Language Reference*.

How Are Aliases Handled?

Aliases are files that actually represent a volume, application, folder, or file stored in another location and possibly under another name. Aliases were introduced in the Macintosh OS in System 7.0. REALbasic contains commands that allow you to either resolve the alias and work with the actual object or work with the object directly. The GetFolderItem function automatically resolves an alias when it encounters it, while the GetTrueFolderItem function works with the alias itself.

Getting a File at a Specific Location

If you know the full path to a file and you wish to access the file, you can do so using the GetFolderItem function. This function when passed the full path to volume, folder, application, or document, will return a FolderItem object that represents that item.



The "path" to a volume, folder, application, or document, is a string of characters that indicates the location of the file. A path starts with the volume name followed by the path delimiter character (a colon on the Macintosh), the name any folders in the path (each separated by the path delimiter) and ending with the name of the item. For example, say you had a document called "Schedule" stored in a folder called "Stuff" that was on a volume called "My Disk". The path to the document would look like this "My Disk:Stuff:Schedule".

The following code creates a FolderItem object in the local variable "f" that represents the document mentioned above:

```
Dim f as FolderItem
f=GetFolderItem("My Disk:Stuff:Schedule")
```

Once you have a FolderItem, you can (depending on what type of item it is) copy it, delete it, rename it, read from it or write to it, etc. You will learn how to read and write to files using FolderItems later in this chapter.

Verifying that you have accessed the Item

When you pass a full path to GetFolderItem, either of two things can go wrong. First, the path may be invalid. An invalid path contains a volume name and/or a directory name that doesn't even exist. For example, if the volume name is HardDisk instead of My Disk, the path used in the above example is invalid and GetFolderItem returns a Nil value in the FolderItem instance, f. If you try to use any of the FolderItem class's properties or methods on a Nil FolderItem, a NilObjectException will occur. If the

exception is not handled in some way, the application will crash. This is no good.

Second, the path may be valid, but the file you are trying to access may not exist. In that case, GetFolderItem returns a valid FolderItem representing the item, ready for your application to create it. If, however, your intention is to open an existing item, then you have to verify that the FolderItem exists before proceeding.

The following shell code checks for these two situations:

```
dim f as FolderItem
f=GetFolderItem(My Disk:Stuff:Schedule)
If f <> Nil then
 If f.Exists then
  //proceed to access the folderitem
 Else
  MsgBox "The document does not exist!"
 End if
Else
 MsgBox "You supplied an invalid path!"
End if
```

If the path is valid, the code checks the Exists property of the FolderItem to be sure that the file already exists; if the file doesn't exist or the path is invalid, a warning message is displayed.

You can also handle an invalid path using an Exception Block. They are discussed in the section "Runtime Exception Errors" on page 388.

Getting Information About a FolderItem

If GetFolderItem returns a valid FolderItem to an existing item, you can now get information about the FolderItem using the local variable "f". For example, you can get the modification date of the FolderItem. This example displays the modification date of the FolderItem above:

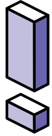
```
Dim f as FolderItem
f=GetFolderItem("My Disk:Stuff:Schedule")
if f <> Nil then
 if f.Exists then
  MsgBox f.ModificationDate.ShortDate
 End if
End if
```

Deleting A FolderItem

Once you have a FolderItem that represents an item that can be deleted, you can call the FolderItem's Delete method. The following example deletes the file represented by the FolderItem returned:

```
Dim f as FolderItem
f=GetFolderItem("My Disk:Stuff:Schedule")
If f <> nil Then
 if f.Exists then
  f.Delete
 End if
End if
```

If the FolderItem is locked, an error will occur. You can check to see if the FolderItem is locked by checking the FolderItem's Locked property.



Deleting a FolderItem does not simply move the FolderItem to the trash. The FolderItem is deleted permanently from the volume.

Getting The Path To Your Application's Folder

Passing a null string (two quotes with no characters in between them) to the GetFolderItem function returns a FolderItem representing the folder your application or project is in. You can then use the FolderItem's Item property to access all the items in the folder your application is in.

Getting Specific Items In the Application's Folder

If the first item in the path is not a volume, the GetFolderItem function assumes that the first item in the path is in the same folder as the application. If you are running your project in REALbasic, GetFolderItem looks for the item in the folder your project is in. If you haven't saved your project yet, GetFolderItem will look in the folder that REALbasic is in.

The following example returns a FolderItem that represents a file called "My Template" in a folder called "Templates" that is located in the same folder as the application:

```
Dim f as FolderItem
f=GetFolderItem(":Templates:My Template")
```

If the file is in the same folder as the application, then you don't need the leading colon. The following works:

```
Dim f as FolderItem
f=GetFolderItem("My Template")
```

Accessing Specific System/Finder Folders

REALbasic provides several functions that return FolderItems representing various folders that are part of the System software or the Finder. When you need to access one of these folders, use the appropriate function from the list below. These functions will still work properly even if the folder's name changes. They are also language independent. However, the TrashFolder and ControlPanelsFolder functions return Nil on Windows and others may return FolderItems that you aren't expecting if the application is run on Windows. For more information on these functions, see the Language Reference.

- AppleMenuFolder
- ControlPanelsFolder
- DesktopFolder
- ExtensionsFolder
- FontsFolder
- PreferencesFolder
- ShutDownItemsFolder
- StartupItemsFolder
- SystemFolder
- TemporaryFolder
- TrashFolder

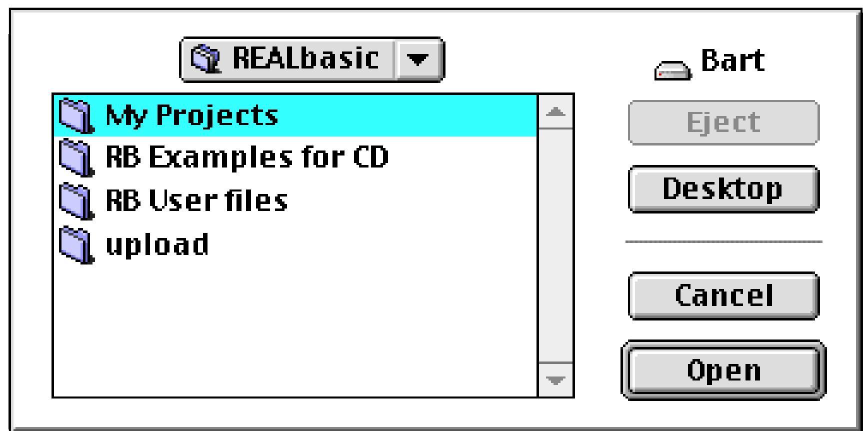
This example displays the number of items that are in the Trash:

```
Dim f as FolderItem
f=TrashFolder
MsgBox "Items in Trash: "+Str(f.Count)
```

Getting The Selected File From An Open File Dialog Box

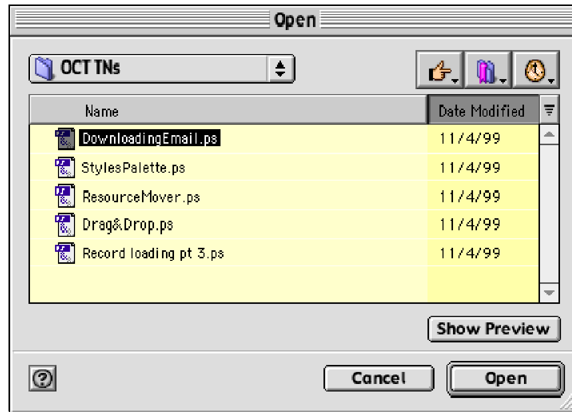
The Open File dialog box lets the user navigate to a particular location on any mounted volume and select a file to open. Figure 105 shows an example of the Open File dialog box.

FIGURE 105. The "classic" Open File dialog box.



If the user is running MacOS8.5 or above, the new Navigation Services open-file dialog box appears instead.

FIGURE 106. **The Navigation Services Open File dialog box.**



To present the user with an Open File dialog box, call the GetOpenFolderItem function. This function displays the Open File dialog box and returns a FolderItem object that represents the file the user selected. One or more file types (that have been defined in the File Types dialog box) must be passed to the GetOpenFolderItem function. It presents only those file types to the user in its browser. In this way, the user can only open files of the appropriate type. To pass more than one file type, separate them with semicolons.

The following example displays the Open File dialog box, allowing the user to select only jpeg files, and then displays the selected file's modification date:

```
Dim f as FolderItem
f=GetopenFolderItem("image/jpeg;image/x-pict")
MsgBox f.ModificationDate.ShortDate
```

If the user clicks the Cancel button rather than the Open button in the Open File dialog box, GetOpenFolderItem returns Nil. You will need to make sure the value returned is not Nil before using

it. If you don't, a NilObjectException error will be generated. The following example shows how the code from the previous example should be written to check for a Nil object:

```
Dim f as FolderItem
f=GetOpenFolderItem("image/jpeg;image/x-pict")
If f <> Nil Then
 MsgBox f.ModificationDate.ShortDate
End if
```

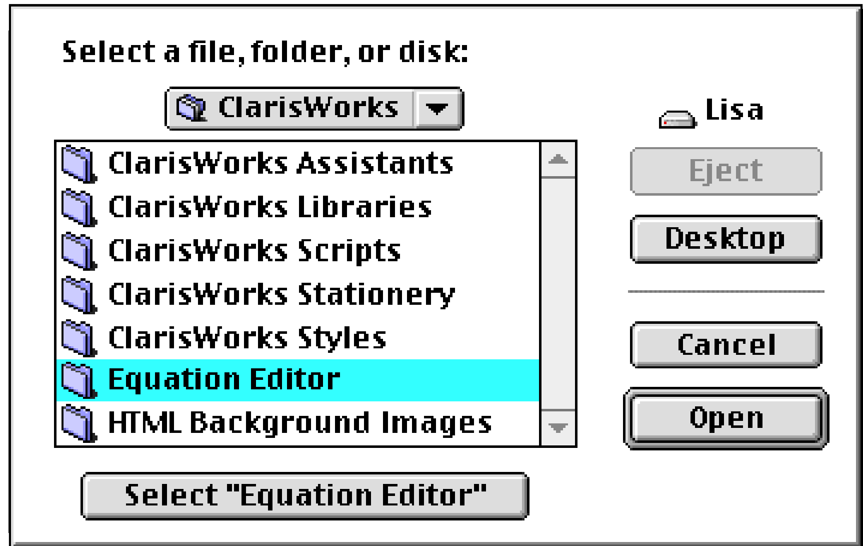
For more information, see "GetOpenFolderItem Function" on page 169 of the Language Reference.

For more information on file types, see "Understanding File Types" on page 278.

Getting The Selected Folder From An Open Folder Dialog Box

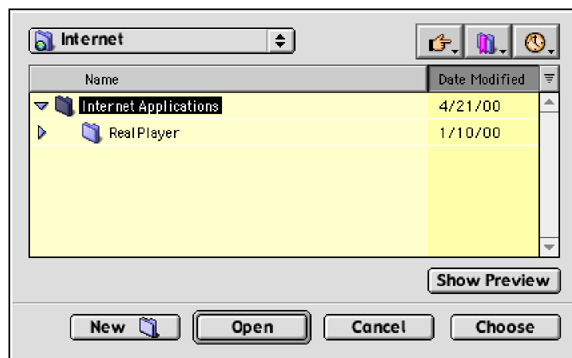
The Open File dialog box doesn't allow the user to select a folder. Fortunately, REALbasic's SelectFolder function displays an Open Folder dialog box that lets the user choose a folder rather than a file. Figure 107 on page 292 shows and example of this dialog box.

FIGURE 107. The "classic" Open Folder dialog box.



If the user is running MacOS8.5 (or above) the Navigation Service's Open Folder dialog box appears instead.

FIGURE 108. The Navigation Services Open Folder dialog box.



The SelectFolder function returns a FolderItem that represents the folder the user selects when he clicks the Select button at the

bottom of the dialog box (or the Choose button in the Navigation Services Open Folder dialog box). If the user clicks the Cancel button rather than the Select/Choose button, SelectFolder returns Nil. You need to check for it before using the returned value.

The following example displays the number of items in the folder selected by the user:

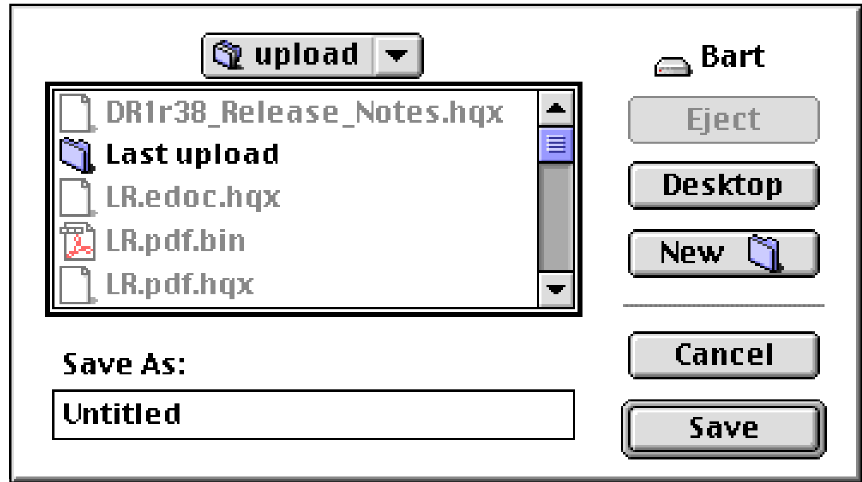
```
Dim f as FolderItem
f=SelectFolder
If f <> Nil Then
 MsgBox Str(f.Count)
End if
```

For more information, see "SelectFolder Function" on page 368 of the Language Reference.

Using the Save As Dialog Box

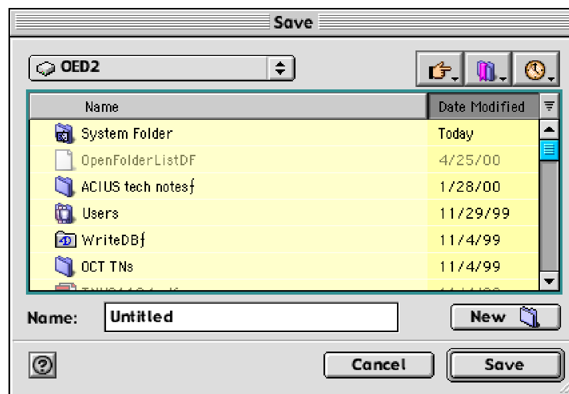
The Save As dialog box is used to let the user choose a location in which to save a file and give the file to be saved a name. Figure 109 on page 294 shows an example of the Save As dialog box.

FIGURE 109. **The “classic” Save As dialog box.**



If the user is running MacOS8.5 (or above), the Navigation Services' Save As dialog box appears instead.

FIGURE 110. **The Navigation Services Save As dialog box.**



REALbasic's GetSaveFolderItem function presents the Save As dialog box and returns a FolderItem that represents the file the

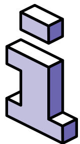
user wishes to save. This is an important distinction because the file doesn't exist yet. You must provide additional code that will create the file and write the data to the file. You will learn about creating files and writing data later in this chapter.

When you call the GetSaveFolderItem function, you define the type of file and the default name for the file (that will appear in the Name field in the Save As dialog box). The file type (which is the first parameter of the function) is the name of any file type defined for the project in the File Types dialog box.

Like the other functions that return FolderItems, you should make sure the FolderItem returned by GetSaveFolderItem is not Nil before using it (The FolderItem will be Nil if the user clicked Cancel).

The following example presents the Save As dialog box. The dialog presents a default file name of "Untitled". It also returns a FolderItem whose Type and Creator match the "image/x-pict" file type as defined for the project in the File Types dialog box. If the user clicks the Save button, the name the user chose for the file is displayed:

```
Dim f as FolderItem
f=GetSaveFolderItem("image/x-pict","Untitled")
If f <> Nil Then
  MsgBox f.name
End if
```



If you are going to create a text file with the FolderItem returned, you can pass an empty string as the first parameter of the GetSaveFolderItem function. The method that creates a text file (CreateTextFile) will assign the file type and creator automatically.

For more information on file types, see "Understanding File Types" on page 278.

For more information, see the “GetSaveFolderItem Function” on page 182 of the *Language Reference*.

Working With Text Files

Text files are files whose Type is “TEXT”. Text files can be read by text editors (like SimpleText and BBEdit) and word processors (like Microsoft Word). Text files can easily be created, read from, or written to with REALbasic. Text files are convenient since they can be read by many other applications.

Whether you are going to read from a text file or write to a text file, you must first have a FolderItem that represents the file you are going to read from or write to.

Reading From a Text File

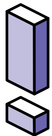
Once you have a FolderItem that represents an existing text file you wish to open, you open the file using the OpenAsTextFile method of the FolderItem. This method is a function that returns a “stream” that carries the text from the text file to your application. The stream is called a *TextInputStream*. This is a special class of object designed specifically for reading text from text files. You then use ReadAll or ReadLine methods of the TextInputStream to get the text from the text file. The TextInputStream keeps track of the last position in the file you read from.

The ReadAll method returns all the text from the file (via the TextInputStream) as a string. The ReadLine method returns the next line of text (the text after the last character read but before the next carriage return). As you read text, you can determine if you have reached the end of the file by checking the

TextInputStream's EOF (end of file) property. This property will be True once the end of the file is reached. When you are finished reading text from the file, call the TextInputStream's Close method to close the stream to the file, making the file available to be opened again.

This example lets the user choose a text file using the Open-file dialog box and displays the text in an EditField:

```
Dim f as FolderItem
Dim stream as TextInputStream
f=GetOpenFolderItem("text/plain")
If f<> Nil Then
  stream=f.OpenAsTextFile
  EditField1.text=stream.ReadAll()
  stream.Close
End if
```



Because ReadAll reads all of the text in the file, the resulting string will be as large as the file. Keep this in mind because reading a large file could require more memory than the user has allocated to your application.

This example reads the lines of text from a file stored in the Preferences folder in the System folder into a ListBox.

```
Dim f as FolderItem
Dim stream as TextInputStream
f = PreferencesFolder.child("My Apps Prefs")
if f <> Nil then
 stream = f.OpenAsTextFile
 While Not stream.EOF
  ListBox1.addrow stream.ReadLine
 Wend
 stream.Close
end if
```

Writing to a Text File

Once you have a FolderItem that represents the text file you wish to open and write to, you open the file using the AppendToTextFile method of the FolderItem. If you are creating a new text file or overwriting an existing text file, use the CreateTextFile method of the FolderItem. These methods are functions that return a "stream" that carries the text from your application to the text file. The stream is called a *TextOutputStream*. This is a special class of object designed specifically for writing text to text files. You then use the WriteLine method of the TextOutputStream to write the text to the text file. Text written to a text file is always appended to the end of the text file.

The WriteLine method, by default, adds a carriage return to the end of each line. This is controlled by the TextOutputstream's Delimiter property which can be changed to any other character.

When you are finished writing text to the file, call the TextOutputStream's Close method to close the stream to the file making the file available to be opened again.

This example displays the Save As dialog box then writes the contents of three EditFields to the text file and closes the stream.

```
Dim file As FolderItem
Dim fileStream As TextOutputStream
file=GetSaveFolderItem("plain/text","My Info")
fileStream=file.CreateTextFile
fileStream.WriteLine namefield.Text
fileStream.WriteLine addressfield.Text
fileStream.WriteLine phonefield.Text
fileStream.Close
```

Limitations of Text Files

Text files can only be accessed sequentially. This means that to read some text that is in the middle of the file, you must read all of the text that comes before it. It also means that to write some text to the middle of a text file, you have to write all of the text that comes before the text you wish to insert, then write the text you wish to insert, then the text that follows the text you wish to insert. You can not read text from a text file and write to the same text file at the same time. If these limitations are going to be a problem for your project, consider using a binary file instead. For more information on binary files, see "Working With Binary Files" on page 304.

Working With Styled Text Files

REALbasic makes it easy to read from and write to text files that support styled text. SimpleText is an example of an application that supports styled text.

Loading Styled Text Into an EditField

Once you have a FolderItem that represents the styled text file you wish to read text from, you can read the styled text using the OpenStyledEditField method of the FolderItem. To use this method, pass it the EditField you wish to display the styled text in. This EditField must have its Styled property set to True.

This example displays an Open File dialog box. It then reads the styled text from the file chosen and displays it in an EditField:

```
Dim f as FolderItem
f=GetOpenFolderItem("SimpleText Files")
```

```
If f <> Nil Then
 f.OpenStyledEditfield EditField1
End if
```

Writing Styled Text From an EditField to a File

Once you have a FolderItem that represents the styled text file to which you wish to open and write to, you can write the styled text using the SaveStyledEditField method of the FolderItem. To use this method, pass it the EditField from which you wish to get the styled text. This EditField must have its Styled and MultiLine properties set to True.

This example displays the Save As dialog box. It then writes the styled text from EditField1 to a new file:

```
Dim f as FolderItem
f=GetSaveFolderItem("plain/text","Untitled")
If f <> Nil Then
 f.SaveStyledEditField EditField1
End if
```

Working With Picture Files

REALbasic has built-in support for opening and saving PICT files. This is the most common Macintosh picture file format. Before opening or saving a PICT file, you must have a FolderItem that represents the PICT file you wish to work with. From there, you can open PICT files with the FolderItem's OpenAsPicture method and save a picture to the file with the SavePicture or SaveAsJPEG methods.

Saving Pictures

To save a picture to a PICT file, you need a FolderItem that represents a new PICT file or an existing PICT file. Next you call the FolderItem's SaveAsPicture method passing it the picture you wish to save. This example saves the backdrop of a Canvas control to a PICT file, the name of which is specified by the user in a Save As dialog box:

```
Dim f as FolderItem
f=GetSaveFolderItem("image/x-pict","Untitled")
If f <> Nil Then
 f.SaveAsPicture Canvas1.backdrop
End If
```

To save in JPEG format, simply substitute "image/jpeg" as the first parameter of GetSaveFolderItem.

Saving the image drawn into the graphics property of a Canvas control (perhaps by its Paint event handler) is a bit trickier. That's because the graphics property isn't a picture. The way to solve this is to add a picture property to the window. Any drawing you do in the Canvas control's graphics property should also be drawn into the picture property. The picture can then be saved using the SaveAsPicture method. The picture property you add to the window must be filled with a reference to a new picture before you attempt to write to it. This is accomplished using the NewPicture function in the window's Open event handler. In this example, the picture property (called "p") is set to a new picture:

```
p=newpicture(canvas1.width,canvas1.height,32)
```

In this example, the MouseDown event handler of the Canvas1 control draws a black pixel when the user clicks on the Canvas1 control. The drawing is also done to the window's p (picture) property:

```
Me.Graphics.Pixel(x,y)=Rgb(0,0,0)
p.Graphics.Pixel(x,y)=Rgb(0,0,0)
```

Finally, the picture property "p" can be saved to a picture file:

```
Dim f as FolderItem
f=GetSaveFolderItem("image/x-pict","Untitled")
If f <> Nil Then
 f.SaveAsPicture p
End If
```

Opening Pictures

To open a picture, you need a FolderItem that represents the PICT file you wish to open. Next, you call the FolderItem's OpenAsPicture method which returns the picture. This example displays the Open File dialog box that lets the user choose a PICT file which is then placed in the Backdrop property of a Canvas control:

```
Dim f as FolderItem
f=GetOpenFolderItem("image/x-pict")
If f <> Nil Then
 Canvas1.Backdrop=f.OpenAsPicture
End if
```

Working With Sound Files

REALbasic supports opening Macintosh sound files but not saving them. Specifically, Macintosh sound files are those files whose "Kind" field in the file's Get Info dialog box is listed as "Sound." To open a sound file, you must first have a FolderItem that represents the sound file you wish to open. Next, you can open

the sound file and place its contents into a Sound object with the FolderItem's OpenAsSound method. This example opens a sound file and plays it:

```
Dim f as FolderItem
Dim s as Sound
f=GetFolderItem("Doh!")
If f<> Nil Then
 s=f.OpenAsSound
 s.Play
End if
```

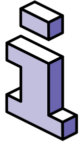
You can also get sounds stored in a snd resource inside your application. For more information, see "Supported Resource Types" on page 311.

Working With QuickTime Movie Files

Like sound files, REALbasic supports opening QuickTime movie files but not saving them. To open a QuickTime file, you must first have a FolderItem that represents the QuickTime file you wish to open. Next, you can open the QuickTime file and place its contents into a Movie object with the FolderItem's OpenAsMovie method. This example opens a QuickTime file, assigns its movie to the Movie property of a MoviePlayer control, and plays the movie:

```
Dim f as FolderItem
Dim m as Movie
f=GetOpenFolderItem("video/quicktime")
If f<> Nil Then
```

```
 m=f.OpenAsMovie
 moviePlayer1.Movie=m
 moviePlayer1.Play
End if
```



If your application needs a specific QuickTime movie, you can drag it into the Project window rather than use GetOpenFolderItem or GetFolderItem.

Working With Binary Files

Binary files are simply files that store values in their binary format rather than as text. For example, the number 30000 stored as text requires 5 characters of text (or bytes) to store in a text file. In a binary file, this number can be written as short integer (or just "short" or short). A short requires only 2 bytes.

Binary files also have the added benefit that you can read and write to a file without having to close the file in-between. For example, you can open a binary file, read some data, then write some data, and close it. You can also read and write anywhere in the file without having to read through all the data preceding the data you want.

Most applications store data in a binary format. The format is simply the arrangement of data within the file. In order to read a binary file, you must know how the data is arranged. If your own application created the file, you will know this, but if the file was created by an application you didn't write, you may not know it. Some formats are made public. For example, the PICT format is public. Other formats are not. Many software vendors do not publish the binary formats that their applications use to create documents.

BinaryStreams

Data read from or written to a binary file travels through a *BinaryStream*. A BinaryStream is a class of object in REALbasic that represents the flow of information between the a FolderItem and the file it represents. Unlike the TextInputStream class (which can only be used to read from a text file) and the TextOutputStream class (which can only be used to write data to a text file), BinaryStreams can be used for both reading data and writing data. You can even indicate to the BinaryStream that you will only be reading data from the file so that the file can continue to be available to other applications for writing.

BinaryStreams can read and write specific types of data, such as strings, short integers, long integers, and single bytes. They can also be used to read and write raw unformatted binary data.

Reading From a Binary File

Once you have a FolderItem that represents the file you wish to open, you open the file using the OpenAsBinaryFile method of the FolderItem. This method is a function that returns a BinaryStream. You then use Read, ReadByte, ReadLong, ReadPString and ReadShort methods to read data from the stream. The BinaryStream keeps track of the last position in the file you read from in its Position property. However, you can change this property to move the position to any location in the file.

This example presents the Open File dialog box, reads a file made up of strings, and displays those strings in a ListBox. Notice that since the code is only reading data and not writing, False is passed to the OpenAsBinaryFile method to indicate the file should be opened in "read-only" mode. Also, reading continues in a loop until the stream's EOF (end of file) property is True.

REALbasic will set the EOF property to True automatically once the end of the file is reached.

```
Dim f as FolderItem
Dim stream as BinaryStream
f=GetOpenFolderItem("myFileType")
If f<> Nil Then
 ListBox1.DeleteAllRows
 stream=f.OpenAsBinaryFile(False)
 do
 ListBox1.AddRow stream.ReadPString
 ListBox1.Cell(ListBox1.ListCount-1,1)=
         stream.ReadPString
 Loop Until stream.EOF
 stream.Close
End if
```

This code would run about 25% faster using a For...Next loop instead of a Do loop. However, the format of the file would have to be different because you need to know in advance how many rows of data to read in order to provide the ending value to the For loop. For example, if the first four bytes of the file format was a long integer that was the number of rows in the file, you could use that integer in your For loop. This is illustrated in the following example:

```
Dim f as FolderItem
Dim stream as BinaryStream
Dim count,i as Integer
f=GetOpenFolderItem("myFileType")
If f<> Nil Then
 ListBox1.DeleteAllRows
 stream=f.OpenAsBinaryFile(False)
 count=stream.ReadLong
 For i=1 to count
```

```
  Listbox1.AddRow stream.ReadPString
  Listbox1.Cell(ListBox1.Listcount-1,1)=
       stream.ReadPString
 Next
 stream.Close
End if
```

Writing to a Binary File

Once you have a FolderItem that represents the file you wish to open and write to, you can open the file using the OpenAsBinaryFile method of the FolderItem. If you are creating a new file, use the CreateBinaryFile method of the FolderItem. This method is a function that returns a BinaryStream. You then use Write, WriteByte, WriteLong, WritePString, and WriteShort methods to write data to the stream. The BinaryStream keeps track of the last position in the file you wrote to in its Position property. However, you can change this property to move the position to any location in the file.

When you are finished writing data to the file, call the BinaryStream's Close method to close the stream to the file making the file available to be opened again.

This example displays the Save As dialog box and writes the contents of two columns of a ListBox to the file and closes the stream. This code creates the file that is opened and read in the read binary file example that uses a For...Next loop.

```
Dim f as FolderItem
Dim i as Integer
Dim stream as BinaryStream
f=GetSaveFolderItem("myFileType","Untitled")
If f<> Nil Then
 stream=f.CreateBinaryFile("myFileType")
```

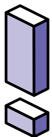
```
 stream.WriteLong ListBox1.ListCount
 For i=0 to ListBox1.Listcount-1
  stream.WritePString ListBox1.List(i)
  stream.WritePString ListBox1.Cell(i,1)
 Next
 stream.Close
End if
```

Working With Macintosh Resources

All Macintosh files (including applications, which are really just files) can have two sections called "forks." The "data" fork holds data that is in whatever format the application that created the file chose to put it in. The resource fork can contain formatted information such as icons, sounds, menu bars, pictures, string lists, etc.

REALbasic provides support for reading from and writing to the resource fork of a file. This is done using a FolderItem class object that represents the file whose resource fork you wish to access or create.

If you need more information on Macintosh resources, read *Inside Macintosh: Resources* published by Addison-Wesley.



REALbasic supports operations on the resource fork only on the Macintosh platform. The discussion and examples in this section assume that the application is running on a Macintosh. If your application manages resources, you can use the TargetWin32 or TargetMacOS constants to verify that the application is running on Macintosh before doing any resource fork operations. There is one exception to this: it is possible to install custom cursors in your application using the 'CURS' resource and access them from

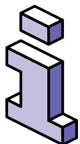
a built Windows application. This is described in the section "Custom Cursors in Windows Applications" on page 313.

Opening a File's Resource Fork

Once you have a FolderItem, you can open the resource fork for the file the FolderItem represents. This is done using the OpenResourceFork method of the FolderItem. This method returns a ResourceFork class object which can then be used to access the resource fork of the file. If the file has no resource fork, the OpenResourceFork method returns Nil.

This example displays the Open File dialog box, allowing the user to choose a file. It then reports if the file has no resource fork or tells the user how many different types of resources are in the file's resource fork:

```
Dim f as FolderItem
Dim rf as ResourceFork
f=GetOpenFolderItem("any")
If f <> Nil Then
 rf=f.OpenResourceFork
 If rf=Nil Then
  Beep
  MsgBox "This file has no resource fork."
 Else
  MsgBox "This file has "+Str(rf.TypeCount)+"
resource types."
 End if
End if
```



The "any" file type passed to GetOpenFolderItem was defined in the File Types dialog box and uses the string "????" as both Type and Creator. The "?" is a wildcard character, matching any type or creator code.

Adding a Resource Fork to a File

Before you can write to the resource fork of a file, it must have one first. You can use the FolderItem's OpenResourceFork method to determine if the file has a resource fork. If it doesn't, you can use the FolderItem's CreateResourceFork method to add a resource fork to the FolderItem. Once the file has a resource fork, you can begin writing to it.

This example displays an Open File dialog box and adds a resource fork to the file (if the file doesn't already have one):

```
Dim f as FolderItem
Dim rf as ResourceFork
f=GetOpenFolderItem("any")
If f <> Nil Then
   rf=f.OpenResourceFork
 If rf=Nil Then
  rf=f.CreateResourceFork("any")
 End if
End if
```

Adding a Resource Fork to a Project

You can add a resource fork to a REALbasic project by naming the resource file "Resources" and dragging it into the Project Window. If the resource file is not named "Resources," you cannot import it into the Project Window.

The following example opens the application's resource fork:

```
Dim rf as ResourceFork
rf=App.ResourceFork
```

Supported Resource Types

REALbasic provides high level support for PICT, CICN, CURS, and snd resources. You can used the AddPicture method to add PICT resources to the resource fork and use GetPicture or GetNamedPicture to get PICT resources from the resource fork. You can use GetCicn to get a cicn (color icon) resource. Sounds can be read from snd resources using the GetSound method of the ResourceFork class. However, you can access any type of resource. REALbasic provides method for getting and setting raw data from any type of resource in a resource fork. However, you must know the format of the resource data to be able to successfully read from it or write to it.

Reading Resources

The ResourceFork class has methods for reading data from four different types of resources. You can read PICT resources using the GetPicture and GetNamedPicture methods of the Resource-Fork class. You can get a color icon as a picture by calling the GetCicn method. You can get a large (32 x 32) icon using the GetIcl method and a small (16 x 16) icon with the GetIcs method. For example, the following line of code displays a picture resource in an ImageWell:

```
Me.Image=App.ResourceFork.GetPicture(128)
```

Use GetCicn, GetIcl, and GetIcs in the same way.

You can load sounds from ‘snd ’ resources using the GetSound method of the ResourceFork class.

Reading Custom Cursors

Use GetCursor to assign a custom cursor to the MouseCursor property of the Application, a Window, or a Control. For

example, the following line in a window's MouseEnter event handler assigns a custom cursor to the MouseCursor property of the window.

```
self.MouseCursor=App.ResourceFork.GetCursor(128)
```

This line causes REALbasic to display the custom cursor whenever the mouse enters the window's region. If you want to change the cursor when the mouse is over a control in the window, you can use a line such as this in the control's MouseEnter event handler

```
self.MouseCursor=App.ResourceFork.GetCursor(129)
```

and then restore the window's custom cursor with the following line in the control's MouseExit event handler:

```
self.MouseCursor=App.ResourceFork.GetCursor(128)
```

Notice that these lines do not assign a cursor to the control's own MouseCursor property; they only change the assignment to the control's parent window.

Assign a custom cursor to a control's MouseCursor property only when you don't use either the Window's or the App class's MouseCursor properties. If either the parent window or the application has a custom MouseCursor property, then the control's MouseCursor property is ignored.

To assign a custom cursor to the application as a whole, create a new class called "App" and make its Super Class "Application." Then add a line such as this to the App class's Open event handler.

```
MouseCursor=App.ResourceFork.GetCursor(128)
```

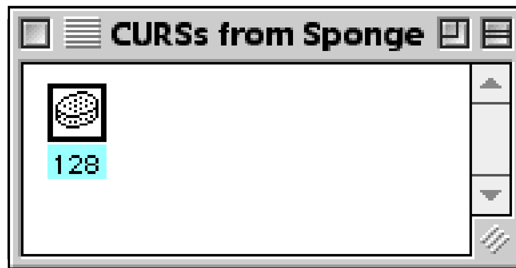
This causes the REALbasic application to display this custom cursor all the time. That is, the MouseCursor properties of any of the application's windows or controls will be ignored.

Custom Cursors in Windows Applications

You can use CURS resources to assign custom cursors in built Windows applications. This, at present, is the only case in which resources are supported on Windows builds. The relationships among the MouseCursor properties of the Application, Window, and Control are the same as for Macintosh applications, but you must create the resource files in a special way. This technique also works for Macintosh builds but is not mandatory.

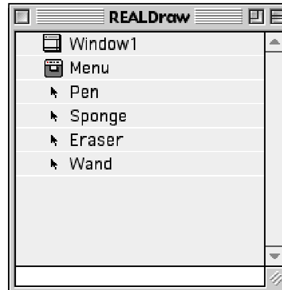
The key is that you must create a separate resource file for each custom cursor that you add to the project. Each resource file must have only a CURS resource that contains one cursor. Typically, you will assign the custom cursor to ID 128. An example is shown in Figure 111.

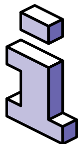
FIGURE 111. **A CURS Resource with one cursor.**



You then drag all of these resource files to your project. These special resource files will appear in the Project Window with a cursor icon, as shown in Figure 112.

FIGURE 112. **A Project with Four Custom Cursors.**





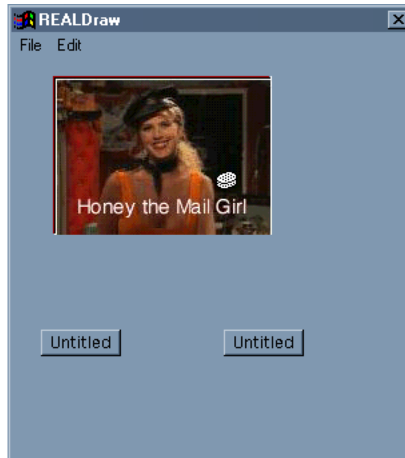
For Macintosh-only applications, you can instead simply create one resource file that contains multiple resource types and multiple cursors in the CURS resource.

You can then access the custom cursors by referencing their names. For example, the following statement in a control's MouseEnter event handler changes the pointer to the Sponge cursor when the mouse enters the region of the control:

```
Self.MouseCursor=Sponge
```

(You would then restore the cursor with a statement in the control's MouseExit event handler.) The result is shown in Figure 113.

FIGURE 113. **A Custom Cursor in a Windows Application.**



Reading Other Resources

To read data from other resources, you must know the format of the resource. For example, to read the STR# resource, you can use the GetResource method of a ResourceFork class. This will return the bytes that make up the resource ID you specify. To then do anything useful with the data, you will need to know that the first two bytes are the number of strings in the resource following by the strings themselves. The strings are Pascal strings so their first byte is the length of the string.

When you build your application, you can enter version information about the application in the Build Application dialog box. That information is stored in a 'vers' resource that becomes part of your application. An application can access the 'vers' resource using the GetResource method of the ResourceFork class. The information written to the 'vers' resource is described in the section "Get Info" on page 433.

Writing To Resources

REALbasic provides methods via the ResourceFork class that can be used to write to resources. You can use the AddPicture method to write REALbasic pictures into a PICT resource. For all other types of resources, you can use the AddResource method to create new resources and the RemoveResource method to delete specific resources. To modify a resource other than PICT resources, you read the data of the resource using the GetResource method, then write the data back by deleting the resource with the RemoveResource method and then recreating the resource using the AddResource method.

More Information on the ResourceFork

For more information on the ResourceFork class, see "ResourceFork Class" on page 344 of the *Language Reference*.

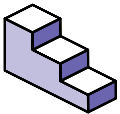
Files Opened From the Desktop

If your application is designed to read from and/or write to files, you may need to consider how your application will react when the user accesses files from the Finder (the desktop).

Files Opened by Double-Clicking

If the user double-clicks on a file whose creator code matches your stand-alone application's creator code, the user will be expecting your application to open the file automatically. If your application is prepared to open a file and take some action, then you should also support the user's double-clicking on the file from the desktop. This is done by adding a new class based on the Application class. This new class represents your application

as a whole and will receive information when the user double-clicks on a document whose creator code matches your application's creator code. The application class you are adding has an OpenDocument event handler that is executed when the user double-clicks on a file at the desktop. This event handler is passed a FolderItem as a parameter. This FolderItem represents the file the user double-clicked on.



To take action when the user double-clicks on a file from the desktop, do this:

1. If your project doesn't already have a class based on Application, choose File ▶ New Class.

   A new Code Editor window appears.
2. In the Properties window, choose Application from the Super pop-up menu.
3. In the Properties window, type "App" as the Name field.
4. Expand the Events list in the Code Editor browser.
5. Click on the OpenDocument event to select it.
6. Enter the code that should execute when the user double-clicks on a file at the desktop. You can access the file using the item parameter passed to the OpenDocument event handler.

Files Dropped On Your Application's Icon

REALbasic treats a file dropped on your application's icon at the desktop the same way it treats the user's double-clicking on a file from the desktop. For more information, see "Files Opened by Double-Clicking" on page 316.

Creating New Files

When the user launches your application without opening a file, REALbasic assumes that the user will probably want to create a document (assuming you application is document/file based). If

you have created a class based on the Application class, that class's NewDocument event handler will execute. This event handler also executes when your application receives an Open Application AppleEvent (oapp) or when a user uses AppleScript to tell the Finder to open your application.

You can call the NewDocument event handler by entering NewDocument in your code. This allows you to have a single location to put the code for your application that creates new documents. Using this event handler, your application will respond to all the appropriate calls to create a new document.

CHAPTER 9

Creating Reusable Objects with Classes

Classes act as templates for objects much in the same way that the windows listed in the Project window act as templates for the windows you open in your application. This chapter will introduce you to the benefits of classes, explain how to modify them, and how you can create custom interface controls using classes.

Contents

- The Benefits of Classes
- Understanding Subclasses
- Modifying Classes
- Managing Menus within Classes
- Using Classes in Your Projects
- The Application Class

- Creating Custom Controls with Classes
- Virtual Methods
- Interface Inheritance
- Custom Object Bindings

The Benefits of Classes

There are lots of benefits to creating classes. They are:

Reusable Code

When you add code to a PushButton control to customize its behavior, you can only use that code with that PushButton. If you want to use the same code with another PushButton, you need to copy the code and then make changes to the code in case it refers to the original pushbutton (since the new pushbutton will have a different name than the original).

Classes store the code once and refer to the object (like the PushButton) generically so that the same code can be reused any number of times without modification.

Smaller Projects/Applications

Because classes allow you to store code once and use it over and over in a project, your project and the resulting application is smaller in size and may require less memory.

Easier Code Maintenance

Less code means less maintenance. If you have basically the same code in several places in your application, you have to keep that in mind when you make changes or fix bugs. By storing one copy of the code, you will spend less time tracking down all those places in your project where you are using the same code. Making a change to the code in a class automatically updates any places where the class is used.

Easier Debugging

The less code you have, the less code there is to debug.

More Control

Classes give you more control than you can get by adding code to the event handlers of a control in a window. In fact, some classes can even manage menus. You can also use classes to create custom controls. And with classes, you have the option to create versions that don't allow access to the source code of the class, allowing you to create classes you can share or sell to other REALbasic users.

As you can see, there are many benefits to creating classes. Overall, classes make your programming effort more efficient.

Understanding Subclasses

REALbasic has many classes built-in to it. Pushbutton, StaticText, EditField, and ListBox are examples of some of the built-in classes. You may find situations where you would like to have an object that is a slightly altered version of one of the built-in classes. For

example, you might want a version of the EditField control that disables the Cut and Copy items on the Edit menu, preventing the user from putting sensitive data on the Clipboard. You might want to create a ListBox that, by default, has the months of the year in it. You can create your own versions of these built-in classes by creating *subclasses*.

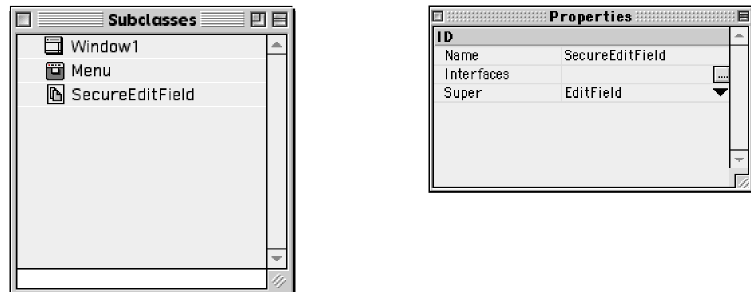
What is a Subclass?

A subclass is simply a class that has a super class. A super class is a class the subclass is based on. The super class is also sometimes called the "parent" class. Subclasses inherit all of their super's properties, methods, and events. The subclass can then modify them. In fact, a subclass is identical to its super class until you start modifying it. After that, it's different from its super class only in the ways you make it different by adding properties, modifying events, and adding or modifying methods.

Examples of Subclasses

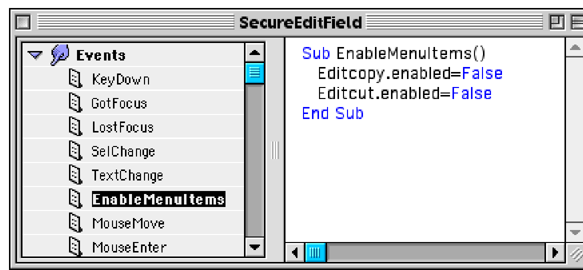
For example, to create an EditField that prevents the user from copying data to the Clipboard (Let's call it a SecureEditField), you create a new class and choose EditField as its super class. The new subclass is shown in Figure 114.

FIGURE 114. **SecureEditField based on the EditField class.**



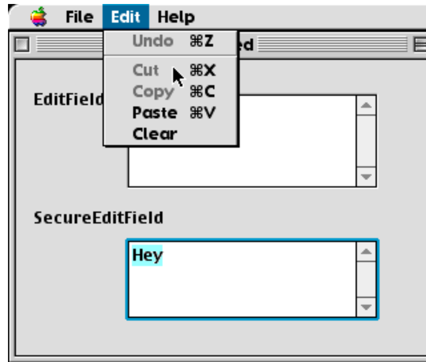
REALbasic automatically enables the Cut and Copy menu items on the Edit menu when characters are selected in an EditField. Since SecureEditField is based on EditField, it inherits these features. Because EditFields can get the focus, any subclass of the EditField control has an EnableMenuItems event handler. This allows SecureEditField to control the menus when it has the focus. To prevent the user from using the Cut and Copy menu items, you set the Enabled property of these menu items to False in your SecureEditField's EnableMenuItems event handler.

FIGURE 115. **Disabling the Copy and Cut menu items in SecureEditField.**



To use an instance of SecureEditField, just drag it from the Project window to any Window editor. It behaves exactly like a 'regular' EditField, except that the Cut and Copy menu items are disabled when the user has selected text.

FIGURE 116. **Selected text cannot be copied from an instance of SecureEditField.**



Since SecureEditField is listed in your Project Window, you can create instances of it anywhere you like and maintain its code in one central place.

Suppose you want to create a ListBox that, by default, displays the names of the months of the year, with the current month selected. You create a new class and choose ListBox as its super class. In the Open event handler of your new subclass, you use the AddRow method of the ListBox to add the month names (In the section "Constructors" on page 326 we will show you another way of accomplishing this). You then write code to select the appropriate month in the list.

You might want to create an EditField that only allows the user to enter numbers. Let's call it "NumbersOnlyEditField." To do this, you create a subclass of the EditField control and put code in the KeyDown event handler that allows only numbers and rejects all other characters. Once created, you can use your new subclass in many different places in your project, but the code exists only in one place.

Subclasses are classes. They are called subclasses to differentiate them from classes that have no super class and emphasize the point that they inherit the properties, events, and methods of their parent class. Because subclasses are classes, they can be the super class to other subclasses. For example, suppose you had already created the NumbersOnlyEditField subclass mentioned earlier. Now, you need an EditField that allows only numbers within a certain range. You could duplicate the NumbersOnlyEditField subclass and then modify its code. However, this would make your project larger and more difficult to maintain. If you found a bug in the code of the NumbersOnlyEditField, you would have to remember that you used that code in other places as well, track them down, and fix them. A more efficient way is to create a new subclass and choose the NumbersOnlyEditField as its super class. The new subclass (let's call it "NumberRangeEditField") would utilize all of the properties, events, and methods of its super class. However, you can add code to the TextChanged event handler that allows only numbers within a specific range.

Referring To A Class's Properties and Methods From Within the Class

When you add code to a control like a PushButton in a window, you are really adding code to an instance of the PushButton class. Consequently, you must include some reference to the instance or REALbasic would have no way of knowing which PushButton, for example, your code is referring to.

However, when you are adding code to a class or subclass, there is no need to refer to any instance because the code is part of the class to which you have added code, not the instance of the class. Consequently, you don't include object references to the

class in its own code. For example, suppose you create a pushbutton named "Pushbutton1" in a window that should be disabled after the user clicks it. The code in the PushButton's Action event handler would be:

```
Pushbutton1.Enabled=False
```

If you had instead created a subclass with Pushbutton as its super class, you would not include the instance reference, so the code would be:

```
Enabled=False
```

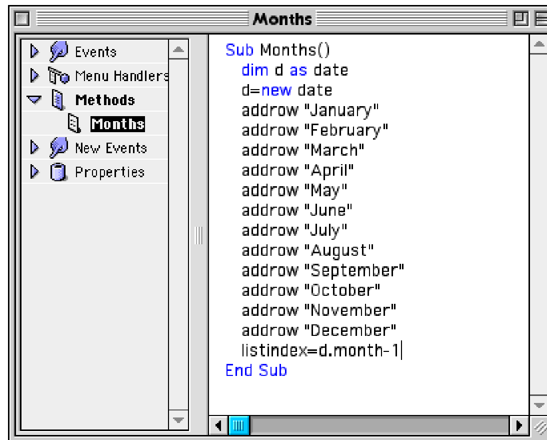
When the subclass is used, the code will automatically be operating on the instance of the class that's in use.

Constructors

When you create a new object, you will sometimes want to perform some sort of initialization on the object. The constructor is a mechanism for doing this. To create a constructor, simply define a method with the same name as the class. This method will then be called automatically when an instance of the parent class is created.

Suppose you want to display the months of the year in a ListBox with the current month highlighted. Create a subclass of ListBox called "Months" and then add a method to the subclass that is also called "Months". It uses the following code:

FIGURE 117. The "Months" Constructor.



Notice that the code doesn't refer to an instance of the ListBox because it is added to the subclass, not any particular instance.

You would then add an instance of the Months class to a window by dragging it from the Project window to a Window editor. In the instance's Open Event handler, use the following line of code:

```
me.months
```

You can also create a destructor by creating a method that has the same name as the class, preceded by the tilde (~). The destructor is called automatically when an instance of the parent class is deleted or goes out of scope.

Overloading

REALbasic also supports what is known as *overloading* of methods. A language that supports overloading allows you to

have two or more methods with the same name but have a different number of parameters or parameters with different data types. When that method name is called, REALbasic will figure out which method you 'mean' to call from its parameters.

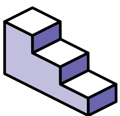
A good example of a built-in overloaded function is the '+' operator. If its arguments are numbers, it computes the sum; if the arguments are strings, it concatenates the strings.

Modifying Classes

One of the big advantages of classes is the ability to modify existing classes. You do this by adding properties, adding or changing events, and adding or changing methods.

Adding Properties

You can add properties to a class to store values that its super class doesn't store. For example, you might want to create a subclass of the EditField control that stores the last value the user entered. This would allow you to selectively reject the current entry and restore the last entry. You add properties to a class the same way you add properties to a window.



To add a property to a class, do this:

1. If the class is not already open, double-click on it in the Project window to open it.
2. Choose Edit ▶ New Property. The dialog box shown in Figure 118 appears:

FIGURE 118. **The Property Declaration dialog box.**

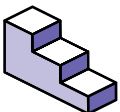
3. Enter the property definition (like: "Name as String").
4. (Optional) Check the Private and/or Visible check boxes
5. Click the OK button.

The "Private" and "Visible" checkboxes add the following features:

- Making a property Private means that the property can be accessed only by the event handlers and methods of the class.
- Making a property Visible means that, if you drag an instance of the class onto a window, the property can be assigned a value from the Properties window (rather than only with code). This option is available only for non-control classes.

Adding Methods

You can add methods to classes to provide functionality that the class previously didn't have. For example, you might want to add a new method to a class based on the ListBox control that inserts a row rather than appends the row to the end of the list (the way the built-in AddRow method does).



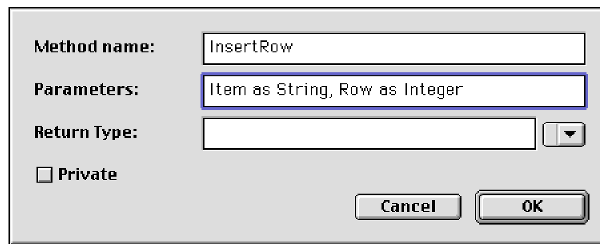
To add a method to a class, do this:

1. If the class is not already open, double-click on it in the Project window to open it.
2. Choose Edit ▶ New Method.

The New Method dialog box appears. Figure 119 on page 330 shows an example of the New Method dialog box.

3. Enter a name for the method.
4. Enter the parameters if any, separating multiple parameters with commas, as shown in Figure 119.
5. Enter the data type of the value to be returned if the method will be a function (or choose the data type from the pop-up menu).
6. Click the OK button.

FIGURE 119. **The New Method dialog box.**



Adding New Events

Any events in a class you have added code to will not, by default, be available to any instance of the class. Consider this example. You create a class based on the ListBox class and you put some code in its Open event handler. Any instances of that class that appear in a window will not have an Open event handler. The assumption is that since the event handler of the class has code for the event, it is handling that event.

There may be times, however, when you want the class to have code in an event handler but you also want to be able to put code in that event handler for any instance of the class. An example of this is when you set up default values. In the Open event handler, you might set the default values of the class. For example, in a class that displays the names of the months in a

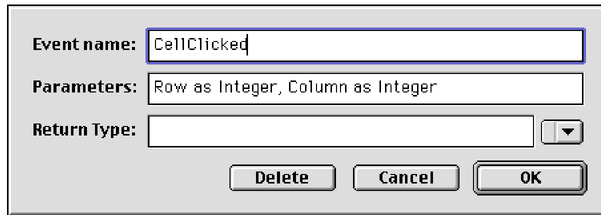
ListBox, you might want to select the current month name by default. However, when you use this class in a window, you might want to be able to override the default action and choose a different month instead. The instance of the month's ListBox won't have an Open event handler because its class is handling the Open event.

Adding new events solves this problem. You add a new Open event to the class and then call it from the class's Open event handler. New events are available only to the instances of the class. When you add a new Open event, you are adding that event to any instance of the class. When will this new Open event occur? Since you are calling it in the class's Open event handler, it will occur when the window opens — just like a regular Open event handler.

Let's look at another example in which you would want to add new events. Suppose you are creating a custom class that will display a grid. The grid allows the user to click on individual cells to turn them on and off. You might want to add an event that occurs when the user clicks on a cell in the grid. Let's call this event "CellClicked." You also want the event to be passed the row and column numbers where the click occurred. In any particular instance of the class, you could then use the CellClicked event as a place to take action when the user clicks in a cell.

So how do you go about adding the CellClicked event? First, add a new event called CellClicked to the class. You want to pass the row and column numbers to this event, so include them as parameters for the event. Figure 120 shows what the New Event dialog box might look like when you are adding the CellClicked event.

FIGURE 120. The New Event dialog box



The next step is to determine when this event will occur. Since the user clicks the mouse to select a cell, it makes sense that this event is generated when he clicks the mouse. For the Canvas control (the class the grid class would be based on), this means calling this event in the MouseDown and MouseDrag event handlers. To do this, call the CellClicked event as if it were a method. You do the necessary calculations to determine the row and column numbers and pass these to the CellClicked event.

When the user clicks on a cell, the MouseDown event handler of the class is executed. This causes the CellClicked event to be called and passed the row and column numbers. This causes the CellClicked event to occur for the instance of the class the user clicked on in the window. The class is basically calling a subroutine of the instance of the class. And, because the CellClicked event could be designed to return a value, the instance of the class can return data back to the super class. This could be beneficial in this particular example if you wanted to filter the click. You could code the class to only continue with handling the click should the CellClicked event return False (use False since this is the default value returned by a function). This would allow any instance of the class to determine which cells are valid for clicking and which cells are not.

See the "Gridlock" project on the REALbasic CD for an example of this kind of new event.

Managing Menus within Classes

Classes that can receive the focus can control the menus when they have the focus. This make it even easier to encapsulate code within a control. The EditField and ListBox classes are the only classes that can receive the focus (on Macintosh). Any classes you create with either of these classes as the super class will have an EnableMenuItems event handler and can have menu handlers for any of the menu items in your project.

When an instance of a class based on the EditField or ListBox has the focus and the user clicks in the menu bar (or presses a keyboard shortcut for a menu item), the class's EnableMenuItems event handler is executed. This gives the class the opportunity to enable or disable any menu items. The window's EnableMenuItems event handler will be executed next, followed by the application class EnableMenuItems event handler (assuming you have created a class with Application as its super property). If a menu item is then selected, REALbasic first checks the class to see if it has a menu handler for the selected menu item. If the menu handler exists, it is executed, followed by the window's menu handler (if it has one for the selected menu item), followed by the application's menu handler (if it has one for the selected menu item).

The SecureEditField class mentioned in the section "Examples of Subclasses" on page 322 is an example of a class controlling menu items. When the SecureEditField has the focus and the user clicks in the menu bar, the SecureEditField's EnableMenuItems event handler sets the Enabled property of the Cut and Copy menu items to False, disabling them. These menu items would normally be enabled automatically by REALbasic.

Another example of a class that manages menus is a class based on the ListBox that allows the user to use the Cut and Copy

menu items to move menus between ListBoxes. See the ClipListBox project of the REALbasic CD for an example.

Using Classes in Your Projects

Before you can use a class in your project, you must first understand a few concepts and terms. The use of a class in a project involves three items: the class, the instance, and the reference.

The Class

The class is a template set of the events, methods, and properties.

The Instance

An instance is a place in memory that stores a copy of the properties of the class. Methods are not stored in memory with each instance. Instead, they are loaded from the class into memory when they are called.

The Reference

The reference is a value stored in a property or local variable that keeps track of where the instance is in memory. You use the property or local variable holding the reference to access the instance of the class. In this example, "person" is a local variable storing a reference to the instance of the class "Programmer." The reference is then used to access the value in the name property of the instance created using the New operator.

```
Dim person as Programmer
person=New Programmer
person.name="Jason"
```

You will learn more about using the New operator later in this chapter.

How you use a class in your project depends on whether the class is based on a control.

SubClasses Based on Controls

To create an instance of a class based on a control in a window, simply drag the class from the Project window to the window in which you want the new instance.

Classes Based on Classes Other Than Controls

Classes don't have to be based on controls. You can also create classes based on classes that are not part of the Control class. For example, the Thread class is not part of the control class. You might need to create a subclass of the Thread class and add properties to it to store information used or created by the thread. You might even need to create classes that have no super class. This is often the case when you need to store complex information. For example, you could create a class called "People" that had properties like Name, Age, and Height to store information about people. You could then create a subclass of people called "ComputerUsers" which would add additional properties that define a computer user.

To create an instance of a class based on a class other than one of the control classes, you must first have a place to store the instance. You can store the instance in a property or a local variable. The property or local variable must be of the same type as

the class or one of the class's super classes. For example, if class Programmers is a subclass of ComputerUsers which is a subclass of People, then the property or variable must be of type Programmers, ComputerUsers, or People.

The New operator is used to create a new instance of the class in memory and then assign a reference to the new instance to the property or local variable you have typed. In this example, the local variable "person" is typed as class Programmer. The New operator is then used to create a new instance of Programmer and assign a reference to this new instance to the variable person.

```
Dim person as Programmer
person=New Programmer
```

Although you can type the property or variable as the class or any of the super classes above it, the property or variable will only have access to the properties and methods of the class you type it as. For example, in the code below, the local variable person has access only to the properties and methods of the ComputerUser and People classes, even though it was created as a Programmer class object with the New operator.

```
Dim person as ComputerUser
person=New Programmer
```

Accessing the Properties and Methods of a Class

Once you have created an instance of a class and stored a reference to it in a local variable or property, you can access its properties and methods the same way you access any object's properties and methods. In this example, a new instance of a class called Programmers is created and a value is assigned to one of its properties:

```
Dim person as Programmer
person=New Programmer
person.name="Jason"
```

When are Instances of Classes Removed From Memory?

REALbasic manages memory for you automatically using something called *garbage collection*. This means that instances of classes are removed from memory automatically when they are no longer used. Suppose you create a class based on a ListBox. You then create an instance of that class in a window. When the window is opened, the instance of the class is created in memory automatically. When the window is closed, the instance of the class is automatically removed from memory. If you store the reference to a class in a local variable, when the method or event handler is finished executing, the instance of the class is removed from memory. If you store a reference to an instance of a class in a property, the instance will be removed from memory when the object owning the property is removed from memory.

The Application Class

This special class is used to create a subclass that represents your application rather than a window or a control. Consequently, you can only have one class based on Application. If you create more than one, REALbasic ignores all other classes based on the Application class.

Special Event Handlers

The Application class has special event handlers. They are:

- Open

  Executes when the you run the application by choosing Debug ▶ Run (⌘-R) or when launching a stand-alone version of your application.

- Close

  Executes when you quit your application either from the Runtime environment or in a stand-alone application.

- NewDocument

  Executes when the stand-alone version of the application is launched without double-clicking one of the application's documents.

- OpenDocument

  Executes when one of the application's documents is double-clicked at the Finder.

- EnableMenuItems

  Executes when the user clicks in the menu bar but before any menu items are displayed. The EnableMenuItems event handler executes after the EnableMenuItems event handler of any classes with instances in the frontmost window and after the window's EnableMenuItems event handler. This is the event handler that should be used to enable menu items that should be enabled regardless of whether there is a window open or not.

- HandleAppleEvent

  Executes when an AppleEvent is received by the application.

Properties Are Global

Properties of the Application class are accessible to all code in your project.

Methods Are Global

Methods of the Application class are accessible to all code in your project.

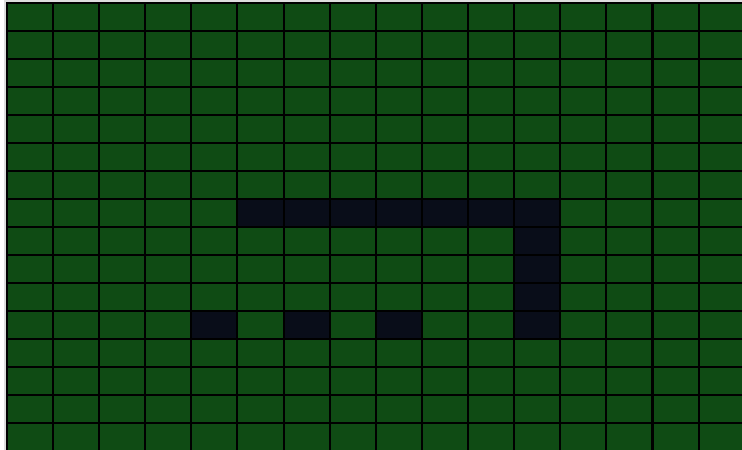
Naming Your Application-Based Subclass

REALbasic creates an application object when your application runs regardless of whether you have an application-based subclass in your project. The App function returns a reference to this application object. If you name your application-based subclass “App,” it will make your code more clear as the App function and your App subclass will effectively operate as the same thing.

Creating Custom Controls with Classes

One of the most important uses of classes is for creating custom interface controls. While REALbasic provides most of the interface controls you will need in your project, you may find you need to create interface controls that are not built-in to REALbasic. Suppose you need to create a control that displays a grid of cells. You want the user to be able to click on the cells in the grid to select them. Figure 121 shows an example of what such a grid control might look like.

FIGURE 121. **Custom grid control**



Custom interface controls are created by building a subclass based on the Canvas control. The Canvas control gives you an area in which you can draw your control and it receives events allowing you to interact with the user. For example, in the grid control example above, the Paint event handler of the Canvas control is used to draw the grid. This is actually the Gridlock class example that you can find on the REALbasic CD. The Gridlock class has properties that store the number of rows and columns the programmer wants for a particular instance of the Gridlock. There are also properties that store the selected cell color and the unselected cell color. When the user clicks in the grid area, the MouseDown event handler for the Gridlock class executes. The code for this event handler determines which cell was clicked and then determines if the cell should now be selected or unselected. A new event called CellClicked has been added to the Gridlock class that is executed when the user clicks on a cell. The purpose of this event is to allow an instance of a Gridlock class to react to a cell click. The CellClicked event handler is passed the row and column numbers of the cell that was clicked. The CellClicked

event handler also acts as a function. If an instance of the Gridlock class returns True in the CellClicked event handler, the Gridlock class assumes the programmer wants to filter the click, so it acts as if the user didn't click in the cell.

Drawing Your Custom Control

The Paint event handler of a Canvas control (or a Canvas Control based subclass) is executed any time the control needs to be redrawn. For example, if a window is covering part of the control and it is then moved to uncover more of the control, the Paint event handler executes to redraw the control. If the look of the control doesn't change at all when it's used, you can do all of the drawing of your control in the Paint event handler. However, if your control changes, you will need to take a different approach. For example, the Gridlock control changes when the user clicks on a cell. The Gridlock control also has a method that allows the number or rows and columns in the grid to be changed on the fly. This requires the grid to be redrawn.

In the Gridlock example, the grid needs to be redrawn at two different times. It needs to be redrawn in the Paint event handler in case something (like a window position over the control) has uncovered a portion of the control, and when the grid is redefined to have a different number of rows and columns. Because of this, the code to do the actual drawing is placed in its own method. The method is called DrawGrid and it is passed the Graphics property of the Canvas control that the Gridlock class is based on. The DrawGrid method can then use this property to redraw the grid. By placing this code in a separate method, the same code can be used by the Paint event handler and by the DefineGrid method. The Paint event handler is passed a reference to the Graphics property of the Canvas so this reference can be passed on to the DrawGrid method when calling it from the Paint event handler. The DefineGrid method calls the DrawGrid

method as well since the grid is being resized and needs to be redrawn. The DrawGrid method can be passed the graphics property in this case by using the syntax:

```
DrawGrid Me.Graphics
```

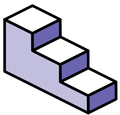
Me is a reference to the instance of the class in the window. So although this code is being called from inside the Gridlock class, the use of Me allows it access to properties of the instance in use.

Virtual Methods

Virtual methods provide a way for a subclass to have its own version of a method. Ordinarily, a subclass inherits the methods belonging to its parent.

When a subclass has a method that has the same name as its parent, the subclass's version is called unless you use the syntax:

```
parentclassname.methodname
```



To create a 'virtual' method, do this:

1. Create a class.
2. Add a method to the class.
3. Create a subclass of the first class.
4. Add a method to the subclass with the same name as the method you added in step 2.

When the subclass calls the method, it will call its own version and not its parent class's version.

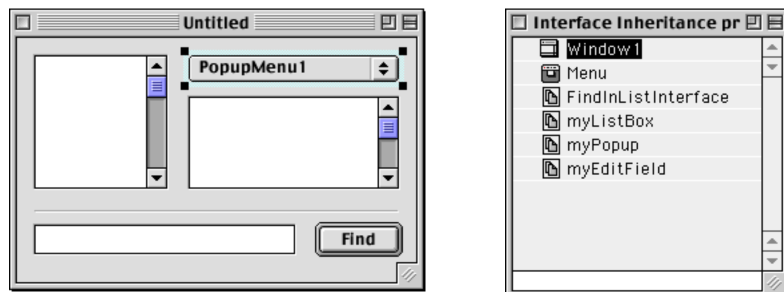
Interface Inheritance

Although Interface Inheritance sounds complicated when described in abstract language, it actually addresses a simple problem. If you have several controls that need to perform the same task but in a different way (depending on the specifics of the types of control) you can write and execute interface-specific code in an elegant way.

Figure 122 on page 343 that shows an application that uses interface inheritance. The purpose of this application is to conduct a search for a user-entered string and find the string in the three controls located above the separator: The EditField, ListBox, and PopupMenu are all based on custom classes. Although the task is identical (a find operation), it cannot be done with exactly the same code for all three objects, since the three objects store and manipulate data differently. Therefore, each custom class has its own implementations of the methods used to do the search.

The ListBox, EditField, and PopupMenu are all derived from custom classes that use a custom interface, FindInListInterface. They all have a Find function that takes the same parameter, but all implement it differently. The code for the Find button can call all of their Find functions using the same syntax.

FIGURE 122. An example application that uses interface inheritance



The user enters a search string in the EditField, FindValue, to the left of the Find button. When he clicks the Find button, the following code is executed:

```
FindIt FindValue.text, listBox1
FindIt FindValue.text, popupMenu1
FindIt FindValue.text, editField1
```

The same method, FindIt, is called for each of the three controls, but each line of code actually executes a different version of FindIt—the one that is appropriate for that type of control. The second parameter is the name of the control; each control inherits methods from the custom class on which it is based.

The EditField, PopupMenu, and ListBox are all instances of custom classes. The custom classes have two methods, Find and

SelectRow, that implement the correct search routines for that object type. This is shown in Table 57.

TABLE 57. Find and SelectRow methods by Control Type

Control	Find Function	SelectRow Method

EditField	Function Find (FindValue as string) as Integer
	dim rows, foundPos, foundCR-pos as integer
	rows=-1
	foundPos=instr(text,find-Value+chr(13))
	do until foundCRpos>=found-Pos
	    foundCRpos=instr(foundCR-pos+1,text,chr(13))
	    rows=rows+1
	loop
	return rows
		Sub SelectRow (Row as Integer)
		dim counter, startPos, endPos as integer
		do until counter=row
		    startPos=instr(startPos+1, text, chr(13))
		    if startPos <> 0 then
		        counter=counter+1
		    end if
		loop
		endPos=instr(startPos+1,text,chr(13))
		selStart=startPos
		selLength=endPos-startPos

ListBox	Function Find (FindValue as string) as Integer
	dim i as integer
	for i=0 to listcount-1
	    if list(i)=findValue then
	        return i
	    end if
	next
		Sub SelectRow (Row as Integer)
		listindex=row

PopupMenu	Function Find (FindValue as string) as Integer
	dim i as integer
	for i=0 to listcount-1
	    if list(i)=findValue then
	        return i
	    end if
	next
		Sub SelectRow (Row as Integer)
		listindex=row

The FindIt method itself uses the FindInListInterface:

```
Sub FindIt (findValue as String, source as
FindInListInterface)
 dim row as integer
```

```
 source.selectRow source.find(findValue)
End Sub
```

The FindInListInterface class simply has two blank methods, Find and SelectRow. It simply defines the methods and their parameters. When the FindIt method runs, it actually executes the versions of the methods that are appropriate for the control passed as the second parameter to FindIt.

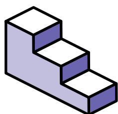
Custom Object Bindings

Object binding allows you to add functionality to your interface without writing any code. There are many binding actions you can choose that are built-in to REALbasic. Built-in binds are listed in the section "Object Binding" on page 110. You can also program your own binds using custom classes. Your custom binds will appear in the New Binding dialog box when you Command-Shift drag from eligible objects.

You can drop your custom binding into any project and it will work as transparently just as if it was built-in to REALbasic. This section shows two examples of custom object binds.

A "Delete All Rows" Bind

This first example creates a user defined bind between PushButtons and ListBoxes. It allows the user to create a PushButton that deletes all rows from a ListBox. You first create a custom class that implements the binding Interface.



To create the custom bind:

1. Add a new class to your project.

2. Name this class “DeleteAllRowsBind”.

Since this class is going to be a bind, it needs to support the bindingInterface. Also, you will want to assign an action to the Action event of the pushbutton that, when clicked, tells the ListBox to delete all the rows. In order to add code that will execute when the user clicks the PushButton, you will need to support the ActionNotificationReceiver interface.

3. To support the bindingInterface and ActionNotificationReciever interface, enter "bindingInterface,ActionNotificationReceiver" in the Interfaces property of the class.

Now that the class supports the bindingInterface, you can add the Bind method.

4. Add a new method to the class and call it “Bind”. Include the parameters “Source as Object, Target as Object”.

Since the code that executes when the user presses the pushbutton will need to know which pushbutton and listbox have been bound, these will need to be stored as properties in the class.

5. Add the following properties to the class: BindSource as Pushbutton button and BindTarget as Listbox.
6. Add the following code to the Bind method:

```
#pragma bindingSpecification pushbutton,list-
box,"Delete all rows in %2 when %1 is pushed"

BindSource=pushbutton(source)

bindTarget=listbox(target)

BindSource.addActionNotificationReceiver self
```

The #pragma statement defines what will appear in the Bind dialog box when the user binds the two objects by command-shift dragging from the source to the target (the pushbutton to

the listbox). It indicates the source class, the target class and the text that will appear in the Bind window. %1 will be the name of the source control and %2 will be the name of the target control.

When the bind occurs, the controls that are bound will be passed to the Bind method. However, they are passed as generic objects. In order for the code to store them in the BindSource and BindTarget properties you just added to the class, these two object parameters have to be recast as PushButton and ListBox. Lines 2 and 3 in the Bind method do just that.

Finally, the last line connects the bind class to the Action event of the pushbutton by calling the addActionNotificationReceiver method of the PushButton class. Self is passed as a parameter to this method and represents the class itself.

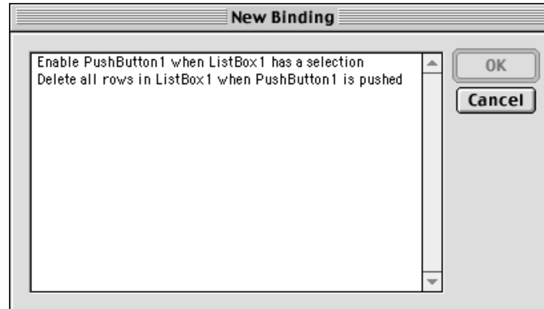
Now you need to add the code that will delete the rows when the user clicks the pushbutton.

7. To do this, add a method called “PerformAction” to the class.
8. The code for the PerformAction method is simple:

```
bindTarget.DeleteAllRows
```

To try out this new bind, drag a ListBox and a PushButton onto a window and add some rows of data to the ListBox. Bind the two controls by Command-Shift dragging from the PushButton to the ListBox. When the Bind dialog appears, the custom binding appears in the list, as shown in Figure 123.

FIGURE 123. **The Custom Binding added to the list of binds**



9. Choose the option “Delete all rows from listbox1 when pushbutton1 is pushed” and then test the binding in the Runtime environment.

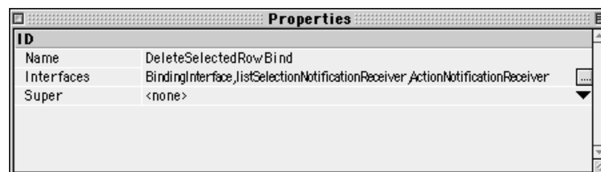
A “Delete Selected Row” Bind

To create a bind that will delete the selected row from a listbox when a pushbutton is pushed, create another class in your project and duplicate all the steps you did above.

Since this version of the bind should work only when a row is selected, you will need to support a few more interfaces in order to detect when the user selects a row.

1. Add a class called DeleteSelectedRowBind.
2. Add listSelectionNotificationReceiver to the list of interfaces in the Interfaces property of the class. The class’s Property window should look like Figure 124.

FIGURE 124. **The Property window for the DeleteSelectedRowBind**



3. Add a new method to the class and called it “Bind”. Include the parameters “Source as Object, Target as Object”.
4. Enter the following code:

```
#pragma bindingSpecification pushbutton, list-
box,"Delete the selected row in %2 when %1 is
pushed"
BindSource=pushbutton(source)
bindTarget=listbox(target)


BindTarget.addListSelectionNotificationRe-
ceiver self
BindSource.addActionNotificationReceiver self
bindSource.enabled=(bindTarget.listindex >= 0)
```

The first new line links the bind class to any change to the list selection. The last line, sets the source (the PushButton) to enabled if a row of the target (the ListBox) is selected and disables the source if no row is selected when the window opens.

Now you need to add the code that will disable or enable the pushbutton once the window is open the user begins selecting or deselecting rows in the ListBox. In order to do this, you will need to add the methods that are part of the ListSelectionNotificationReceiver interface to the class.

5. Add methods “SelectionChanged” and “SelectionChanging” to the class. Neither has any parameters. Add the following code to the SelectionChanged method:

```
bindSource.enabled=(bindTarget.listindex >= 0)
```

Any time the selection in the ListBox is changed, the bind will be notified and the SelectionChanged method will fire.

The SelectionChanging method has no code.

Lastly, you need to add the code that will delete the selected row when the user clicks the pushbutton.

6. Add the PerformAction method and enter the following line of code:

```
bindTarget.removeRow bindTarget.listindex
```

Now you can add another button and bind it to the ListBox.

7. Choose the new bind as the bind action and test it in the Runtime environment.

Importing Classes From Other Projects

Because classes can be exported, they can also be imported. When a class is exported, it appears on the desktop with a cube icon. Figure 125 on page 351 shows an example of an exported class. To import a class, just drag the class file into your Project window. This copies the class into the Project so you can delete the class file if don't need to use it elsewhere. The Project is not dependent upon it.

FIGURE 125. An exported class file

If a class you are importing is based on another class, that other class must be present in order for the class you are importing to function. If that class is based on one of the built-in classes (like the EditField for example), this isn't an issue. However, if the class

is based on a class that isn't built-in, then that other class must be present in your Project window.

Exported classes can be protected. This means that, while you can import and use the class, you cannot view or edit the source code for the class. You can check to see if a class you have imported is protected by double-clicking on the class in the Project window. If the class is protected, a dialog box will be displayed informing you of this.

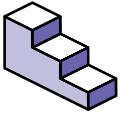
Exporting Classes For Use In Other Projects

Classes can be easily exported from your Projects for use in other projects. There are two ways to export a class. You can export a class simply by dragging it from the Project window to the desktop. This will create a file on the desktop with the name of the class. Figure 125 on page 351 shows an example of this.

Sometimes the folder you want to place the exported class in is not easy to get to. For example, the folder might be a folder within another folder that isn't open at the moment. In these cases, you can export the class by clicking on the class to select it in the Project window, then choosing File ▶ Export Class.

Protecting Your Source Code

You may want to share classes with other REALbasic users. If you wish to share a class with other users but you don't want to share the source code itself, you can protect the class when you export it. This creates an exported class that can be imported and used but cannot be edited. The user cannot even view the source code. This is especially important if you plan to create sophisticated classes that you wish to sell as third party add-ons to REALbasic.

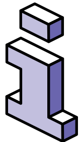


To export a protected class, do this:

1. Click on the class in the Project window to select it.
2. Choose File ▶ Export Class.

   The Export Class dialog box appears.
3. Click the Protect checkbox to select it.
4. Click the OK button to export the class.

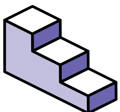
Protected classes that have been exported use the same desktop icon as unprotected classes, so you will need to keep track of which ones are protected.



Since other users cannot open a protected class to view its source code, you will need to provide them with a list of methods and properties if they should have access to them.

Deleting Classes From a Project

Before deleting classes from a Project, make sure you are not using the class in your code anymore. Also be sure to check for other classes that may have this class as their super class.



To delete a class from a Project, do this:

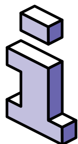
1. Click on the class in the Project window to select it.
2. Press the Delete key on the keyboard or choose Edit ▶ Clear.

If you delete a class accidently, choose Edit ▶ Undo (⌘-Z).

CHAPTER 10

Creating Databases with REALbasic

With REALbasic, you can create database front-end applications that can be used with a variety of database engines, including REAL Software's own database.



Database-related features are fully supported in the Professional version of REALbasic. In the Standard version, you can experiment with all database features, but you are limited to 50 rows of data and you cannot build runtime database applications.

Contents

- REALbasic's Database Architecture,
- Structured Query Language,
- REALbasic's Database tools,
- Creating and Modifying databases from the Project Window,

- Using Object Binding,
- Creating a Database Front End Programatically.

REALbasic's Database Architecture

You use REALbasic to build a "front-end" to your database. It works in conjunction with a database "back-end" that actually stores the data itself. The database back-end may be a separate application or it may be REALbasic's own database back-end. The front-end serves as the user interface and the means by which queries are sent to the database itself. The end user uses the front-end to view, enter, and modify records, search for and sort records, and print reports.

You use another database application, such as 4th Dimension's 4D Server or Oracle, to actually store the data. The database application that actually holds the data is referred to in REALbasic as the "data source." For your convenience, REALbasic ships with an internal database that you can use for development and/or for deployment of single-user databases.

A great feature of this architecture is that any database front-end that you create in REALbasic works with any supported data source—or multiple data sources. You can develop a database application with the internal REAL database and then deploy the system simply by switching the data source. Your database code will work without any other modifications.

A REALbasic front-end can also use two or more data sources simultaneously. For example, you can access data locally on 4D Server while simultaneously accessing remote data on a SQL or ODBC-compliant database.

REALbasic uses its plug-in architecture to support multiple data sources. Except for 4th Dimension and OpenBase, the plug-ins are built into REALbasic and don't appear in the plugins folder. You (or a third-party) can add support for additional data sources by writing a plug-in for that back end. REAL Software's plug-in SDK contains information on writing database plug-ins.

Structured Query Language

A REALbasic front-end uses the Structured Query Language (SQL) to communicate with its data sources. The plug-in for your data source receives a SQL statement from REALbasic, translates the statement into a form that the data source understands, and sends it to the data source.

If you are unfamiliar with SQL, you will need to learn its basics before implementing your REALbasic front-end. This manual does not attempt to teach you SQL; rather, it describes the subset of SQL that is currently supported by REALbasic. Please consult one of the many good SQL references, such as *SQL for Dummies* by Allen G. Taylor (ISBN: 0-7645-0105-4) or the SQL tutorial available online at http://w3.one.net/~jhoffman/sqltut.htm.

SQL in REALbasic

REALbasic currently supports a subset of SQL. This section provides an overview of SQL in REALbasic. Table 58 lists the supported SQL statements.

TABLE 58. **Supported SQL statements in REALbasic.**

Statement	Example	Description
SELECT WHERE ORDER BY GROUP BY	SELECT * from Customers SELECT [Last Name], Phone from Customers	Returns a group of rows, known as a cursor. You can specify the columns (fields), the table(s), search conditions, grouping, and sorting columns.
CREATE TABLE	CREATE TABLE invoice(id integer not null, Cust_ID integer not null, Amount varchar (25), Date varchar (25), primary key (id))	Creates a table and specifies the fields and their attributes.
UPDATE	UPDATE Customers SET City='Toldeo',Telephone='312 555-1212' WHERE Cust_ID='0121'	Updates existing records.
Set functions: Min, Max, Avg, Count, Sum	SELECT Count (*) from Customers SELECT Sum(Total) from Invoice SELECT Name from Invoice where total=(select max(Total) from Invoice)	Returns calculated values from a group of rows. Used with SELECT statement and optional WHERE clause.

Select Statement

The following is the syntax for a Select statement in REALbasic:

SELECT *columnlist* FROM *tablelist* WHERE [LIKE] *searchcondition* GROUP BY *groupingcondition* ORDER BY *sortcondition*

TABLE 59. **Arguments in SELECT Statement**

Argument	**Description**
columnlist	List of fields separated by commas. Using the asterisk (*) in place of the fieldnames retrieves all fields in table. If a column name has spaces in it, it must be surrounded by brackets, e.g., [first name] should be used for the field ‘first name’. If *tablelist* refers to multiple tables, use the syntax *tablename.fieldname* to refer to a field.
tablelist	List of tables separated by commas. If a tablename has spaces in it, it must be surrounded by brackets, e.g., [Product Groups]
searchcondition	Expression that specifies a subset of the rows in the table or tables. Must evaluate to a Boolean whose value is True for the desired rows. If *tablename* refers to multiple tables, use the syntax *table-name.fieldname* in *searchcondition*, i.e., Custom-ers.Customer_ID=Invoices.Customer_ID. *Searchcondition* may use the following comparison symbols: **Comparison** **Symbol** Equality = Not equal <> Less than < Less than or equal to <= Greater Than > Greater Than or equal to >= You can create a compound *searchcondition* using the OR, AND, and NOT conjunctions, e.g. ‘Customers.Customer_ID=56 AND Invoices.Customer_ID=56’. To use a SQL wildcard, use the LIKE operator, e.g., select * from Customers where name LIKE 'smith%'

TABLE 59. **Arguments in SELECT Statement**

Argument	Description
groupingcondition	Field or fields on which you wish to group the rows in the cursor. If a GROUP BY clause is used, the rows are organized in groups defined by the fields in this clause. The groups appear in alphabetical order.
sortcondition	Field or fields on which you wish to sort the individual rows in the cursor. Use ORDER BY rather than GROUP BY if the field is unlikely to define groups, such as Last Name. By default, the rows appear in ascending order. If you wish to use descending order, include the modifier DESC, i.e., 'ORDER BY invoices.date DESC'.

Joins

You do relational operations (e.g., "joins") by specifying the tables to be joined in the SELECT statement's *tablelist* and indicating in the WHERE clause how the tables are to be joined, i.e., rows in the 'many' table whose foreign key matches the primary key in the 'one' table. Please refer to examples of joins in the Database Class in the REALbasic Language Reference and a SQL reference book for detailed information on relational operations.

Create Table Statement

The Create Table statement creates a table structure on the current data source. It specifies the table name, field names, and field attributes. The syntax is:

CREATE TABLE *tablename* (*fieldname1 fieldtype [not null]*, ….*FieldnameN fieldtype [not null]*, Primary key *fieldname*)

TABLE 60. **Arguments in the CREATE TABLE Statement**

Argument	Description
TableName	Name of the new table.
FieldName	Name of a field
FieldType	Data type. The REAL database supports the following data types: Smallint, integer, float, double, boolean, date, and time. If you are using another vendor's data source, see their documentation on supported data types.
Primary Key	The field whose values uniquely identifies the row, i.e., the identifying field in the table for relational operations.
Not Null	Optional. If Not Null is used, the database will require a value for that field in ever row, i.e., the field may not be missing.

Use the SQLExecute method when using the CREATE TABLE statement.

Update statement

The Update statement changes existing data in a table. Its syntax is:

UPDATE *tablename* SET *fieldname1=expression1*,… *fieldnameN=expressionN* WHERE *searchconditio*n

TABLE 61. **Arguments in UPDATE Statement**

Argument	Description
TableName	Name of table containing fields to be updated.
FieldName	Field in *TableName*
Expression	Value to be assigned to Field
SearchCondi-ton	Boolean expression that identifies the row or rows to be updated.

You can also perform updates using the Edit…Update methods within the REALbasic language. See the section "Editing Records" on page 373 for more information.

Set Functions

The Set functions apply to sets of rows in a table and return the results of an arithmetic calculation. A Set function returns a value that is the result of the calculation. You determine the number of rows on which the calculation is based using a standard WHERE clause and include the Set function in a SELECT statement. You can use several Set functions in a SELECT statement and each calculates its value on the rows specified by the WHERE clause. The SELECT statement returns a DatabaseCursor. If the WHERE clause is omitted, the function is computed on all the rows in the table.

TABLE 62. **Set Functions in REALbasic**

Function	Description and Example
Count	Number of rows in cursor that have nonmissing data on fields specified in select statement. SELECT COUNT (Name, Phone) from Customers WHERE Zip='48070'
Min	Minimum value of the field specified in SELECT statement. SELECT MIN (Price) from Products
Max	Maximum value of the field specified in the SELECT statement. SELECT MAX (Population) from Cities
Avg	Computes the average value of the field specified in the SELECT statement. SELECT AVG (Price) from Products WHERE Product='Database'
Sum	Computes the total of the values in the field specified in the SELECT statement. SELECT SUM (Population) from States WHERE Region='NE'

The following code produces a database cursor that summarizes information on a particular individual. It sums the Amount column for the customer whose ID is equal to 1 and returns the person's name and the sum of the amount column in a database cursor. The new DatabaseCursor has one row and two columns (Name and sum of Amount), but the sum is computed across many rows in the invoices table.

```
Dim db as Database
Dim cur as DatabaseCursor
.
.
cur=db.SQLSelect("select customers.name,SUM
(invoices.amount) from invoices,customers
where invoices.Cust_ID=1 and customers.id=1")
```

REALbasic's Database Tools

A database front-end typically uses a mixture of database-specific and generic controls and commands. There is one database-specific control, the DatabaseQuery control, and several classes and methods that are database-specific. Beyond that, you will use generic controls such as EditFields and ListBoxes to display and edit data, PushButtons and menu items to perform actions, and tab controls and other interface elements to polish the user interface.

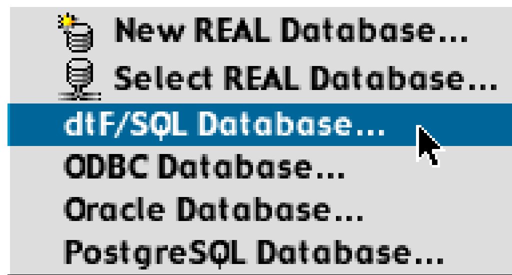
REALbasic's built-in data source can be used for development purposes and for the deployment of single-user solutions. The built-in data source is limited to single-user applications and each record may contain no more than 8,174 bytes, and each varchar field may contain no more than 8,174 bytes. Each table can have

a maximum of 254 columns. There is no fixed maximum number of rows.

Selecting a Data Source

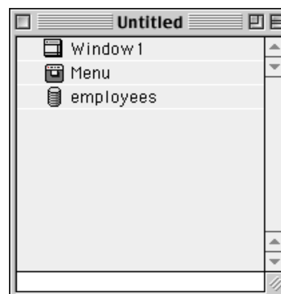
You can select a data source using either the language or the File ▶ Add Data Source submenu in the Design environment. The latter provides the options shown in Figure 126.

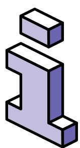
FIGURE 126. **The Add Data Source submenu**



When you choose a data source in this manner, it appears in the Project window, as shown in Figure 127.

FIGURE 127. **A database in the Project Window.**





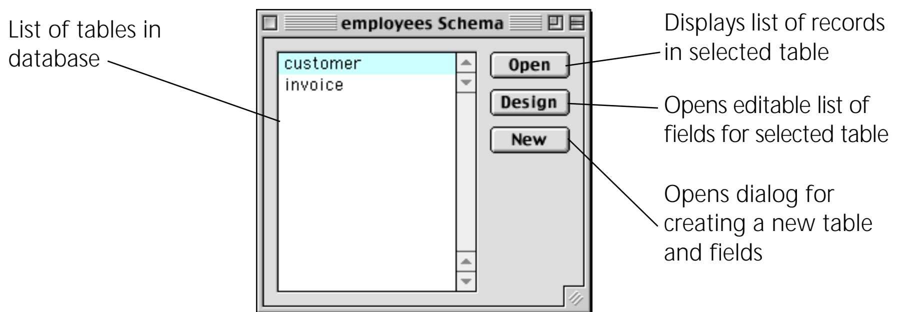
Alternatively, you can simply drag an existing REAL database from the Finder into the Project Window. To remove a data source from the Project Window,

highlight the data source and press the Delete key on the keyboard or choose Edit ▶ Clear.

Creating and Modifying Databases from the Project Window

You can double-click a database in the Project Window to display its Schema—the list of its tables. From that list, you can view the data itself and the list of fields and field properties.

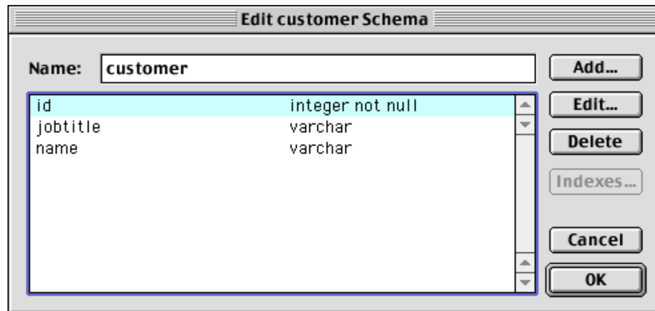
FIGURE 128. **A database Schema for an existing database.**



If the database is new (i.e., you chose File ▶ Add Data Source ▶ New REAL database), the list of tables will be blank; you can add tables by clicking the New button.

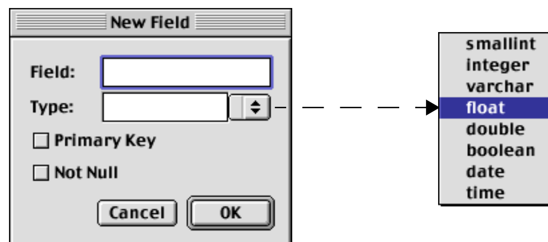
The Design button displays the schema for the selected table. An example table schema is shown in Figure 129.

FIGURE 129. **Fields and Field Properties.**



The Add button (or Command-A) lets you add a new field using the dialog box shown in Figure 130. The Edit button displays the same dialog box showing the properties of the selected field, allowing you to modify its name and properties.

FIGURE 130. **The Add Field dialog box.**



The Open button in the Database Schema dialog box (Figure 128 on page 365) opens a static list of the data in the selected table. You can view the list but not edit or open a record. An example of such a list is shown in Figure 131 on page 367.

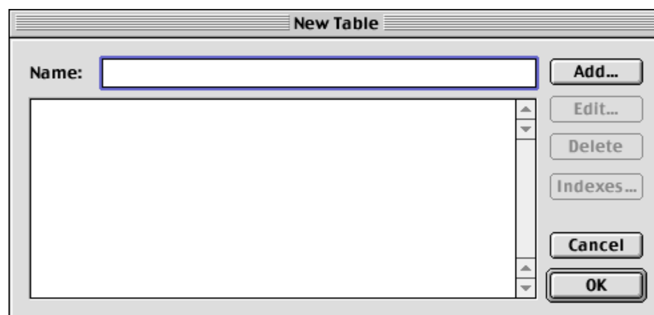
FIGURE 131. **A listing of records.**

employees.customer

1	VP	Lois Lane
2	Reporter	Jimmy Olson
3	Editor	Perry White
4	Receptionist	Lana Lang

The New button in the database Schema dialog box (Figure 128 on page 365) lets you create a new table using the following dialog box.

FIGURE 132. **New Table dialog box.**

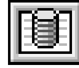


Use the Add button to add fields to the database using the dialog box shown in Figure 130 on page 366. Name the table and click OK.

The DatabaseQuery Control

The DatabaseQuery control can be used to send queries to the data source, but this function can also be performed entirely with the language. It is up to you.

FIGURE 133. **The DatabaseQuery control**



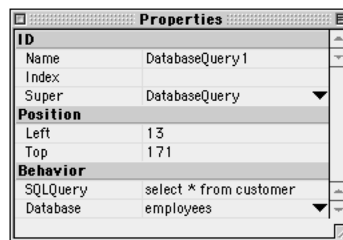
You add a DatabaseQuery control to a window like any other control, but it is not visible to the end-user. It is used only as an object that performs database queries.

TABLE 63. **DatabaseQuery Properties**

Name	Description
Database	The data source that will be queried.
SQLQuery	The text of the SQL query to be run against *Database*

SQLQuery is executed automatically when its window appears. For example, the properties shown in Figure 134 will retrieve all rows and columns from the customer table when the window opens. However, the DatabaseQuery control cannot *display* the rows and columns all by itself.

FIGURE 134. **A DatabaseQuery control's properties**



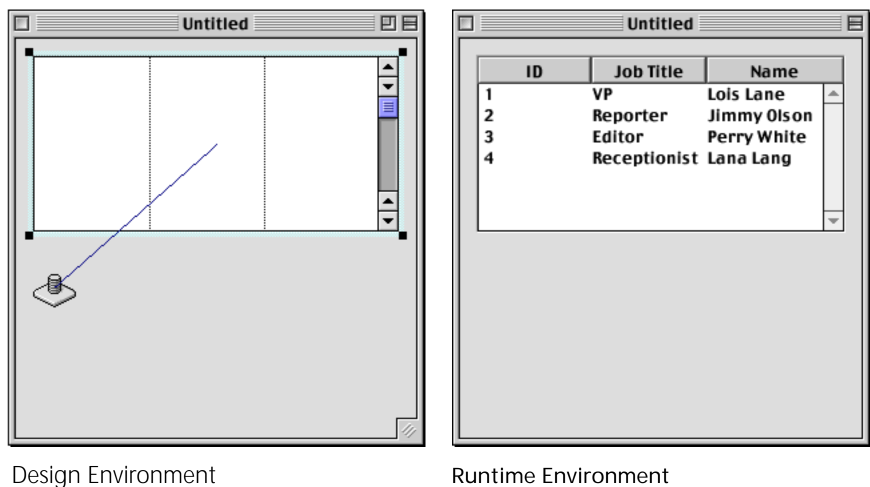
The DatabaseQuery control has one method, RunQuery, which executes *SQLQuery* against *Database*.

Using Object Binding

The DatabaseQuery control, together with object binding, allows you to create a simple database front-end with no programming. Since the DatabaseQuery control automatically executes *SQLQuery*, you only need a means to display the results. Simply add a ListBox to the window and bind the DatabaseQuery and the ListBox controls (hold down Shift and Command and draw a line connecting the two controls). An Object Binding dialog box will appear. Select the option that binds the ListBox to the DatabaseQuery control's results.

When you choose Debug ▶ Run, the data appear in the ListBox, as shown in Figure 135.[1]

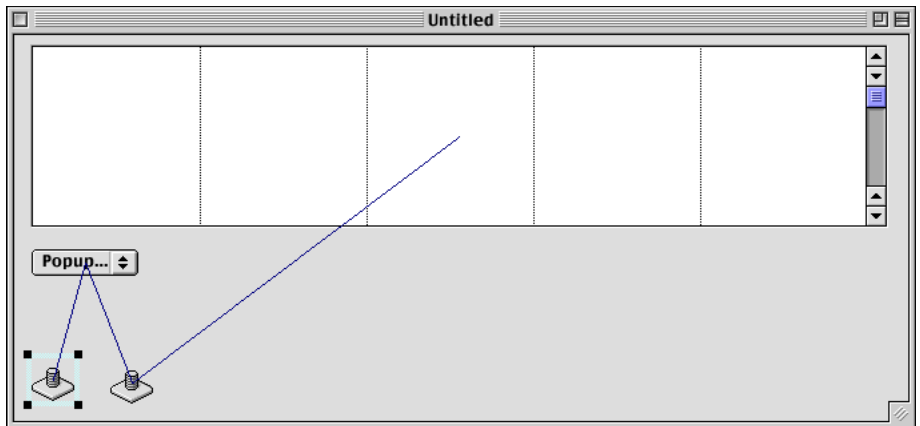
FIGURE 135. **A Simple database built with no programming.**



1. Headings were added to the ListBox using its Properties window.

Similarly, you can bind a DatabaseQuery control to a popup menu so that the results of the query populate the pop-up menu. You can also bind the Popup to the DatabaseQuery control so that the query can be changed by selecting an item from the popup menu. Figure 136 illustrates this concept:

FIGURE 136. **Using object binding to populate a PopupMenu control.**



The selected DatabaseQuery control contains the SQL query:

```
Select name, id from airports
```

The binding to the PopupMenu control is simply: Bind DatabaseQuery1 to PopupMenu1.The first column of the query is displayed by the PopupMenu control (name of a city). The second column, the ID, populates the PopupMenu's RowTag property, an invisible column that can be used as an identifier.

The binding from the PopupMenu to the second DatabaseQuery is:

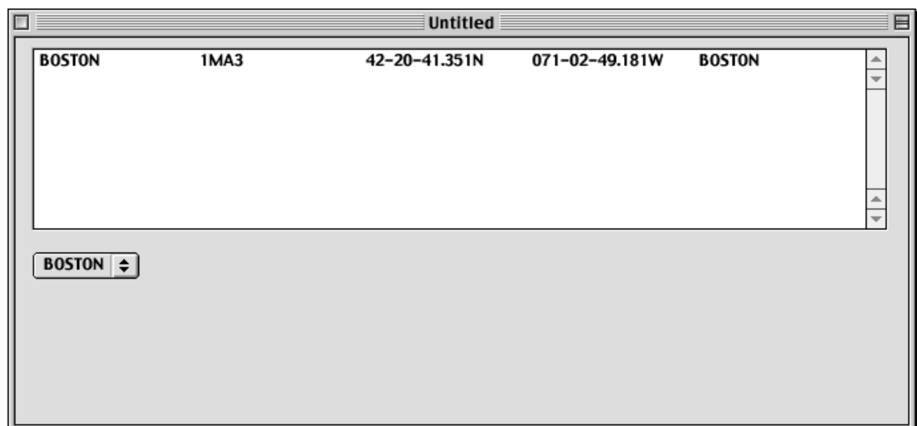
```
Bind DatabaseQuery2 parameter to PopupMenu1
RowTag
```

and the SQL statement associated with DatabaseQuery2 is:

```
select * from airport where id='%1'.
```

This statement includes the parameter %1, which is assigned to the value of the RowTag for the selected item in the popup menu, i.e., the ID belonging to the city selected by the user. In other words, this query selects all columns for the selected city, but it uses the invisible column rather than the city to do the query. The last binding, of course, binds the results of the DatabaseQuery to the ListBox. An example is shown in Figure 137.

FIGURE 137. **A simple database that uses only object binding.**



Creating a Database Front End Programmatically

To fully exploit REALbasic's capabilities, you will need to write some code. The commands that are listed in the Database theme in the *Language Reference* provide you with all the necessary tools to build sophisticated database front-ends.

Choosing a Data Source

The following methods or functions allow you to choose the database back end(s) used in your application:

TABLE 64. **Methods and Functions for choosing a data source**

Method	Description
NewREALDatabase	Creates a new REAL Database.
OpenREALDatabase	Opens an existing REAL Database.
OpenOracleDatabase	Opens an Oracle database.
OpenODBCDatabase	Opens an ODBC database.
SelectODBCDatabase	Displays a dialog allowing the user to choose an ODBC database.
Select4DDatabaseByADSP	Displays a dialog allowing the user to choose an 4th Dimension database.
Open4DDatabaseByADSP	Opens a 4th Dimension database using ADSP.
Open4DDatabaseByTCPIP	Opens a 4th Dimension database using TCP/IP.
OpenDBFCursor	Opens an xBase format file (i.e., a dbf file from dbase).
OpenDTFDatabase	Opens a dtF database.
OpenCSVCursor	Opens a comma delimited text file as a data-base cursor.
OpenPostgreSQLDatabase	Opens a (Linux) Postgre database.

With the exception of OpenCSVCursor, these methods return an object of type Database. Table 65 gives the Database Class methods.

TABLE 65. **Database Class methods**

Name	Parameters	Description
Close		Closes the database
Commit		Commits (saves) changes to records. Use Commit and Roll-back for transaction process-ing.

TABLE 65. **Database Class methods**

Name	Parameters	Description
FieldSchema	TableName as String	Returns DatabaseCursor with a information about all fields in the table.
InsertRecord	TableName as String Data as DatabaseR-ecord	Inserts *Data* as the last row of *TableName.*
Property	Name as String	Returns the database property specified by *Name* from the data source. Currently sup-ported only for 4D Server and for getting the connection string.
Rollback		Cancels a set of changes to records.
SQLSelect	SelectString as String	SQL Select statement. Returns a DatabaseCursor.
SQLExecute	ExecuteString as String	SQL Execute statement
Table-Schema		Returns DatabaseCursor with a list of all tables in the data-base.

The SQLSelect method can be used instead of a DatabaseQuery control to send queries to the database. The SQLExecute statement can be used in place of the database Schema dialog boxes to build database schema (as well as perform many other functions). For example, the following statement can be used to create the table schema shown in Figure 129 on page 366:

```
db.SQLExecute("create table customer(id integer not null, name var-
char (25), jobtitle varchar (25), primary key (id))")
```

Editing Records

The SQLSelect method returns an object of type DatabaseCursor. The cursor contains the rows of data that meet the selection

criteria (defined by the SELECT statement's WHERE clause) and, in multi-user applications, locks them against modification by other users. You can then display the rows and/or edit them. If you need to edit the rows, you must process the records one row at a time.

The properties and methods of a DatabaseCursor are shown in Table 66.

TABLE 66. **DatabaseCursor Properties**

Name	Type	Description
BOF	Boolean	Beginning of cursor
EOF	Boolean	End of cursor
FieldCount	Integer	Number of fields in cursor

TABLE 67. **DatabaseCursor Methods**

Name	Parameters	Description
Close		Closes an open cursor.
MoveNext		Moves the record pointer to the next record in the cursor
Field	Name as String	Returns value of Field in row the record pointer is pointing to.
IdxField	Index as Integer	1-based array. Use to refer to i^{th} field in the cursor.
Edit		Call prior to performing modifications to the current record.
Update		Call to update cursor to reflect changes to the record the record pointer is pointing to.
DeleteRecord		Deletes the record in the cursor the record pointer is pointing to.

When an SQL statement returns a database cursor, the user has 'possession' of those records for his exclusive use. If the cursor contains more than one record, you can use the cursor's properties and methods to cycle through the rows and columns of the cursor.

As the record pointer moves to a particular record, you can use the Edit…Update methods to edit the record or the DeleteRecord method to remove the record. To modify the record, cell the Edit method, perform the modifications, and then call Update. This process updates the record within the DatabaseCursor. When you are finished, call the Database object's Commit method to commit the set of modifications to the database, or call the Rollback method to cancel the modifications.

If you don't call Commit, REALbasic issues an implicit Commit when the user quits the application.

See the examples for the classes and methods in the Database theme in the *Language Reference* for more information.

Adding Records

You add a new record using the Database object's InsertRecord method. It has two parameters, the database object and an object of type DatabaseRecord.

TABLE 68. **DatabaseRecord Properties**

Name	Type	Description
Column	Name as String	Column in current table.

For example, the following code adds a record to the "employees" table.

```
dim db as Database
.
dim rec as DatabaseRecord
rec = new DatabaseRecord
.
rec.Column("id") = "09"
rec.Column("name") = "Clark Kent"
rec.Column("jobtitle")="Reporter"
db.InsertRecord("employees",rec)
```

See the descriptions of the Database, DatabaseCursor, and DatabaseRecord classes in the *Language Reference* for additional discussion and examples.

CHAPTER 11

Debugging Your Code

Wouldn't it be great if every line of code executes just the way you want without a single error? Well, for those times when it doesn't work out that way, REALbasic provides you with some tools to track down the bugs and fix them.

Contents

- What is Debugging?
- Displaying the Debugger By Setting Breakpoints
- Watching Your Variables and Properties
- Following the Execution of Methods
- Interrupting Code Execution at Runtime
- Handling Runtime Exception errors

What is Debugging?

Debugging means removing errors, both logical and syntactical, from your programming code. Errors in programming code are referred to as bugs. You are probably wondering why errors are called bugs. Well, back in the 1940's, the United States Navy had a computer that occupied an entire warehouse. At that time, computers used vacuum tubes and the light from the tubes attracted moths. These moths would get inside the computer and short out the tubes. Technicians would have to go in and remove the bugs to make the computer work again. Since this was a government project, everything had to be logged, so they would put down "debugging computer" in the log. But enough of the history lesson.

Debugging is part of programming. It's the part of programming most programmers like the least. Fortunately, REALbasic makes it easy to track down those nasty bugs and squash them like a, well, bug. REALbasic comes with a Debugger which is a set of windows that help you see what is going wrong.

Logical Bugs

These are bugs in your programming logic. You will know you have found one of these when your code zigged when it should have zagged. REALbasic's built-in Debugger can help you find these by letting you watch your code execute one line at a time.

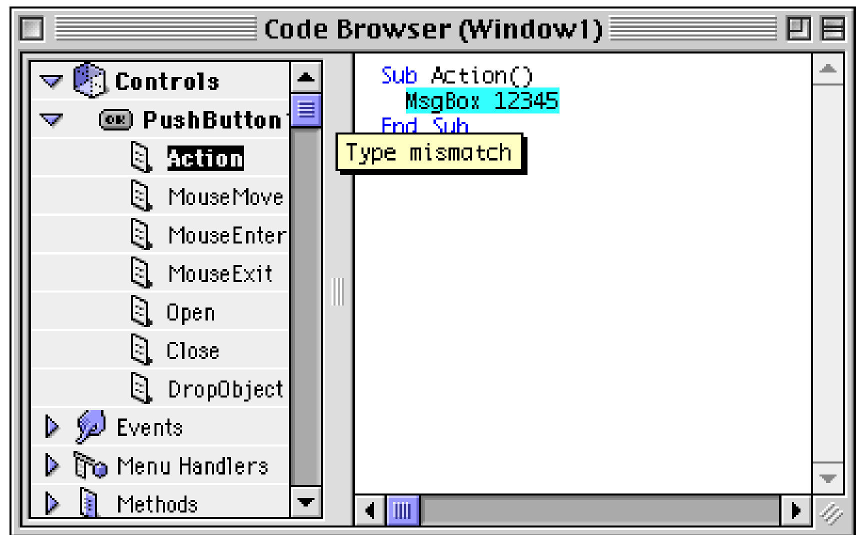
Syntactical Bugs

These are bugs where you have mistyped the name of a class, property, variable, or method. You may have also tried to use two values together that don't go together. For example, if you try to assign a string value to a variable or property of type integer, you will get a Type Mismatch error because they are different data

types. REALbasic makes finding syntax errors a snap. As soon as you choose Debug ▶ Run (⌘-R), REALbasic checks your code for syntax errors and reports them instead of compiling your project. Yes, you have to fix them before you can run your code.

Figure 138 shows an example of an error displayed by REALbasic.

FIGURE 138. A syntax error message.

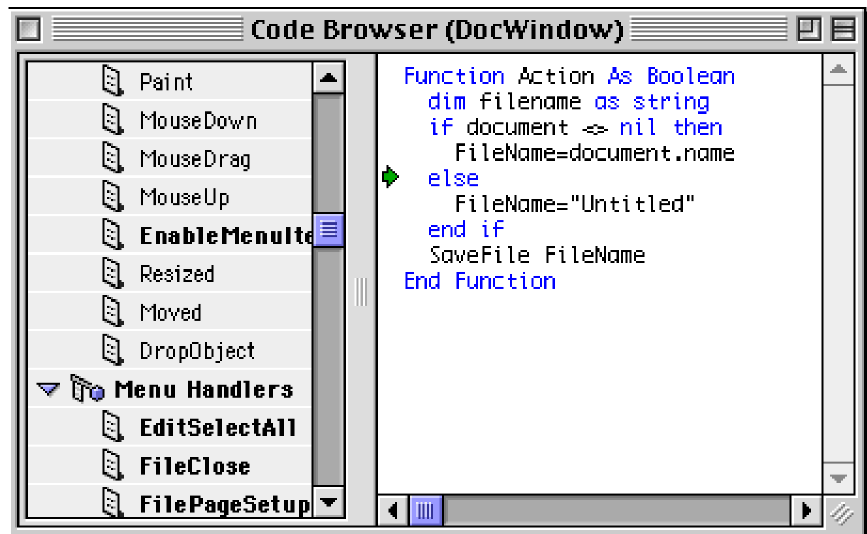


REALbasic highlights the line with the error and displays an error message below the offending line. This syntax error occurred because the MsgBox method is expecting a string as its parameter, not a number. To pass the value 12345 to the MsgBox method, it would have to be in quotes (making it a string) or be passed to the Str function which would convert it to a string.

The Debugger

The Debugger window looks just like the Code Editor window you used to write your code, with a few important differences. First, the Debugger displays a little green arrow to the left of the line of code that is about to be executed. Also, the when the Debugger is active, the other menu items on the Debug menu become active.

FIGURE 139. The Debugger



There are three ways you can display the Debugger:

You Have A Syntax Error In Your Code

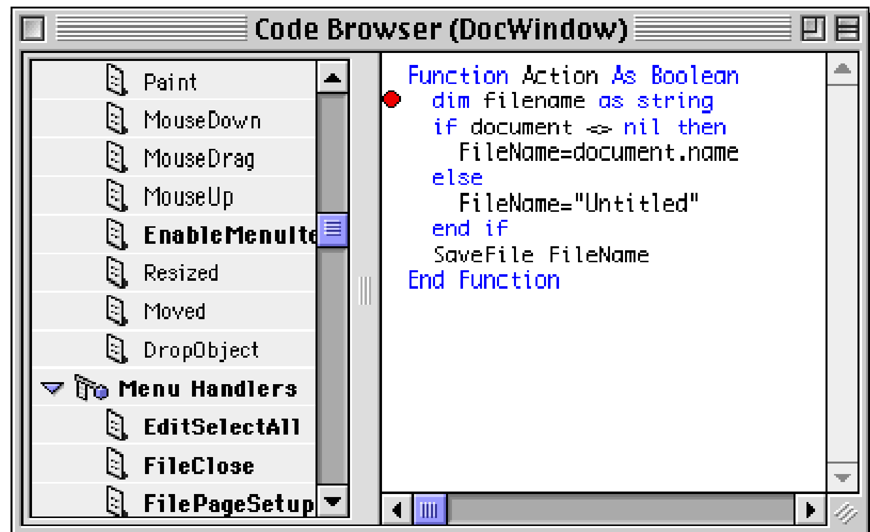
When you attempt to run your project by choosing Debug ▶ Run (⌘-R) or by choosing File ▶ Build Application and then clicking the Build button, REALbasic checks your code for syntax errors. If

it finds one, it stops immediately and displays the Debugger along with the error message.

You Have Set A Breakpoint In Your Code

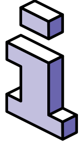
A *breakpoint* is a marker you can set for any line of code that tells REALbasic to display the Debugger when it reaches that line of code but before it executes it. You set a breakpoint in the Code Editor by clicking on a line of code to place the cursor there and then choosing Debug ▶ Set Breakpoint. A red circle will appear to the left of the line of code to indicate that a breakpoint has been set. Figure 140 shows the Code Editor with a breakpoint set on a line of code.

FIGURE 140. **A breakpoint in the Code Editor**



Breakpoints are persistent. This means they will stay in your code until you remove them. You remove a breakpoint by clicking on

the line that contains the breakpoint then choosing Debug ▶ Clear Breakpoint.



Breakpoints have no effect in stand-alone applications you build by choosing File ▶ Build Application.

You Have Pressed ⌘-Shift-Period

If you are running your project and you need to interrupt the code that is executing, you can hold down the Command key (⌘) along with the Shift and Period keys to halt code execution and display the Debugger. This is handy if you find yourself in an endless loop.

You can also switch back to the Design environment by clicking on any Design environment window.

Following the Execution of Methods

When your code isn't cooperating or you're just not sure what is executing and when, it's helpful to be able to watch your code as each line executes. The Debugger makes this easy. Once the Debugger is displayed, the current line icon (the green arrow) indicates which line of code is about to be executed. When you tell the Debugger to continue, it executes that line and goes on to the next line of code. What it does next depends on the command you give it when you wish to continue. The Debug menu gives you three commands, each of which will execute the current line and then take a different course of action for the next line of code.

Step Over

Choosing Debug ▶ Step Over (⌘-]) executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger executes the method but will *not* step through the method's code. When the method is finished executing, the Debugger will continue from the next line of code in the current method. Consider the following code:

```
EditField1.SelBold=True
EditField1.Text=ToFrench(EditField1.Text)
EditField.SelBold=False
```

Let's assume that "ToFrench" is a method that to translates English to French. If you step through this code using the Step Over menu item, the second line of code is executed, but the Debugger won't display the code in the ToFrench method. It executes the ToFrench method and continues with the next line of code.

Step Into

Choosing Debug ▶ Step Into (⌘-[) executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger displays the method and steps through the method's code. When the method is finished executing, the Debugger returns to the calling method or event handler and continues with the next line of code.

Step Out

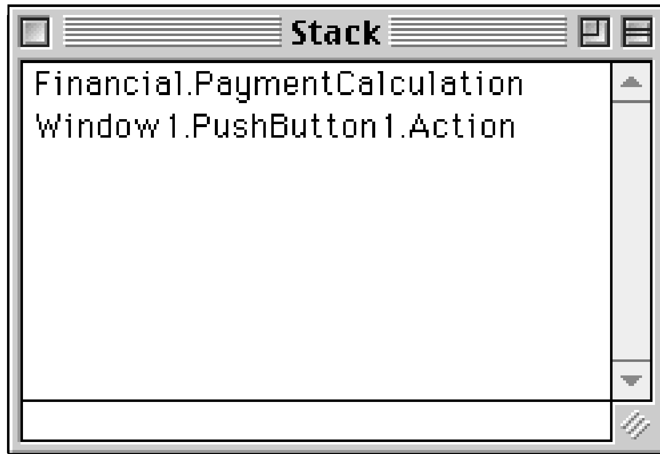
Choosing Debug ▶ Step Out (⌘-Y) executes the rest of the method without stopping on each line. This is handy when you have used Step Into to step through a method that was called by another method and now wish to continue code execution

without stopping on each line. If you entered the current method or event handler using Step Into, then stepping out executes the rest of the method and stops on the next line of code in the method that called the method you are stepping out of.

Tracking Method Execution with the Stack

A method or event handler can call another method or event handler which can call another one. This can go on for a while and you may need to keep track of the path of methods that were executed to get you where you are now. The Stack window does just that. When code execution begins (for example, when a button is clicked), the Stack window lists the pushbutton's action event handler. If the action event handler calls a method, that method is added to the top of the list in the Stack window when it's called. If that method calls another method, it is added to the top of the list. Once the current method finishes executing, it is removed from the list as REALbasic returns to the method that called it. Figure 141 on page 385 shows the Stack window listing a few methods. The window is called the Stack window because the methods are "stacked" one on top of the other in the order they were called.

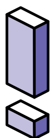
FIGURE 141. The Stack window



In Figure 141, The Action event handler of Pushbutton1 of Window1 called the PaymentCalculation method of the Financial module.

If you need to see the code from a method or event handler called earlier in the stack, you can simply double-click on its name in the Stack window to display the Debugger for that method.

The Stack window is displayed when the Debugger is displayed. But you can hide it if you aren't using it by choosing Debug ▶ Hide Stack.



The larger the Stack list gets, the more memory is being used. If you run out of memory it could be because your stack is so long that it takes up all the memory that has been allocated to the stack. The solution is try to make fewer method calls and use fewer local variables.

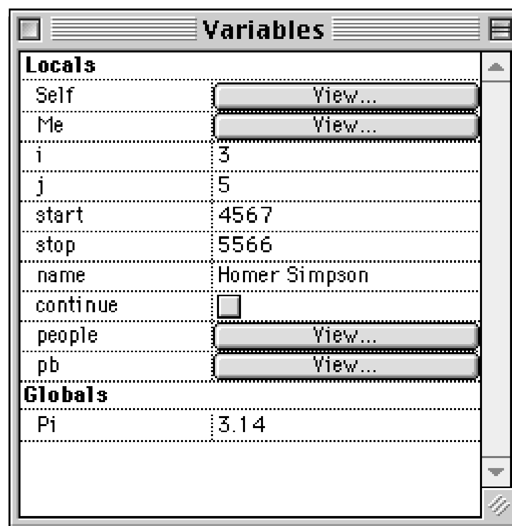
Watching Your Values

Part of debugging is monitoring the conditions under which certain lines of code execute. Another part of debugging is monitoring the values of variables, objects, and properties as your code executes. The Variables window is used for these purposes. This window displays any local variables, parameters, the current object, and its super class. It also displays global properties from modules and the application subclass if there is one.

Local Values

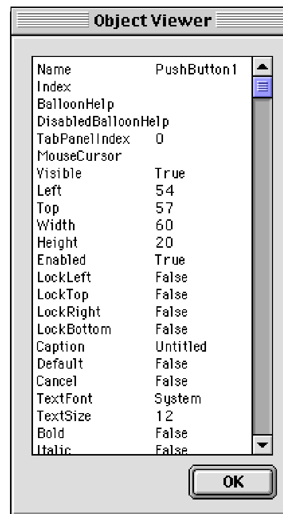
In Figure 142, a pushbutton's action event handler is executing. The local variable Self refers to its parent, which is the window. Me refers to the object whose code is executing (the pushbutton in this case). Pb is a local variable defined as a PushButton1 and is storing a reference to a PushButton1 created with the New operator.

FIGURE 142. The variables window



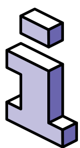
Because all of these items (Self, Me, and pb) are objects, they each have a View button next to them. This button displays the Object Viewer that shows the current values for all the properties of the object.

FIGURE 143. The Object Viewer displaying Pushbutton1



I, J, Start, and Stop are local integer variables while Name is a local string variable. Continue is a local boolean variable. All of these variables can be edited in the Variables window. This comes in handy when you figure out that a variable has the wrong value due to a bug but you want to see how the rest of the code would act if it had the right value. You can simply change the value and continue executing the code.

People is a local array. The Object Viewer can be used here to view the elements of the People array.



The Object Viewer currently only supports viewing single dimension arrays.

Global Values

Because Pi is a property of a module, it is global in scope. Appropriately, it's listed under the Globals label. If your project has a class based on Application, any properties of this subclass appear under the Globals label as well, since they are global in scope.

Starting and Stopping Your Project

You can switch back to the Design environment while your application is running in the Runtime environment by clicking on any Design environment window. This causes the execution of code in the Runtime environment to pause. Should you decide to resume execution of code in the Runtime environment, choose Debug ▸ Run (⌘-R). To stop the execution of code and quit from the Runtime environment, choose Debug ▸ Kill (⌘-K).

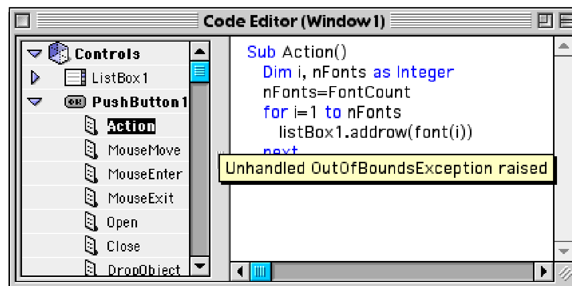
Runtime Exception Errors

When you test your application in the IDE, REALbasic performs basic syntax checking and, if it finds a problem, it stops compilation and alerts you to the problem (see Figure 138 on page 379).

There are certain types of errors that can be detected only at runtime, (i.e., when the lines of code that contain the problem are actually executed). This is because these errors depend on values that are not known until the code is running. An application containing a *runtime error* compiles without error and may even run without problems for a very long time before the lines of code containing the error are actually called. But

when these lines are executed, REALbasic has no choice but to stop execution. In more blunt language, it crashes. If you are testing the application in the IDE, the Code Editor will reappear and an error message will be shown below the line that contains the runtime error. An example runtime error message is shown in Figure 144.

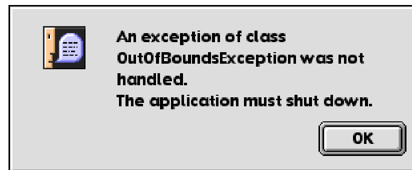
FIGURE 144. **An Unhandled Runtime Exception error in the IDE.**



This error occurs when the value of the counter, i, reaches the value of nFonts. The programmer has forgotten that font numbering starts at zero rather than one; the loop should be from zero to FontCount-1. Until the value of nFonts is actually reached at runtime, the error cannot be detected. If the error is buried in an obscure method that is very rarely called, the application may survive many hours of testing without encountering the error.

If the error occurs in a built application, REALbasic displays a generic message box that identifies the type of problem that it encountered. An example message box is shown in Figure 145.

FIGURE 145. **An Unhandled Runtime Exception Error in a Built Application.**



The application will quit when the end-user accepts the message box.

Handling Runtime Errors

REALbasic provides a way to detect and handle runtime errors. There are five types of so-called *runtime exceptions* that you can detect in your code and handle to prevent REALbasic from crashing. These runtime exceptions are shown in Table 69.

TABLE 69. **Runtime Exception types in REALbasic.**

Error Type	Description	Example
IllegalCastException	You attempted to recast an object and sent it a message its real class can't accept.	Recasting a BevelButton as a Pushbutton: Dim c as Control c=New BevelButton1 PushButton(c).Push
NilObjectException	An attempt was made to access an object that does not exist.	Accessing a Nil FolderItem: Dim f as FolderItem f.Delete

TABLE 69. **Runtime Exception types in REALbasic. (Continued)**

Error Type	Description	Example
OutOfBoundsException	An attempt was made to read from or write to a value, character, or element outside the bounds of the object or data type, i.e., you tried to access an array element that doesn't exist.	Accessing an array element that doesn't exist: Dim a(3) as String a(4)="Hugo"
StackOverflowException	If your application runs out of stack space, a StackOverflowException will occur.	Calling a method recursively until the stack overflows: The method: Sub Square (i as Integer) as Integer Return Square (i) End Sub The calling routine: Dim i as Integer i=Square(2)
TypeMismatchException	You tried to assign to an object the wrong datatype.	You tried to assign a variant that was initially assigned a picture (that was dragged into the Project Window) to an integer: Dim v as Variant Dim i as Integer v=HoneytheMailGirl i=v

Please see the description of each error type in the *Language Reference* for more examples.

You can 'catch' and handle runtime errors using an *Exception block*. With an Exception block, you can display a more informative message box (letting you track down the problem)

and prevent the application from crashing. Exception blocks always appear at the end of a method (not where you think the error might occur) because every line after the Exception line is considered part of the exception block.

When an Exception block 'catches' an exception error, the code in the Exception block runs, allowing you to obtain information on the location of the problem and the values that caused the error.

In the Code Editor, the Exception line has the same level of indentation as the Sub or Function line. You can use **Exception** alone if you wish to handle any type of exception in the Exception block, as shown below:

```
Sub…
  .
  .
Exception
 MsgBox "Something really bad happened,
       but I don't know what."
End Sub
```

The syntax of an Exception block is as follows:

Exception *errorParameter* **As** *errorType*

Both *errorParameter* and *errorType* are optional; *errorType* cannot be used without errorParameter.

The example shown above is sufficient to prevent the application from quitting, but the message is not very informative because you don't have a clue what type of exception occurred.

One way to test *ErrorParameter* is with an If statement in the Exception block:

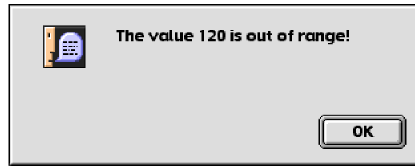
```
Sub…
.
.
Exception err
If err IsA TypeMismatchException then
MsgBox "Tried to retype an object!"
elseif err IsA NilObjectException then
MsgBox "Tried to access a Nil object!"
.
.
End if
End Sub
```

For example, the runtime exception error shown in Figure 144 on page 389 can be handled with the following exception handler.

```
Sub Action()
Dim i, nFonts as Integer
nFonts=FontCount
for i=1 to nFonts
    listBox1.addrow(Font(i))
next
Exception err
if err IsA OutOfBoundsException then
msgBox "The value "+Str(i)+" is out of
      range!"
end if
End Sub
```

When this method runs, it displays a message box such as shown in Figure 146:

FIGURE 146. **A Handled OutOfBoundsException error.**



When the end user accepts this message box, the application does not quit.

Instead of using multiple If statements, you can also use multiple Exception blocks, each of which handles a different runtime exception type:

```
Sub…
.
.
Exception err as TypeMismatchException
MsgBox "Tried to retype an object!"
Exception err as NilObjectException
MsgBox "Tried to access a Nil object!"
End Sub
```

CHAPTER 12

Communicating With The Outside World

Some applications need to communicate with other applications or even serial hardware devices to exchange information. Sometimes this is done automatically while other times it is initiated by the user. For example, when you use your computer to connect to the Internet, you are initiating communications between an application on your computer and an application on another computer at your Internet Service Provider (ISP). Fortunately, REALbasic provides controls that make communications between applications on different computers, and even communications between a computer and a serial hardware device easy.

Contents

- Communicating with Serial Devices
- Communicating with Other Computers Via TCP/IP

Communicating With Serial Devices

A serial device is a device that communicates by sending and/or receiving data in serial. This means that it is either sending data or receiving data at any one moment. It doesn't send and receive at the same time. The most common serial device is a modem. Some printers are serial devices. Serial communications using REALbasic are done with the *Serial* control. To communicate with a serial device you configure a Serial control, open the serial port to make the connection, read and/or write data to and/or from the serial device connected to one of your serial ports, and finally close the serial port when you are through to disconnect from the serial device.

Getting Set Up

So the first step is to place a Serial control in one of your project's windows or instantiate a Serial object using code. Before you can begin communicating with a serial device using a Serial control, you need to set up the Serial control so that it will know which serial port your serial device is connected to. You will also need to set the speed at which communications will occur, as well as a few other settings. This can all be done at design time using the Properties window or at runtime using code.

How you configure the Serial control's behavior properties will depend on what the serial device is expecting. Some devices can only communicate with one specific configuration. Other devices (like modems) can communicate using many different configurations. In the case of a modem, you will not only have to consider what configurations the modem will accept but also what configuration the modem your modem will be connecting to will accept. The Serial control's default configuration should

work for most modems. You may need to change the default configuration for other serial devices.

Opening the Serial Port

Once you have configured the Serial control, you can open the serial port to initiate communications with the serial device. This is done by calling the Open method of the Serial control. This method is, in fact, a function that returns True if the connection is opened and False if it is not. For example, suppose you have a Serial control whose name is "Serial1." To open the serial port using this control, you can use the following code:

```
If Serial1.Open then
  MsgBox "The serial port was opened."
Else
  MsgBox "The serial port could not be opened."
End if
```

Once you have successfully opened the serial port, it will be unavailable to all other applications (and in fact, to other Serial controls as well) until it's closed.

Reading Data

When the serial device sends data back to the Serial control that is connected to it, the Serial control's DataAvailable event handler executes. The data that has been sent back goes into a place in the computer's memory called a *buffer*. The buffer is simply a place to store the data that has been sent by the serial device because most serial devices don't have much memory of their own. When new data arrives in the buffer, REALbasic executes the DataAvailable event handler of the Serial control.

In the DataAvailable event handler, you use the Serial control's Read or ReadAll methods to get some or all of the data in the buffer. Both of these methods act as functions. Use the Read method when you want to get a specific number of bytes (characters) from the buffer. If you want to get all the data in the buffer, use the ReadAll method. In both cases, the data returned from the buffer is removed from the buffer to make room for more incoming data. If you need to examine the data in the buffer without removing it from the buffer, you can read the data from the Serial control's LookAhead property.

This example appends any incoming data to an EditField:

```
Sub DataAvailable()
  EditField1.Text=EditField1.Text+Me.ReadAll()
End Sub
```

You can clear all data from the buffer without reading it by calling the Serial control's Flush method.

Writing Data

You can send data to the serial device at any time as long as you have opened the serial port with the Serial control's Open method. You send data using the Serial control's Write method. The data you wish to send must be a string, as the Write method accepts only a string as a parameter.

The Write method is performed asynchronously. This means that the next line of code following the Write method can already be executing before all the data has actually been sent to the serial device. If you need your code to wait for all data to be sent to the serial device before continuing, call the Serial control's XmitWait method immediately following a call to the Write method.

Changing a Serial Control's Configuration on the Fly

There may be times when you need to change a Serial control's behavior properties while the serial port is open. While you can change these properties, the changes won't take effect until you close the serial port and reopen it. If you need the behavior properties to update immediately, call the Serial control's Poll method. This updates all properties immediately and calls the DataAvailable event handler immediately if there is any data waiting in the buffer.

Closing the Port

Once you are finished communicating with a serial device, you must close the serial port to end the communications session and make the port available to other Serial controls or other applications. To close the serial port, call the Close method of the Serial control that opened the serial port.

Communicating With Modems

Modems have a set of commands you can send them to tell the modem to do things such as dial a particular number. Most of these commands are the same for every modem. Your modem probably came with a guide that lists its commands. Consult that guide for more information.

TCP/IP Communications with the Socket Control

Sometimes applications need to communicate with other applications on the same network. This can be accomplished using the REALbasic's *Socket* control. The Socket control can send and receive data using TCP/IP.

TCP/IP is the protocol of the Internet. It's the way most data is transmitted via the Internet. In fact the "IP" in TCP/IP stands for "Internet Protocol."

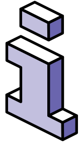
The Socket control can be used to communicate with other computers on the same network, provided they have TCP/IP drivers. In the case of the Macintosh, this driver is implemented as part of Open Transport which comes with the system software. When you connect to the Internet, you are part of the Internet network. This allows you to communicate with other computers on the Internet via TCP/IP.

Getting Set Up

You can either add a Socket control to a window or instantiate a Socket object using code. Before you can connect to another computer using the Socket control, you must first set the *port*. The port is to TCP/IP what channels are to television or frequency assignments are to radio stations. Ports give an application the ability to focus on specific data rather than receiving all the data transmitted to your computer via TCP/IP. This allows you to browse the web and send email at the same time because the web uses one port and email uses another. The port is represented by a number and there are thousands of available ports. Some have already been designated for specific functions like web browsing, email, FTP, etc. If you are designing an application

that will need to communicate with another application, you will need to find out what port the other application is using. For example, if the other application is an SMTP server, it's probably using port 25 since that is the port that is reserved for SMTP (Simple Mail Transfer Protocol).

A Socket control has a Port property that can be assigned at design time or runtime but it must be assigned a value before you can connect to another computer. If you plan on initiating the connection, you must also assign the IP address of the computer you wish to connect to the Address property of the Socket control that will make the connection.



Note: A Socket control can only be connected to one application at a time. If you need to maintain multiple connections simultaneously, you will need to have multiple Socket controls; one for each connection.

Making a Connection to Another Computer

Once you have assigned a port and an IP address, you can connect to an application on the computer at that IP address, provided that the application is listening for TCP/IP connections on the port you have specified. To initiate a connection, you simply call the Socket control's Connect method. If a connection is established, the Socket control's Connected event handler executes. If a connection is not established, an error occurs and the Socket control's Error event handler is executed. Once a connection is established, your application can begin sending and receiving data with the application at the other end of the connection.

Listening For a Connection From Another Computer

In some cases you may want your application to wait for another application to connect to it rather than initiate the connection. To do this, you use the Socket control's Listen method. For example you have a button that when pressed, causes the application to listen for a TCP/IP connection on the port number that is assigned to the Socket's Port property. Let's assume that the Socket control is named "Socket1." In the pushbutton's Action event handler, you use the following code:

```
Sub Action()
  Socket1.Listen
End Sub
```

Once a connection is established, the Socket control's Connected event handler executes, letting you know that you have a connection.

Reading Data

When the application at the other end of the connection sends data back to the Socket control that it's connected to, the Socket control's DataAvailable event handler executes. The data that has been sent back goes into a place in the computer's memory called a *buffer*. The buffer is simply a place to store the data that has been sent by the other application. When new data arrives in the buffer, REALbasic executes the DataAvailable event handler of the Socket control.

In the DataAvailable event handler, you can use the Socket control's Read or ReadAll methods to get some or all of the data in the buffer. Both of these methods act as functions. Use the Read method when you want to get a specific number of bytes

(characters) from the buffer. If you want to get all the data in the buffer, use the ReadAll method. In both cases, the data returned from the buffer is removed from the buffer to make room for more incoming data. If you need to examine the data in the buffer without removing it from the buffer, you can read the data from the Socket control's LookAhead property.

This example appends any incoming data to an EditField:

```
Sub DataAvailable()
  EditField1.Text=EditField1.Text+Me.ReadAll()
End Sub
```

Writing Data

You can send data to the application you are connected to at any time. You send data using the Socket control's Write method. The data you wish to send must be a string, as the Write method accepts only a string as a parameter. In this example, the text from an EditField is being sent via a Socket control:

```
Socket1.Write EditField1.Text
```

Handling Errors

Errors can occur attempting to connect, sending data, or receiving data. Errors are not always what they seem. For example, when the other computer closes the connection, an error is generated. When an error occurs, the Socket control's Error event handler is executed. Errors are represented by numbers. The Socket control's LastErrorCode property will contain the number of the last error that occurred. See the Socket Control in the Language Reference for a complete list of error numbers.

Errors are simply ways to alert your application to conditions it may not have anticipated or be able to anticipate. For example, if you attempt to make a connection or listen for one and you don't have Open Transport installed, an error is generated.

Closing the Connection

When you are finished communicating and wish to disconnect from the other application, you do so by closing the connection. The connect is closed by calling the Socket control's Close method. Suppose you have a Socket named "Socket1" that has established a connection. To close the connection, you can use the following code:

```
Socket1.Close
```

Understanding Protocols

Any kind of communication requires that all parties involved agree on a method of communication and a language. For example, if you want to communicate with a friend, you might go talk to them face to face, call them on the phone, or send them email. Both of you must be able to communicate using the same language or you won't be able to communicate at all. Communications via TCP/IP work the same way. The language used is called a *protocol*. A protocol is simply an organized way of sending and/or receiving information.

If you are writing an application that will communicate with another application via TCP/IP, you will need to understand the protocol the other application will be expecting in order to communicate with it. For example, on the Internet, the protocol for the world wide web is called HTTP (HyperText Transfer Protocol), the protocol for sending email is called SMTP (Simple Mail Transfer Protocol), and the protocol for receiving email mail

is called POP3 (Post Office Protocol 3). Complete descriptions of these Internet protocols and others are available on the Internet. The descriptions of these protocols are called RFCs (Request For Comments). The easiest way to find information on RFCs is to go to www.yahoo.com and search for "RFC". This will give you a list of links to various web sites that explain all of the various Internet protocols.

If you are writing an application that communicates with another applications you have written, then you can define your own protocol. Your protocol will simply be a set of commands you define that allow the applications to understand what the other wants.

CHAPTER 13

Extending the Capabilities of REALbasic

One of the things that makes REALbasic easy to learn and use is that it abstracts you from the inner workings of the operating system. You don't have to know any of the 8,000 commands that make up the API (application programming interface) used to work with the Mac OS. This also means that REALbasic may not have a particular capability that you require. Fortunately, REALbasic provides several ways to extend its capabilities, allowing you to add just about any functionality you need.

Contents

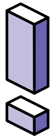
- Using XCMDs and XFCNs
- Making Toolbox calls
- Calling AppleScripts
- Communicating with AppleEvents

- Using and Writing REALbasic Plug-ins
- Using PowerPC Shared Libraries

Using XCMDs and XFCNs

XCMDs are individual commands written in a language like Pascal, C, or C++ and then compiled. XFCNs are the same thing except that they are functions so they return a value. The "X" in the name is short for "external," which simply means a command that is external to the environment or "not built-in." XCMDs and XFCNs became popular with Apple Computer's HyperCard application. HyperCard does not allow the programmer direct access to the Mac OS, so programmers write XCMDs and XFCNs when they need this kind of access. There are thousands of public domain XCMDs and XFCNs available, especially on the Internet. Fortunately, REALbasic supports this external format.

There are two types of XCMDs: type 1 and type 2. Apple Computer created the type 2 format to add additional capabilities to XCMDs. Most existing XCMDs are type 1. REALbasic currently supports type 1 XCMDs and will support type 2 XCMDs in a future release.



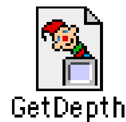
Note: XCMDs and XFCNs are not supported on Windows builds.

Getting an XCMD Out of a HyperCard Stack

To use an XCMD or XFCN with REALbasic, you must have it in its standalone format. Most of these commands and functions are installed inside HyperCard files called "stacks." There are applications available that can look inside a HyperCard stack and extract any XCMDs or XFCNs and save them has individual

desktop files. One such application is written in REALbasic and is called "Xtractor." It was written by Red Designs and is available on the REALbasic CD and at the REAL Software web and ftp sites. An XCMD or XFCN once extracted appears as a ResEdit file.

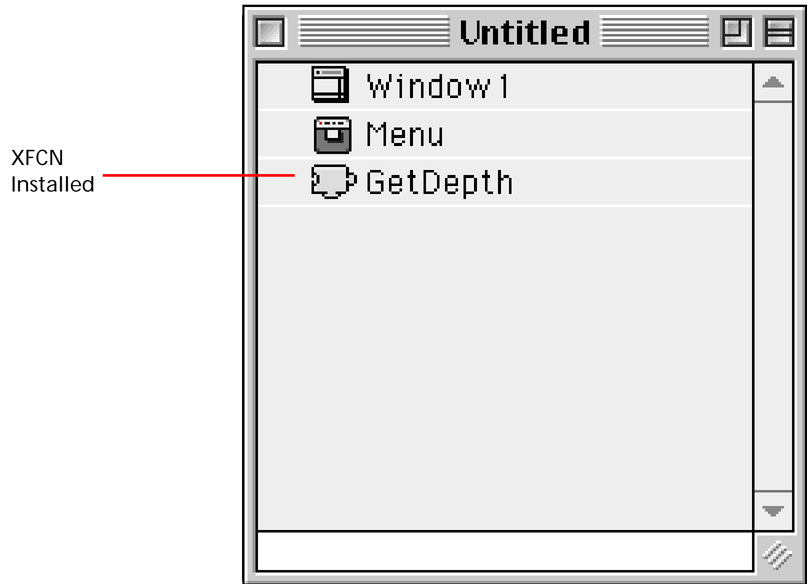
FIGURE 147. An Extracted XFCN



Installing an XCMD/XFCN in a Project

Once you have an XCMD/XFCN extracted from a HyperCard stack, loading it into your project is easy. To load the XCMD/XFCN into your project, just drag it into the Project window. It will appear in the Project window with a special icon that indicates that it's an XCMD or XFCN.

FIGURE 148. **An XFCN installed in a project**



Calling an XCMD or XFCN in a Method

XCMDs are called the same way methods are and are passed parameters (when they require them) the same way parameters are passed to methods. The following is an example of how a XCMD called SetSound would be called:

```
SetSound 10
```

XFCNs are called and passed parameters the same way functions in REALbasic are. The following is an example of how an XFCN called GetSound would be called:

```
Dim i as Integer
i=GetSound()
```

Removing an XCMD or XFCN

To remove an XCMD or XFCN from a project, click on the XCMD/CFCN in the Project window to select it and press the Delete key or choose Edit ▶ Clear.

Where to Find XCMDs and XFCNs

The best place to look for XCMDs and XFCNs that might provide some functionality you need is on the Internet. A quick search using Yahoo on the keyword "Hypercard" listed about 40 sites. The XCMD Hideout is one place to start. It's located at http://www.nmc.csulb.edu/projects/xcmdhideout/ and includes source code for some of the XCMDs available there.

Making Toolbox Calls

Using the Declare statement, you can access the toolbox on either the Macintosh or Windows platforms. PPC, 68K, and Intel machines are supported. You need to use the conditional compilation feature to isolate your Declare statements for each platform. With the Declare statement, you specify the name of the toolbox call and its shared library, and the parameters the call uses. If the call returns a value, you specify the data type of the value that is returned.

If the functionality is available on both platforms, you can use the same name for both platforms. However, often the parameters for the call will be different. Use conditional compilation to isolate your calls as well.

The following button Action uses the Macintosh Speech manager to speak the text in an EditField:

```
dim s as string
dim i as integer
#if TargetMacOS then
 Declare Function SpeakString lib "SpeechLib"
  (SpeakString as pstring) as Integer
#endif
s=editField1.text
#if TargetMacOS then
 i=SpeakString(s)
#else
 MsgBox "Speech is supported only on Macintosh!"
#endif
```

If the name of the toolbox call is the same as a REALbasic method, use the Alias keyword to refer to the call. For example, if SpeakString was the name of a REALbasic method, you could not use the above syntax. You could use, for example:

```
Declare Function MySpeakString lib "SpeechLib"
Alias "SpeakString" (SpeakString as pstring)
as Integer
```

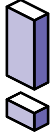
You would then use MySpeakString in your code to invoke the toolbox call.

See the description of the Declare statement in the *Language Reference* for more information.

Calling AppleScripts

AppleScript is Apple Computer's system-level scripting language that makes controlling applications easy. REALbasic supports

AppleScript. You can write a script in AppleScript and then call that script in your REALbasic project.



Note: Applescript is available for use only on Macintosh.

Preparing an AppleScript to Work in REALbasic

In order for REALbasic to run an AppleScript, the entire script must be enclosed in an on run handler like this:

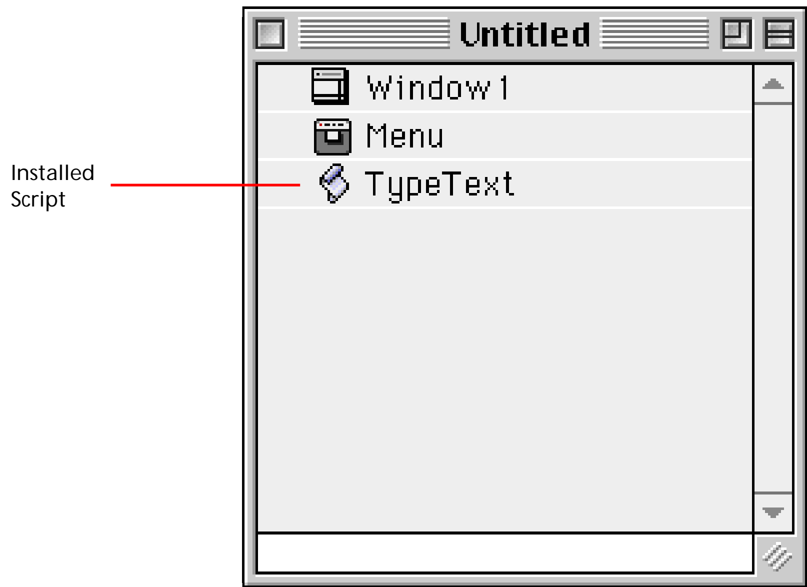
```
on run
  //your script code goes here
end run
```

Next, your script must be saved as a compiled script. In the Script Editor supplied by Apple Computer, choose File ▶ Save As. Then choose Compiled Script from the Kind popup menu.

Loading an AppleScript into a Project

To load an AppleScript, just drag your compiled script file into the REALbasic Project window. The script will appear with a script icon next to it. Figure 149 shows an example of a project with a script installed.

FIGURE 149. A compiled AppleScript in Project window



When you drag a compiled script into your Project window, REALbasic copies the script into the Project. Therefore, once you have dragged the script into the project, you can delete the compiled script if you don't need to use it elsewhere.

Passing Values To an AppleScript

If you are writing a script you want to pass parameters to, the parameters must be enclosed in curly braces following the on run statement. In the following example, the x and y are parameter variables that will hold the values of the two parameters passed to the script:

```
on run {x,y}
  //your script code goes here
end run
```

Returning Values From an AppleScript

You write a script to act as a function by having it return a value. To return a value from a script, simply use the return command in AppleScript followed by the value you wish to return. This simple example takes a number of days and returns the equivalent number of years:

```
on run {daysOld}
  return daysOld/365
end run
```

Calling an AppleScript

Scripts are called just like the built-in methods and functions. Type the name of the script as it appears in the Project window. If the script requires parameters, the parameters follow the name of the script just as they do with any of the built-in commands. This example calls a script that sets the sound level of the Macintosh to 5:

```
SetSoundLevel 5
```

Scripts that return values (acting as functions) work just like the built-in REALbasic functions. This script gets the current sound level and assigns it to a variable:

```
Dim level as Integer
level=GetSoundLevel()
```

Removing an AppleScript

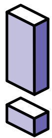
To remove a script from a project, click on the script in the Project window to select it then press the Delete key.

Communicating with AppleEvents

AppleEvents is the core communications system between applications on the Macintosh. As a matter of fact, when you are calling AppleScript code, AppleScript is actually performing all of its magic with AppleEvents. When you choose Special ▶ Restart in the Finder, the Finder sends a "Quit" AppleEvent to any open applications. This particular AppleEvent is one that all applications are required to support.

You can perform some very fast and powerful actions with AppleEvents. You create AppleEvent objects in REALbasic using the NewAppleEvent function. AppleEvents have three parts: an event class, an event ID, and the creator code of the target application.

The Event Class and Event ID together uniquely define a particular AppleEvent. The EventClass acts as a category for logically grouping events together. While there are many standard (and even some required) AppleEvents, many applications have several custom AppleEvents for performing actions specific to the application. Consult the application's documentation or its author to get information on what custom AppleEvents may be available.



Note: AppleEvents work only on Macintosh.

Sending AppleEvents

Once you create the AppleEvent object and populate the necessary parameters with data, you then send the AppleEvent to the target application using the AppleEvent object's Send method.

In this example, an AppleEvent is created to tell the Finder to restart the Macintosh. "FNDR" is the class of AppleEvent and "rest" is the event ID. "rest" is clearly short for "restart". Finally, "MACS" is the creator code for the Finder. The AppleEvent class has a Send method. This method is a function that returns True if the AppleEvent is successful and False if it fails.

```
dim ae as AppleEvent
ae=newAppleEvent("FNDR","rest","MACS")
if not ae.send then
 msgBox "The computer couldn't be restarted."
end if
```

Receiving AppleEvents

In order to receive AppleEvents your project must have a subclass that has Application as its Super property value. That's because the Application class is the only class with a HandleAppleEvent event handler. When your application receives an AppleEvent, the application class HandleAppleEvent event handler is executed and the AppleEvent is passed as a parameter to the event handler.

This event handler, when called, is passed an AppleEvent object, the event class, and the event id. There are required AppleEvents that your application should support. One of the them is the Quit AppleEvent.

In this example, if the HandleAppleEvent event handler receives a quit AppleEvent from the Finder, it calls the Quit method.

```
Function HandleAppleEvent(Event as AppleEvent,
eventClass as String, EventID as String) as
Boolean
  if eventClass="aevt" and eventID="quit" then
```

```
    //the Finder wants the app to quit
    beep
    msgBox "I must quit now."
    quit
  end if
End Function
```

You can create your own set of AppleEvent classes and event IDs for your application that represent various actions your application can take in response to them.

Sophisticated AppleEvents

AppleEvents can actually contain quite a bit of very specific data. AppleEvents for example, can be used write to applications that process data for web servers. For more information on AppleEvents, see the AppleEvent class in the *Language Reference*.

Using and Writing REALbasic Plug-ins

Many applications have their own plug-in format. Netscape Navigator, Adobe PhotoShop, 4th Dimension, are just a few examples of applications that have a plug-in format. Plug-ins are a way for an application to be extended by other programmers. For example, there is a plug-in for Netscape Navigator that allows it to play QuickTime movies that have been embedded into web pages.

REALbasic also has its own plug-in format. Plug-ins are written in languages like C and C++. For example, James Milne of REAL Software wrote a plug-in for REALbasic that plays a particular type of music file. REALbasic also uses plug-ins to manage connectivity to database back ends. You can add support for

other database engines simply by writing (or obtaining from a third-party) the plug-in for that database engine.

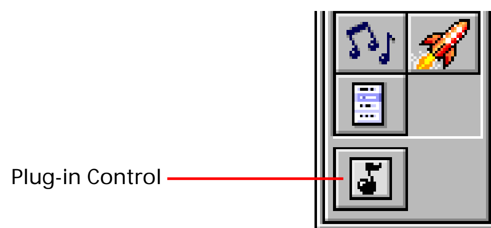
Loading Plug-ins

Loading plug-ins is easy. Simply create a folder called "Plugins" in the same folder that contains REALbasic. Then drop your plug-in files into that folder. Any plug-ins in this folder will automatically be available to your projects.

Using Plug-ins

Some plug-ins are in the form of controls similar to those that appear in the REALbasic Tools window. When you have this type of plug-in in your Plug-ins folder, a new control will appear in the Tools window. Plug-in controls are visually different from the built-in controls. Plug-in controls appear raised while the built-in controls appear sunken. Figure 150 shows an example of a plug-in control as it appears in the Tools window.

FIGURE 150. **A plug-in control in the Tools window**



You use a plug-in control the same way you use any other control in the Tools window, by dragging it to a window. The properties window will then display any properties that can be set from the Design environment.

Plug-ins can also be a set of methods that has no interface whatsoever. Plug-ins of this type do not appear anywhere in the interface. You must have some documentation to know which methods exist in the plug-in, what the methods do, and how to use them.

Including Plug-ins in Your Stand-Alone Applications

When you build a stand-alone application from your project, any plug-ins you are using in your project will automatically be built-in to the stand-alone application.

Writing Your Own Plug-ins

If you know C or C++, you can write REALbasic native plug-ins. The REALbasic Plug-in Software Development Kit (SDK) is available on the REALbasic CD and at the REAL Software web and ftp sites. This kit contains all the information you need to write plug-ins including sample plug-ins and include files for Metrowerks CodeWarrior.

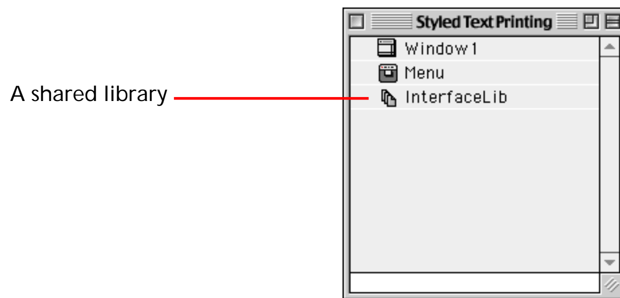
Using PowerPC Shared Libraries

PowerPC shared libraries are files that have subroutines that can be called and passed parameters. These parameters are referred to as "entry points" and, as the name suggests, these libraries run only on PowerPC-based Macintosh computers. REALbasic supports shared libraries. This can be a convenient way to write external code for REALbasic, especially if you want to use the same code with other applications that support shared libraries.

Accessing Commands In Shared Libraries

To access the commands (or "entry points") in a shared library from within a REALbasic project, you must first load the shared library into the project. This is done by dragging the shared library into the Project window. Figure 151 shows an example of a shared library loaded into a project.

FIGURE 151. A shared library loaded into a project

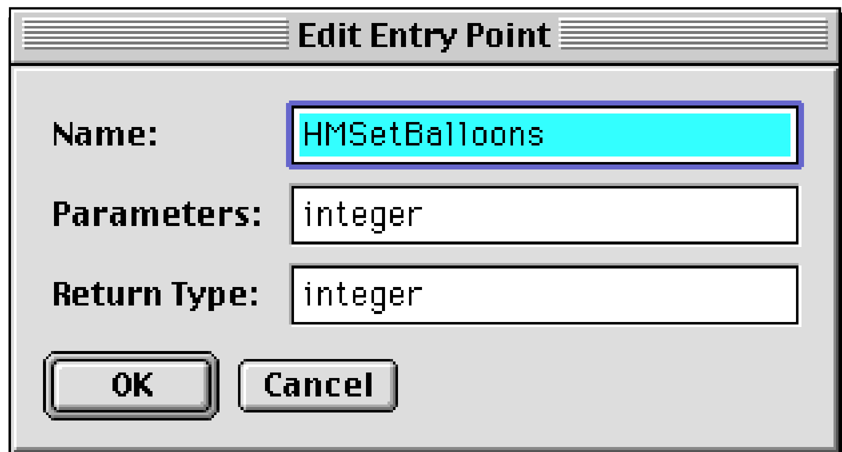


Next, you need to define the entry points you are going to access in the shared library. This is done with the Entry Point Editor. You can access this editor by double-clicking on the shared library in the Project window. The Entry Point Editor displays a list of entry points you have defined. Figure 152 shows a list of entry points.

FIGURE 152. **The Entry Point Editor.**

Click the New button to define a new entry point, select an entry point from the list, and click the Edit button to edit it. This displays the Edit window where you name the entry point and define its parameters. Figure 153 shows this Edit window.

FIGURE 153. **The Edit Entry Point window.**



You must know the name of the entry point and its parameter types in order to successfully use a PowerPC shared library in your code.

Calling Commands In Shared Libraries

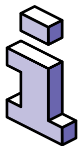
To use an entry point from a shared library in your code, simply call the entry point as if it were a method of a module. In this example, the HMSetBalloons entry point (for turning Balloon Help on or off) is passed a 1 to turn Balloon Help on and a 0 to turn it off. This particular entry point is a function that returns an error code, so it is being called as such.

```
Dim err as Integer
err = InterfaceLib.HMSetBalloons(1)
```

CHAPTER 14

Building Stand-Alone Applications

When you are ready to turn your project into a stand-alone application, there are a few things you will need to know. This chapter will help you understand what finishing touches your application may need to make it complete.



If you have the Standard Edition of REALbasic, you can only build demo versions of Windows applications. A demo version quits automatically after 5 minutes. You can build fully functional Macintosh applications with either the Standard or Professional editions.

Contents

- Building Your Application
- Project Window Items
- Assigning Custom Icons

- Registering Your Creator Code
- Using and Writing REALbasic Plugins
- The Thread Manager

Building Your Application

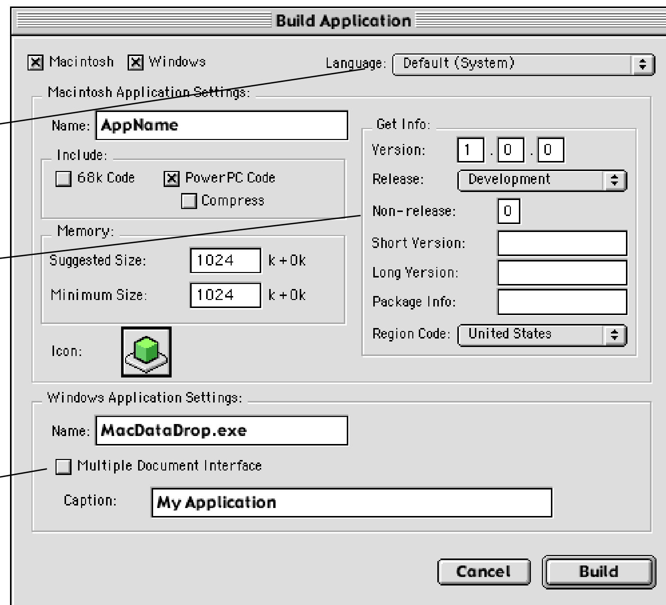
Building a stand-alone version of your project as an application couldn't be easier than it is in REALbasic. Just choose File ▶ Build Application or press ⌘-M. This displays the Build Application dialog box. Figure 154 shows this dialog box.

FIGURE 154. **The Build Application dialog box.**

The Language pop-up defaults to the default language selected in Edit ▶ Project Settings.

Some Get Info settings appear in the Apple Get Info dialog box for the built application.

Check MDI if you want your Windows app to run inside a 'master' window. Enter the text of the master window title in the Caption area.



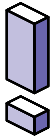
In this dialog box you assign a few settings and then click the Build button to create the stand-alone version of your application.

You have the options of:

- Building for either Macintosh or Windows (or both).
- For Macintosh, building only 68K or PPC code (or both).
- Compressing the PPC version (if applicable).
- Setting the minimum and preferred memory size for Macintosh.
- Specifying MDI (Multiple Document Interface) for the Windows build and setting the text that appears in the master window's title bar. The MDI option encloses all of your application's windows in one 'master window' whose Title bar text is entered in the Caption area of the Build Application dialog.

  If you deselect Multiple Document Interface, your application's windows will appear alongside other applications' windows (like a Macintosh).
- Setting the Language for the build. The default language set in the Project Settings dialog box is the default.
- Entering version information that will appear in the Get Info dialog box, the Finder's List view, and written to the 'vers' resource of your application.

A standalone application includes a library of routines that support almost all the controls, classes, global methods, etc. that comprise REALbasic, plus any plug-ins that are used in your application. This means, for example, that a standalone application contains code to manage ListBoxes even if your application doesn't use any ListBoxes.



The larger the project, the more memory REALbasic needs to build the application. If REALbasic cannot successfully build a standalone application, allocate more memory to the REALbasic application.

Compiling for Windows

Most notably, the Build Application dialog box lets you compile your application for Windows computers. Simply check the Windows checkbox and enter the name of the .exe file in the Windows Application Settings area. The compiler will create a single executable application for the Windows environment. On Macintosh, the Windows application will have a Virtual PC™ icon; if Virtual PC is installed, you can double-click the Windows build to launch Virtual PC and open your application.

When you drag the .exe file to your Windows computer or to a Virtual PC directory, it will have the standard REALbasic application icon:

FIGURE 155. **Windows builds of a REALbasic application.**

Windows Considerations

Before building an application for deployment on Windows, you should check the following issues for compatibility:

- **End of Line Characters**: When inserting text into EditFields via code, keep in my that carriage returns must be followed by line feeds.

- **High ASCII Codes**: Characters above ASCII 127 are different on Macintosh and Windows. For example, the bullet character on the Macintosh is ASCII 165 but on Windows it's ASCII 149. See Appendix B in the Language Reference for a table of ASCII codes.
- **MDI Interface**: The size of the MDI Frame (the parent window) is not currently configurable. That means you have to consider this when determining the size of any window you are going to open.
- **Windows GUI**: It's important to consider Windows user interface guidelines. Otherwise, your application will look like a port from the Mac to Windows rather than a genuine Windows application.
- **Interface Elements**: Many Windows interface elements are different from the Macintosh. For example, a resizable document window on Macintosh has a grow icon. On Windows it doesn't. If any of your interface controls are expecting a grow icon (because of their placement) they won't look right on Windows. You will need to consider this.
- **Control Order**: The control order of interface controls becomes more important on Windows since controls like Checkboxes and Pushbuttons can get the focus. Make sure you test the control order on Windows before building your application.
- **Single Document Interface windows**: Single Document Interface applications (created when the Multiple Document Interface checkbox is left unchecked in the Build Application dialog) should have a separator line control at the top of the main window to separate the content region of the window from the menus.
- **Mac-only Controls**: Some controls (like the ChasingArrows) are unique to the Macintosh and don't have a Windows counterpart. Although REALbasic provides Windows version of these

controls, you should consider whether or not they will be appropriate for the Windows version of your application.

- **Windows Menus**: The text of certain standard menu items on Windows are different. For example, "Exit" is used instead of "Quit". You can use the REALbasic constants system to localize your application's interface for Windows in this respect (see "Using Constants to Localize Your Application" on page 222). Also Windows application menu items usually have keyboard accelerators. These can be inserted into your menu items (see "Using Constants to Add Windows Keyboard Shortcuts to Menus and Menu Items" on page 224).
- **Mac-only Features**: A few REALbasic capabilities are Macintosh-specific. AppleEvents and AppleScript, XCMDs, resources, and shared libraries. You can use conditional compilation to isolate this code. It uses the structure:

```
#If TargetBoolean then
 //platform-specific code, included in
 //the built app when TargetBoolean is True
#Endif
```

TargetBoolean is a boolean constant, TargetMacOS, Target68K, TargetPPC, or TargetWin32, that lets you selectively include code that will be included only on a particular platform or environment. See the section in the *Language Reference* on Cross-Platform Development and the descriptions of TargetMacOS and TargetWin32 for more information.

- **Mac-specific FolderItems**: Some of the functions that return references to MacOS-specific folders, such as TrashFolder and ControlPanelsFolder, return Nil on Windows or return FolderItems that are different than what you might be expecting. Be sure that your code checks for Nil values and/or the platform on which the application is running when using these functions.

- **Toolbox Calls**: With the Declare statement, you can make direct toolbox calls for either the Macintosh or Windows platforms. Of course, the nature of the toolbox call will differ by platform. Use conditional compilation to isolate both the Declare statements and your usage of your toolbox calls later in your code.
- **Database data sources**: Only the ODBC driver and the built-in REAL database engine are currently supported on Windows.

Default Language

If you have provided support for more than one language via constants (see "Using Constants to Localize Your Application" on page 222), you can choose the default language for this build from the Default Language pop-up menu. The default language is the language that was selected in the Project Settings dialog box.

Including 68K and PowerPC Code

You can choose to include 68k code, PowerPC code, or both. A 68K code application will run on 68K machines and in emulation mode on PowerPCs. A PowerPC application will run in native mode on a PowerPC computer but won't run at all on a 68K machine. Unless your application is doing lots of computations or running lots of loops, a PowerPC user might not notice the difference between a 68K and PowerPC application. Including both makes the application larger, so you might want to do some testing with both to see whether your users will benefit from the extra code.

Compressing PowerPC Code

If you are going to include PowerPC code, you can choose to compress it. This reduces the size of the code that REALbasic adds to your application by about 50 per cent. However, it also adds 600k to the memory requirement and adds to the time

required to launch the application. You are sacrificing memory for a smaller disk footprint.

Memory Settings

By default, REALbasic sets the suggested and preferred memory settings to 1024k. Is this too much? Is this too little? How do I know what the right amount of memory is for my application? Well, as it turns out it's not all that straightforward. For the most part, it just takes some experimenting. One indicator is the Finder's About This Computer dialog box located in the Apple menu. This dialog box lists all running applications along with the amount of memory reserved for them and the percentage of the memory they are actually using. For many applications you will find that 1024k is plenty of memory. It may even be overkill. For other applications it won't be nearly enough because they are doing things that are using up memory. For example, loading lots of data into memory, especially pictures, increases the memory requirement. If you are using the sprite engine, the more sprites displayed at once, the more memory you will need. Unfortunately, there is no really straightforward logic to determining the memory requirement for your application. You will have to experiment.

Custom Application Icons

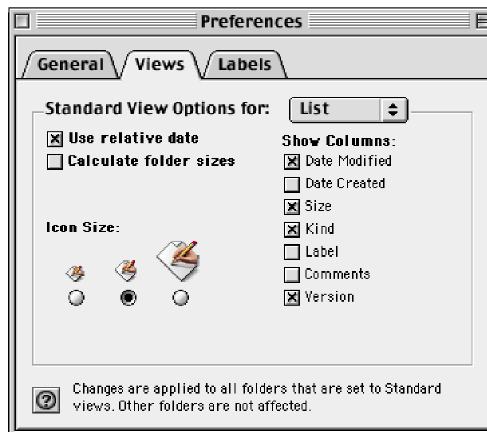
The Build Application dialog box has a place for you to paste in a custom application icon. Copy your icon to the Clipboard, click on the icon field in the Build Application dialog box, and choose Paste from the Edit menu. The icon you paste will be preserved once you build your application. If your stand-alone application doesn't display your custom icon, you may need to rebuild the desktop. This can be done by restarting your Macintosh and holding down the Command and Option keys until a dialog box appears asking you if you wish to rebuild the desktop.

Get Info

Some of the information in the Get Info area of the Build Application dialog box will be displayed when the user clicks on your application icon and chooses File ▶ Get Info (⌘-I). The text you enter in the Package Info area appears directly below the application's name. The text you enter in the Long Version entry area appears in the Version area below the modification date.

If the user chooses Edit ▶ Preferences in the Finder and, in the Views panel, selects Version in the Show Columns list, a "Short Version" info column will be added to the Finder's List views. The text you enter in the Short Version area will then be shown in the Version column in the user's Finder List views.

FIGURE 156. **The Version option in the Views Preferences panel.**

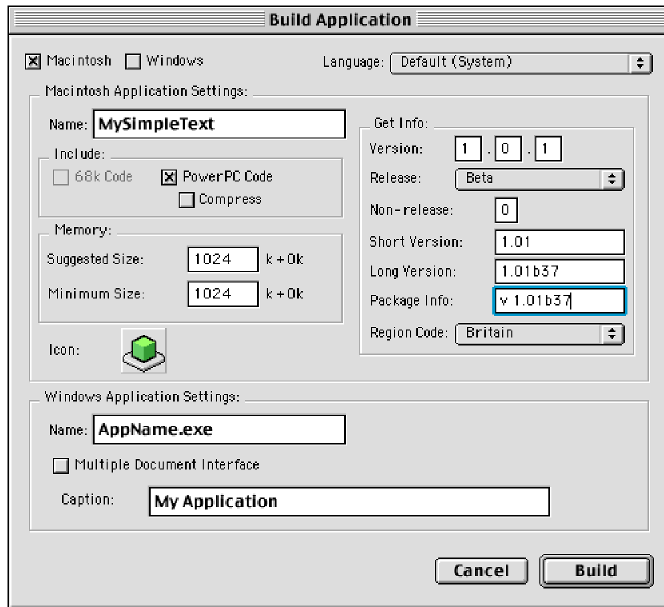


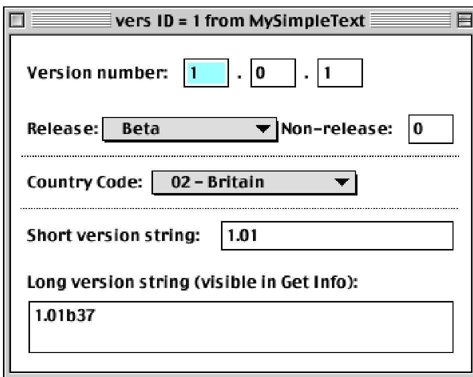
The remaining Get Info settings are not visible but are stored in your built application's 'vers' resource. The 'vers' resource can be used to store information about a individual application or, if it is part of a set of files, to the group of files. The 'vers' resource with an ID of 1 specifies version information for the application; the

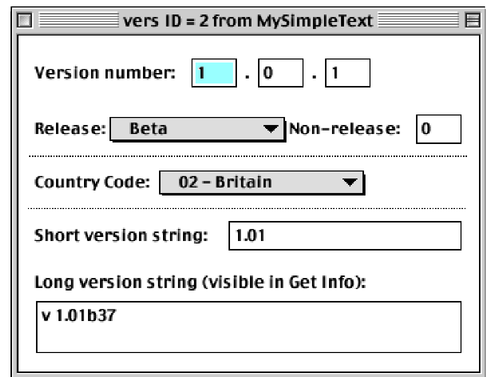
version resource with an ID of 2 specifies version information for a set of files.

Figure 157 shows the relationship between the REALbasic Build Application dialog and the ‘vers’ resource settings:

FIGURE 157. **Build Application dialog and ‘vers’ resources.**







Although ResEdit's labelling is confusing, the string in the Long Version area for vers ID2 corresponds to the Package Info field in the REALbasic Build Application dialog box.

The following elements are stored:

- Major version level, in binary-coded decimal format. Accessible only from the 'vers' resource in the first byte.
- Minor version level, in binary-coded decimal format. Accessible only from the 'vers' resource in the second byte.
- Release level. Accessible only from the 'vers' resource in the third byte. Levels are coded as follows:

TABLE 70. Values for Release Level

Value	Description
x20	Development
x40	Alpha
x60	Beta
x80	Release

- Pre-release version level. Accessible only from the 'vers' resource.
- Region code. Identifies the script system for which the application is intended. The values stored in the 'vers' resource are given in Table 71 on page 439.
- Short version. Identifies the version number of the software. This may be displayed, at the end-user's option, in the Finder's List views.
- Long version/Package Info. Long Version contains the version number and other identifying information about the company, developer, or group of files. For 'vers' ID 1, the Long Version is displayed in the version field of the built application's Get Info

window. For ‘vers’ ID 2, the string is displayed under the application’s name and next to the application’s icon at the top of the Get Info window. It is referred to as ‘Package Info’ in the Build Application dialog box.

You can find more information about the ‘vers’ resource at http://developer.apple.com/techpubs/mac/Toolbox/Toolbox-454.html.

Using the GetResource method of the ResourceFork class, you can access the information contained in the ‘vers’ resource of your built application.

Project Window Items

Your Project window lists many different kinds of resources used in your code. You may have:

- QuickTime movies
- Pictures
- Sounds
- Databases
- PowerPC Shared Libraries
- AppleScripts
- Classes
- Modules
- Windows
- XCMDs

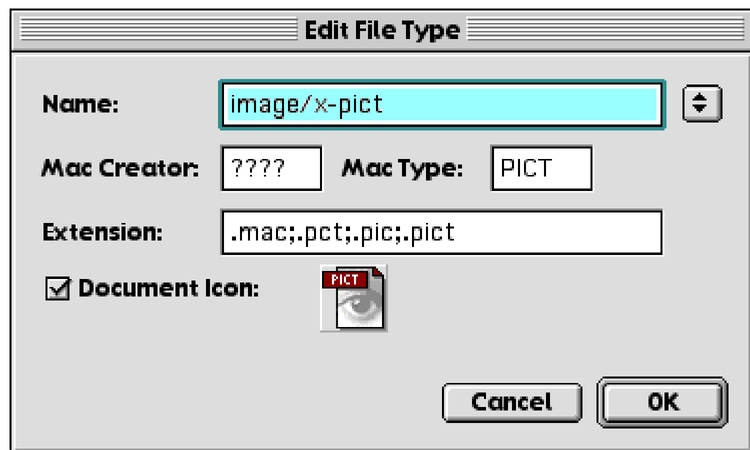
- REALbasic Plugins

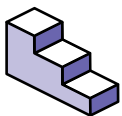
Databases that you include in your Project Window are not included in the built application. End users must have access to the data sources that your application references.

Assigning Custom Icons

As you already know, you can assign a custom application icon in the Build Application dialog box. If your application creates documents, you will probably want to assign icons that match the theme of your application icon, to the documents it creates. This can be done through the File Types dialog box.

FIGURE 158. **A File Type in the File Types dialog box**





To assign a custom icon to a particular file type, do this:

1. Use any graphics program to create your icon. Remember, its size is 32 x 32 pixels.

2. Copy the icon you wish for a particular file type to the Clipboard.
3. Open the File Type in the File Types dialog box.
4. Click the Document Icon checkbox.
5. Click on the blank document icon and choose Edit ▶ Paste (⌘-V).
6. Click the OK button.

Registering Your Creator Code

Each application's creator code should be unique. This is because the Finder uses these codes to determine which application to launch when a file is double-clicked. The Finder simply locates the first application it can find with a matching creator code.

You can register your application's creator code with Apple Computer to be reasonably sure that it's unique. To register your creator code, go to the following web address:

http://developer.apple.com/dev/cftype/information.html

The Thread Manager

While REALbasic requires the Thread Manager to be installed in order for it to run, your stand-alone application may not. Your application will not require the Thread Manager unless you are creating subclasses based on the Thread class. The Thread Manager was installed as an extension in prior to System 7.5 when it was built-in to the Mac OS.

Appendix: Region Codes

The following table gives the Region Codes that are used in the 'vers' resource for built applications.

TABLE 71. **Region codes used in the 'vers' resource.**

Code	Value	Code	Value
00	US	22	Malta
01	France	23	Cyprus
02	Britain	24	Turkey
03	Germany	25	Yugoslavia
04	Italy	33	India
05	Netherlands	34	Pakistan
06	Belgium-Lux.	36	It. Swiss
07	Sweden	40	Anc. Greek
08	Spain	41	Lithuania
09	Denmark	42	Poland
10	Portugal	43	Hungary
11	Fr. Canada	44	Estonia
12	Norway	45	Latvia
13	Israel	46	Lapland
14	Japan	47	Faeroe Isl.
15	Australia	48	Iran
16	Arabia	49	Russia
17	Finland	50	Ireland
18	Fr. Swiss	51	Korea
19	Gr. Swiss	52	China
20	Greece	53	Taiwan
21	Iceland	54	Thailand

CHAPTER 15

Converting Visual Basic Projects to REALbasic

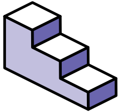
Because of the similarities between REALbasic and Visual Basic, creating a Macintosh version of a Visual Basic application is fairly easy. REALbasic can save you hours of time by handling the tedious job of recreating the interface and pasting in your code into all the various event handlers and methods.

Contents

- Importing Forms and Code
- Tips that make the process easier
- What about VBX and ActiveX controls?
- Database Options

Importing Forms and Code

REALbasic can import Visual Basic 2.0 (or greater) forms and the code associated with them. REALbasic recreates the interface and imports the code for all of the various controls into the appropriate event handlers and methods.



To import VB forms into a REALbasic project, do this:

1. Move your Visual Basic form files (those ending in.frm) over to your Macintosh.
2. Open a new or existing project in REALbasic.
3. Drag the form files from the desktop into the Project window to import them.

Because there are differences between REALbasic and Visual Basic, there are going to be errors in your code that you will need to correct. Once you finish importing the form files, you can run your project to begin tracking down those errors.

Making The Conversion Easier

There are a number of steps you can take proactively to make the process of converting a VB project to REALbasic easier. First, there is a preprocessor for VB applications on your REALbasic CD,VB Cleaner. It preprocesses VB forms, projects, and classes and does an extensive job of preparing these files for import, saving hours and hours of time. As a first step, run VB Cleaner on your VB application.

Here are some specific tips for manual preprocessing VB applications. Special thanks to Tony Hansen for compiling this list of do's and don'ts based on his first-hand experience:

- Don't use the object!property syntax, use object.property
- Remove all type declaration characters (no A$, n%, etc.)
- Don't use the Call syntax on subs
- Don't use '* x' declarations on strings (i.e. dim strA as string * 10)
- Don't use REDIM
- Minimize use of variants
- Don't change passed vars within functions and subs
- Use 'one place in and one place out' design in functions and subs
- Make sure source files are being maintained in text
- Only one statement per line

In REALbasic you will have to:

- Remove the $ on str$, trim$, etc
- Change ucase to uppercase, .clear to .deleteallrows, .additem to addrow, etc.
- Change the way files are opened and read

What about VBX and ActiveX controls?

VBX and ActiveX controls are usually written using quite a bit of Windows API calls, making them very Windows dependent. However, if you are using these for Internet, for serial

connectivity, or for animation, there are controls that are built-in to REALbasic that provide this functionality. The Visual Basic importer won't do the conversion for you in these cases but you do have a solution.

What Are My Database Options?

Visual Basic applications often use Microsoft Access or the Jet database engine that comes with Visual Basic to provide single-user database capabilities. You can convert these applications to use the built-in REALbasic database engine or any other supported data source.

Index

Numerics

4D Server 356
4th Dimension 356, 357

A

Aaron extension 230
accelerators 224
  Windows (OS) 224
Action event handler 163, 326
ActiveX controls 443
Add File Type dialog box 280
AddMacData method 271
AddPicture method 311, 316
AddResource method 316
Address property 401
AddRow method 144
aliases 283
  importing 181
And operator 148
animation
  starting and stopping 273
  with sprites 272–274
App function 339
Appearance Control panel 51
Appearance Manager 51
AppendToTextFile method 298
AppleEvent object
  Send method 416
AppleEvent objects
  creating in REALbasic 416
AppleEvents 430
  communicating with 416–418
  receiving 417–418
  required 417
  sending 416
  sophisticated 418
AppleMenuFolder function 288
AppleScript 430
  adding to a project 413
  adding to REALbasic 413
  calling 415
  passing values to 414
  returning values from 415
AppleScripts
  calling 413–415
Application class 312, 316, 318, 337–339, 417
  event handlers 337
  methods of 339
  properties of 339
application icons
  custom 432
application object
  creating a 214
application-based subclass
  naming recommendation 339
array
  definition of 137
  passing as a parameter 138
array element
  referring to an 138
arrays 137–138
  zero-based 138

B

Backdrop property 302
  of Canvas control 249
  to display a picture 246
BASIC
  compiled 10
  disadvantages of interpreted 128
  history of 10, 128–129
  interpreted 10
  object-oriented 10
binary file
  reading a 305–307
  writing to 307–308
binary files
  benefits of 304
  definition of 304
  random read/write access to 304
BinaryStream 307
  compared to TextInputStream 305
  definition of 305
BinaryStream class 305, 306, 307
binding (see object binding)
branching
  definition of 156
breakpoint
  definition of 381
  removing a 381
  setting a 381
breakpoints
  in stand-alone applications 382
Browser 163–167, 173
  contents of 164
  contextual menus in 176
  expanding and collapsing categories 176
  Find and Replace window 177–179

hiding the 173
hiding using keyboard equivalent 175
use of bold in 167
using to access code 167
viewing items in 166
buffer
definition of 397
bugs
logical 378
syntactical 378
Build Application dialog box 426, 428, 432
button controls 197
ByRef 146
ByVal 146

C

Canvas control 67, 93–95, 332
Backdrop property 301, 302
copying a picture in a 251
creating custom controls with 340
MouseDown event handler 301
redrawing 258
saving image drawn in 301
carriage return
used in writing to text files 298
CellClicked event handler 332, 340
Checkbox 59–61
Chr method 145
CICN resource 311
class
accessing properties and methods of 336
adding a property to 328
creating based on controls 335
definition of 334
with no super class 335
classes
adding methods to 329
adding properties to 328
built-in 321
constructors 326
custom 344
deleting 353
destructors 327
exporting 352
exporting protected 352
importing 351–352
not based on controls 335–336
Clipboard
getting data from 269–270
putting data on 271–272
testing for data types 269
transferring text and graphics with 268–272
Clipboard class 268
Close event handler 338
CMY function 259
code
commenting 143
exporting 182
line by line execution 382–384
protecting 352
protecting exported 182
step into 383
step out 383
step over line of 383
Code Editor 163–179
accessing the 219
auto-completion in 170
Browser 163–167
contextual menus in 176
Editor Settings dialog box 166
entering code in 170
opening the 165
printing code in 179
resize bar 173, 174, 175
resizing panels in 173
role of 165
Code Editor Window 30
code examples 16
code execution
step into option 383
step out option 383
step over option 383
color
working with 259–264
Color class object
properties of 259
Color Picker 261
Crayon 262–263
RGB 262
System 7 262
colors
changing using Color Picker 29
Colors Window 29
Colors window 260
comments
adding to code 143
multiline 143
comparison operators 147–148
conditional compilation 430–431
constructors 324, 326
contextual menus
accessing in Code Editor 176
in Browser 176

ContextualMenu 74–75
control
  Timer 110
control array 200
  creating a 200
  for managing several controls of the same type 200
control layers 49, 115
Control order
  changing the 115
Control Order dialog box 115
ControlPanelsFolder function 288
controls 196–201
  adding to a window 44
  aligning 116
  appearance of 51–53
  BevelButton 80
  button 197
  Canvas 93–95
  changing properties of 46–48
  Checkbox 59–61
  ContextualMenu 74–75
  creating custom 339–342
  creating new instances on the fly 198
  customizing the appearance of 51–53
  definition of 196
  distributing evenly 117
  dragging from Tools window 44
  duplicating 51
  EditFields 65–67
  events for 196
  GroupBox 82–84
  line 88
  ListBox 75–78
  mouse events for 197
  MoviePlayer 97–100
  moving 45
  NotePlayer 100–101
  oval 91–93
  PopupMenu 78–80
  Position properties 45
  Progressbar 72–73
  Pushbutton 53–55
  Radiobutton 61–63
  rectangle 88–90
  removing 48
  RoundRectangle 90
  Scrollbar 68–69
  selecting 45
  selecting in reverse order 45
  selecting several 45
  Serial 108–109
  Slider 70–72
  Socket 109–110
  SpriteSurface 101
  StaticText 63–64
  TabPanels 84–87
coordinates system
  description of 245
counter
  choice of "i" as 153
  incrementing a 153
counter variable
  in loops 153
counter variables
  in nested loops 155
  typing as Integer 152
Crayon Color Picker 262–263
Create Table statement 360–361
CreateResourceFork method 310
CreateTextFile method 298
creator code
  registering 282, 438
CURS resource 311, 313, 314
CURS resources 311–314
cursor
  database 358, 362
cursors
  custom 311–314
  custom, in Windows 313–315
custom control
  drawing 341–342
custom controls
  257–259
  creating 339–342
custom document icons
  in stand-alone applications 437
custom icons
  adding to custom file types 281
  displaying at Finder level 282

D

data source
  adding and removing a 364
  selecting a 364
  specifying a 372
data sources
  two or more 356
data type
  Boolean 132
  changing 132
  Double 131
  establishing with Dim statement 136

Integer 131
Single 131
String 130
data types
definition of 130–132
data typing
in parameter line 169
DataAvailable event handler 397
database
cursor 358
Database class 372
DatabaseCursor class 373, 374
DatabaseQuery control 367–371
properties 368
databases
adding fields to 366
adding records 375
adding tables 367
back-end 356
building interface for 363
choosing a data source 372
creating a new REALdatabase 365–367
data source 356
editing records 373
field types 361
foreign key 360
modifying records 375
Not Null attribute 358, 361
OpenBase 357
overview of 356
plug-in architecture 357
primary key 360, 361
REALbasic internal 356
selecting a data source 364
SQL 357
viewing data in 366
viewing Schema 365
date format properties 241
dates
formatting 241–243
Debug menu 382
Step Into command 383
Step out command 383
Step Over command 383
Debugger window 380
debugging
definition of 378
Declare statement 411, 431
Delete method 286
DesktopFolder function 288
destructors 327
dialog icons
drawing 252
Dim statement 136
for declaring an array 137
Do loop 150–151, 306
Document window 37
documentation
conventions 12
double-clicking
to open a file 316
drag and drop 201–206
implementing 201
in Code Editor 171
in Online Reference 172
MacData 209–211
MacData property 201
multiple items 201
PrivateMacData 209–211
dragging
from a ListBox 202
dragging text
in EditFields 202
DragItem object 205
DrawCautionIcon method 252
DrawLine method 254
DrawNoteIcon method 252
DrawOval method 254
DrawPicture method 250, 251
DrawPolygon method 255, 256
DrawRect method 255
DrawRoundRect method 255
DrawStopIcon method 252
DropObject event handler 205, 206
dropping
implementing 204–206
on EditFields 206

E

EditField 403
dragging text in 202
implementing drag and drop 202
MultiLine property 202
Styled property 299
EditField control 50, 115
subclass of 322
EditField subclasses of 333
EditFields 50, 65–67
cut, copy, and paste in 269
Editor Settings dialog box 166
Else clause
in If statement 157
ElseIf statement

in If statement 157
Enabled property 333
EnableMenuItems
event handler 338
EnableMenuItems event 212, 213
EnableMenuItems event handler 333
end of file
checking for 296
endless loop
escaping from an 151
endless loops 151
entry point
defining a new 422
Entry Point Editor 421
EOF property 297, 305, 306
event 168
event handler 168, 183
Close 338
definition of 163
EnableMenuItems 338
HandleAppleEvent 338
Open 338
OpenDocument 338
passing parameters to 169
event handlers
for controls 196
for ListBox 197
event-driven programming 162–163, 183
concept of 25
definition of 161
events
adding to classes 330–332
examples of 162, 183
indirect 162
Exception block 391–394
syntax of 392
exported class
desktop icon of 351
exporting
items 182
protected module 228
source code 182
exporting classes 352
ExtensionsFolder function 288

F

file
accessing a 284
File menu
New Window command 42
file type
adding a 279
definition of 278
deleting 281
editing 280
file types
APPL 278
custom 281–282
overview of 278
PICT 278
TEXT 278
File Types dialog box 279, 290
files
creating new 317–318
opened by dropping 317
opening from desktop 316–317
FillOval method 254
FillPolygon method 255, 256
FillRect methods 255
FillRoundRect method 255
Find scope 178
Find/Replace window
in Browser 177–179
Floating windows 39–40
flow of control 156
focus 333
definition of 50
FolderItem
checking for nil 284
definition of 136
deleting a 286
getting for application’s folder 287
getting info on a 286
locked 286
OpenAsSound method 303
relative paths 287
SaveAsPicture method 301
uses of a 283
FolderItem class 283, 284, 286, 287, 288, 290, 291, 293, 295, 297, 298, 299, 300, 301, 302, 303, 306, 307, 308, 309, 310, 317
FolderItems 282–283
definition of 282
properties of 283
representing System folders 288
font
System 230
font attributes
determining the 235–237
setting 237–238
Font function 231
Font menu
adding a 232

font style
determining the 236
font style properties 236
font styles
toggling the 238
FontAvailable function 237
FontCount function 231
fonts
determining available 231–233
setting attributes of 230–238
FontsFolder function 288
For loop 152–156
ForeColor property 254, 255, 259, 260
Format function 239
characters used with 239
FormatSpec 241
Format menu
used to move controls 115
formatting
numbers, dates, and times 239–243
FORTRAN
historical note on 153
frame redrawing speed 273
FrameSpeed 273
function 169
functions
definition of 144–145

G

garbage collection
definition of 337
Get Info dialog box
in stand-alone applications 433
GetCicn method 311
GetFolderItem
relative paths 287
GetFolderItem function 283, 284, 286, 287, 303
GetIcl method 311
GetNamedPicture method 311
GetOpenFolderItem function 290, 291, 297, 299, 302, 303, 306, 309, 310
GetopenFolderItem function 290
GetPicture method 311
GetResource method 315, 316
GetSaveFolderItem function 294, 295, 300, 301, 302, 307
GetSaveFolderItem method 298
GetSound method 311
GetTrueFolderItem function 283
Global Floating Window 40
global values
monitoring 388
graphical user interface
characteristics of 24
grid
drawing a 254
Gridlock class example 340
GroupBox 82–84
for organizing RadioButtons 82
GroupBox control 49

H

HandleAppleEvent event handler 338, 417
help
context-sensitive 14
online 14
Help menu
adding a 120
HSV function 259
HTTP 404

I

If…Then structure 156–160
IllegalCastException error 390
Image property
in NextFrame event handler 272
importing
into a project 180–181
importing classes 351–352
Index parameter 233
Index property 214, 232
index property
used to differentiate multiple instances 199
InsertRow 145
installation requirements
for Mac OS System 7 11
instance
definition of 334
interface controls 339
interface inheritance 343
interfaces
character based 24
Internet 109
interrupting execution 382
IP address 401

J

Jet database engine 444
joins
specifying in SQL 360

K

Kaleidoscope 53, 231
keyboard shortcuts 121
  Windows (OS) 224
KeyDown event handler 324
KeyTest method
  SpriteSurface 274

L

language reference
  online 14
LastErrorCode property 403
lines 88
  drawing 254
ListBox 75–78, 144, 145
  event handlers for 197
  hierarchical 75
  implementing drag and drop 202
  multicolumn 155
  selection in 75
ListBox class 142
ListBox control 50, 115, 306, 307, 329
ListBox subclasses of 333
ListBoxes 50
local values
  monitoring 386
Locked property 286
loops 149–156
  endless 151, 382
  lengthy 151
  nested 155
  optimizing speed of 154

M

Mac OS System 7 11
MacData property 269
MacDataAvailable method 269
Macintosh Color Picker 29
MacType string 269
mathematical operators 139
mathematical precedence 139
Me 342
Me function 258, 342, 386
memory errors
  caused by excessive calls 385
memory management 337
  examples of 337
memory requirement
  of stand-alone apps 432
Menu Editor 30, 118
menu handler
  adding a 211
menu handlers
  definition of 211
menu item array 215
menu item separators
  adding 124
menu items
  adding 121–122
  creating on the fly 214
  enabling 212
  enabling or disabling 333
  handling from controls 213
  handling when a window is open 213
  handling when no windows are open 213
  implementing 211–215
  keyboard shortcuts for 121
  moving 123–124
  removing 124
  removing programmatically 215
menus
  adding 118–123
  managing within classes 333–334
  moving 124
method
  adding to a module 220, 222
  definition of 142
  parameter line 168
methods
  AddRow 142
  associated with objects 183
  built-in 142
  components of 168
  parameters passed to 144
  passing values to 143
  private 192
  referring to in subclasses 325
  tracking path of 384–385
  values returned from 144
Microsoft Access 444
Modal Dialog windows 38–39
modems
  communicating with 399
module
  adding a 219
  compared to class based on Application
    object 221
  importing a protected 227
modules
  importing and exporting 227–228
  role of 217
moths

role of in history of computing 378
MouseCursor property 311, 312, 314
MouseDown event handler 258, 332, 340
MouseEnter event handler 312
MouseExit event handler 312
Movable Modal window 37–38
Movie class 303
movie controller
default appearance of 97
MoviePlayer control 97–100, 303
Movie property 304
movies
importing into projects 33
MsgBox method 252, 290, 291, 293, 295, 379
Multiline property
in EditField 234
multiple connections 401

N

nested loops 155
New Binding dialog box 346
new events
reasons for adding 331
New operator 334, 335, 336, 386
to open a window 187
used to create menu item on the fly 214
NewAppleEvent function 416
NewDocument event handler 318, 338
NewPicture function 301
NextFrame event handler
sprite animation 272
NextItem function 207
NextPage method 266
Nil object 284, 286, 291, 293, 295, 297, 300, 301, 302, 303, 306, 307, 309, 310, 430
NilObjectException error 285, 291, 390
NotePlayer 100–101
numbers
formatting 239–241

O

object binding 110–115, 369–371
custom 346–351
DatabaseQuery to Listbox 369
DatabaseQuery to PopupMenu 370–371
Object Viewer 387
object-oriented programming
advantages of 11
definition of 183
on run handler 413
on run statement 414
online help 14
On-Line Reference
using code examples 16
Open Application AppleEvent 318
Open event handler 324, 338
open file dialog
limiting file types displayed in 290
Open File dialog box 289
Open Transport 400, 404
OpenAsBinaryFile method 305, 306, 307
OpenAsMovie method 303, 304
OpenAsPicture method 300, 302
OpenAsSound method 303
OpenAsTextFile method 296, 297
OpenBase 357
OpenDocument
event handler 338
OpenDocument event handler 317
OpenPrinter function 265
OpenPrinterDialog function 265, 266
passing SetupString to 266
OpenPrintertDialog function 267
OpenResourceFork method 309, 310
OpenStyledEditField method 299, 300
Or operator 148
Oracle 356
order of mathematical operations 139
OutOfBoundsException error 391, 393
oval 91–93
controlling "ovalness" of 91
ovals
drawing 254
overloaded function 328
overloading 327

P

Page Setup dialog box 264
PageSetupDialog method 264
Paint event handler 258, 259, 263, 301, 340, 341
Palindrome 100
parameter line
data typing 169
in method 168
parameter passing
in parameter line 169
parameters
definition of 144
more than one 144
password field
creating a 234

path
  absolute 284
PenHeight property 254
PenWidth property 254
PICT file
  opening a 302
  saving a 301–302
PICT files 300–302
PICT resource 311, 316
picture
  copying a portion of 251
  displaying a 250
  displaying in a portion of a window 249
  displaying in a window 246–249
  scaling a 251
Picture property 269, 271
PictureAvailable method 269
pictures
  creating 250
  importing into projects 33
Pixel property 253
  of a Graphics object 263
pixels
  drawing 253
Plain Box windows 40–41
plug-ins 418–420
  formats of 418
  including in stand-alone apps 420
  loading 419
  used to create custom controls 419
  using 419–420
  writing 420
polygons
  drawing 255–257
  filled 257
PopupMenu control 78–80
port
  definition of 400
Port property 401, 402
PowerPC code
  compressing 431
PowerPC shared libraries
  definition of 420
  using 420–423
PowerPC shared library
  accessing commands in 421
PreferencesFolder function 288
Print dialog box
  displaying the 266
PrinterSetup class 264
PrinterSetup class objects 265
PrinterSetup settings 265
printing 264–267
  in Code Editor 179
  overview of process 264
  sending a page to the printer 266
  without the Print dialog box 267
printing scope 179
private methods 218
Private option
  properties 190, 329
private properties 218
  of a module 220
Progressbar 72–73
project
  adding and removing items from a 33
  creating a new 32
  removing items from 33
  saving a 33
  starting and stopping 388
project templates
  creating 33
Project window 27
  adding data source to 364
  adding shared library to 421
  importing files into 180
  items included in 27
properties
  adding to a module 220–221
  assigning values to 133
  data type mismatches 137
  definition of 129
  getting value from 134
  private 190, 329
  referring to in subclasses 325
  visible 190, 329
Properties window 29, 46
property
  adding to a class 328
protected classes
  documenting 353
  importing 352
Protected option
  in Save As dialog box 182
protecting
  exported code 352
protocol
  defining your own 405
  definition of 404
Pushbutton 53–55
PushButton control 162

Q

QuickTime 97
QuickTime movie file
  opening a 303
QuickTime movies 303–304
QuickTime Musical Instruments 97, 101

R

Radiobutton control 61–63
RadioButtons
  compared to Checkbox 60
Read method 398
ReadAll method 296, 297, 398
ReadLine method 296
REAL Software
  contacting 20
REALbasic
  adding AppleScripts to 413
  advantages of compiled 129
  Code Editor Window 30
  Colors Window 29
  controls 43–117
  converting Visual Basic apps to 441–444
  coordinates system 245
  debugging in 26
  developer release 21
  development process in 25
  differences from BASIC 128
  electronic documentation 17
  ftp site 18
  IDE 26–31
  Interface Assistant 35, 126
  mailing lists 19
  memory management 337
  Menu Editor 30
  new commercial releases 21
  On-Line Reference 15
  overview of 10
  plug-ins 418–420
  project 27
  project templates 33
  Project window 27
  projects in 31
  Properties window 29
  reporting bugs 20
  reserved words in 141
  SDK 420
  stationery pads 33
  technical support 19
  third-party book 18
  third-party web sites 18
  Tools Window 29
  using code examples 16
  using database with 444
  web page 18
  Window Editor 28
  window types 36–43
REALbasic CD 19
REALbasic data source
  specifications 363
rectangle 88–90
rectangles
  drawing 255
recursion 391
reference
  definition of 334
RemoveResource method 316
reserved words 141
resize bar
  in Code Editor 175
resizing 173
resource fork 308
  adding a 310
  adding to a project 310
  contents of 308
  opening a 309
resource forks
  definition of 308
resource types
  supported 311
ResourceFork class 309, 310, 311, 315, 316
resources
  reading 311–315
  writing to 316
reusable code 320
RGB Color Picker 262
RGB function 259, 263
Rgb function 259
RGB values
  getting the 261
Rounded windows 42
RoundRectangle 90
Run method
  SpriteSurface 273
runtime exceptions 390–394

S

Save As dialog box
  managing the 293–296
SaveAsJPEG method 300
SaveAsPicture method 300, 301, 302
SaveStyledEditField method 300

schemes
  in Kaleidoscope 53
Scrollbar 68–69
ScrollBar control 67
Scrollbars
  event handlers for 198
SelChange event handler 234
Select statement
  primary key 358
Select...Case statement 158–160
  restrictions on 160
selected file
  getting folderitem for 290
selected folder
  getting folderitem for 291
selected text
  determining attributes of 235
  working with 233–234
SelectFolder function 291, 293
Self function 194, 314, 386
SelLength property 233
SelStart property 233
SelText property 233
SelTextFont property 235
SelTextSize property 236, 237
Serial control 108–109, 396, 397
  changing configuration of 399
  Close method 399
  configuring 396
  DataAvailable event handler 397, 398
  Flush method 398
  LookAhead property 398
  Open method 398
  overview of use 396
  placing in a window 396
  Poll method 399
  reading data with 397
  Write method 398
  writing data 398
  XmitWait method 398
Serial controls
  event handlers for 198
serial device
  definition of 396
serial devices
  communicating with 396–399
serial port
  closing the 399
  opening the 397
SetText method 271
SetupString property 265
  storing the 265
shared libraries
  accessing commands in 421
  calling commands in 423
  definition of 420
  using 420–423
ShutDownItemsFolder function 288
Slider 70–72
Sliders
  event handlers for 198
SMTP protocol 404
SMTP server 401
snd resource 311
  getting sounds from 303
  reading 311
Socket control 109–110, 400, 402
  Close method 404
  Connect method 401
  Connected event handler 401, 402
  DataAvailable event handler 402
  Error event handler 403
  for communicating via the Internet 109
  Listen method 402
  Port property 401
  Write method 403
Socket controls
  event handlers for 198
Sound class 303
sound file
  opening a 302
sound files 302–303
sounds
  importing into projects 33
sprite animation
  detecting keystrokes during 274
sprite area 273
sprites 272–274
  animating 272–273
  definition of 101
SpriteSurface 101
SpriteSurface Backdrop property 273
SpriteSurface control 272
SQL 357
  Create Table statement 358, 360–361
  cursor 362
  Group By clause 358, 359, 360
  Like clause 359
  Order By clause 358, 359, 360
  Select statement 358, 359, 360, 362, 374
  Set functions 358, 362–363
  specifying joins 360
  Update statement 358, 361–362
  Where clause 358, 359, 360, 362, 374

wildcard 359
SQL in REALbasic 358–363
SQL Set functions
Avg 362
Count 362
Max 362
Min 362
Sum 362
Stack window 384
hiding 385
viewing code from 385
StackOverflowException error 391
stand-alone applications
building 426–433
compressing PPC code 431
custom document icons 437
custom icons for 432
ease of building 426
including 68K and/or PPC code 431
memory requirements for 432
StartupItemsFolder function 288
StaticText control 63–64
stationery pad
creating a 33
step into option 383
step out option 383
step over option 383
Step statement
for incrementing a counter 153
Str function 145, 379
Structured Query Language 357
Styled property
in EditField 299
styled text
definition of 234
handling 234–238
reading into an EditField 299
writing to a file 300
styled text files 299–300
subclass
customizing a 322
definition of 322
subclasses
examples of 321, 322–325
submenu
adding a 122–123
super class 324
definition of 322
syntax errors
checking for 379, 380
System font 230
SystemFolder function 288

T

Tab order 45
changing the 115
TabPanel control 49
TabPanels 84–87
advantages of 84
Target property 194, 195, 196
used to change properties of controls 195
Target68K constant 430
TargetMacOS constant 430
TargetPPC constant 430
TargetWin32 constant 430
TCP/IP 109, 400
supporting multiple connections 401
TCP/IP communications 400–405
TCP/IP connection
closing 404
error handling 403
listening for a 402
reading data 402–403
to another computer 401–404
writing data 403
TCP/IP protocols 404
TemporaryFolder function 288
text
getting and selecting 233
text file
creating a 295
reading from 296
writing to 298
text files
compared to binary files 299
limitations of 299
working with 296–300
Text property 269
TextAvailable method 269
TextInputStream
definition of 296
TextInputStream class 297
TextOutputStream
definition of 298
TextOutputStream class 298
themes
in Mac OS 51
Thread class 438
Thread Manager 11
requirement of 438
time format properties 243
Timer control 110
times
formatting 243

toolbox calls 411
Tools Window 29
TrashFolder function 288
type selection 50
TypeMismatchException error 391

U

user interface
  importance of 35

V

vacuum tubes
  use of in computers 378
values
  changing the data type of 132
  debugging 386–388
  getting and setting in variables 135
  getting from properties 134
variables
  definition of 130
  getting and setting values 135
Variables window 386
VBX controls 443
virtual methods 342
Visible option
  properties 190, 329
Visual Basic
  converting to REALbasic 441–444
  importing forms and code 442
  tips in converting to REALbasic 442–443

W

While loop 149–150
While…Wend loop 297
Win32 accelerators 224
window
  accessing properties of 191
  adding a method to 192
  adding a property to 190
  deleting a method 193
  deleting a property 190
  editing a method 192
  editing a property 190
Window Editor 28
windows
  accessing controls, methods, and properties of other 193
  adding methods to 191–193
  adding properties to 189–191
  creating 42
  deleting 43
  events 184–186
  multiple instances of 194
  opening 186–188
  opening with New operator 187
Windows (OS)
  accelerators 31, 224
  keyboard shortcuts for 224
Windows API calls 443
WriteLine method 298

X

XCMD 408
  extracting from HyperCard 408
XCMD/XCFN
  calling in a method 410
XCMD/XFCN
  installing in a project 409
  locating on the Internet 411
  removing 411
XFCN 408

Z

zero-based array
  definition of 138