# Beginning Mac®


## For Classic Mac® OS and Mac® OS X

Premier Press

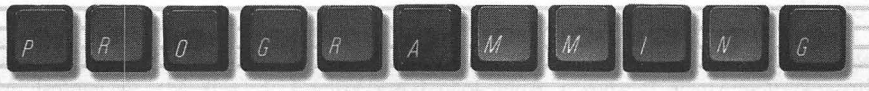
KEVIN SPENCER AND JEFF THOMPSON


CD Included

# Beginning Mac PROGRAMMING

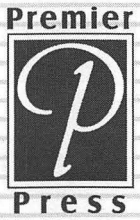
## For Classic Mac OS and Mac OS X

# Beginning Mac




## For Classic Mac OS and Mac OS X

KEVIN SPENCER AND JEFF THOMPSON






**Publisher:** Stacy L. Hiquet
**Associate Marketing Manager:** Heather Buzzingham
**Managing Editor:** Sandy Doell
**Acquisitions Editors:** Jawahara K. Saidullah, Kevin Harreld
**Project Editor:** Brian Thomasson
**Technical Editors:** Geoff Perlman, Dan P. Sydow
**Copy Editor:** Kate Welsh
**Interior Layout:** Marian Hartsough
**Cover Design:** Mike Tanamachi
**Indexer:** Katherine Stimson



**Important:** If you have problems running REALbasic, go to REAL Software's Web site at http://www.realsoftware.com. Premier Press, Inc. cannot provide software support.



ISBN: 1-931841-00-4
Library of Congress Catalog Card Number: 00-110736
Printed in the United States of America

00 01 02 03 04 II 10 9 8 7 6 5 4 3 2 1

# Acknowledgments

All my thanks go to my co-author, Jeff, who spearheaded the code writing for this book. Your dedication is a rare find. Many thanks also to Brian, Kate, Kevin H, and the crew at Premier Press, Inc., who provided a solid base in times of trouble. Many thanks to Geoff Perlman and the folks at REAL Software for their excellent help with technical corrections and suggestions. A special thanks to Marta Justak, for her guidance. And a very special thank you to my wife, Rebecca, who suffered for agonizing hours as she watched over 150 satellite television channels while I worked.

—KS

My thanks go out to those that Kevin has already thanked, along with thanks to Kevin as well. Your experiences and talents as a writer, not to mention your assistance along the way, are much appreciated. Special thanks go to my High School computer-programming instructor, G. Ross Buckman, for convincing me that a career in computer programming is right where I belong. Very special thanks go to my wife, Kathie, and my children, Neil and Samantha, for their encouragement and patience. Lastly I'm sure Kevin would like to join me in thanking all of our friends, whose support was very important to us.

—JT

# About the Authors

KEVIN SPENCER is a certified Apple Service Technician, computer hobbyist, and writer. Kevin has worked with some of the earliest personal computers of the late 1970's and still thinks the BASIC computer language was a pretty nifty idea. When not bicycling or wading in weeds while working in the yard, he spends his time with his wife, two sons, and two computer-illiterate cats in Indianapolis.

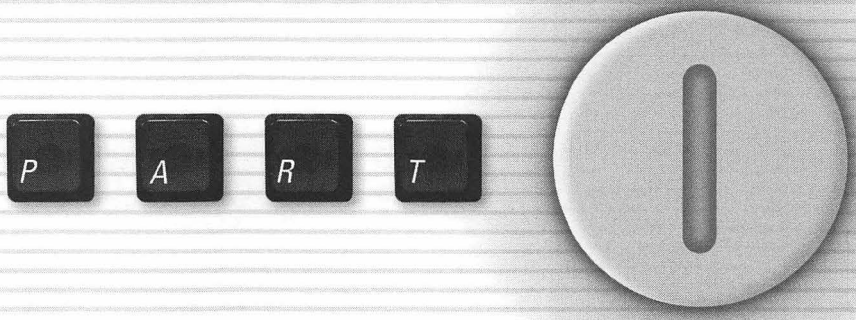
Working as a software developer since 1985, JEFF THOMPSON has written applications in various languages such as BASIC, Z-80 Assembler, 6502 Assembler, DBL, 8088 Assembler, C and C++. Jeff worked with and developed applications on various platforms from the good old days of the TRS-80, Apple II, DEC minicomputers, IBM PC's, and the Macintosh Plus all the way up to today's latest Pentium PC's and Macintosh PowerPC systems. Jeff's computer career also includes a couple of years work as a computer technician, salesman and a short stint where he taught classes on the BASIC programming language. Jeff also had a passing acquaintance with HTML, Java, and Perl. Jeff has developed a wide variety of applications including, payroll, accounting, shareware entertainment, billing systems, and billing analysis tools. He's currently employed by CTI Billing Solutions as Senior Software Engineer and Technical Lead on one of the highest rated billing analysis software applications in the country. Jeff is currently listed as an inventor on a software patent used in CTI Billing Solutions' billing analysis software, Smart Bill™. The patent is also licensed for use by more than half of the long distance and local telephone carriers in the United States. Jeff currently resides in Indianapolis, Indiana with his wife, two children, two dogs, four cats and an indeterminate number of ducks which have taken up permanent residence in his pond. He also has an ever-increasing collection of Macintosh computers, which require a lot less care and feeding than the other members of his family.

# Contents at a Glance

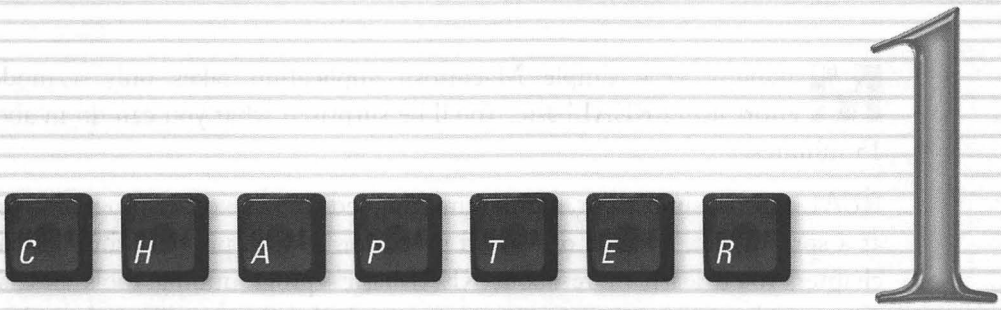
# Contents

# Part IV
# Advanced Things to Do 315

# Beginning Mac® PROGRAMMING




# Getting Your Feet Wet




# Getting Acquainted with REALbasic

## In This Chapter

- What's on the CD-ROM
- Installing REALbasic
- The REALbasic design interface
- Making your first application

Making a very simple Macintosh application takes only a modest effort using REALbasic. You'll be surprised what you can do in about 15 minutes.

REALbasic is, simply put, a program designed to make other programs, such as a spreadsheet, a game, or a word processor. As you progress through each chapter, you'll learn more about REALbasic's programming tools and language, and programming concepts in general. But first, you'll need to obtain and install a copy of REALbasic.

# What's on the CD-ROM

To teach you the ins and outs of Macintosh programming, we decided to go against the grain in the programming world. Many programmers prefer to use a programming language known as C++ (that's pronounced *cee-plus-plus*); many programming tools use C++ for development as well. Although C++ is very powerful and versatile, it's difficult to learn for a novice programmer, and a bit arduous for even seasoned programmers.

A few of you may have taken a computer-programming course before, or remember way back in the late 1970s when the first microcomputers for home and business use arrived. Back then, you couldn't buy many computer programs—you were stuck having to type computer programming instructions into your computer. Later, you could buy some canned programs from the local store, instead of typing in instructions like these:

```
10 LET A=5+10
20 LET B=5*10
30 LET C=A+B
40 PRINT C
50 END
```

The computer language used on many of the first personal computers of the 1970s was BASIC. No, not *basic* as in *simple*, but BASIC (*Beginner's All-Purpose Symbolic Instruction Code*). Although BASIC wasn't a very strong programming language, it wasn't very hard to understand. Like many programming languages in history, BASIC is based on FORTRAN, another programming language used on early large computers.

### In the Beginning

The buyers of the earliest personal computers had to type in their programs from books or other printed sources. The TRS-80 from Radio Shack and other early computers added an audio cassette deck for loading in software created by others, and later added floppy disk drives. So, basically, the idea of manually writing applications for a personal computer isn't a new one—it just became less efficient to do it.

In the following years, some companies improved and strengthened their own versions of BASIC to make it more like C++ and other high-level programming languages. One of these evolved versions is REALbasic, a programming tool with many easy-to-use features.

REALbasic is available in a Standard and Professional version. REAL Software has provided a trial Standard version on the CD included with this book. The makers of REALbasic, REAL Software, continue to improve and update this program, so your copy on the CD might be off a few version points by the time you get our book. The latest version of REALbasic is available online at REAL Software's Web site, http://www.realbasic.com/realbasic.

The Professional version of REALbasic allows you to create Windows 95/98/NT4/ 2000/ ME versions of your applications at the same time as your Macintosh applications, and has extensive database support. The Standard version has demo support of the Professional version's features so you can get an idea how cool it is to make a Windows and Macintosh app all at once.

If you find REALbasic to your liking, you can purchase a serial number from REAL Software that will turn the trial Standard version to a live Standard version with full functionality. The trial Standard version of REALbasic has one major limitation you should remember. After 30 days, the program ceases to operate until you purchase a serial number from REAL Software to activate the program. Any applications you build with a trial version of REAL-

basic will also cease operating within 30 days. Any Windows applications you create with the trial version will only work for 5 minutes. You'll also be greeted by a little dialog box that, when you start up REALbasic (or any applications you create using REALbasic), reminds you that REALbasic and anything you create using it will stop working soon. Needless to say, to get the most out of REALbasic and to get rid of the reminders, you should purchase a copy.

Depending on the version you purchase, once you register REALbasic, it transforms into a fully functional Standard or Professional version right before your eyes!

The great thing about REALbasic is that it runs on older Macs as well as the new PowerPC G4 systems. Before you rip the CD out of the book, make sure your Macintosh computer meets these minimum requirements for our projects:

- Mac OS system software 8.1 or later
- A PowerPC processor
- A CD-ROM or DVD-ROM drive (for using the CD-ROM)
- 32MB of total computer memory (with virtual memory on)
- 6.5MB of free hard disk space.

As you go through REALbasic's system requirements (as found in its Read Me and documentation), you might notice that our system requirements are steeper, and for good reason. Yes, I, too, love the fact that a Macintosh is the Maytag of computers—they keep running forever. But so do certain potentially explosive cars and politicians. After a point, it's not cost effective, practical, or (if you so happen to start a career from this book) profitable to support older Macintosh hardware.

Don't get us wrong—REALbasic is quite able to create applications for older Macintosh systems as well as Power Macintosh systems. Nonetheless, although the programs you create in REALbasic could likely run on older Mac hardware (that is, computers that aren't Power Macintosh class) or on Mac OS 7.6.1 or earlier, we're not able to show you how to support applications created for these systems.

There's a line that must be drawn because, basically, Apple drew it first, and we're toeing the line so that we keep this book lean with information you

require for modern application development. Eventually (read: *soon*), Apple will likely offer hardware and software support for only Power Macintosh G3 systems and newer, as these are the only systems that can officially run Mac OS X, the successor to Mac OS 9. Many companies that create Mac programming software (including REAL Software) are designing their new versions of applications to work only on PowerPC systems. If you're considering dusting off that old Quadra so you can make a Mac OS 9 or Mac OS X application with REALbasic or another programming tool, it's likely you may be out of luck. While you can use REALbasic on your Quadra to create new applications meant for other Quadras and older Macintosh systems, doing so is much like designing a new, high-efficiency engine manifold for a Ford Pinto. There's just not a lot of point to it unless you have a real need for an application you can't find anywhere else.

Needless to say, the more modern your Mac, the easier it will be to create new programs.

# Installing REALbasic

Installation's a breeze for REALbasic if you use a Power Mac running Mac OS 9 or Mac OS X. Follow these steps:

1. Insert the book's CD-ROM in your CD-ROM or DVD-ROM drive. The Beginning Mac Programming CD-ROM icon appears as an icon on the Mac OS Desktop.
2. Double-click the CD-ROM icon on the Desktop and locate the "Open Me for REALbasic 3" folder.
3. Drag the REALbasic folder from the CD-ROM to your applications folder on your hard disk.

If you use an older Macintosh (that is, a non-Power Mac, such as a Quadra or Centris), see Appendix B for more detailed instructions

That's all there is to it! Inside the REALbasic folder is the REALbasic application. Additionally, the CD includes documentation, sample code, and other items that you'll find useful as you gain experience with REALbasic programming.

# What REALbasic Looks Like

The REALbasic application has five windows that serve as your workplace for creating (or developing) the parts of your program. The folks at REALbasic officially call it the *design interface*; it's shown in Figure 1.1.

The design interface consists of the Project window, which is, in a sense, a container. It holds all the pieces that form a REALbasic application, including the following:

- **Tools window.** The Tools window is a toolbar of a sort that contains Mac OS interface elements, called *controls*, which you can drag into the Project window.
- **Colors window.** The Colors window is a palette where you can place colors that you use often in your projects. You can use the Colors window to give colors to properties that accept color assignments.

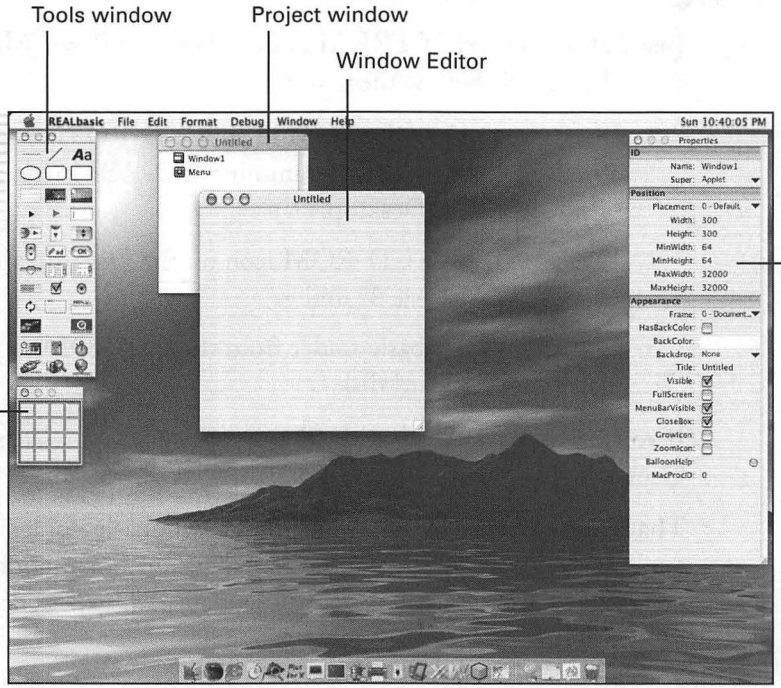



**Figure 1.1**
Welcome to the REALbasic design interface.

- **Properties window.** The Properties window shows information about an item selected in the Project window or Window Editor.
- **Project window.** The Project window shows all the components that make up your project as you build it.
- **Window Editor.** The Window Editor represents a window in the program you are building. You add interface controls to it by dragging them from the Tools window into the Window Editor. When you first launch REALbasic, a new project is created with a single, blank window to get you started. You can add more windows to your project later as you need them.

# Making Your First Application

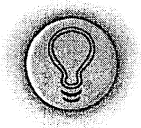
REALbasic provides you access to the key graphical elements available within most Macintosh applications. These graphical elements can form a program quickly. The program I'll have you write here won't do a great deal and hasn't a lot of substance. You might even call it a "simple" application. But, as my friends at REAL Software point out, a "simple" application can be quite useful and packed with a feature or two that's invaluable. For instance, *docklings* are applications that are placed in the Mac OS X dock. A dockling is a simple application, yet it can provide users a quick way to change a system preference without having to open the System Preferences application from the Apple menu. In short, programming is literally what you make of it.

Essentially, you'll take advantage of built-in programming within the Mac OS that forms windows, scroll bars, and other pictures on the screen. Apple refers to these programming parts collectively as the *Toolbox*. The Toolbox makes your programming life much easier by simplifying and minimizing the programming skills you need to know to create a program. You'll learn more about the Toolbox in Chapter 19, "The Carbon Environment."

You'll be using all of REALbasic's interface items except the Colors window for this exercise. Don't worry if you don't know anything about programming at this point. The object of this exercise is to show you how relatively easy it can be to write a Macintosh program.

The application we're going to write simply displays a message window with a button that quits the application. Most experienced programmers would

recognize this little gem as a version of the revered "Hello World" application that demonstrates the most basic of programming instructions—a message on the screen. A Hello World application like this one is always a good way to get your feet wet, so let's get started.



**TIP**

For my examples, you'll be seeing REALbasic 3.2 running in Mac OS X. Don't worry about the subtle appearance differences between REALbasic running in Mac OS 9 and X. You will follow the same steps no matter which operating system is running your copy of REALbasic.

1. Start by launching REALbasic on your computer. Locate the REALbasic program and double-click its icon to start the program. The REALbasic application will launch and a new Window Editor appears as shown in Figure 1.2. You just started a brand-new REALbasic programming project and, even as an untitled project with nothing done yet, REALbasic still views it as an application that can run!
2. Right now, you should be looking at a blank project window, simply named "Untitled." The Properties window currently displays the properties, or programming characteristics, of the Project window as shown in Figure 1.3. The properties of the Project window start with the Name field at the top of the window, under the ID header. The default (or preset) name of the Name property is *Window1*.

**NOTE**

If you don't see *Window1* under the Name property, you may have clicked on another item in REALbasic and the Properties window is dutifully showing you the properties of whatever you accidentally clicked. Just click on the Project window near the center of the screen to view the Project window properties. The Properties window changes its contents to display the properties of the item you just selected. When you click on controls or other elements later, the contents of the Properties window will change again to reflect the properties of that item.

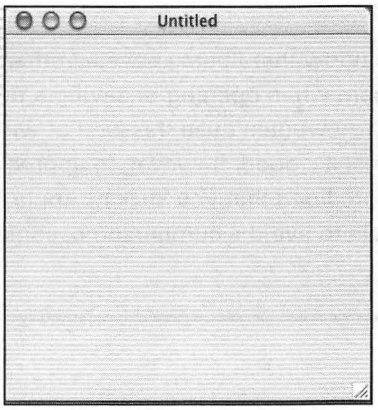



**Figure 1.2**
The Window Editor looks like this when you create a new project: a totally empty and featureless window.

3. Next, you'll change the window's appearance by adjusting the Project window's properties in the Properties window. To begin, look for the properties listed under the Appearance header at the bottom third of the Properties window.
4. Click to the right of the first property field in this area, Frame, and select Movable Modal.

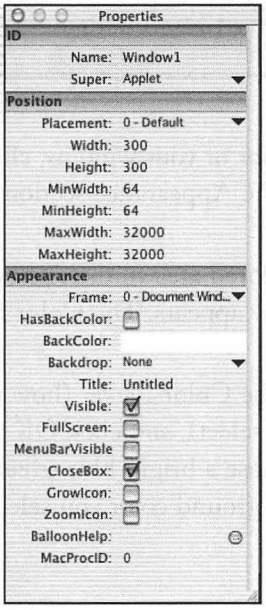


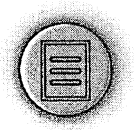

**Figure 1.3**
The Properties window here shows what you should see once you select the Window Editor to work on.

**NOTE**

A *modal window* is a window that floats above all other windows. You've typically used them when opening or saving a document in an Open/Save dialog box. You wouldn't use this kind of window in most conventional applications. It's good, however, to appreciate a different way of viewing things, provided that you stick to Apple interface conventions as much as possible to avoid a cumbersome, confusing application interface.

5. Change the numbers that appear under the Width and Height, properties as follows:
   - Width: 318
   - Height: 166

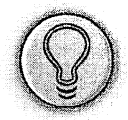
   The minimum width and height properties should remain at their preset, or default values, so you won't need to adjust them.



**NOTE**

The Window Editor doesn't always show the changes you make. You'll see the results when you actually run your program for the first time.

6. To change the background color of your window, click on the HasBackColor check box in the Appearance section of the Properties window to activate that option.
7. Double-click on the white-colored area to the right of the BackColor label. A Mac OS color window appears. The Color Picker options should be selected.
8. Pick any color you like from the Color Picker (how it appears to you depends on which option you select), and click OK. In my case, I used the crayon color picker and chose a bright blue-green. The Window Editor will change to the background color you selected.

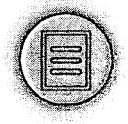


**TIP**

We suggest that you choose a light color. Selecting a dark color will make the text you later add in the window a bit hard to read.

9. See the large A item in the Tools window? That's the Static Text control. It enables you to create text or labels within your program to show information. Drag and drop that control anywhere inside the window editor.
10. Change the properties that appear under the Position header to the following:
    - Left: 35
    - Top: 30
    - Width: 250
    - Height: 50
11. You just specified that the label's relative size fit some text you'll add in a moment. Now you need to adjust the alignment of the text within the field. With the label still selected, click on the TextAlign property and select 1-Middle from the pop-up menu. Now, any text will be centered in the label.
12. Next, you need to change the text to something meaningful. Find the Text field just below the Appearance property header. Right now, it should simply say *Label:*. Click on the button to the right of this to open an Edit Value window.
13. The Edit Value window is just a place to change the information in specific controls. In your case, you're just typing in some new text. To do so, delete the *Label:* text and type the following:

```
This is MyLittleWindow.
It's not much of a program, but it's mine!
```



**NOTE**

Be sure to press Return to break the text into two lines, as shown above.

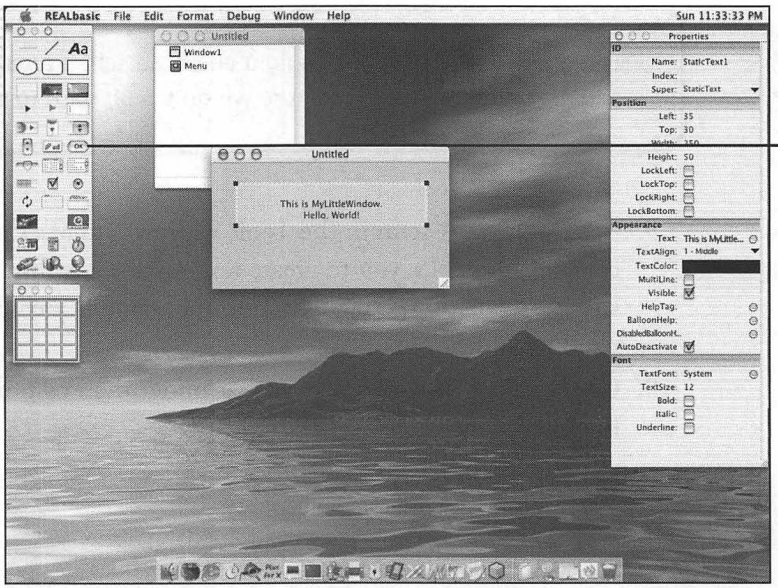



**Figure 1.4**
Your new window editor shows the label, but it looks a little bare. Time to add a button using the PushButton control

14. Click on OK to close the Edit Value window when you're done. Figure 1.4 shows the completed window thus far.
15. Because your program will be a floating modal window with no menus, you need to be able to turn the silly thing off once you've marveled at it. You can do that easily enough by adding a button to the window. To begin, find the PushButton control on the Tools window (it looks like an OK button) and drag it just below the text inside the Window Editor.
16. Change the Position values in the Properties window for your button to the following:
    - Left: 130
    - Top: 100
    - Width: 60
    - Height: 20
17. The button should now be centered under the text. Click on the button to the right of the Caption field under Appearance in the Properties window. Another Edit Value window appears for you to change the text of the button.

18. Type in this really complex bit of text and click OK to close the Edit Value window:

    ```
    Great!
    ```

19. Now it's time for you to write your first REALbasic code to activate the button and make it operate. Get your thinking caps on. Begin by double-clicking the PushButton control in the Window Editor to open the Code Editor window.

20. The Code Editor window assumes that you want to program the Button control to perform a specific set of commands; as a result, REALbasic has placed the text cursor exactly where you need to type, between the `Sub Action()` and `Sub` lines. Type in the following text (be sure not to enter any other characters, and don't press the Return key).

    ```
    quit
    ```

    The results are shown in Figure 1.5.

21. Click on the Close box at the top-left of the Code Editor window to close the window.

22. To follow Apple graphical interface conventions, you should make your little button the default button so it has that familiar doubled outline around it, or in the case of Mac OS X, the button pulsates. To begin, select the Great! Button control, then click the Default check box under the Appearance heading in the Properties window.

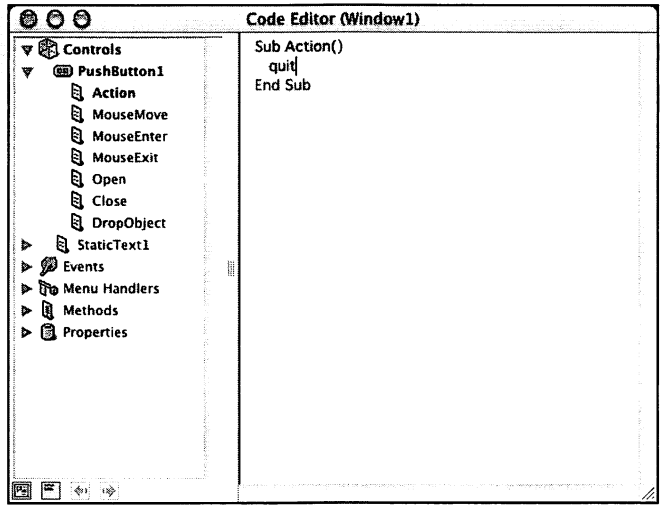



**Figure 1.5**
Between the two pre-entered commands, you'll enter one magic command.

### It Knows What You're Thinking

You might have noticed that your `quit` command appeared automatically in the Code Editor after you typed a couple of letters. That feature is what REAL Software appropriately calls *auto-complete*. The Code Editor window knows all legal commands and will try to complete any command as you type then. If the command that the Code Editor shows is the right one, just press the Tab key to let REALbasic complete the suggestion. If the command REALbasic suggests isn't the right one, just keep typing out the command you intended.

23. Let's complete matters by saving your work. Open the File menu, and choose the Save command. Save your work under the name *MyLittleWindow* in your REALbasic folder. Wonderful! Your program is ready to be tested.

REALbasic allows you to test programs you've made in a *debugging environment*. This debug mode lets your application operate as if you were running it as a standalone program in the Mac OS. The main advantage of the debug environment is that you won't lose control of the application there. Should a program you test in debug mode ever get out of hand, press Command+Shift+Period, or click on any design environment window to return to REALbasic.

24. To run your new program, click on the Debug menu and choose Run.
25. You should see your new program as it appears on the screen in a floating window. When you click the Great! button, the program quits and returns you to the REALbasic environment.
26. Finally, let's make this program a true standalone Macintosh application that you can run from the desktop. To begin, open the File menu and choose Build Application.
27. The Build Application window appears, as shown in Figure 1.6. Leave the Macintosh option checked. (In case the Windows option catches your eye, don't worry—we'll touch on making Windows versions of your programs in Chapter 22, "Porting Your Applications to Microsoft Windows.")

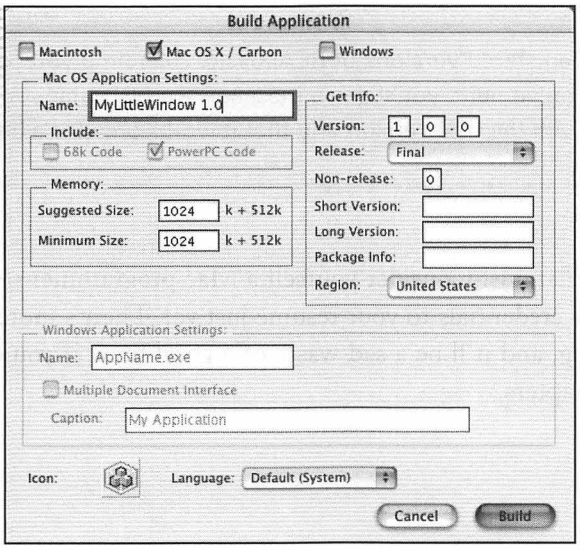



**Figure 1.6**
The Build Application window is your final step to turning your REALbasic project into a working standalone program.

28. In the Name field in the Mac OS Application Settings area, type `MyLittleWindow 1.0`.
29. In the Get info area, select Final from the Release pop-up menu.
30. Click the Build button to build your new application.

The new application will have a generic icon appropriate to your version of the Mac OS. Your new application is stored where you saved the project file.

Try using your program. When you do, you'll see exactly what you created—a modal window, as shown in Figure 1.7. When you click on the Great! button, the program will quit.

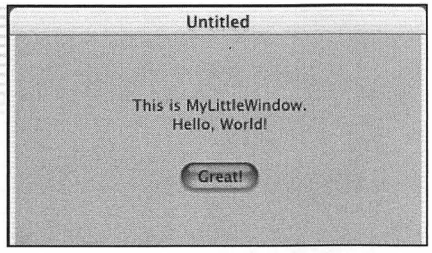



**Figure 1.7**
Ta-da! Your finished program

**NOTE**

In unregistered versions of REALbasic, each time you open an application you've built, you'll see a message warning that your application was built with the trial version and that it works only for a limited time.

Congratulations! Consider yourself a Mac programmer—but don't go adding your new credentials to your resume just yet. There's more to learn about programming, and it'll be a sad waste of a good book if you don't peruse the rest of the chapters.


# Programming's Big Picture

## In This Chapter

- The phases of programming
- *Sprecken sie* REALbasic?

After installing REALbasic and having a chance to play around with it, you're probably itching to get started writing your own programs. Well, hold on a minute there, partner! You can't run before you learn to walk, and you can't code until you learn a little more about programming.

The next few chapters are for the benefit of those readers who are unfamiliar with the concept of programming. If you already have a good grasp of the simple concepts associated with programming, you may just want to skim these chapters. On the other hand, if you have no idea what programming is all about, then this is the place to start.

# The Phases of Programming

Depending on whom you ask, computer programming is either a science or an art. I like to think of programming as a mixture of the two. Even though the process of writing a computer program is methodological, the design, look and feel, and even the programming source code can express the programmer's artistic talents. For instance, a sculptor can make a beautiful fountain. It's art, naturally, for its aesthetics. The fountain also holds practicality on a hot day as its cool spray drifts on nearby people. The function of the fountain goes beyond its original purpose by serving as a wishing well as it gathers coins dropped inside it.

The objects in Figure 2.1 have both an aesthetic quality and at least one function. As a developer, remember that both concepts are needed to form a good application. Of course, a comb with no teeth is not a comb. You'll still need to provide a clear primary function for your application, although it can have more than a single function.

Be it science or art, the goal of computer programming is to allow a computer to perform a predefined set of tasks accurately, predictably, reliably, and repeatedly. The tasks being performed vary from program to program. In one program you might keep track of your checkbook register, and in another you might simulate the experience of protecting the world from an alien invasion.

Even a sculptor must follow a plan or a set of rules to create art. Programming is no different. Many developers find it useful to use drawings to illuminate. With that in mind, Figure 2.2 shows a path to programming enlightenment.

**Figure 2.1**
Programming is much like a collection of everyday objects—having both aesthetic and functional qualities.

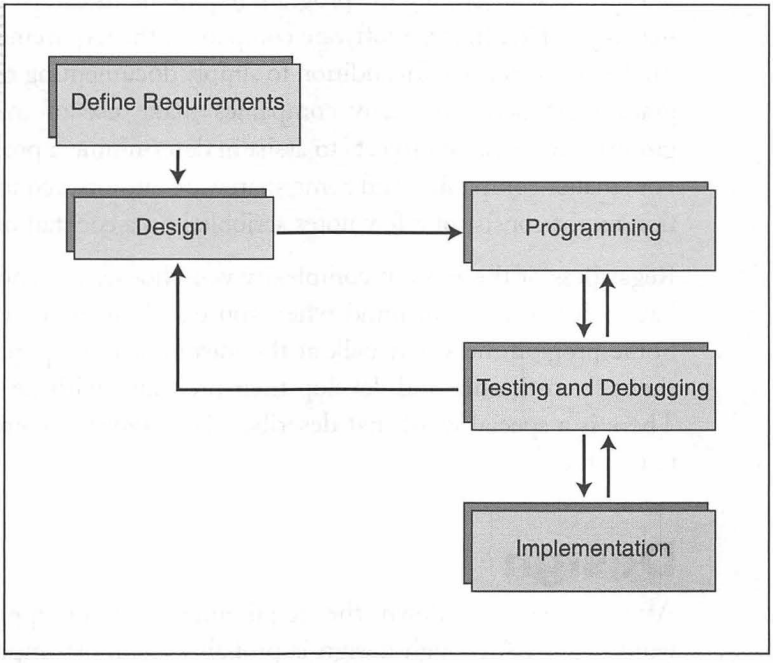



**Figure 2.2**
The better your application's design, the less likely you are to have to backtrack.

Even though this book deals primarily with the process of writing computer programs, there is more to programming than that. The complete process of computer programming includes many phases:

- Defining requirements
- Design
- Programming
- Testing and Debugging
- Implementation
- Support

# Defining Requirements

All computer programs exist to fulfill one or more needs. Defining a program's requirements is the process of documenting everything the computer program should be capable of doing. When you decide to write a program, you should always begin by defining the requirements of the program.

The process of defining the program requirements can be as formal or as informal as you like. In large software companies, the requirements-definition phase can be very extensive. In addition to simply documenting the tasks that the program must perform, many companies make use of market analysis, focus groups, and research projects to assist in determining a program's requirements. For smaller companies, and some shareware authors, requirements documentation might consist of a few notes scribbled on a cocktail napkin.

Regardless of the level of complexity you choose, you should at the very least have a list of goals in mind when you decide to write a computer program. Some programmers may balk at the idea of strict requirements, preferring to shoot from the hip and develop their programs with no clear goals in mind. There is a special word that describes those types of computer programmers: unemployed.

# Design

After you've laid down the requirements of an application, the design work begins. Although design is probably the most important part of development, we're not covering it in this book. Plenty of books on the market deal

with design methodologies; we don't need to reinvent that wheel. For the purpose of Macintosh programming, I recommend Apple's Human Interface Guidelines, available through the Apple Developer Connection Web site at http://www.apple.com/developer.

An application design includes many aspects, not just the visual ones. Although the user interface—that is, your program's windows, dialog boxes, and menu items—is quite often the thing that beginning programmers think of when they think of design, it is far from the only thing. Other aspects of an application's design include

- **Source code design.** Your source code should be thought through—that is, designed—before you start writing your program. You should document the ways your source code addresses the various requirements. Additionally, you should consider how the various parts of the application's source code will interact.
- **Data design.** If your program reads and writes any data, you should design the layout of these data files before writing any code. Heavily data dependant applications require very extensive data design.
- **Testing-process design.** Think about how to best test your program. How your user interacts with your application via the user interface will help you determine how to test your application.

These aren't the only aspects of application design, just a few to give you an idea of the things that you should count on doing.

As with the development of the application requirements, the application design can be as formal or informal as is needed. Just make sure that you spend some time thinking about your design before writing the program source code.

# Programming

Finally, you get to the fun stuff. The programming phase is where you actually write the source code that controls what the program does and how it does it.

The process of programming, however, often called *coding*, involves more than simply typing source code. Programmers, or *developers*, as they are often referred to, are responsible for testing their work as they go along. This testing is not meant to provide a complete overview of the system and all its functionality—just the parts of the program on which the developer is working at

any given time. As the developer finishes any given unit of code, he or she needs to test it to ensure that it works properly. This type of testing is referred to as *unit testing*.

There is no hard and fast definition of a unit. It can be as small as one line of source code, or it can be made of multiple sections of source code, all of which are responsible for a single application feature. That said, unit testing is something that should be done often. The more often you unit test, the fewer changes you are testing. This means that your individual unit tests will go faster and you'll be less likely to miss something.

## Testing and Debugging

After programming comes testing and debugging; this is when you get a chance to be really hard on your application. You need to test every aspect of your program. Make sure it does everything it's supposed to do, and does it correctly. Try to crash your application before you set it loose in the world and someone else figures out how to crash it for you.

In relatively small projects, you may decide to complete all your programming before testing and debugging. In larger projects, however, you should break your work down into multiple milestones and perform complete testing and debugging of all features included in each milestone. Just as with unit testing, the more often you test and debug, the less likely you are to miss something.

*Debugging*, in case you've never heard of it before, is where you correct the mistakes you've found. If you didn't find any mistakes, then you didn't look hard enough. If there's one rule about programming, it's that there's always one more bug. In large applications, such as operating systems and office-productivity tools, the number of bugs is often in the thousands—and these applications are released to the public with these bugs still in them! Fortunately, most have been identified and can be easily fixed.

## Implementation

After you've identified all the bugs, determined those that can and will be repaired, repaired the bugs, and re-tested the application, you're ready to go on to the next stage of the development process: implementation.

You might be writing an application for friends or co-workers. Alternatively, you might plan to sell your application as a retail product, or to distribute it directly to the public on the Internet. Regardless of your intentions, the process of making your application available for use is referred to as a *release.*

Various methods of release include

- Retail
- Shareware
- Freeware
- Open source

## Releasing a Retail Product

A retail product is the standard store-shelf, packaged-delivery option that is usually associated with larger commercial software packages. A lot goes into the commercial release of a retail product—more than we can go into here. Suffice it to say that the bygone days of a single developer writing and releasing a retail software package are long gone.

## Releasing a Shareware Product

Sometime back in the 1980s, the majority of computer development was moved to large software manufacturers. Because this made it harder for smaller software companies to get noticed, they did something radical: They gave their software away—well, almost. The idea was that you could get the software free of charge, and pay for it after you'd had a chance to try it out and decide whether you liked it. You could also share the software with your friends and co-workers, hence the name *shareware.*

The concept is still around, and is stronger than ever. Most new software companies offer shareware versions of their applications. Some shareware either limits the features, or the amount of times it can be used, until the user decides to pay for a full-feature version. This type of product is sometimes jokingly referred to as *Heroinware*, because, like a drug dealer, the software company gives you a little free of charge just to get you hooked, and charges you from then on.

## Releasing Freeware

You also have the option of not charging for the use of your software at all. Simply give it away. Sounds like a great idea if you're a computer user, but a stupid idea if you're a software developer, right? Maybe not. A lot of software, such as TechTool (shown in Figure 2.3), is released as freeware. Some is released simply to give the product some exposure, or to gain fame for the developer. Even commercial software manufacturers have begun releasing their software as freeware; Internet browsers, word processors, and various other applications are simply given away, in hopes that users will want to buy other commercial products offered by the company.

## Releasing an Open-Source Program

In a variation of the freeware concept, some developers have decided to forgo all potential economic gain by giving away not only their applications, but also their source code. The idea is to allow other developers to look at the code and propose ideas for making your product better.

Why would people want to help other developers make their products better? Because those people are probably also *users* of those developers' products. So think of open source as a big programming commune, where everyone works together for the good of the application, making a stronger product for the future.

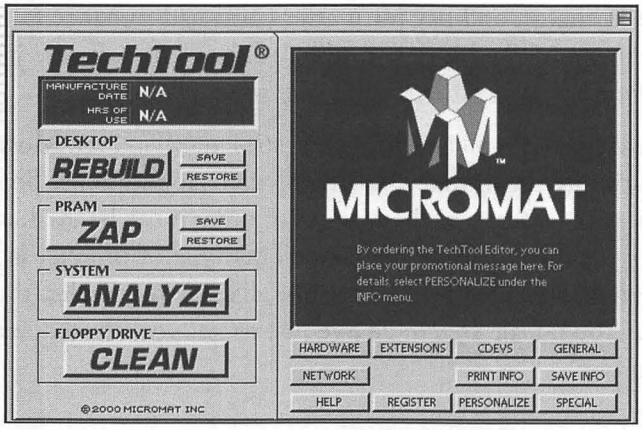



**Figure 2.3**
TechTool, from Micromat Systems, is an example of a free application.

Shades of communism aside, open-source products are making tremendous inroads. Operating systems (think Linux), Internet browsers, and word processors are only some of the products that are being released as open-source products.

## Support

After you've released your software, it is inevitable that the people who use it will have questions about it. Use of your software, no matter how well it's documented, and no matter how well it's been debugged, will always prove to be problematic to some user somewhere. For this reason, regardless of your release and marketing plan, you should plan for product support.

Assuming that you plan to develop shareware or freeware applications, putting your e-mail or Web address in your software's documentation will go a long way toward handling support issues, enabling people who use your software to contact you.

If you plan to release your product in retail channels, then e-mail and Web support might be enough. Even so, you should plan to offer telephone support of some kind, whether it's a long-distance or toll-free phone number. You might even want to consider on-site support of your product if it is a particularly complex application.

Finally, plan on keeping a record of all of your support calls and e-mails. They will help you target those areas of your business that have caused problems in the past. Keeping a database of everyone who has contacted you is a good idea. You can send e-mails to all these users when a new version of your software is released, or when you need help testing a beta version of your software in the future.

# *Sprecken sie* REALbasic?

Or, "Do you speak REALbasic?" for those not familiar with German.

Programming, coding, developing, or whatever else you want to call it is the process by which you instruct the computer to do what you want it to do. Unfortunately, the computers of *Star Trek*, which can be programmed via

## Development Names Are All Greek to Us

Various releases of software are often referred to using the Greek letters *alpha* and *beta*. Meaning first and second, respectively, these letter designations are used to indicate what type of software release is being done.

An *alpha release* usually refers to a release that is meant for in-house testing purposes only. An alpha release usually is not a complete release. Portions of the application may be missing, or there might be spelling errors, debugging code, and other testing specifics in the application, which you would never want the actual users of your product to see. Alpha versions should never be sent to end-users unless you know they are trustworthy or they are willing to sign a non-disclosure agreement.

*Beta releases* are versions in which all the coding for the application is complete and only debugging of the application needs to be done. Software authors will often select a set of users to beta-test their applications. These beta testers get an early look at the application in exchange for free software-testing work. A *public beta* is a release in which anyone in the world can obtain and test the software, usually by downloading it from a Web site. A *private beta* is a release in which potential beta testers must apply, usually with a resume of testing experience, before being approved as a beta tester.

simple spoken commands, are light years in the future. You must be able to speak to computers in languages that they understand, and sadly, English is not one of them.

Throughout the history of computers, countless languages have been used to write computer programs. So many languages have been used that no one

programmer could claim to be proficient in them all. With names like RPG, ALGOL, PASCAL, BASIC, FORTRAN, LISP, C, C++, ASSEMBLE, COBOL, DIBOL, DBL, and ACTOR, computer languages vary almost as much as spoken languages.

Like spoken languages, each programming language can have different dialects. Since its creation, BASIC has spawned perhaps the most dialects of any computer-programming language, including GWBASIC, BASIC-A, QUICK BASIC, BUSINESS BASIC, Visual Basic, and REALbasic, to name a few. So although you may "speak" a particular computer language, keep in mind that there may be other versions of that language that vary slightly from the one you know.

At the very least, computer-programming languages do have the equivalents of verbs and nouns. They have to know what to do, and what to do it to. Additionally, all computer languages can handle conditional statements—that is, they can instruct the computer to do a certain thing only if a specific condition is met. The one thing that computer languages lack is a conversational aspect. You wouldn't want to communicate person-to-person using computer-programming languages. They are very command oriented. Computer languages are best used when issuing commands that must be obeyed without question.

# Review

This chapter covers the "the meaning of programming," or how computer programs are simply lists of instructions that tell a computer what tasks to perform. We discussed the various stages of computer development, including requirements (these determine what your application will do and how it will do it), design (this determines what your application and its source code will look like), programming (where you actually write the source code for your application), testing and debugging (where you find and correct any mistakes that you made in programming), implementation (this refers to the delivery of your completed program), and support (this involves keeping your customers happy by addressing any problems they might have with your program). Finally, we discussed how programming languages are both similar to and different from spoken languages.


# The Parts of a Mac Program

## In This Chapter

- From the source: programming code
- The GUI: why Mac programming can seem a little tougher
- Resources: how pictures and icons are connected

In the previous chapters you got a taste for programming in general. Now it's time to explore a bit of the early days of programming and operating systems, and to show what makes programming on a Macintosh different from programming on other operating systems.

# From the Source: Programming Code

Computers aren't very smart or very conversational. Human languages are way beyond a computer's comprehension, and will be for many years to come despite recent advances. And, no, shouting at your computer when it gets screwy never really helps.

Yes, I know, you're saying, "But I can buy software that lets me talk to my computer!" That's true, but the voice-recognition software had to be created by a programmer using programming code, or simply *code*.

## In the Beginning

A computer's native language really consists of binary numbers—ones and zeros. These binary numbers act like a light switch. Flip it up, the light goes on; flip it down, the light goes off. Likewise, binary numbers act as a series of on and off electronic pulses that the computer's hardware understands as "yes" and "no," respectively. Send the correct binary instructions to a computer component (usually the processor, which is the eyes, ears, heart, and brain of a computer) and the computer starts work on a particular task.

In the earliest days of computing, scientists sent instructions to computers using programming languages that weren't much different from strings of numbers like `1010111010`. Each computer was built by hand—you couldn't run down to a store to buy one. Additionally, each computer required you to communicate with it in a specific manner—usually not involving a keyboard. Switches, buttons, and punch cards were common in early computers, used to start programs and enter data needed by the program to complete its work. Disks, such as floppy disks, hard disks, and CD-ROMs, didn't exist.

Early computers used *vacuum tubes*, large switch devices that could start or stop the flow of electrons through it so that binary signals could be sent. But vacuum tubes generated a lot of heat, used a lot of power, and were slow. Because of the size of these tubes, early computers occupied entire rooms—sometimes even entire floors.

Not surprisingly, programming with one of those early computers was slow, and, because of the limitations of vacuum tube-based computers, it took hours for the computer to spit out the results gleaned by the program. If scientists entered the program or data incorrectly, the computer's data would be wrong, or the program might abort before it completed, resulting in hours of reprogramming the computer and awaiting the results.

Oh, and did I mention that displays like your VGA monitor didn't exist, either? Scientists had to view their works in progress by using a few lights on a panel. The results of a program were usually just a series of numbers or a few words on a printed page. As computer technology advanced in the 1960s and 1970s, early video displays appeared, making it easier to enter and display computer information. Even so, computer data still appeared as dull numbers and letters on a video display or piece of paper.

## The First Modern Circuits

In the early 1950s, three wise men from AT&T discovered that certain nonmetallic substances could be joined to form a sort of switch with no moving parts—a *solid-state device*. These scientists took a chunk of one substance, then mashed a smaller chunk of a second substance in the middle of the first. Next, they placed a wire in the center of the second substance, and placed one wire on each side of the first substance. When the scientists applied an electrical charge to the wire connected to the second substance, electricity could flow from one end of the other two wires, through the first compound, to the other. Thus, the scientists created the first *transistor*.

Transistors were very small because they required the use of only a small amount of the substances, called *semiconductors*. Semiconductor materials were also in plentiful supply because the primary ingredient, silicon, can be refined from ordinary sand. With the development of the semiconductor, electrical circuits could be miniaturized, enabling computers to shrink in size.

# From Interpreting Systems to Operating Systems

The first personal computers of the late 1970s could be programmed to display text, and to draw simple lines and shapes on a screen, and perhaps to spin around a little. Still, telling the computer what to do required you to type in many lines of computer-programming code. Early personal computers didn't come with an operating system like the Mac OS or Windows, but only with hardware designed to translate, or *interpret*, programming code a user would enter into the keyboard. Although the first personal computers were much, much smaller than their 1940s counterparts, they didn't work very differently.

## Early Personal Computers Get Smarter

Radio Shack's TRS-80, the first mass-produced, commercially sold personal computer, included built-in software that understood the computer-programming language known as BASIC. Based on a more-complex programming language used on the larger university and scientific computers, BASIC was designed for novice computer users to create computer programs. Combined with a cassette tape player, a slow but efficient way to store a completed program, TRS-80 users could load or save their work.

BASIC was a start, but it didn't lend itself to being very versatile. Most importantly, BASIC couldn't talk very well to the computer hardware or to things attached to the computer, making it difficult to make the computer more autonomous—that is, more able to perform tasks without continuous supervision.

IBM, an office-products company, took its turn at making a personal computer, but aimed it at both the home and office. Instead of installing a BASIC interpreter and requiring users to type in computer code, IBM took some lessons from the large university computers running a promising operating system known as UNIX, and developed an operating system for their PC.

IBM's PC-DOS and other new operating systems helped establish the computer-software industry, where programmers become producers of software for others to buy and use. Still, the simple flashing prompt was quite daunting to people who didn't really know what to type in to make their programs work.

### Do It with Pictures

At a research division of Xerox, the company that makes office photocopiers, some computer whizzes were toying with the idea of creating a new operating system that presented itself much differently from PC-DOS. Each element of the operating system with which people would interact was represented with a picture. Floppy disks (a recent invention) and directories appeared as a representation called an *icon*. The contents of the disk could be viewed in a listing in a frame called, appropriately enough, a *window*. Moving or selecting icons and windows on the screen called for the use of a *mouse*, an unusual device at the time, which moved a cursor anywhere on the screen. To instruct the computer to perform a command on a selected item, an object called a *menu* could be opened with the cursor to display a list of options.

It was a very intriguing experiment for the few that saw this prototype operating system. Of these few people were two with whom you're probably familiar: Steve Jobs and Bill Gates. After the visit, Jobs took the idea back to Apple Computer to consider. Gates thought the idea was novel, but not significant, probably owing to his programming experience. As history would later prove, Jobs had a greater vision of what computers could do, and so licensed Xerox's concepts to build his first attempt at a personal computer with a graphical user interface, or *GUI*.

Jobs' vision, manifested in a personal computer called *Lisa*, was promising but cost a ridiculous $10,000. Worse, like Bill Gates, others did not view machines with a graphic interface as serious computer systems. Apple stopped production and later buried hundreds of unsold Lisas in an unknown landfill. Back at the drawing board, Apple simplified and refined the GUI concept and built a new, much smaller box. In January, 1984, the Macintosh was born.

## The GUI: Why Mac Programming Can Seem a Little Tougher

Apple encountered many of the pitfalls and challenges of a complex operating system like the original Mac OS (then, it was known simply as *system software* or *the System*). The first challenge was to allow software developers to generate applications without having to create code for the graphics as well as the program itself.

Creating all the graphic code as well as the program's code would be so time-consuming that developers would be very uneasy about developing Macintosh applications, or perhaps discouraged altogether. Even if a programmer cared to write the necessary code that displayed the windows, buttons, and icons on the screen, it was impossible for the graphic interface to look or work the same from one application to the next

Apple solved this programming challenge by creating the programming needed for all the graphic elements and installing it permanently in each Macintosh as read-only programming. These program parts could be called up by a developer's application to create a window, menu, dialog boxes, alert, and so on with the relative simplicity of the old BASIC command for subroutines. Apple named this collection of graphic interface tools the *Toolbox*. (We'll discuss the Toolbox and how it is transformed in Mac OS X in Chapter 19, "The Carbon Environment.")

Despite the Toolbox and other aids, programming on the Macintosh was more complex than creating a simple DOS application, and still a bit more complex than creating a Microsoft Windows application because of Apple's requirements in any Macintosh design.

For instance, on a Macintosh, a programmer must design an interface for any application with which the user must interact. That sounds obvious, but consider the many applications made in DOS that didn't show you much more than a blinking cursor until you pressed a button on the keyboard. Because the Mac OS stresses ease of use and simplicity, developers must adhere to Apple's requirements so that users aren't confused by the appearance and functioning of, say, the Open dialog box in one application versus another.

Because graphic elements such as menus and windows are added to practically every facet of a Macintosh program, it takes a bit more time to check not only the program code, but the interface elements themselves, for errors.

# Resources: How Pictures and Icons Are Connected

In DOS, every data file or application is formatted more or less the same. The only difference in DOS programming may be in whether the file contains executable code—that is, programming that comprises an application on the computer that would start up when called.

## How Many Viruses?!?

A *computer virus* is a program designed to cause weird or harmful things to occur in another computer's applications or operating system. Many of you may remember the "Melissa" and "I Love You" computer viruses of 2000. These viruses were designed to infect users of Microsoft Office applications. That is, Microsoft Office *for Windows*.

Macintosh users have a version of Office that's compatible with Office 97 for Windows documents, but these viruses didn't affect the users of the Mac OS. One reason Mac users kept typing along without much concern involves how Office 98 Macintosh Edition was designed, or not designed as the case may be. The Windows version of Office contains software that runs Visual Basic applications that, in the right hands, give Office for Windows applications and documents extra features. Microsoft, however, left a few loopholes in their use of Visual Basic as well as their Word and Excel macro languages. When infected documents (or, in the case of the "I Love You" virus, file attachments) were opened, very nasty things happened that clogged e-mail servers around the world.

Macintosh versions of Microsoft applications have very limited support for Visual Basic applications, so most VB applications simply don't operate—especially not in the Mac version of Microsoft Outlook for Exchange Servers, where the "I Love You" virus presented its payload to millions of users. Likewise, most Word macro viruses are written with Windows file directory structures in mind because the virus makers aren't familiar with or don't care about creating a Mac version of their virus.

There are about 20,000 viruses that infect Intel PC hardware. Because there are fewer Macintosh developers, there are, per capita, even fewer Mac virus makers. Because of the complexity and rules involved in a Macintosh application, there are only about 60 viruses that can affect Mac OS 9.

That doesn't mean, however, that Macintosh users can't be carriers of viruses. Be sure not to send documents that are infected to PC users. Remember that Mac OS X is a completely new operating system based in BSD, which has its share of viruses as well that could be mutated into something nasty.

And please, use your blossoming programming powers for good, not for evil.

A Macintosh file is actually composed of two parts known as *forks*. The *data fork* contains either document data or executable code. The *resource fork* contains information about the document as it relates to the Mac OS—essentially, the icons, menus, windows, and other graphic pieces found in a Mac application.

The great thing about resources for non-programmers is how easy they make it to change, or hack, parts of a document or even an application. For instance, suppose you love your favorite word-processing application so much that you want to add a new menu signifying your love. Using Apple's free resource-editing tool, ResEdit, you could add a new menu that did nothing more than show itself on your word processor's menu bar. Or you could change the colors of various menu commands to make things livelier.

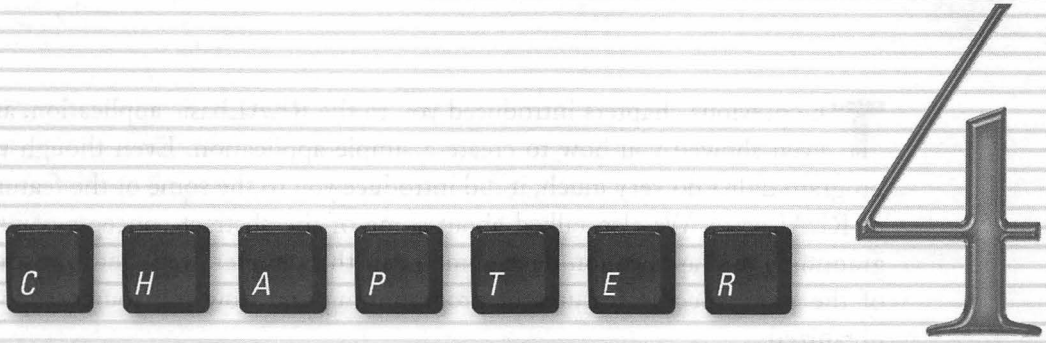
While ResEdit is still a great development tool, there are additional tools you'll discover in Macintosh development that change the old rules about resources. Mac OS 9 applications rewritten for Mac OS X begin to separate the resources from the data of a file to conform more to the UNIX and Windows methods of data distribution. By the time you develop an application designed for use only in Mac OS X, resources and data are completely separate.

You'll want to use resources with care and determination so that your application works as you expect. Resources work much like REALbasic and other programming environments in the sense that they are called on as objects in an application. To simplify, resources are "plugged in" to your application as modules; that's not much different in effect from true object-oriented programming such as in REALbasic.

As you'll learn in Chapter 19, "The Carbon Environment," Apple revised the list of resources available in the original Mac OS so that developers like yourself can modify their applications to take advantage of Mac OS X, the next-generation operating system.

# Review

This chapter introduces the parts that make up a Macintosh application. In the next chapter, we'll introduce you to REALbasic commands, the instructions that make up an application that interacts with itself, the computer, and you.




# Under Your Command

## In This Chapter

- What commands do
- Trying out some REALbasic commands
- Good documentation makes happy programmers

The previous chapters introduced you to the REALbasic application, and even showed you how to create a simple application. Even though the program didn't do very much, it did introduce you to the some of the features of REALbasic. We also talked about some of the abstract concepts of programming and application development. In this chapter, we'll get into some of the more specific features of programming and how the source code is organized.

# What Commands Do

As was mentioned in previous chapters, the source code of a program is the text portion of a project that includes all the instructions that tell your program how to behave. The source code is arranged in lines, just like any other text document. A program's source code is executed from the top down. If you've ever cooked anything by following a recipe, then you should already be familiar with this concept. The recipe is a list of instructions, which must be performed in a certain order, just like a computer program.

The instructions in a computer program are referred to as *commands*. Most commands are single, simple instructions. Commands usually do one thing and one thing only.

Suppose for the purposes of example that you have children, and that your children are programmable just like computers (oh, how I've wished). Say they have their own programming language, which we'll call *KidTalk*, that looks pretty much like English. If you wanted to write a KidTalk program called *BedTime*, the source code for the program might look something like this:

```
Wash Hands
Brush Teeth
Put on Pajamas
Go to Bed
Go to Sleep
```

From here, your Kid–o–Matic should compile your KidTalk instructions into usable steps that it can understand and implement. Realizing that my kid was late for bed, I hastily executed the BedTime application. Figure 4.1 shows that it appears to be a successful test, but perhaps we should add a `clean the room` instruction next time.

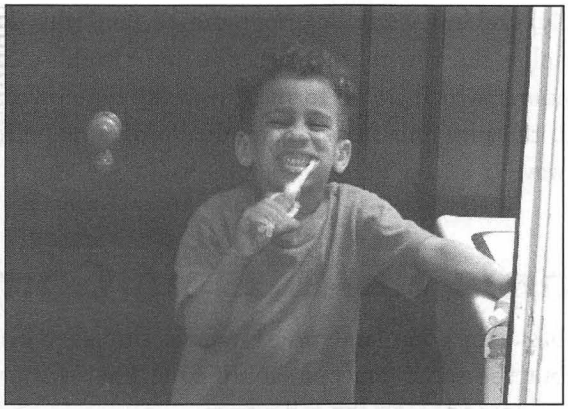
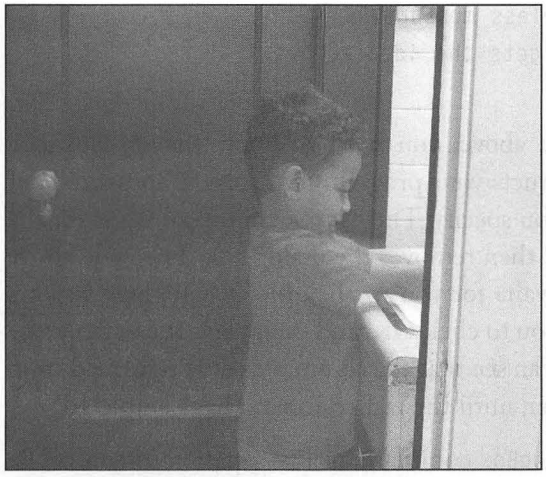
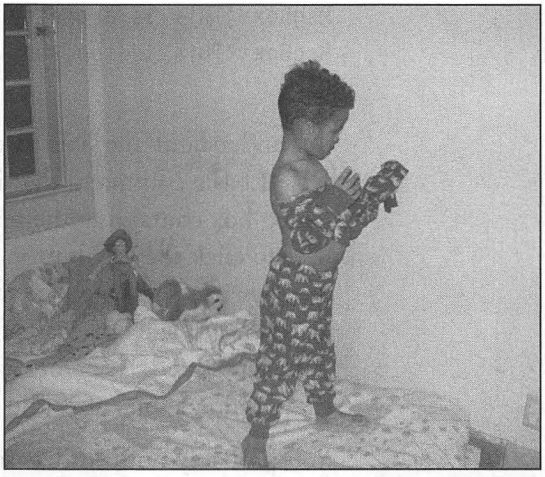
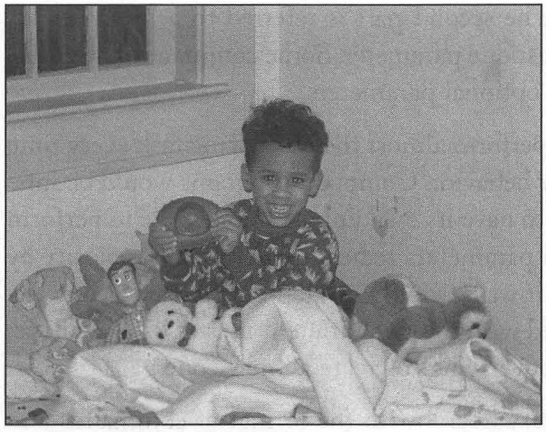


**Figure 4.1**
Computers don't respond to yelling, and neither do kids. But, the right commands in the proper sequence can make for a quiet night.







Realistically, children can't be programmed, and the commands above are pretty complex. But you get the idea. The point is, this example is a list of simple instructions, which are performed from the top down. After the last task is completed, the program stops. Computer programs perform in pretty much the same way.

# Trying out Some REALbasic Commands

In REALbasic, the commands aren't quite as similar to English as the KidTalk example, but the same concepts apply. One line of source code contains at most one command, as in the following:

```
MsgBox "This is a pretty useless message."
MsgBox "This is too, but it gets the idea across."
Beep
```

To understand the source code above, you need to know that `MsgBox` is a REALbasic command that instructs your program to display a "message box" dialog box containing the text you specify. The `MsgBox` command waits for the user to click the OK button and then removes the dialog box. The code above displays one message box and waits for you to click on OK. It then displays another message box, waits for you to click OK, and then plays the default system alert sound. As before, you can see that the commands are performed from the top down, one after the other, until the last command has completed.

You probably noticed that the `MsgBox` command is a two-part command. The first part is the `MsgBox` command, and the second part is the text that you want to appear in the message box. The second part is referred to as a *parameter*. Almost all commands have at least one parameter. Some commands have multiple parameters and some have optional parameters.

Parameters allow commands to perform almost the exact same task every time while altering one aspect of their behavior. Computer programs would be infinitely complex if every task had to have its own unique source code to perform each operation. Commands and parameters reduce the level of complexity by doing one task, in many different ways, simply by altering the parameters. By altering the parameters, as we did with the two different uses of `MsgBox` above, you can alter the behavior without having to create all new code.

Some computer languages are very liberal with the format of commands and their parameters, allowing commands and their parameters to be on separate

lines. Some computer languages also allow multiple commands in one line of code. However, in REALbasic, a command and its parameters must reside in the same line of code. For example, the following source code will generate an error if you attempt to run it in REALbasic:

```
MsgBox
    "This is a program that won't run"
```

Likewise, the following source code, with more than one command on a line, will generate an error, because REALbasic only allows one command per line:

```
MsgBox "One" MsgBox "TWO" MsgBox "THREE!"
```

Although commands and parameters offer a lot to computer language, they can't do everything. Using just commands and parameters, your programs can't alter their behavior in reaction to changes in conditions. Programs wouldn't have the capability to efficiently perform the same task multiple times if all you had to work with were commands and their parameters. We'll discuss the ways in which programs can become more flexible in the next few chapters.

# Good Documentation Makes Happy Programmers

Before we get much deeper into more programming concepts, we should touch on documentation. One important thing to remember when writing any program is that the source code can be confusing, as you can tell from the simple examples above. Source code can be hard to understand for developers who haven't worked during all stages of the project, and even to developers who are returning to a project they haven't worked on for a long time. To reduce confusion, it is essential that you maintain good source-code documentation.

In this section we'll discuss some of the various methods of source-code documentation and the benefits of some methods over others.

## The Myth of Self-Documenting Code

If you talk to developers long enough, one of them is bound to mention something called self-documenting code. Usually they refer to it jokingly, but some actually believe in the practice and use it religiously.

The concept of self-documenting code is simple enough: If you write your source code properly, then anyone can simply read the code and tell what it does. It's a noble concept, but in practice source code, no matter how "self-documenting" it is, is never easy to read.

Self documenting–code advocates will argue that all source-code modules should be as simple and compact as is feasible to reduce complexity and confusion. Of course, every piece of paper in a filing cabinet is simple and compact, but you wouldn't want to read every single word on every single page just to find the part you want. Documents are organized in a filing cabinet in folders, and the documents themselves often contain headings and explanatory text to help the reader find what he is looking for. So why shouldn't source code be treated the same way?

When you start talking about projects with hundreds, thousands, or even hundreds of thousands of lines of source code, the idea of self-documenting code becomes absurd. Forcing a developer to search through the multitudes of source-code files and modules squanders their talents. They should be programming, not trying to figure out your code.

As bad as it sounds, the concept of self-documenting code does have one benefit: It requires that the developer write code that is easy to understand. A lofty goal, but not impossible.

## Documentation Repositories

Because we've all but given up on the idea of self-documenting code, we need to come up with an alternative. One way to document your source code is to create a separate document, in the word processor of your choice, detailing each source-code file and module, and what all of the various parts do.

After all your source code has been documented, you then place all your source-code documentation in one simple-to-find location so that you can reference it in the future.

This documentation repository becomes one of the most important assets of your business, and must be maintained and protected at all cost. Several tools on the market provide centralized document repositories, which maintain version histories of the documents and provide simple methods of backing up your documents so that their safety is ensured.

A *third-party tool* is an application, or tool, that a developer purchases to perform functions not included within the application-development tool. These tools could include code beautifiers, which clean up your source code; documentation tools; design tools; paint programs, and so on.

This "Fort Knox" approach has problems of its own. First, it forces you, and any future developers, to switch between the word processor or paper documents and the application-development tools to reference source code to its documentation. This approach either requires the manual maintenance of your documentation repository or the purchase of third-party tools to maintain your repository. These third-party tools can be expensive and unwieldy. Remember, too, that the less-expensive documentation-repository tools have fewer features and may not be a robust when it comes to maintaining the integrity of your documentation.

# The Promise of Inline Documentation

Fortunately, almost every programming language allows for inline source-code comments—documentation embedded within the source code itself. This allows the developer to place notes, annotations, explanations, or full-length novels right in the source code itself. Future developers, or the original developer years later, will appreciate the convenience of seeing the explanations of what the source code does right there along with the source code itself.

This practice makes the maintenance of source code a lot easier, but lengthens the initial development time. The "there wasn't enough time to document the code" argument is the most often quoted reason for not using inline documentation. It's a poor excuse for making someone else's job harder, though.

The easiest way to force yourself to do inline documentation is to do it before you write the code. As soon as you create a new source code module, document what this module will do, what its purpose is, and the steps it takes to achieve this. Then go back and add the source code around and between the documentation.

## Inline Documentation and REALbasic

In REALbasic, inline documentation is referred to as *comments*. The REALbasic development tool ignores any comments in your source code. These lines are skipped during compilation and execution. When you compile your source code, the comments aren't included in the executable application. As far as REALbasic is concerned, it's as though these comments don't even exist. But, they're of great use to the developer.

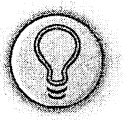
Back in the "good old days" of BASIC programming, the only way to introduce a comment in the source code was to use to the REM (short for "remark") keyword. Here's an example:

```
10 REM This is a stupid, useless, and annoying program!
20 PRINT "Hello World! ";
30 GOTO 20
```

REALbasic allows for multiple methods of commenting source code. For the most part, these different methods are used to make life easier on developers who have worked in other programming languages. REALbasic comments are marked using any of the following methods:

- `REM` The classic BASIC language REM keyword
- `'` The classic BASIC language REM shortcut single-apostrophe comment delimiter
- `//` The C language-style double-slash single-line comment delimiter

Because I started my development career as a BASIC programmer, I tend to use the apostrophe comment delimiter. One good reason to use this comment delimiter is that the REALbasic Comment Lines menu item uses this format as well. That said, a lot of people prefer the double-slash single-line comment delimiter. Use whatever is easiest for you.



**TIP**

To use the REALbasic Comment Lines command, open the Edit menu and select the Comment Lines item. If you use this option with no text, selected a single-apostrophe comment delimiter is placed at the beginning of the current line of code. If multiple lines of code are selected, a single-apostrophe comment delimiter is placed at the beginning of each selected line of text. This is often referred to as *commenting out* lines of code, because the source code has been turned into a comment. This is a quick way to remove source code without actually deleting it (in case you change your mind later).

Whatever commenting method you choose, remember that everything to the right of the comment delimiter is ignored, as in the following lines:

```
'This entire line is a comment
MsgBox "Test" 'This portion of this line is a comment
```

## The One and Only Documentation Solution

The one and only solution to the source code-documentation issue is to do whatever works best for you. For almost everyone, this is going to be a combination of any or all of the methods mentioned above. Each method has its problems and benefits:

- Self-documenting code requires that developers be somewhat psychic, knowing where to go to find the documentation they need. It's talked about more often than it's actually used. However, it does enforce clean, easy-to-understand source code.
- Source code repositories cause maintenance hassles and disassociate the documentation from the code. On the other hand they do provide a single source of information that can be effectively browsed without previous knowledge of the source code.
- Inline documentation, like self-documenting code, requires pre-knowledge of the source code, but allows for concise annotations that can be quickly referenced. Pre-commenting your code can be an effective method of improving development efficiency.

By using a combination of these methods you can reap the benefits of each while reducing the problems inherent in them. Remember, use what works best for you with the tools and resources you have available. You might choose to simply create a project overview document and include source-code comments. As long as this satisfies your documentation needs, it's sufficient. Never do more work than is needed to maintain your project. Wasted effort is wasted time.

## Documentation Standards

Whatever documentation method or combination of methods you choose, it's a good idea to set some standards for what the documentation will look like. I'm not going to preach any one standard here; there are plenty of books on the market already for that. Find a standard you like, or make one up, and stick to it. Having all your documentation in one format means that every document is instantly recognizable and easy to use.

I know, this may sound obvious, but you'd be surprised how many developers balk at the idea of documentation standards. Developers are a unique breed, very bohemian and free thinking. The idea of being told how to do their job often

drives them to distraction. But, when handled properly, documentation standards actually can save precious time due to reduction of misunderstandings.

Coding standards fall into this category as well. Many different coding standards exist and are documented in numerous books, journals, and papers. REALbasic does a pretty good job organizing the source code for you, so I won't go into coding standards here. About the only thing you have control of are things like variable, method, and class names (more on these later). Just like with your documentation, you should strive to make your source code have a similar look and feel.

Trust me, documentation and coding standards make the maintenance of an application much easier down the road. The extra effort put forth at the beginning of a project is an investment in time saved later on.

# Review

This chapter covers commands and parameters. Commands are simply the instructions that the computer program follows when it's running. Commands can have parameters, which can alter their behavior and allow for variations of their use. Commands and their parameters must reside on the same line; only one command is allowed per line.

This chapter also discusses the importance of source code documentation, which I can't stress enough. Good documentation equates to reduced effort in the future. Maintaining and modifying existing programs is a difficult task under the best of circumstances; poorly written or nonexistent documentation can make it even more painful. Do yourself a favor and learn to document your code. You'll be glad you did. You can self-document your code, add inline code comments, and use documentation repositories. REALbasic allows for inline source code documentation.

Documentation and coding standards can help reduce confusion and simplify your documentation practices. Coding and documentation standards can be almost as important as the documentation itself. But don't get so bogged down in standards that your work doesn't progress. It's important to remember that it's a balancing act between productivity now and productivity later. If you get buried in the standards and never actually get any work done, you'll never realize the future productivity benefit, because your project will have no future.


# Variables, Operations, and Constants

## In This Chapter

- Keeping track with variables
- Common types of variables
- Declaring variables
- Operations and variables
- Constants are constant
- Where to use variables and constants

The last chapter introduced the concept of commands and parameters. A *command* is a single instruction for your program to perform, and *parameters* are used by the commands to alter the behavior of the command. Like we said, commands and their parameters provide some flexibility, but not enough.

In this chapter, we'll be discussing one way to add flexibility to your programs. When an application is running it needs to vary it's behavior based on changes in its environment. In order to do this a program needs to be able to keep track of values that can change. This is done using variables.

# Keeping Track with Variables

If you've taken a high-school algebra class, then you're probably familiar with variables. In algebra, variables are the letters *X*, *Y*, *Z*, and so on, for which you seem to spend all year solving. If you've never had the pleasure of dragging yourself through algebra, then this will be your first introduction to variables.

In computer languages, variables are used to store values, numbers, or text, which can change, or *vary* (hence the name *variables*), in response to different situations. The variable itself is merely a label that is used to reference the value. When you declare a variable you're telling your application to set aside a portion of the computer's memory, which will be used to store some sort of value. The variable name is a label, which you use to reference that memory. Think of the variable's name like the address on a mailbox as shown in Figure 5.1.

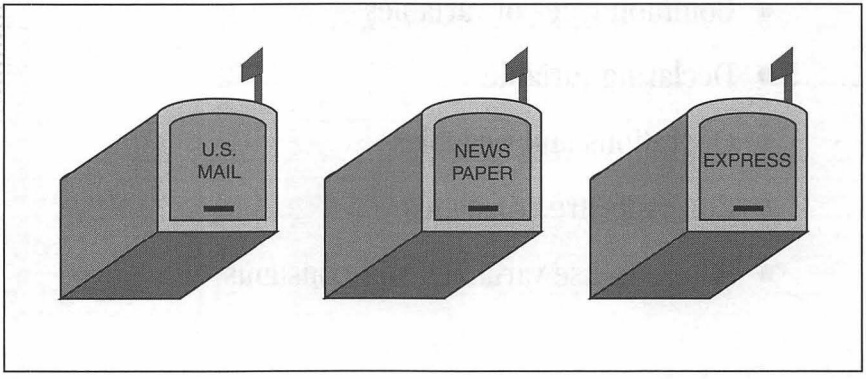



**Figure 5.1**
Variables are like delivery boxes for mail and the like. Only one type of item is allowed in each container, and their contents can change.

The fact that the values can vary allows your program's behavior to vary in response to those situations and conditions.

The contents of variables can be compared to each other, and they can be operated on with math operations. You can add, subtract, multiply and divide using the contents of the variables. A variable's contents can be written to disk for later retrieval. The contents of variables can be displayed onscreen or used to control the display of other items onscreen. Variables allow your programs to do just about everything they need to do.

# Common Types of Variables

Variables can contain, or *reference*, many types of information—just about any type of information you'd want to keep track of. Some of the variable types that are supported in REALbasic are

- **Integer.** Used for whole numbers (0, 1, 2, and so on). Either positive or negative values can be stored in integer variables. The valid range of integer variables is –2,147,483,648 to 2147483647.
- **Single.** Used for single-precision floating-point real numbers (for example, 3. 141592653). Single-precision variables can contain numbers accurate up to nine decimal places.
- **Double.** Used for double-precision floating-point real numbers (for example, 3. 141592653589793). Double-precision variables are accurate up to 15 decimal places.
- **Boolean.** Used for true/false (Boolean logic) values.
- **String.** Used to store text values.
- **Variant.** A variable that can contain numbers, text, or other types of values.

# Declaring Variables

To use a variable in your program, you must first define the variable so that REALbasic knows what type of variable it is. You declare variables in

REALbasic by using the `Dim` statement (short for *dimension*). Some examples of variable declarations are

```
Dim nAge, nWeight AS Integer          'Declare two integer variables ➥
    named nAge and nWeight
Dim dSalary AS Double                 ' Declare a double variable ➥
    named dSalary
Dim strName AS String                 ' Declare a string variable ➥
    named strName
```

Another method of declaring variables allows you to declare arrays of variables so that your variables can store lists of values. When declaring variable arrays, you define the name, dimension, and the type of the array. Some examples of array declarations are

```
Dim nDaysPerMonth (11) AS Integer     'Declare an array of 12 integers ➥
    named nDaysPerMonth
Dim strWeekDayNames (6) AS String     'Declare an array of 7 string ➥
    variables named strWeekDayNames
Dim nAnInteger(0) AS Integer          'Declare an array of 1 integer ➥
    variable named nAnInteger
Dim intEggs (1, 5) AS Integer         'A two-dimensional integer array ➥
    (2 rows 6 columns–like an egg carton)
```

Something you probably noticed about the examples above is that the array size is always one smaller than the number of values it can store. This is because the first element of the array is always numbered `0`. So, in the example above, the elements in the `nDaysPerMonth` per month array are numbered `0` through `11` for a total of twelve. So, if you want to declare an array of one value, you define an array size of `0` (like in the `nAnInteger` example above). Coincidentally, you can choose to forgo the use of the array size when you are defining an array that has a dimension of one. REALbasic defaults variable declarations with no array size to a dimension of one. So, all the first examples could be written to define the dimension of the array as `0`, but you don't need to; REALbasic assumes that the dimension is `0`.

You can also `Dim` an empty array in REALbasic by using a size of `-1`. Doing so lets REALbasic know that you want to create an array of unknown size. You'd do this in those cases in which you don't know what size you want an array to be, until the application is running. You can then re-dimension the

array, using the Redim keyword, once you know what size the array should be. For example

```
Dim nMonthDays(-1) AS Integer      'Declare an array of an unspecified ➥
    number of integers named nMonthDays
If (nMonthNumber = 2) Then
ElseIf (nMonthNumber)
End If

Dim strName AS String          ' Declare a string variable named strName
```

**TIP**

> You may have noticed that all of the variable names in the preceding examples started with lowercase letters. This variable-naming convention is a programming technique referred to as *Reverse Hungarian Notation*. *Hungarian Notation* is a practice used by some programmers that helps remind them of a variable's type without requiring them to track down the variable declaration. You'll see variables using this type of notation all throughout this book. Use of this type of notation is not necessary; it's simply a method to make your life easier.

All this variable declaration is well and good, but variables are of little use if you don't know how to store and retrieve values in variables. That's what we're going to look at next.

# Assigning Values to Variables

Variables contain values, and the values in variables can be assigned, modified, or operated on using many different operators. The assignment operator, =, is used to store a value in a variable.

One thing to keep in mind is that the assignment operator, along with other operators, works only with similar data types—so, for example, text data can't be stored in an integer variable. The one exception to this rule is a *variant variable*, which is the multiple personality–disorder variable of REALbasic.

Variant variables can act like any other variable type.

Some examples of valid, and invalid, variable assignments are

```
Dim nAge AS Integer
Dim dSalary AS Double
Dim strName AS String
nAge = 34            ' OK--an integer value
dSalary = 25350.50   ' OK--a real number (a.k.a. floating-point) value
strName = "Jeff"     ' OK--a string value
nAge = "forty"       ' Wrong--not an integer value
strName = nAge       ' Wrong--these variables are different types
```

All variables are assigned initial default values when they are declared with the `Dim` statement. Numbers (integers, singles, and doubles) are assigned a default value of `0`. The initial default value for `Strings` is an empty string (that is, `""`). Boolean variables are assigned a default value of `false`.

Back to variant variables: They are variables that can contain just about any data type and can, in some cases, convert their data from one type to another. One example of this is in the code below; where a variant is assigned values of many data types. These data types are displayed using the `MsgBox` command, which only accepts a string as its argument.

```
Dim varJustAboutAnything As Variant
Dim bTrueFalse As Boolean

varJustAboutAnything = 42      ' Assign a numeric value to your ➥
                               variable
MsgBox varJustAboutAnything    ' It will be displayed as a string

varJustAboutAnything = "Text"
MsgBox varJustAboutAnything

varJustAboutAnything = 1.23
MsgBox varJustAboutAnything

varJustAboutAnything = bBoolean
MsgBox varJustAboutAnything
```

# Operations and Variables

We mentioned that there are many operators that can be used along with variables. Some of the more common operators are

- **The addition operator (+).** Used to add two numbers together.
- **The subtraction operator (–).** Used to subtract one number from another.
- **The multiplication operator (*).** Used to multiply two numbers.
- **The floating-point division operator (/).** Used to divide one number by another.
- **The integer division operator (\).** Used to divide one number by another while truncating the result to only the integer portion of the number.
- **The equivalence operator (=).** Used to determine whether the values of two expressions are equal to each other. Not to be confused with the *assignment operator* (=), which assigns a value to a variable.
- **The less-than operator (<).** Used to determine whether the value of one expression is smaller than the value of another.
- **The less-than-or-equal-to operator (<=).** Used to determine whether the value of one expression is smaller than or equal to the value of another.
- **The greater-than operator (>).** Used to determine whether the value of one expression is larger than the value of another.
- **The greater-than-or-equal-to operator (>=).** Used to determine whether the value of one expression is larger than or equal to the value of another.

The following source code demonstrates the use of these operators:

```
Dim dNumber1, dNumber2, dNumber3 As Double
Dim bResult As Boolean
dNumber1 = 9
dNumber2 = 4


dNumber3 = dNumber1 + dNumber2    'dNumber3 contains a value of 13 ➥
(9 + 4)
```

```
dNumber3 = dNumber1 - dNumber2    'dNumber3 contains a value of 5 ➥
(9 - 4)
dNumber3 = dNumber1 * dNumber2    'dNumber3 contains a value of 36 ➥
(9 * 4)
dNumber3 = dNumber1 / dNumber2    'dNumber3 contains a value of ➥
2.25 (9 / 4)
dNumber3 = dNumber1 \ dNumber2    'dNumber3 contains a value of 2 ➥
(9 \ 4)

bResult = dNumber1 < dNumber2     'bResult contains "false" ➥
(9 is not less than 4)
bResult = dNumber1 <= dNumber2    'bResult contains "false" ➥
(9 is not less than or equal to 4)
bResult = dNumber1 > dNumber2     'bResult contains "true" ➥
(9 is greater than 4)
bResult = dNumber1 >= dNumber2    'bResult contains "true" ➥
(9 is greater than or equal to 4)
```

There are other operators, but we'll discuss them in later chapters as needed.

# Constants Are Constant

In writing programs, you may occasionally want to store something in a variable, but never change the value while your program is running. Your application's name, or something similar, is a value that will never change during the time your program is running. (That said, you might want to change this value before you compile your application.)

You're probably thinking that a variable whose values don't vary seems like a contradiction—and you're right, it is. To store values that don't, and can't, change you don't want to use a variable. You'll want to use a *constant*.

To declare a constant, you give the constant a name and assign it a value. For example:

```
Const nTheCurrentYear = 2001
Const strTheProgramName = "Hello World"
```

You can think of a constant as a nickname, or synonym, for the value you have assigned to it. Remember, constants aren't variables, so you can't change their values. If you try to change the value of a constant, the REALbasic compiler will generate an error. For example, the following source code will generate an error at runtime:

```
Const nTheCurrentYear = 2001
nTheCurrentYear = 2002
```

REALbasic defines a few constants for use in your programs, as described in Table 5.1.

**TABLE 5.1 REALBASIC CONSTANTS**

| Constant | Type | Description |
|---|---|---|
| `DebugBuild` | Boolean | Returns `true` if your application was launched with the Debug item on the Run menu, versus running as a compiled application. Handy if you want to display debugging messages while testing your application. |
| `TargetMacOS` | Boolean | Returns `true` if your application has been compiled to run on a Macintosh. |
| `Target68K` | Boolean | Returns `true` if your application is compiled to run as Motorola 68000 machine code. |
| `TargetPPC` | Boolean | Returns `true` if your application is compiled to run as PowerPC machine code. |
| `TargetCarbon` | Boolean | Returns `true` if your application is compiled to run as a Carbon application. |
| `TargetWin32` | Boolean | Returns `true` if your application was compiled to run on a Windows computer. |
| `RBVersion` | Double | Returns the major and minor version numbers, as a double-precision floating-point value, of the REALbasic compiler with which you created your application. |
| `RBVersionString` | String | Returns the major and minor version numbers, as a string value, of the REALbasic compiler with which you created your application. |

As you can see, the REALbasic-provided constants can tell you a lot about the operating environment in which your application is running. Chapter 22, "Porting Applications to Microsoft Windows," talks a bit about how you can use these Boolean constants in creating an application for Microsoft Windows as well as for the Mac OS.

# Where to Use Variables and Constants

Variables and constants can be used anywhere in a program where a command parameter is needed or as the parameters of any operation. For example:

```
const   strTheApplicationName = "Hello World"
msgBox  "The " + strTheApplicationName + " now does useless math!"
```

A good general rule is to use variables if you know a value is going to change while your program is running. Use constants if you use the same value, in many places, but you don't expect the value to ever change while your application is running. Constants are also well used in situations in which values change for different versions of your applications, like the program name or the program version number.

Another use of constants is to give a meaningful name to an otherwise meaningless value. For example, you might realize that the value 3.14159 is the mathematical value of PI, but other developers may not. Creating a constant with the name PI and a value of 3.14159 allows you to use this constant throughout your source code, and you don't have to remember what the value of PI actually is.

As time goes on and you get more experience writing your own programs it will become quite obvious to you when and where you should use variables and constants.

# Review

In this chapter, we went over variables and how they can be used to keep track of various values. We talked about some of the various variable types common to REALbasic and other software development tools, went over the integer, single, double, Boolean, and string variable types, and touched on the variant data type, which can act as any of these types of data.

We talked about declaring variables with the `Dim` statement, and went over how both variables and arrays of variables are declared. All arrays are zero based, so arrays are numbered from `0` to `n-1`, where `n` is the number of elements you want to set the array size to. It's no big deal if you declare your arrays too large; you're just using up memory that you don't need to.

We then went into some detail about the more common operators that can be used to modify and inspect the contents of variables. We talked about the most often–used math operators (+, –, *, /, and \) and the comparison operators (=, <, <=, >, and >=).

We then talked about constants, which are like variables whose values don't vary. In a sense, they can be thought of as non-variable variables. Constants are best used for values that never change, are used in multiple places in your application, or might change from one version of your application to the next. You should never attempt to use a constant if its value is going to change while your application is running. Doing so will cause an error in REALbasic and your application won't run.

Last but not least, we talked about how variables can be used as the parameters of commands and operations. Variables have to be used in order to be appreciated, so, if you haven't already, go back to the "Hello World" application from Chapter 1, "Getting Acquainted with REALbasic," and start playing around a bit with some of the sample code. We think you'll get an understanding of how essential variables are in no time.


# Making Your Program Flow

## In This Chapter

- What is flow control and why is it important?
- The If/Then/Else If/Else/End If keywords
- The Select/Case keywords
- The For/Next keywords

In the previous two chapters, we talked about commands and variables, noting that commands are the instructions that the program follows when it's doing its thing, and variables are used to store values, which vary, during the runtime of the program. The program can use these values as parameters to commands and can respond to the values of the variables to behave in different ways. What we didn't get into was *how* a program responds to the variables in order to behave differently. We cover that here and in Chapter 7, "And Still More on Program Flow."

# What Is Flow Control and Why Is It Important?

Although a program is running, it needs to respond to various conditions that change during its runtime. If a program consisted merely of a set of instructions, one after the other, but had no ability to respond to changes in the environment, it would be a very poor program.

Flow control is the capability of a program to respond to different values in variables and take different routes depending on those values. Think of it like a series of branching irrigation channels carrying water. The flow of water into each branch is controlled by blocks that are moved to stop the water flowing into one branch and allow it to flow into another. Coincidentally, the various sections of code, which are executed during flow control, are referred to as *branches*. At every branch, the program decides on which path it will take.

Alternatively, you could think of these branches as being like branches in a stream. Just as you can travel by raft down only one branch of a stream at the same time, your programs can only execute one branch of code at a time.

To better show you visually how flow control works, we'll use a classic programming tool: the flowchart (see Figure 6.1). Basically, computer instructions (shown as parallelograms) travel from top to bottom. To stop the flow so that a decision can be made, add a diamond shape to allow the program to respond to Boolean (yes/no) questions, the answers to which will redirect the flow path. Simple, right? Good.

There are many methods for achieving flow control. A program can choose to execute one of many branches of code based on the contents of a variable. It can execute the same code over and over again until a variable has changed to

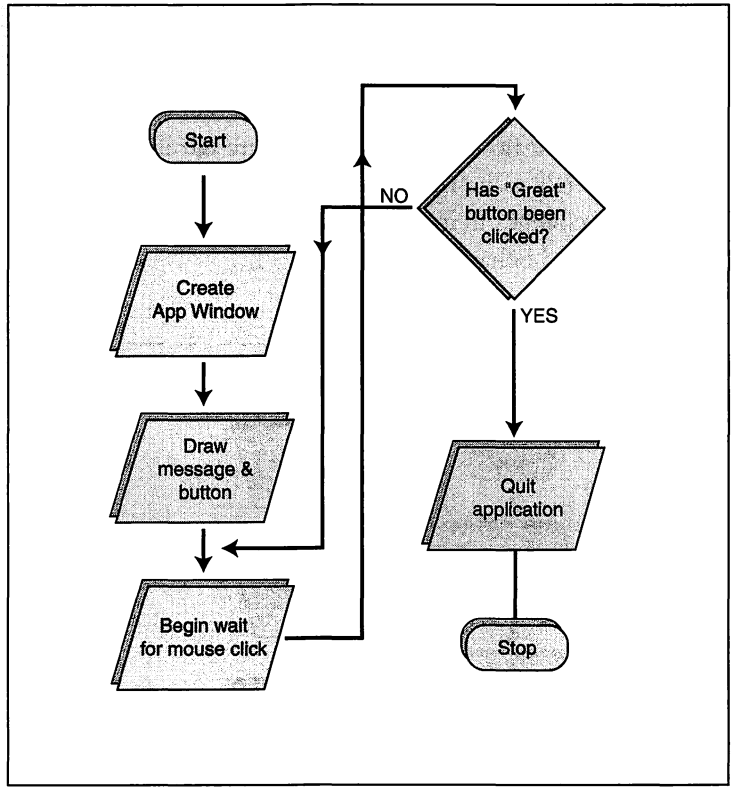



**Figure 6.1**
A flowchart is a diagram of a program's . . . um . . . flow! This is a good tool to use to avoid simple program-design mistakes.

a specific value or while it remains as a previous value. Programs can even skip entire sections of code using flow control.

To put it simply, flow control is not only the most important part of programming, it is the *essence* of programming. Every program you've used has within it some amount of flow control.

It should come as no surprise to learn that some programs have very poor flow control. We won't point to any single software manufacturer, even though there are plenty of them with problems. Instead, we'll quote an example from nature (nature can't get a lawyer to sue us).

There is a species of spider that, like the trap-door spider, makes its burrow underground; rather than springing on its prey, however, it hunts in the open, and then brings its spoils back to its lair to feed. This type of spider has been "programmed" by evolution to follow a certain set of steps when it's hunting.

If its instructions were drawn in a flowchart, the spider's "program" would look like the one in Figure 6.2.

This program is simple enough, but it has its problems. Although there is some implied flow control, the "Ensure lair is safe" step implies that if it's not safe, it needs to be made safe—a major flaw in the spider's program. It can't adapt to changes in its variables, the environment, which alter the way the program should behave.

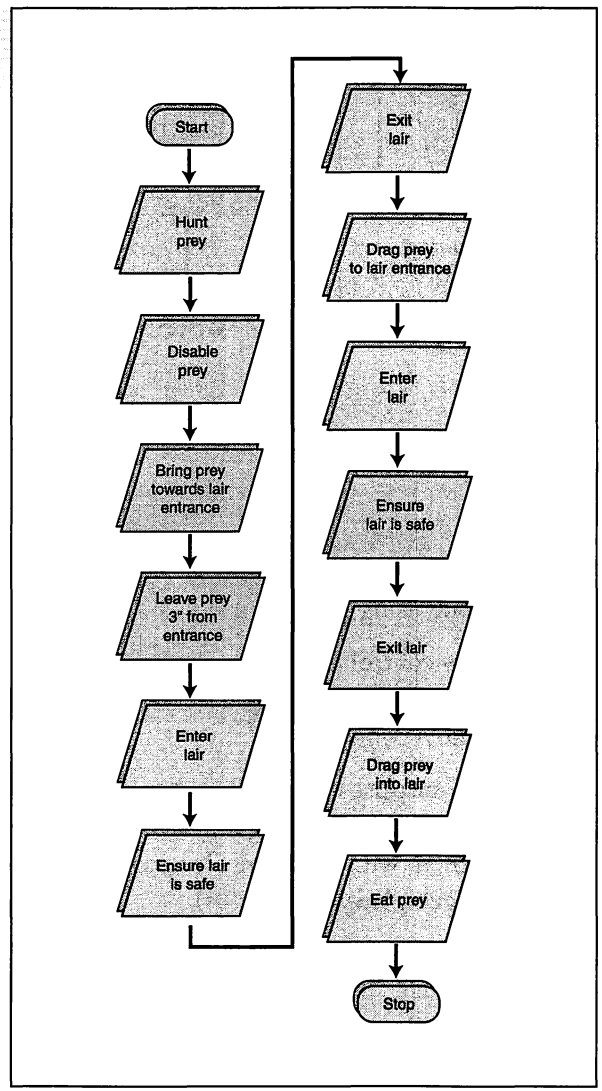



**Figure 6.2**
"Come to my Parlour . . . ," computer style.

Let's introduce a variable to the spider's program with which it is unprepared to cope. The variable you'll be introducing is a particularly inquisitive and, perhaps, somewhat cruel individual. Make him a young boy or a research scientist—they're both about as nasty when it comes to experimentation with bugs. This individual waits for the spider to enter its lair the second time, after dragging its prey to the entrance of the lair. If the individual moves the prey back to its first location, a few inches from the lair, the spider will repeat the previous steps. It will continuously move the prey to the entrance, check the lair, and return to do this again, again, and again. This can, if the experimenting individual is fiendish enough and so chooses, continue until the spider literally falls over dead from exhaustion and starvation. As you can see in Figure 6.3,

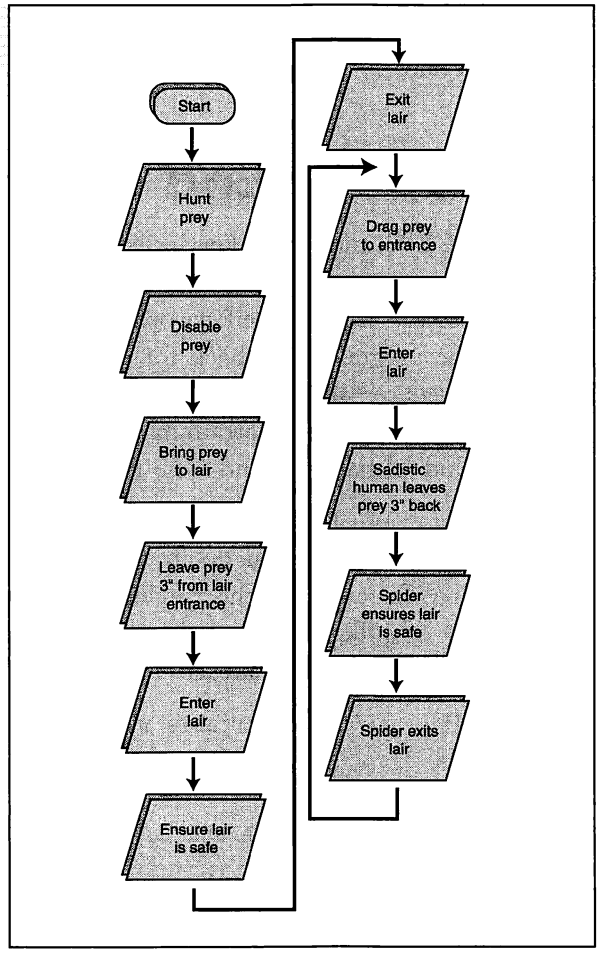



**Figure 6.3**
Because the spider can't adapt to the change, it will go back through the flow over and over until it starves.

the spider simply can't adapt its programming to handle this new situation. There is no variable, condition, or flow control to handle this possibility.

Is that cool or what? Nature and evolution have conspired to create in this unfortunate creature a program that provides endless hours of enjoyment for demented little boys and scientists everywhere. It's like the very first video game, nature's own version of *Doom*. Nature is a bad programmer. It takes thousands of generations to effect changes in its programs and remove bugs from its code (Bugs! Get it?). Even Microsoft works faster than this.

A good developer can't afford to wait for his programs to drop dead to correct flow-control problems. A good developer needs to be prepared for all possible conditions (except for devious research scientists and cruel little boys) and their programs should handle them appropriately. Flow control is a necessity; complete and comprehensive flow control is what differentiates the good programs from the bad ones.

# The If/Then/Else If/Else/End If Keywords

The If, Then, Else If, Else, and End If keywords are used to inspect a variable's value and respond by executing specific source code depending on that value. The simplest form of the use of these keywords is the If/Then/End If usage. Using this form, the program can check for some specific condition—for example, a variable's value—and perform a specific task. The If portion is the condition being checked; the Then portion marks the beginning of the task to perform; and the End If marks its end. The flowchart shown in Figure 6.4 shows an example.

This is no different from what you do in a normal day-to-day decision-making process using, for example, the English-language conditional statement, "If you are out of milk, then go to the store and buy more milk." This process could be written in pseudo code as

```
Dim bWeAreOutOfMilk     // Some variable which contains "true" if ➥
you are out of milk
If (bWeAreOutOfMilk) Then
  GoBuyMilk     // Go to the store and buy more milk
End if
```

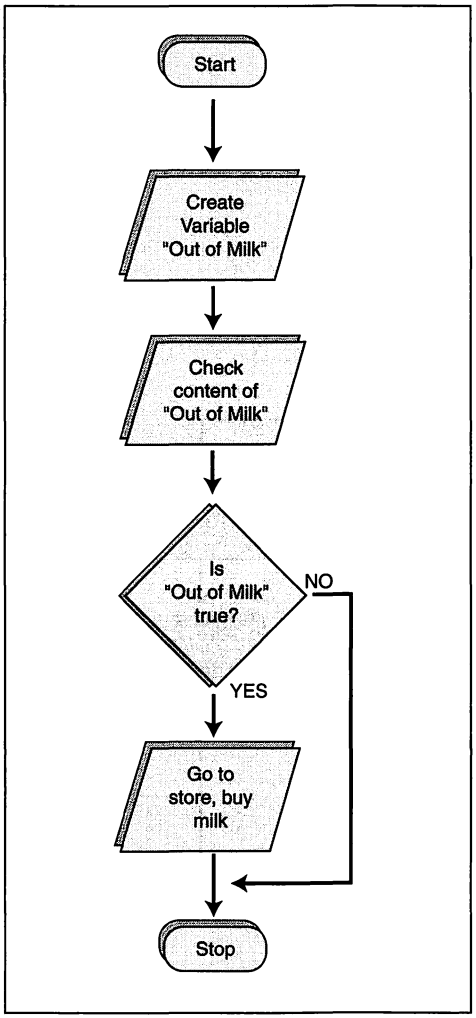



**Figure 6.4**
And you thought you just had to drive to the store.

The second form of If statements contains an Else statement. The Else statement is used in conjunction with the If/Then statement. It allows your program to perform a specific task when the condition in the If/Then statement is not true. The flowchart in Figure 6.5 contains an example of this statement in action.

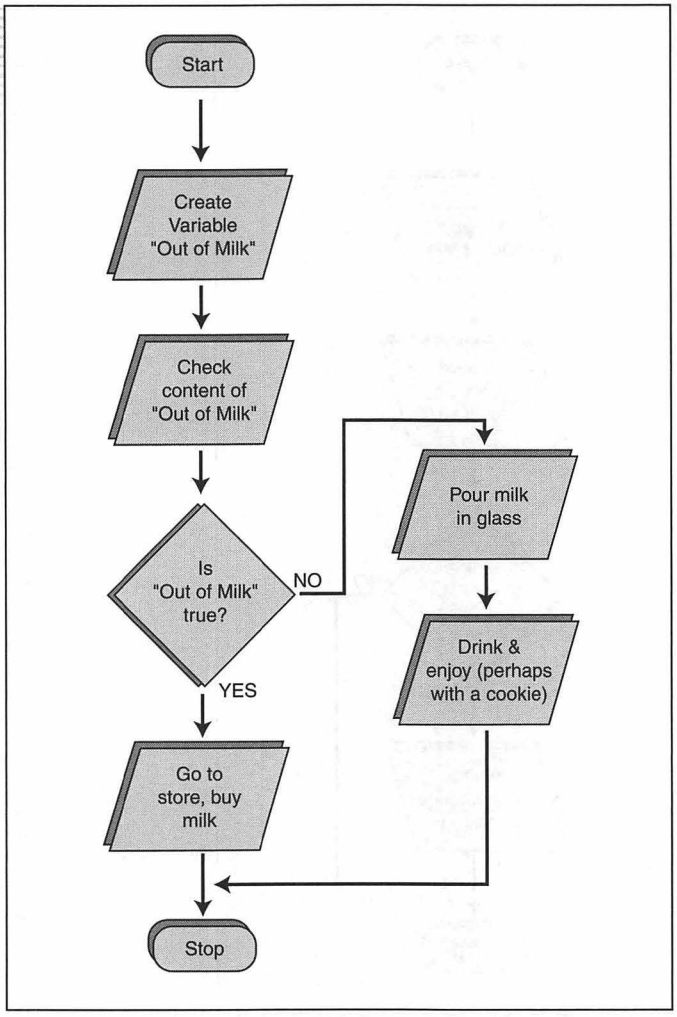



**Figure 6.5**
With an Else decision, you can make the flow more effective.

Again, no different from what you do in a normal day-to-day, albeit not life-altering, decision-making process. This process could be written in pseudo code as

```
If (bWeAreOutOfMilk) Then   // Check to see if you are out of milk
  GoBuyMilk                 // Go to the store and buy more milk
Else
  PourAndEnjoy              // Pour a glass of milk and enjoy
End If
```

One important thing to remember is that If/Then/End If blocks of code are self-contained and don't relate to other blocks. For example, the following code checks two separate conditions. The code for each condition will be executed regardless of the other condition.

```
If (SomeConditionIsTrue) Then
  // Perform some specific task
End If
If (SomeOtherConditionIsTrue) Then
  // Perform some OTHER specific task (regardless of the first ➥
  condition)
End If
```

If/Then/End If blocks of code can be *nested*, or inserted within each other. In the following example, the second block of code is nested, or embedded, within the first block of code, meaning that the second condition won't even be evaluated unless the first is true.

```
If (SomeConditionIsTrue) Then
  // Perform some specific task
  If (SomeOtherConditionIsTrue) Then
    // Perform some OTHER specific task (only if BOTH conditions ➥
    are true)
  End If
End If
```

In the preceding pseudo code, the first task will be performed only if the first condition is true. The second task will be performed only if both the first and second conditions are true.

Code blocks can be nested in the Else statements, as in the following pseudo code:

```
If (SomeConditionIsTrue) Then
  // Perform some specific task
Else
  // Perform some OTHER task
  If (SomeOtherConditionIsTrue) Then
    // Perform yet ANOTHER specific task (only if the first ➥
    condition is false and the second is true)
  End If
End If
```

In this example, the second task will be executed only if the first condition is false. The third task will be executed only if the first condition is false and the second is true. If the second task wasn't needed and excluded, the code would look something like this:

```
If (SomeConditionIsTrue) Then
  // Perform some specific task
Else
  If (SomeOtherConditionIsTrue) Then
    // Perform some OTHER task (only if the first condition is ➥
false and the second is true)
  End If
End If
```

The result of this is that the second task will only be performed if the first condition is false and the second is true.

The use of this form of nested code is so common that most programming languages have a special statement to handle them. Rather than nesting the second If/Then/End If block of code, you can use the Else If statement. Else If allows the developer to check multiple conditions within a single group. It's best if the conditions are related in some sense, but they need not be. Each condition is checked from top to bottom; if any particular condition is true, its code is executed and execution continues after the End If statement. For example:

```
If (SomeConditionIsTrue) Then
  // Perform some specific task (but none of the others)
Elseif (SomeOtherConditionIsTrue) Then
  // The first condition wasn't true, but the second is
  // Perform some other specific task (but none of the others)
Elseif (YetAnotherConditionIsTrue) Then
  // The first two conditions weren't true, but the third one is
  // Perform yet another specific task (but none of the others)
Else
    // None of the conditions above were true
    // Perform a fourth specific task (but none of the others)
End If
// Execution continues from this point after one of the tasks ➥
above has completed
```

The If/Else If/Else usage may seem pretty complicated, but it really isn't. Again, there are mundane examples of this type of decision making in your every day lives, as shown in the flowchart in Figure 6.6.

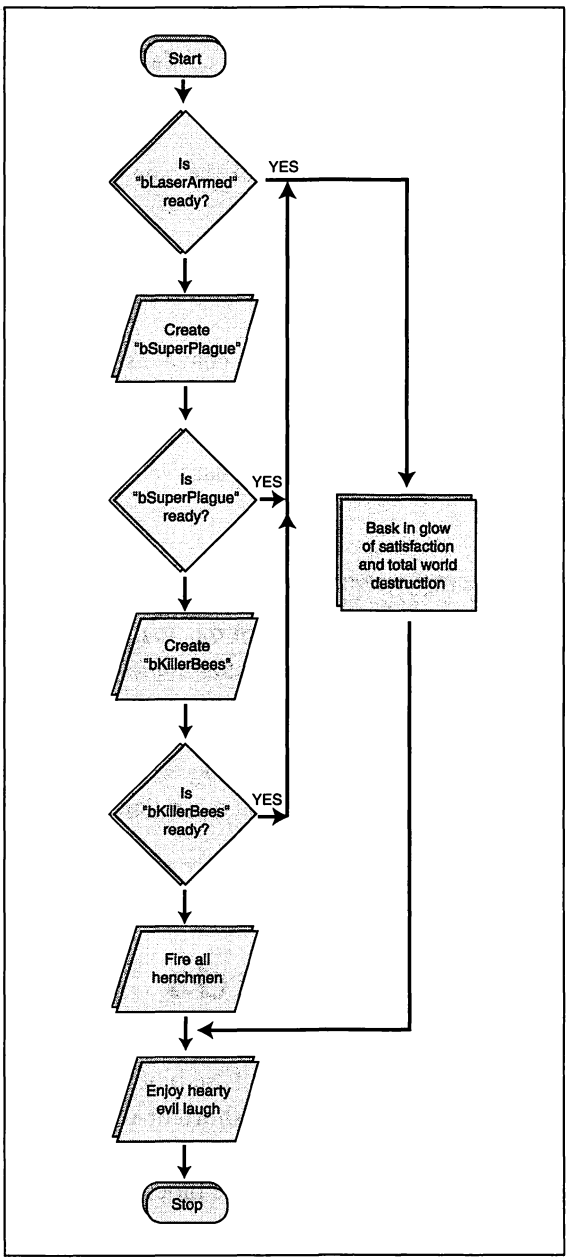



**Figure 6.6**
Planning your day as a villain might be more effective with proper checks to end the world or the jobs of your useless minions.

Okay, so maybe the flowchart in Figure 6.6 is not a mundane example, but you get the idea. The addition of Else If allows you to check multiple conditions and have your program behave accordingly in response to *one* of these conditions being true. Used in a REALbasic program the If, Then, Else If, Else and End If statements would look something like the following

```
Dim nValue As Integer    // nValue contains 0 for now
nValue = 12    // nValue now contains a value of twelve

If (nValue < 12) Then     // check to see if nValue is LESS THAN ➥
twelve
    msgBox ("The nValue variable contains a value less than twelve.")
Elseif (nValue > 12) Then     // check to see if nValue is GREATER ➥
THAN twelve
    msgBox ("The nValue variable contains a value greater than ➥
twelve.")
Else    // nValue is neither GREATER THAN or LESS THAN twelve
    msgBox ("The nValue variable must contain a value of twelve.")
End If
```

**TIP**

In the examples above you'll notice that we used parenthesis around the conditions being checked in the If statements. While this is not necessary in REALbasic, other programming languages, like C and C++, do require the use of parenthesis around conditions. If you're planning on picking up other programming languages, you might want to get in the habit of doing this. Besides, it does make your code a lot easier to read.

# The Select/Case Keywords

We just talked about using the Else If keyword instead of nesting a second block of conditional code in another Else block of code. Using Else If reduces your code complexity and provides a shortcut when you are writing the source code. The Select/Case keywords can also be used as a shortcut of sorts. If you are writing multiple If, Else If, and Else blocks of code, which are all checking

the contents of a single variable, you can use the Select/Case keywords instead. For example, in the following code, multiple Else If statements are used to check for multiple values for the nValue variable:

```
Dim nValue As Integer     // nValue contains 0 for now
nValue = 2                // nValue now contains a value of two
If (nValue = 1) Then
  msgBox ("The nValue variable contains a value of one.")
Elseif (nValue = 2) Then
  msgBox ("The nValue variable contains a value of two.")
Elseif (nValue = 3) Then
  msgBox ("The nValue variable contains a value of three.")
Else
  msgBox ("The nValue variable contains something other than 1, 2 ➥
  or 3.")
End If
```

This code can be simplified using Select/Case syntax, which enables you to compare the nValue variable, multiple times, to various values without all the redundant Else If code. The preceding code could be re-written as

```
Dim nValue As Integer     // nValue contains 0 for now
nValue = 2    // nValue now contains a value of two
Select Case (nValue)
Case 1
  msgBox ("The nValue variable contains a value of one.")
Case 2
  msgBox ("The nValue variable contains a value of two.")
Case 3
  msgBox ("The nValue variable contains a value of three.")
Else
  msgBox ("The nValue variable contains something other than 1, 2 ➥
  or 3.")
End Select
```

Not only does the code above look nicer, it will actually perform better than the multiple Else If version with the same functionality. The difference in performance is pretty small, but in a program with lots of these kinds of operations, it begins to add up.

One important thing to remember when using the Select/Case keywords is that they are limited to checking for integer and string equality. For example, there is no method for using a Select/Case statement to check whether the value of a variable is less than or greater than a specific value.

# The For/Next Keywords

In the previous two sections, we discussed program flow control that is based on conditional branching. A choice is made between two or more branches of code based on the condition of the variable being evaluated. Although these conditional branching types of flow control are all well and good, they don't provide the other major type of flow control, which is looping.

*Looping* is when a computer program performs the same operations over and over again. Now uncontrolled looping—when a program gets stuck in a loop and can't stop—is bad . . . very bad. Infinite loops can either crash an application or make it appear that the application has locked up. Flow control using looping, however, enables the developer to perform code in loops, without the risk of infinite loops.

The first type of looping is achieved with a For/Next statement. Using a For/Next statement, the developer can repeatedly perform a specific set of tasks, while the program increments or decrements the value of a counter variable. There are two forms of the For/Next statement, the first of which looks like this:

```
For Counter = StartValue To EndValue Step StepValue
  // Perform some specific task (maybe using the Counter variable ➥
  in the task)
Next
```

At the beginning of the first form of the For/Next loop, the `Counter` variable, which must be an integer, is assigned the initial value of the `StartValue` parameter. The task within the For/Next loop is executed at least once and the `Counter` variable is incremented by the value of the `StepValue` parameter. If the new value of the `Counter` variable is less than or equal to the value of the `EndValue` parameter, then the task within the For/Next loop is repeated. The task within the For/Next loop will continue to be executed until the value of the `Counter` variable is greater than the `EndValue` parameter.

The second form of the For/Next loop looks like this:

```
For Counter = StartValue Downto EndValue Step StepValue
  // Perform some specific task (maybe using the Counter variable ➥
  in the task)
Next
```

The second form of the For/Next loop works just as the first, except that the `Counter` variable decrements by the value of the `StepValue` parameter at the end of each loop rather than being incremented. The loop terminates when the value of the `Counter` variable is less than the value of the `EndValue` parameter.

It's not as difficult as it looks. Everyone performs tasks like this every day of their lives. Whenever you repeatedly perform the same task a certain number of times, you are in one sense executing a For/Next loop in your head. For example, say you have a dozen chocolate-chip cookies to go with that milk from the previous examples, and you've decided to sit down and eat them one at a time. Your flowchart would look like the one in Figure 6.7.

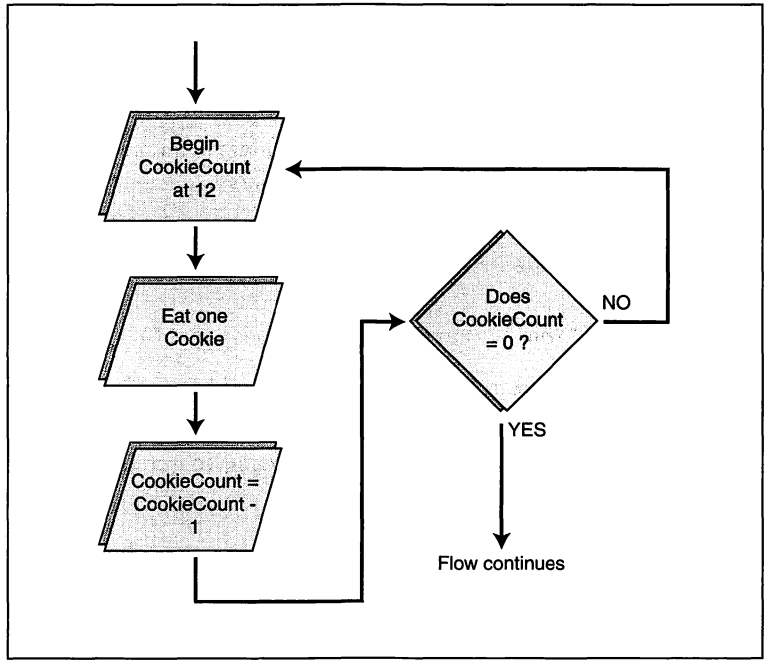



**Figure 6.7**
You can't eat just one, but you will for now.

See? Nothing to it. Like we said above, you don't need the Step parameter if you're incrementing by one, which is the default Step value. For example, this REALbasic code is just like the pseudo code above, but it leaves out the Step parameter:

```
Dim nValue As Integer
For nValue = 1 To 12 // note, no step specified (a step of one is ➥
assumed)
  msgBox ("Eat cookie number " + str(nValue) + "!")
Next
msgBox ("There is no cookie number " + str(nValue) + "!") // note ➥
that nValue is now 13
```

A good example of the second form of the For/Next loop would be

```
DIM nValue As Integer
For nValue = 5 Downto 1 // Note the use of down to
  msgBox (str(nValue))
Next
msgBox ("BLASTOFF!")
```

Like we said before, infinite loops are very, very bad, and you have to watch out for them. Doing something like what is done in the sample code that follows could be very bad for the users of your software:

```
Dim nCounter, nStepValue As Integer
nStepValue = 0    // Warning, something stupid is about to happen
For nCounter = 1 To 2 Step nStepValue
  msgBox ("This (" + str(nCounter) + ") will get very boring and ➥
very annoying very quickly!")
Next
msgBox ("You'll never see this message!")
```

Because you've told the program to add 0 to the initial value of 1 until it is greater than or equal to 2, you've created an infinite loop. No matter how many times you add 0 to 1, you're never going to get an answer of 2, so this loop will continue running forever.

Something else you need to watch out for is the ending value of a For/Next loop changing during the processing of the loop. This could create an infinite loop as well. For example, the following REALbasic code would appear to be

valid at first, but closer inspection of the code within the For/Next loop will show that it also is an infinite loop:

```
Dim nValue1 As integer
Dim nValue2 As integer
nValue2 = 2
For nValue1 = 1 To nValue2   // would appear to be counting from ➥
1 to 2
  msgBox ("Performing task " + str(nValue1) + " of " + ➥
str(nValue2))
  nValue2 = nValue2 + 1   // WHOA! Now you're counting to 3 not ➥
2, etc.
Next    // will be task "1 of 2", "2 of 3", "3 of 4", etc. , ➥
etc. , etc.
```

As you can see, For/Next loops give a developer many ways to handle loops in their application. However, they don't provide all the answers. Sometimes you want to loop based on something other than an incremented or decremented counter. There are types of looping to handle these situations as well, which you will learn about in the next chapter.

# Review

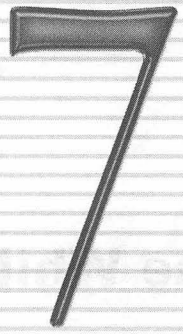
Flow control is so important in computer programming and software development. Without flow control, no conditional branching or repetitive task can be performed in computer software—at least not easily.

The If/Then/Else If/Else/End If keywords are used to create conditional branches in your source code, which will be executed if the condition you are checking for evaluates as true. A simple If/End If block allows you to include or exclude the execution of certain code when the specified conditions are met. If/Else blocks let you choose one of two paths through your code, based on the conditions being checked. Last, but definitely not least, the use of multiple Else If statements allows you to check multiple conditions and take the appropriate primrose path through the garden of your source code.

The Select/Case form of flow control can be used in place of If /Then/ Else If/Else/End If blocks of code if an integer variable is being compared to

specific values in all the conditions being tested. Using Select/Case blocks not only makes your code look nicer, it may actually improve the performance of your application. Plus, you will impress all your programmer friends with your knowledge of REALbasic programming.

The For/Next loop is best used when the developer needs his program to perform a specific task a fixed number of times, based on a counter that is either being incremented or decremented. Tasks that process a known quantity of items are a perfect fit for For/Next loops.

# And Still More on Program Flow

## In This Chapter

- The While/Wend keywords
- The Do/Until keywords
- The Goto and Exit keywords

There are even more ways to control the path of your code so your application responds properly to input. So, without further ado, let's pick up where we left off with flow control keywords.

# The While/Wend Keywords

The While/Wend loop is the first variant of non–counter based loops. Unlike For/Next loops, While/Wend loops don't initialize a variable to a specific value. Also, While/Wend loops don't automatically increment or decrement a counter variable the way For/Next loops do. The condition being checked in the While/Wend loop is completely under the control of the developer, unlike with the For/Next loop, which can only compare ending values to the counter values.

The form that While/Wend loops take is

```
While (SomeConditionIsTrue)
  // Perform Some Task
Wend
```

As you can see, the condition is checked first, before the task within the loop is performed. This too is different from the For/Next loop, which checks the condition after the task within the loop has been performed at least one time. The code in the While/Wend loop may not be performed even a single time if the condition is initially false.

You can also see that the While/Wend loop is a simpler form of looping than the For/Next loop. Though the While/Wend statement is less complex, you'll actually be doing more work coding *because* the While/Wend loop is a more basic form of loop. But don't despair; the While/Wend loop is not difficult to understand and use. It's similar to looping that you perform in everyday tasks. The following pseudo code shows how simple it is to understand these types of loops, with a particularly gluttonous example:

```
While (There Are Still Chips in the Bag)    // You can't eat just ➥
one!
  // Eat Another Potato Chip
Wend
```

Quite often, While/Wend loops are used to perform complex tasks similar to those performed by For/Next loops, but without relying on the automatic

incrementing of the For/Next loop. In the following code, you can see an example of a value that doesn't change on a regular basis like those in a For/Next loop:

```
Dim nValue1, nValue2 As Integer
nValue1 = 1
nValue2 = 3
While (nValue2 > 0) // do the following as long as nValue 2 is ➥
greater than 0
  nValue1 = nValue1 + 1
  If (nValue1 > 4) Then
    nValue2 = nValue2 - 1
    MsgBox ("nValue1 is " + str(nValue1) + " nValue2 is " + ➥
    str(nValue2))
  Else
    MsgBox ("Wait for it... nValue1 is just" + str(nValue1))
  End If
Wend
MsgBox ("All done. nValue1 is " + str(nValue1) + " nValue2 is " + ➥
str(nValue2))
```

As you can see, the logic in a While/Wend loop can be a lot more complex than the logic in a For/Next loop. In addition to the relative complexity, While/Wend loops give the developer much more control over the conditions being checked. The only thing the developer doesn't control is when the condition is checked; it's always checked at the beginning of the While/Wend loop. For more control of when the condition is checked, you'll have to use the Do/Until loop.

# The Do/Until Keywords

Unlike While/Wend loops, which run as long as the conditions being tested remain true, Do/Until loops are used to execute a specific set of tasks while the condition being tested remains false.

There are two major forms of Do loops. In the first major form, the condition is checked at the beginning of the Do loop. If the condition is true, then the entire block is skipped and execution of the program resumes after the Do loop's block of code:

```
Do Until (SomeConditionIsTrue)
    // Perform some task as long as SomeConditionIsFalse
Loop
```

A good psuedo-code example of the first form of Do loop would be

```
Do Until (There Are No More Chips in the Bag)    // You can't ➥
eat just one!
  // Eat Another Potato Chip
Loop
```

This is pretty much the same as the psuedo-code used in the While/Wend example, except for the condition being checked. In the While/Wend loop, you are eating chips "while there are still chips in the bag." You continue performing the action while the condition is true. In the Do/Until loop, you are eating chips "until there are no more chips in the bag." You continue to perform the action until the condition is true. To make it simple, just remember that Until is the opposite of While. In order to get the two loops to perform identically, you need to reverse your logic.

So the first major form of a Do/Until loop is just like a While/Wend loop with reversed logic. In the second major form of a Do/Until loop, the condition is checked at the *end* of the loop. If the condition being tested evaluates to false, the loop will run again. The result of this is that the tasks within the loop will be performed at least once regardless of the condition being tested—sort of like a For/Next loop. The second major form of the Do loop looks like this:

```
Do
    // Perform some task, at least once
Loop Until (SomeConditionIsTrue)    // Repeat if ➥
SomeConditionIsFalse
```

The only real difference between this form and the one preceding it is that in this form, the condition is checked at the end of the loop. This form of the Do/Until loop should be used only if you want to perform the task in the loop at least one time, regardless of the condition being checked. Use the first form of the Do/Until loop if you want to check the condition before performing the task within the loop.

There is a less-used form of Do/Until that uses an Until at both the beginning and end of the loop, allowing for both an initial conditional test and another conditional test at the end of the loop.

Here's an example:

```
Do Until (SomeConditionIsTrue)
     // Task to perform if SomeConditionIsFalse
Loop Until (SomeOtherConditionIsTrue)     // Repeat if ➥
SomeOtherConditionIsFalse
```

Although this might seem confusing, it's a powerful variant of the loop, enabling you to test for one condition at the beginning of the loop and a completely different condition at the end of the loop. A great day-to-day example of this would be

```
// A typical Sunday afternoon program
Do Until (The Entire Lawn Has Been Mowed)
  // Mow An Unmowed Strip Of The Lawn
Loop Until (The Football Game Has Started) // Mow the lawn until
the game starts
// Now would be a good time for those potato chip loops!
```

In this example, the "program" will first check whether the entire lawn has been mowed. If not, it will mow a strip of the lawn. After mowing one strip, the program will check whether the football game has started; if not, the loop will be processed again. The program alternates between checking the condition of "the entire lawn being mowed" at the beginning of the loop and checking the state of "the game starting" at the end of the loop. This ensures that at least one strip of the lawn gets mowed, while only missing at most a couple of minutes of the pre-game show.

Here's a REALbasic example of this type of Do/Until loop:

```
Dim nVal1, nVal2 As Integer
nVal1 = 1
nVal2 = 1
Do Until (nVal1 = 10)
  nVal1 = nVal1 + 1
  nVal2 = nVal2 + 2
  MsgBox ("nVal1 is " + str(nVal1) + " and nVal2 is " + str(nVal2))
Loop Until (nVal2 > 10)
```

Step through the preceding sample code in your mind, and try to figure out which condition will cause the loop to terminate. Then go ahead and add the

code to the Hello World application from Chapter 1, "Getting Acquainted with REALbasic," and see whether you were right.

Last, and least used, is another form of the Do loop, which doesn't use an Until condition at the beginning or at the end of the loop. The result of this is an intentional infinite loop:

```
Do
  // perform some task for ever and ever
Loop
```

Don't bother creating sample code for a loop like this; you'll see enough mistakes that look like this without doing it on purpose. As we said before, infinite loops such as this one can be very bad, but there are ways to terminate them—even though they're frowned upon. We'll discuss methods for breaking out of infinite loops in the next section.

# The Goto and Exit Keywords

As expressed in previous sections, infinite loops can create a huge problem. In most cases, good programming practices will help you avoid infinite loops. That said, it may sometimes seem impossible to terminate infinite loops. Developers often "code themselves into a corner," writing code in which getting out of a loop at the correct time is almost impossible. In these cases, REALbasic provides two statements to break out of loops (or, in the case of the second statement, to create an all-too-easy-to-abuse kind of loop).

## The Exit Statement

The Exit statement is used to terminate a loop prematurely by jumping to the code immediately following the loop. Upon execution of the Exit statement, the program will exit the loop and continue to the point after the Next in a For/Next loop, the Wend in a While/Wend loop, or the Loop in a Do/Until loop as if the loop had completed naturally.

In the following example, the Exit statement is used to terminate this poorly coded loop:

```
Dim nVal1, nVal2, nVal3 As Integer
nVal2 = 1
```

```
nVal3 = 2
For nVal1 = 1 To nVal3 // stupid loop
  nVal2 = nVal2 + 1
  If (nVal2 > 10000) Then // sanity check...
    Exit // break out of this loop after waiting too long
  End If
  nVal3 = nVal3 + 1 // the loop above is stupid because of this
Next
```

Of course, the use of Exit wouldn't be necessary if you wrote the code properly in the first place. For the most part, you should do your best to use the standard forms of looping and avoid the use of Exit completely.

**NOTE**

One valid use of Exit might be to artificially limit a loop during development and debugging, so that the developer can test for certain conditions without having to wait for them to occur naturally. This technique can also be used to cause a loop to perform fewer iterations than normal so that it can be tested with less difficulty.

# The Goto Statement

Three words best describe the Goto statement: Evil, Evil, EVIL! But seriously, like the Exit statement, the Goto statement allows for control to be passed to another location in the program. Unlike the Exit statement, however, which simply jumps to the end of the current loop, the programmer can determine what location to jump to with the Goto statement. The problem is that because the developer chooses where the program is going, it makes future maintenance very difficult. If you use a lot of Goto statements in your code, things can get very confusing very fast.

You specify the location in the program that the Goto statement should jump to with a label. In REALbasic, a *label* is simply a line of code that

- Ends with a colon
- Does not contain any spaces
- Contains only letters and numbers

- Starts with a letter, not a number
- Starts at the beginning of the line

Listed below are some examples of valid, and invalid, REALbasic labels:

```
// The following lines contain valid labels
ThisIsAValidLabel:
Label1:
YetAnotherLabel:

// The following lines contain invalid labels
1LabelWhichIsBad:    // Can't begin a label with a number
Another Bad Label:   // Labels can't contain spaces
WhatAmI!Thinking?:   // Labels can't contain punctuation
```

Quite simply, a REALbasic Goto statement takes the following form:

```
Goto SomeValidLabelName
```

They are so insidiously easy to use, Goto statements tempt even the best programmer. But don't be tempted by the Dark Side. Even though the path offered by Goto is quicker and easier, forever will you be tricked into using them. It's a slippery slope down which you do not want to tread.

As we said before, the Goto statement allows you, using a label as a parameter, to jump to any location in the code—backward or forward. When jumping backward in the code, you are usually instructing the program to repeat some part of the code. This is similar to the control that you get using any of the more proper forms of flow control via looping. That there are already other looping statements that can perform this type of loop, is one reason that the use of the Goto statement is considered a bad programming practice. Using a Goto is the lazy way out of thinking about proper loop structure. One very bad example of this usage of the Goto statement would be

```
Dim nVal As Integer
StupidIdea:
For nVal = 1 To 20
  If (nVal > 10) Then
    Goto StupidIdea
End If
Next
MsgBox ("You'll never get here")
```

As you can see, the developer chose to jump out of a For/Next loop using a Goto statement even though the location to which he is jumping is right before the loop in question. The developer has intentionally created an infinite loop. If the reason for doing this is valid, then the developer should have used one of the more accepted looping statements—a Do/Until loop, perhaps—as in this example:

```
Dim nVal As Integer
Dim bHellFrozenOver As Boolean
bHellFrozenOver = False
Do
  For nVal = 1 To 20
    If (nVal > 10) Then
      Exit
    End If
  Next
Loop Until (bHellFrozenOver)
MsgBox ("You'll never get here")
```

The point of this code is that the same effect is achieved: The loop never stops executing, but without the use of a Goto statement. In this code, you can see that we used an Exit statement to drop out of the For/Next loop. The use of the Exit statement is slightly more acceptable than the use of a Goto statement. Try to remember it this way: It is less rude to show someone the exit than it is to tell him where to go.

The following code is another, perhaps slightly better, example of the use of Goto statement:

```
Dim nVal As Integer
For nVal = 1 To 20
    If (nVal > 10) Then
     Goto PrematureExit
End If
Next
PrematureExit:
MsgBox ("Decided you didn't like the rest of the FOR loop?")
```

At least this code doesn't create an infinite loop, but again, the same effect could be achieved by using the Exit statement rather than the Goto, which would eliminate the need for the `PrematureExit` label.

Some programmers consider the use of Exit and Goto statements inappropriate and just plain lazy. Some consider the use of a Goto statement even worse, bordering on banal and downright evil. It is best to use other methods of flow control before resorting to Exit and Goto. Try to figure out a way to restructure your code, like we did in the "Frozen Over" example, before giving in to the temptation to use one of these less-structured methods. You should use Exit and Goto only as a last resort or, as we said before, during development and debugging to make these tasks easier. It's okay to use shortcuts like these in a work in progress; just remove them before anyone else has to look at your code.

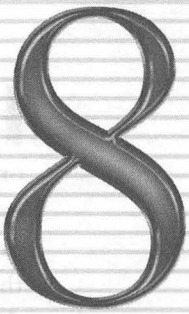
# Review

While/Wend loops are quite different from For/Next loops. They don't directly rely on counter variables, and the loop's condition is checked at the beginning of the loop. A While/Wend loop might not even execute at all if the condition being checked is initially false, unlike a For/Next loop, which always executes at least once. While/Wend loops give you much more control over looping in your source code.

Do/Until loops terminate when the condition being tested evaluates as false. The conditions of a Do/Until loop can be tested at the start of the loop, the end of the loop, both the start and end of the loop, or not at all. The use of Do/Until loops gives you even more control over the flow of your source code.

The use of Exit and Goto should be avoided at all costs . . . unless there's no way around it. The Exit keyword, which is slightly more acceptable than Goto, allows you to prematurely exit any of the looping forms of flow control. Exit is usually used to handle unexpected situations, or during development and debugging to make life easier for the beleaguered developer.

The use of the Goto statement is enough to get you killed in some places and is best avoided, if at all possible. Using Goto, you force your program to jump to a specific location in the source code, which is specified by a label. It's a very powerful way to get things done, but it can make reading your code troublesome at best. This increases the difficulty of maintenance and debugging for other developers, causing them to make statements that would imply that your parents weren't married and that you should go do physically impossible things to yourself. Take my word for it: Don't use Goto statements if you can at all avoid it. You'll live a lot longer and keep more of your friends as well.

# Subroutines, Functions, and Recursion

## In This Chapter

- What are subroutines and functions?
- Subroutine and function declarations
- Parameters and return values
- Recursion, recursion, recursion . . .

In the previous chapters, you learned about commands, variables, constants, and flow control. Although it would be possible to write a program using only these concepts, there are better ways. If you were to write a program of even moderate complexity from the top down, including all the steps that the program needs to perform, the looping would get very complex—so complex that you would eventually need to resort to Goto statements just to make things work (never a good idea). You might even have to duplicate code.

What you need is a way to reduce code complexity and remove the need for redundant code. Fortunately, just about every language, including REALbasic, has ways to do this: subroutines and functions.

# What Are Subroutines and Functions?

Subroutines and functions serve multiple purposes. They help simplify source code by grouping all the code for a specific task in one module. This makes the source code easier to read, because the code can be viewed in small, manageable chunks. For example, assume that you're a parent, and that you've given your children a set of instructions to perform each night before they go to bed. These bedtime instructions are only a small part of all the instructions that your children perform all day, but it makes sense to group them together, because they are related tasks. So, in pseudo code, you would create a subroutine for these tasks along these lines:

```
Sub GetReadyForBed (ChildName)
  Wash face
  Brush teeth
  Brush hair
  If (ChildName is Neil) then
    Clean braces
    Insert retainer
  End if
  Wash hands
  Change into pajamas
  Get into bed
  Go to sleep
End Sub
```

As you can see, all the bedtime tasks have been organized into a single `GetReadyForBed` subroutine. If the programmer wants to know which tasks apply to getting ready for bed, all he needs to do is look at the code in this subroutine. There's no need to go digging through hundreds of lines of code just to find the ones that apply to bedtime tasks. Remember, subroutines provide a nice organizational tool by allowing you to create smaller modules of source code with related tasks grouped together. These subroutines are simply portions of routines that are related in some way.

Subroutines and functions provide one other benefit: They allow for code reusability. In the `GetReadyForBed` example, you'll notice that `ChildName` follows the `GetReadyForBed` subroutine declaration. This is a parameter of the `GetReadyForBed` subroutine. We'll talk more about parameters later; for now, you just need to know that the program using this subroutine can specify for which child the subroutine is currently running. So, in our bedtime subroutine, we've allowed the code to be used by any number of children with a special exception for Neil, because he has braces and needs to perform tasks that other children wouldn't need to perform. We could have created a completely different bedtime subroutine for Neil, which includes all the tasks performed by the other children plus the tasks associated with dealing with his braces, but that would have created redundant code in our program. This way, we can share the code and use it for multiple purposes.

It can be assumed that the `GetReadyForBed` subroutine above is part of a larger program, which uses this subroutine to perform a specific set of tasks. When a program uses a subroutine or a function, it is said to be "executing a subroutine" or "making a function call." After the subroutine or function has finished performing its task, control is returned to the point in the program where the subroutine or function was called. So, our pseudo program would probably call the `GetReadyForBed` subroutine somewhere between a `DinnerTime` subroutine and a `HaveANightmareAndWakeUpYourParents` subroutine.

You may be wondering, what's the difference between subroutines and functions? They both seem to provide the same capabilities. Well, for the most part, you're correct. There is no difference—with one exception. Subroutines perform their tasks and return to the calling code without any communication back to the calling code. They provide no feedback as to what they did, whether they were successful, or whether any further actions should be taken.

This is like issuing a command to your program along the lines of "go do something and return back here when you're finished." Subroutines are used when the calling program either doesn't care what happens in the subroutine, or can determine this on its own.

Functions, on the other hand, perform their tasks and return a value to the calling code. Functions in computer programming are kind of like those x = y×2 functions in math. The y in the formula is a parameter; the y×2 is the task performed by the function; and the x is the return value of the function. Using a function in programming is like asking your program to perform some task that will determine the answer to some question, and return here, with the answer, when it's done.

So subroutines do their thing with no response to the calling program, while functions communicate the result of the tasks they've performed to the calling program. If the `GetReadyForBed` example was a function instead of a subroutine, it might look something like this:

```
Function GetReadyForBed (ChildName) as SomeVariableType
  Dim Result as SomeVariableType
  Wash face
  Brush teeth
  Brush hair
  If (ChildName is Neil) then
    Clean braces
    Insert retainer
  End if
  Wash hands
  Change into pajamas
  Get into bed
  If (Child is thirsty or wants some attention) then
    Result = Child asked for a glass of water
  Else
    Go to sleep
    Result = Child went to bed quietly
  End If
  Return Result
End Sub
```

In this example, the function can tell the calling program whether the child went quietly to bed, or is pulling the old "I want a glass of water" trick to stay awake a few more minutes. The calling program can then take the appropriate action based on the return value of the function—something like

```
Select Case (GetReadyForBed (ChildName))
Case Child asked for a glass of water
  // Grumble under your breath and be a good parent
Case Child went to bed quietly
  // Breathe a sigh of relief and enjoy a quiet night
Else
  // Something must be wrong, go find out what it is
End Select
```

So you see, functions and subroutines provide code reusability and make your code easier to read—and therefore easier to maintain. Both perform a specific set of tasks and return to the calling program when those tasks are complete. When control is returned to the calling program from a function call, the result of the function is returned to the calling program, allowing the calling program to behave accordingly.

# Subroutine and Function Declarations

In REALbasic, you create subroutines and functions by opening the File menu and selecting the New Method option. Regardless of what type of development is being done, and for that matter the language being used, subroutine and function declarations look about the same.

In REALbasic, subroutine and function declarations take the form of

```
Sub SubroutineName (OptionalParameter As VariableType, ...)
Function FunctionName (OptionalParameter As VariableType ...) As
ReturnVariableType
```

As you can see, subroutine and function declarations are nearly identical, other than the fact that functions are declared to be a certain variable type. This is required to allow the function to return a value like we talked about in the

previous examples. Here's an example of the declaration of a typical subroutine, the MsgBox subroutine:

```
Sub MsgBox (message As String)
```

"What's this?" you ask, "I thought MsgBox was a command!" Well, MsgBox *is* a command, but interestingly enough, most of the REALbasic commands are just predefined functions and subroutines written by the REALbasic developers for your use. Like a library of books, written by other authors for you to read and use for your own purposes, these prewritten functions and subroutines are grouped into libraries for your use. Not only can you use the standard library of built-in REALbasic subroutines and functions; you can download other libraries, written by other REALbasic developers, for your use.

# Check out the Bodies on These Subroutines

The code that comprises a subroutine or function is referred to as its *body*. A subroutine and function body contains variable declarations, which must precede any other code, code comments, and the statements, commands, flow control, subroutine, and function calls that perform all the tasks for which the function or subroutine is responsible. About the only thing you can't include in the body of a function or subroutine is another subroutine or function declaration. The end of a function body is marked with an End Function statement. The end of a subroutine body is marked with an End Sub statement. The following two examples represent valid subroutine and function bodies:

```
Sub AddTwoNumbers (nValue1 As Integer, nValue2 As Integer)
    Dim nTheResult As Integer     // Variable declaration(s) must
    Dim strResult As String       // come before other source code

    // Specific tasks to perform
    nTheResult = nValue1 + nValue2
    strResult = Str(nTheResult)

    // Display the result
```

```
    msgBox "And the answer is - " + strResult
End Sub

Function AddStringValues (strText1 as String, strText2 As String) ➥
As String
    Dim nCalculatedValue As Integer     // Variable declaration(s) ➥
must
    Dim strReturnValue As String        // come before other ➥
source code

    // Get the value of each string and add the values together
    nCalculatedValue = Val(strText1) + Val(strText2)

    // Convert the value back to a string
    strReturnValue = Str(nCalculatedValue)

    // Value returned must be the same variable type as the ➥
    function
    Return strReturnValue
End Function
```

# Parameters and Return Values

With subroutine and function declarations, you specify zero or more parameters. A *parameter* is simply a variable declaration representing a value that is passed to the subroutine or function. A comma separates each parameter declaration. When you call your subroutines or functions, you specify either constants or variables to be passed as their parameters. For example, to call the `AddTwoNumbers` subroutine, above you might use code like the following:

```
Dim nValue As Integer
nValue = 1
AddTwoNumbers(nValue, 2)
```

This code declares a variable of type integer called `nValue`, stores a value of `1` in this variable, and calls the `AddTwoNumbers` subroutine, which should display a message box containing the result.

**CAUTION**

When declaring a function, you must specify a return-value variable type for the function. REALbasic doesn't support void functions (that is, functions that don't return a value). In REALbasic, functions that don't return a value are subroutines. There's no way around it. Functions return values, subroutines don't.

When you call a function, not only do you specify the parameters that are passed to the function, you also need to be able to accept the return value by either storing it in another variable or by passing it to another subroutine, function, or REALbasic command. For example, to call the `AddStringValues` function above, you might do something like this:

```
Dim strValue As String
strValue = "1234"
MsgBox AddStringValues("4321", strValue)
```

This code starts by declaring a string type variable, `strValue`, and then assigns `"1234"` to this variable. The code then calls the `AddTwoNumbers` function, which converts the string parameters to numbers, adds them together, and returns the sum as a string. The return value from the `AddStringValues` function, like we said, a string, is passed to the `MsgBox` subroutine, which displays it on the screen.

# Recursion, Recursion, Recursion . . .

One thing you have to watch out for when using functions or subroutines is recursion. *The Smart-Alecky Programmer's Dictionary* definition of recursion is "recursion *noun*. See recursion". Okay, seriously, recursion is a very bad thing that occurs when a subroutine or function calls itself or calls another subroutine or function that in turn calls the first subroutine or function. Doing this can cause your program to enter an infinite loop.

In most cases, you want to avoid recursion like the plague. Uncontrolled recursion can cause your programs to go off to Never-Never Land while they perform the same operations over and over again. Various unpredictable things can happen when recursion occurs: your program might appear to be locked

up, your program could crash, or in extreme cases, your computer could crash (not very likely, but it happens). If you're the daring sort, try using the following function in a REALbasic program sometime:

```
Function VeryBadIdea (nValue As Integer) As Integer
    Dim nReturnValue As Integer
    nReturnValue = VeryBadIdea(nValue + 1)
    return nReturnValue
End Function
```

Better yet, don't use this code. We'll just explain what will happen:

1. When the function is called, it declares an integer variable named `nReturnValue`.
2. The function then attempts to calculate a new value for `nReturnValue` by calling itself with a parameter of `nValue + 1`.
3. The function again declares an integer variable named `nReturnValue` and attempts to calculate a new value for `nReturnValue` by calling itself with a parameter of `nValue + 1`.
4. The function again declares an integer variable named `nReturnValue` and attempts to calculate a new value for `nReturnValue` by calling itself with a parameter of `nValue + 1`.
5. The function again declares an integer—you get the idea. This happens again and again until something bad happens. If you're lucky REALbasic will generate an Unhandled Stack Overflow Exception error and terminate your application, but as we said before, less-pleasant things could happen as well.

Still not convinced that recursion is all that bad? Think back to the `GetReadyForBed` function example. Imagine what would happen if you put a call to `GetReadyForBed` right in the middle of the `GetReadyForBed` function. You would end up causing your poor children to be forever washing their hands and brushing their teeth without ever actually getting to sleep. Not something a good parent or programmer would ever want to do.

On the other hand, you should be aware that recursion isn't always nasty and evil. In the hands of an advanced programmer, recursion is a very powerful tool. Some really cool things, beyond the scope of this book, can be achieved with recursive programming. But like any powerful tool, its use is best left to those that completely understand it.

## All Things Stacked

When REALbasic generates a Stack Overflow Exception, it's telling you that you have written code that has caused the stack to overflow. But just what is a stack, and why is it so bad when it overflows?

Just about every programming language that supports the idea of functions and subroutines uses stacks to pass the parameters to the functions and subroutines. Simply put, the *stack* is an area of memory that has been set aside to pass variables to and from functions and subroutines.

Suppose you have three subroutines in a program, with the following declarations:

```
Sub SomeSub1 (nValue1 As Integer)
    SomeSub2(nValue1 * 2)
End Sub
Sub SomeSub2 (nValue2 As Integer)
   SomeSub3(nValue2 + 4)
End Sub
Sub SomeSub3 (nValue3 As Integer)
    msgBox("The value is: " + Str(nValue3)
End Sub
```

Here's how the stack works. Suppose you call `SomeSub1` with a parameter of 8. Although the program is in the body of `SomeSub1`, the stack contains a value of 8. As soon as the program calls `SomeSub2`, the stack will contain values of 16 and 8. As soon as `SomeSub3` is called, the stack will contain values of 20, 16 and 8. It's called a *stack* because as each function is called, the parameters for the currently executing function are "stacked" on top of the previous parameters. As parameters are added to the stack, they are said to be "pushed" onto the stack.

To complete the example above when SomeSub3 has finished its tasks and returns control to SomeSub2, the stack will once again contain values of 16 and 8. After control returns to SomeSub1, the stack will contain 8. After control returns from SomeSub1 to where you called the function, the stack will again be empty. When parameters are removed from the stack, they are said to be "popped" from the stack.

Because the stack is located in memory, it is by definition of limited size. If you call enough functions or subroutines from within other functions or subroutines, you can cause the stack to fill up and overflow. At that point, your program can no longer operate properly because it has run out of stack space.

Recursively calling a function or subroutine can have the same effect, because values will keep being pushed onto the stack without ever being popped off of it.

# Review

Functions and subroutines provide two valuable features for programmers. They allow you to organize related tasks into small chunks of manageable code, simplifying development and debugging. More importantly, functions and subroutines also allow for code reusability. Code that is used repeatedly throughout an application can be moved to functions and subroutines, reducing the redundant source code, which makes for a smaller overall application.

Functions and subroutines share similar definitions. Both allow for declarations of parameters, which are variables, which contain the values that are passed to the function or subroutine. Functions, unlike subroutines, can return a value to the calling code. A function's return type can be any of the valid REALbasic variable types.

The bodies of functions and subroutines are where all the work is performed. You can do just about anything in a subroutine or function body that you can do in any other source code, except declare another function or subroutine. You should also avoid functions and subroutines that call themselves. Doing so is known as *recursion*.

For the most part, you want to avoid recursion like six-week-old leftovers in the back of your refrigerator. Unless you have a very good reason to use recursion, be on the lookout for it and try to avoid it in everything you write. Some pretty advanced programming algorithms (just a fancy word for formulas), however, actually rely on recursion to work. Data encryption and sorting algorithms are all examples of advanced programming techniques that use recursion to achieve their goals. Although these advanced methods are beyond the scope of this book, you should keep them in mind as potential valid uses of recursion. Just remember that uncontrolled recursion is bad, but recursion, when used wisely, can help you perform some pretty nifty tricks.


# Object-Oriented Programming

## In This Chapter

- Understanding classes and objects
- Two halves of an object: properties and methods
- Encapsulation, inheritance, and Polymorphism
- Events and handlers in REALbasic

The last chapter talked about how subroutines and functions can be used to simplify code and reduce redundant code. Subroutines and functions, however, are just the tip of the structured-programming iceberg. To improve upon the concepts of functions and subroutines, developers rely on object-oriented programming.

A complete tutorial of object-oriented programming is beyond the scope of this book, but by the end of this chapter, you should at least be familiar enough with its concepts to effectively use them within your REALbasic applications. This is, after all, meant to be a beginner's guide to programming, and some object-oriented programming concepts can get pretty advanced.

# Understanding Classes and Objects

In object-oriented programming variables, subroutines and functions are grouped into related sections of code referred to as *objects*. It's a simple concept; but with it, many powerful things can be done.

For example, think of a Swiss Army knife, which is actually the equivalent of many tools—a corkscrew, a knife (or knives), a screwdriver, tweezers, a toothpick, and so on—combined in one simple, easy-to-use package. Not only does grouping these tools make them more compact, it also makes them easier to keep track of. The objects of object-oriented programming are sort of like the Swiss Army knives of the programming world: they group tools into a single package.

In other languages, you actually define the class definition in source code. For example, the following is an example of a simple class definition in C++:

```
class Employee {
public:
  CString strName;     // Employee name property stored in a string
  CString strAddress;       // Employee's home address property
  double dSalary;           // Annual salary property
  CDate dateHired;          // Hire date property
  int nVacationDays;        // Total vacation property
  int nAvailableVacation;   // Vacation days available for use ➥
property
```

```
    RaisePay (int nPercentIncrease);  // Method to raise pay by ➥
whole percentages
};
```

In REALbasic, you don't have to go through the process of defining classes like this. You simply open the File menu and select New Class to create a new class. When you do so, the new class is added to the Project window and the properties of the class are displayed in the Properties window, as shown in Figure 9.1. The Name property is obviously the name of the class. (We'll talk later about the Super property and how you add properties and methods to your new classes later in this chapter.)

# The Terminology

Before we get too far into a discussion of object-oriented programming, let's talk a bit about terminology. In discussions of object-oriented programming, you will hear objects referred to in two different ways: as classes and as instances. The source code and properties that define an object's behavior is referred to as a *class*. It's just the definition of an object. A class by itself is useless unless you do something with it.

An *instance* is the object as it exists for use in your application. When you want to use a class in your application, you create an instance of that object. Creating an instance of an object is a two-part process. First, you define a variable of the class type. This creates a *pointer* to an instance of the specified object. So far you've encountered variable types that are each used to hold a value of a particular type, such as an Integer that holds an integral (whole) number and a String that holds a number of characters. Now you meet a very different

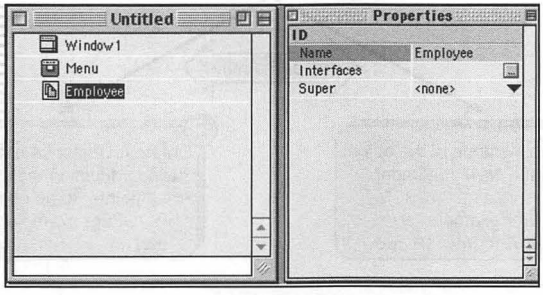



**Figure 9.1**
Creating a new class definition in REALbasic.

variable type—the pointer. A pointer is a type of variable that doesn't hold a value *per se*, such as an integer or a string, but rather holds the address of a memory location. That is, rather than actually holding a value, a variable that's a pointer is used to tell the program where to look in memory for a value. Granted, that sounds like tricky stuff, and it is, but rest assured it's an important way of handling some programming tasks. By default, this pointer contains a value of NIL, which means that the pointer doesn't point to anything at all:

```
DIM oSomeObject AS SomeKindOfClass // oSomeObject contains a nil ➥
pointer to a SomeKindOfClass object
```

To create the actual instance of the object, you have to do the second part, which usually involves allocating the memory for the instance of the object. This is done using the new statement:

```
oSomeObject = new SomeKindOfClass // allocate the memory for this ➥
object
```

What actually happens in this code example is that an amount of memory large enough to contain a `SomeKindOfClass` instance is allocated and the `oSomeObject` variable is assigned a pointer to the allocated memory. From this point on in your code, the `oSomeObject` variable is an instance of a `SomeKindOfClass` object. This process is illustrated in Figure 9.2.

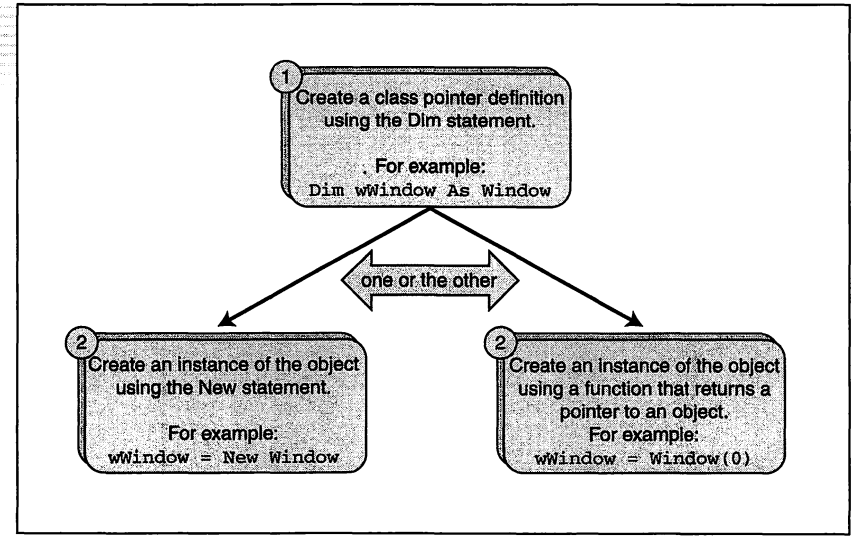



**Figure 9.2**
The REALbasic object-instantiation process

# Properties and Methods: The Two Halves of an Object

When designing a class, the developer must keep two things in mind: state and functionality.

First and foremost, classes maintain their own state. That is, a class contains all the information about itself. The state of a class is maintained in variables, referred to as *properties*. For example, an object of the class `Employee` might contain properties such as `Salary`, `Hire Date`, `Vacation Earned`, and so on. These class properties are defined just like any other variables. Because these variables are members of a class, they are sometimes referred to as *member variables*.

In REALbasic, you add properties to a class by opening the File menu and selecting the New Property item. Figures 9.3, 9.4, and 9.5 show a few examples of properties being added to a class in REALbasic.

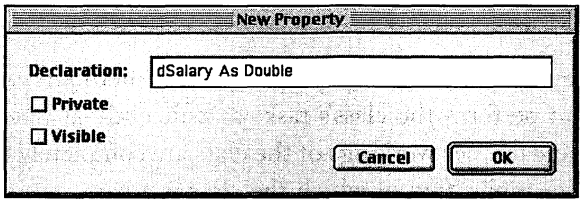



**Figure 9.3**
Creating a `Salary` property in REALbasic.

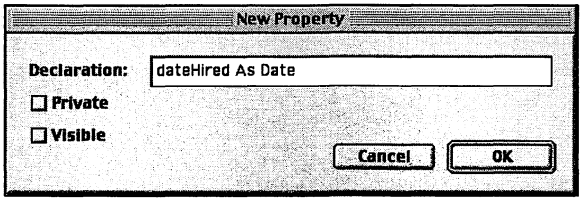



**Figure 9.4**
Creating a `Hire Date` property in REALbasic.

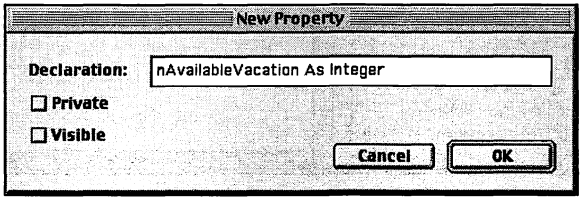



**Figure 9.5**
Creating an `Earned Vacation` property in REALbasic.

After you've added properties to a REALbasic class, you can view them in the class edit window. To view the class edit window, you can either double-click the class name, in the REALbasic project window, or select the class name and press Option+Tab. To view the properties in a class, you can either double-click the Properties item in the class edit window or click the disclosure triangle of the Properties item, as shown in Figure 9.6.

The benefit of class properties is that all the code that has to deal with these properties is contained within the class as well. A class can contain functions and subroutines, referred to as *methods*, which are used to perform specific tasks related to the class in question. The nice thing about this is that only the class needs to know anything about its properties. For example, suppose the `Employee` class discussed previously has a `RaisePay` method, which allows the calling code to raise an employee's pay by a specified percent. The `Salary` property could be defined as an integer, a double, or a string, and the `RaisePay` method could handle changing the employee's salary. The calling code doesn't have to know, or even care, about the type of variable that `Salary` is, and yet it's instructions are carried out as intended.

# Encapsulation

Classes, by definition, group properties and methods so that they can work together to perform the class's tasks as efficiently as possible. As a matter of fact, the actual inner workings of the class are completely unknown to all other parts of the application of which the class is a part.

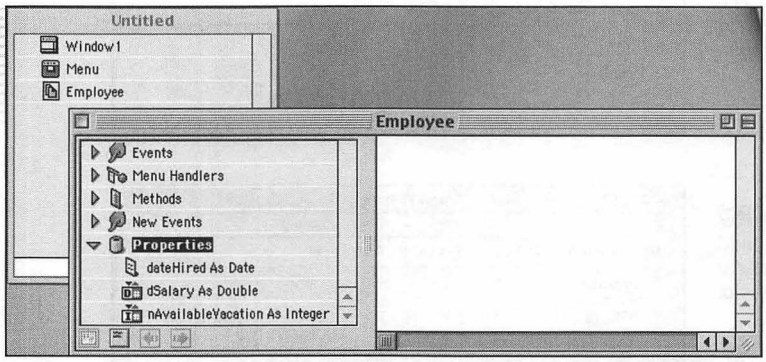



**Figure 9.6**
The properties of the `Employee` class

Think of it this way: When you put a key in your car's ignition and turn it to start the car, lots of complicated things happen, but you don't really care about the details. You just want to see the expected result of your action: The car starts. This is the same type of thing that can occur within a class in object-oriented programming. Some portion of the application asks a class to perform a task. It does so without boring the other parts of the application with the tedious details. It also will perform these tasks without any additional intervention. It won't even ask the other parts of the application to keep track of the properties associated with its actions.

The grouping of properties and methods into a package that hides its inner workings from other parts of an application is referred to as *encapsulation*. As we said before, the definition of the classes, groups, properties, and methods so that they can function in concert. Encapsulation is a very important concept in object-oriented programming.

# Inheritance

Another powerful feature of object-oriented programming is that a class definition can be based on other class definitions. As classes are derived from other classes, they are said to *inherit* the properties of the parent class. Think of inheritance in a family tree: You inherited traits from you father, your father from your grandfather, and so on. The same is true in object-oriented inheritance.

For example, assume you created the following class (we'll use an English-language variation of the C++ class definition here, just to make things easier to understand):

```
class Grandfather {
public:
    GoFishing ();
    PlayPiano ();
private:
    MakeWoodCarvings ();
}
```

In this class definition, two of the tasks that the class can perform, `GoFishing` and `PlayPiano`, are assumed to be skills that will be passed on to descendents of this class. As such, they are marked as public. The other task,

MakeWoodCarvings, will not be passed on to descendent classes, so they are marked as private. This means that if you create a new class based on this class, it will inherit the capability to GoFishing and to PlayPiano, but not the capability to MakeWoodCarvings.

In the following example, the new class has the capability to perform the first two tasks, GoFishing and PlayPiano, by default. It inherits these tasks simply because the parent class has these capabilities. It can't, however, MakeWoodCarvings because the parent class labeled this task as private—it never shared this capability with its descendent class (after all, we all need to keep some things to ourselves). The new class also has the capability to perform a new task, CookChili.

```
class Father : Grandfather{
public:
    CookChili ();
}
```

As we said, this class can GoFishing, PlayPiano and CookChili, but not MakeWoodCarvings. You could then define another class, based on the first two, using the following definition:

```
class Son : Father{
public:
    BuildTreehouse ();
    PlayDrums ();
private:
    PickOnSiblings ();
}
```

So, this class would inherit the capability to CookChili, GoFishing, and PlayPiano but not the capability to MakeWoodCarvings (ah, the lost talents of days gone by). This class also adds new capabilities, specifically the capability to BuildTreehouse, PlayDrums, and PickOnSiblings. The capability to perform the first two tasks would, by default, be passed on to any descendents of this class, but the last (very bad behavior indeed) would not be passed on. Figure 9.7 demonstrates the process of inheritance.

When it comes to inheritance, object-oriented classes mimic the passing down of information from one generation of a family to another. Each class inherits

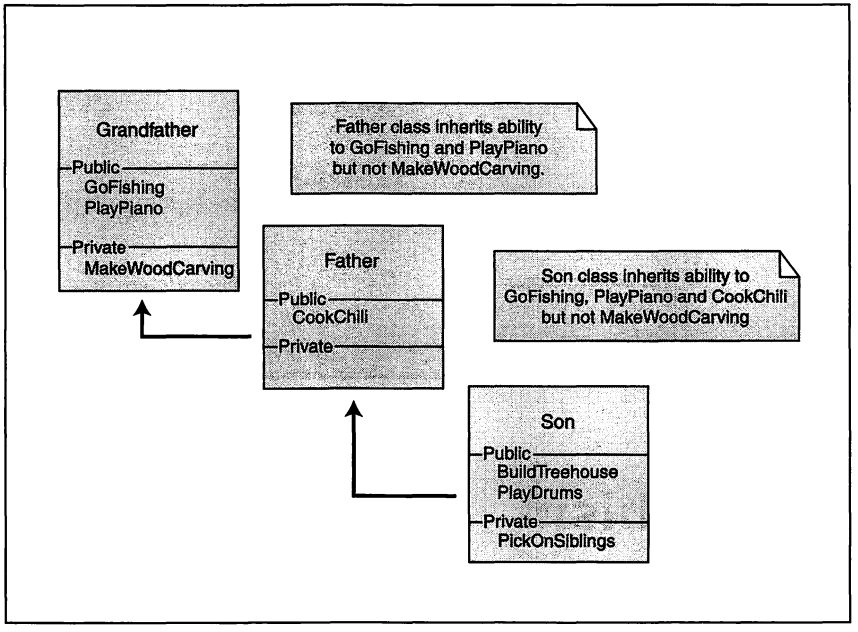



**Figure 9.7**
Inheritance of tasks

the public methods and properties of the class from which it is derived. Because of the resemblance to family trees, the classes from which these descendent classes are derived are often referred to as *parent classes*, and the descendent classes are referred to as *child classes*.

Keep in mind that just as with family trees, some things aren't passed from one generation to the next. Classes that don't wish to share their functionality with their children mark these items as private.

# Inheritance in REALbasic

When you create a new class definition in REALbasic and want to derive your class from another class, all you need to do is specify a class to be the super class of the current class, as shown in Figure 9.8. As classes are derived from previous classes, they tend to get smaller and smaller and do more specific things. Keeping that in mind may help you to remember that the parent of your class is its super class—it's superior to your derived class.

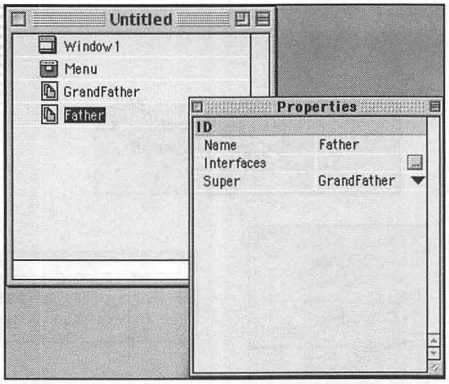



**Figure 9.8**
Creating a derived class in REALbasic.

# Polymorphism

Probably the most powerful capability of object-oriented programming is known as *polymorphism*. Literally meaning *many forms*, polymorphism is the capability to have multiple classes, all derived from the same super class, which can all be used in place of the super class. For example, suppose you had a super class called `Rugrat` from which you derived a `son` and `daughter` class. Anywhere in your application where a `Rugrat` class pointer is required, you can use a `son` or `daughter` class pointer instead.

Additionally, the derived classes can replace the default behavior of the super class's method by overriding the method in each derived class. For example, in the Rugrat class mentioned previously, you define the default behavior of the `CleanYourRoom` method. In all the classes derived from the `Rugrat` class, you would define behavior specific to that class. Your program could use any of the Rugrat-derived classes (thinking that they were actually `Rugrat` classes) and the proper actions of that derived class will occur, instead of the base class. This process is shown in Figure 9.9.

So the `Rugrat` class contains one set of instructions for the `CleanYourRoom` behavior, while the `son` and `daughter` classes contain their own unique instructions for cleaning their rooms. One of the benefits of polymorphism is that your application can call members of these classes without really knowing what the type of class is. Say you have some code, somewhere in your program, that defines a `son` or `daughter` object and calls a subroutine, passing either of those objects as a parameter:

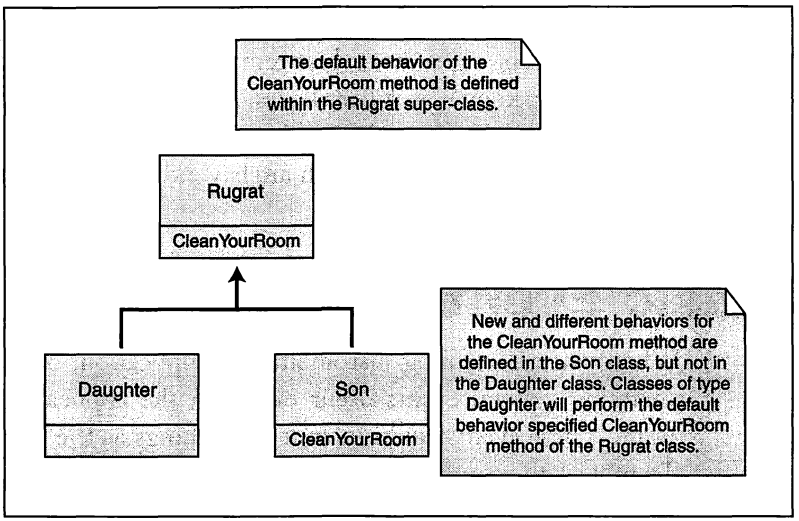



**Figure 9.9**
Two derived classes in which the behavior of one of the super class's methods has been overridden.

```
Dim chWhichKid As Rugrat
If (bTodayIsMonday) Then
  chWhichKid = new Son
Else
  chWhichKid = new Daughter
End If
MakeChildCleanTheirRoom (chWhichKid)
```

The subroutine being called, `MakeChildCleanTheirRoom`, would have the following definition:

```
Sub MakeChildCleanTheirRoom (Rugrat chOneOfTheKids)
  ChOneOfTheKids.CleanYourRoom
End Sub
```

The subroutine accepts a pointer to a `Rugrat` class as a parameter. Even though you passed a son or daughter class to the subroutine, the code still works because the `son` and `daughter` classes are derived from the `Rugrat` class. As you saw in the preceding class definitions, the `daughter` class didn't override the `CleanYourRoom` method. So, if a `daughter` class is passed to the `MakeChildCleanTheirRoom` subroutine, the `Rugrat` class's `CleanYourRoom` method will be called. If a `son` class is passed to the subroutine, then the `son` class's `CleanYourRoom` method will be called.

Polymorphism can be difficult to grasp, so don't sweat it too much if you don't get the concept right away. Just remember that classes can be derived from other classes, known as super classes, and that the derived classes can override the methods of the super classes. Anything past that starts to get into advanced programming techniques, which are beyond the scope of this book.

# REALbasic Events and Handlers

When you are running just about any application, on almost any computer operating system, events occur of which the application needs to be made aware. These events could include such things as the user moving or clicking the mouse, the computer clock ticking off another second, or any other countless numbers of other things that are happening. In the good old days before object-oriented programming, developers had to write huge chunks of code just to see when specific events occurred and act accordingly:

```
If (bTheUserClickedTheMouseButton) Then
  // Do whatever we need to do when the user clicks the mouse ➥
  button
Else If (bTheUserMovedTheMouse) Then
  // Do whatever we need to do when the user moves the mouse
Else If (bTheUserMovedTheMouseWhileTheButtonWasDown) Then
    // Do whatever we need to do when the user click drags the ➥
    mouse
Else If (bTheTimerClickedOffAnotherSecond) Then
  // Do whatever we need to do when the timer clicked off another ➥
  second
Else If (…)
// …etc, etc, etc
End If
```

Handling events in this way is sort of like picking up the phone every five seconds, just to see if someone has called you. It would be much better to have the equivalent of the bell on the phone so that you know when to pick it up, rather than constantly checking. Well, you do. The phone is your application and the bell is the event handler.

Event handlers are pre-existing methods of a class, which, by default, don't contain any code. Depending on the type of class, there are event handlers for

mouse clicks, timer ticks, and all kinds of other things to which your applications must be able to respond to.

Let's go back to the classes we were talking about before, the `Rugrat` classes, and add an `Adult` class that works with the `Rugrat` class. One of the tasks that the `Adult` class must perform is `ComfortInjuredRugrat`. In a traditional application, the `Adult` class would probably have to keep checking the `Rugrat` class to see if it had been injured:

```
Sub CheckRugratForInjury (chSomeChild As Rugrat)
    If (chSomeChild.bIsInjured) Then
        // Do whatever we need to do to comfort the injured Rugrat
    End If
End Sub
```

This would be an inefficient and tedious procedure. The `Adult` class would regularly be asking the `Rugrat` class if it had been injured: "Are you hurt? Are you hurt yet? How 'bout now? Still not hurt?" With events, the `Adult` can respond to the `Rugrat` class only when it needs attention. The operating system could send a `RugratInjured` event to the `Adult` class. All that would need to be done would be to add the code to handle this event. Not so coincidentally, the subroutine that handles an event is referred to as an *event handler*:

```
Sub RugratInjured (chSomeChild As Rugrat)
    // Do whatever we need to do to comfort the injured Rugrat
End Sub
```

So, as you can see, not only does using event handlers remove a lot of useless wasteful checking to determine whether an event has occurred, it also simplifies the code because all the code that checks for the events is not needed.

To add an event handler in REALbasic, as shown in Figure 9.10, all you need to do is

1. Select a class in the Project window.
2. Open the class editor for the class in question by pressing Option+Tab.
3. Double-click on the Events item, or click the Events disclosure icon.
4. Click the event to which you'd like the class to respond.
5. Add the appropriate code for the event in the Code Editor window.

Adding event handlers is something you'll be doing a heck of a lot of. Most of your time programming in REALbasic, as well as other object-oriented

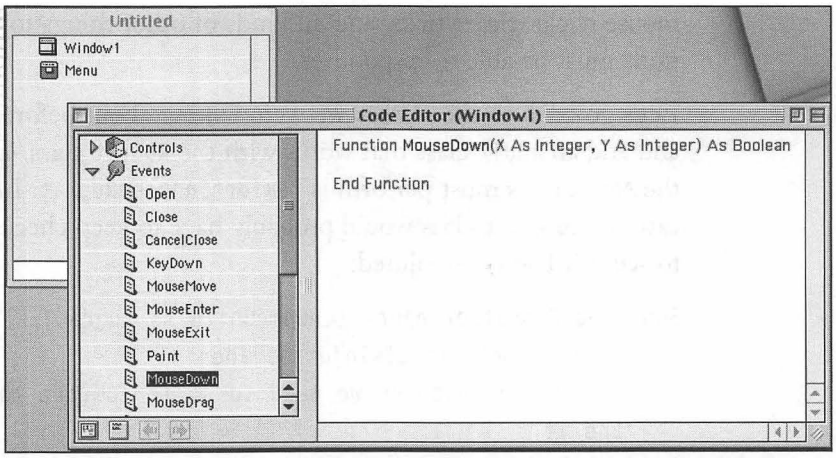



**Figure 9.10**
The event handler for a window's `MouseDown` event, which is sent to a window class when the user clicks the mouse button within the borders of the window

languages, is spent writing event handlers. Even all the new classes you create could be considered to serve the needs of the event handlers, so we guess you could say it's *all* about event handlers.

Because you'll be using them all the time, you might want to explore some of the events that are sent to various REALbasic classes. Refer to some of the various class definitions, the Window class for example, in the REALbasic Language Reference (included on the CD-ROM) for some examples of events.

# Review

A class is a collection of related functions, subroutines, and variables. The functions and subroutines are referred to as *methods* because they are the methods by which the class completes its tasks. The variables are referred to as *properties* because they contain the values that define the state of the class.

A class is just source code. An object is just a variable that has been defined as a specific type of class. An instance of the object exists only after the memory for that object has been allocated and assigned to the variable.

The three main concepts of object-oriented programming are encapsulation, inheritance, and polymorphism. Encapsulation means that all the properties and methods of a class are contained within the class definition and that no

other source code needs to be aware of the contents of the class. Inheritance is the capability to derive one class from another in which the derived class inherits the methods of the super class. Polymorphism is the capability to use derived classes in place of the super class without the application having knowledge of whether the derived class overrides the functionality of the super class.

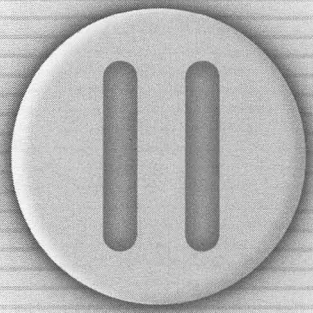
REALbasic, along with other object-oriented programming languages, sets aside certain predefined class methods that are called in response to system events. These methods are referred to as *event handlers*. Most of the coding work that is done in developing a REALbasic application is coding the event handlers or code to support the event handlers.

The concepts of object-oriented programming are simple enough by themselves, but from these simple acorn-like concepts a mighty oak of stable, simple-to-use applications can grow. Understanding classes, objects, methods, properties, and events is one of the essential keys to becoming a great REALbasic developer.

# Beginning Mac®

PROGRAMMING

PART



# Developing Your First Mac Program


# 10

# Making My Paint

## In This Chapter

- Introduction to the tutorial
- Creating the new project
- Adding the main window

In the previous chapters you familiarized yourself with REALbasic, created a very simple application, and learned some of the concepts of programming. You've probably been itching to get down and dirty and actually work on a functional application.

Everything you've done up to now has laid the groundwork for what you'll be working on from now on. By now you should have a fairly decent understanding of the main parts of the REALbasic application and should understand the simpler concepts of programming.

Now you're ready to move on to bigger and better things.

# Introduction to the Tutorial

In the next few chapters you'll be guided step by step in the process of creating a complete application using REALbasic. The subject that has been chosen for this tutorial is a type of application with which almost everyone is familiar: a paint program. After all, one of the first things most Mac users do after setting up their new computer is play with a paint program—or at least that was the case in the old days. So it makes some sense that the first full-blown application you're going to create is a paint program.

In each chapter of this tutorial, you'll progress deeper into the process of developing an application. In this chapter you'll begin by creating a new REALbasic project and creating the main application window. In the ensuing chapters, you'll add drawing controls; file open and save controls; and cut, copy, and paste features. You'll then move on to more advanced topics such as tool and color palettes.

# Creating the New Project

The first thing you'll need to do to start on this tutorial is to launch REALbasic. Locate the REALbasic program icon and double click it. After launching REALbasic, you should be presented with the REALbasic design environment, shown in Figure 10.1.

Just to refresh your memory, the parts of the REALbasic design environment are the

- Project window
- Tools window

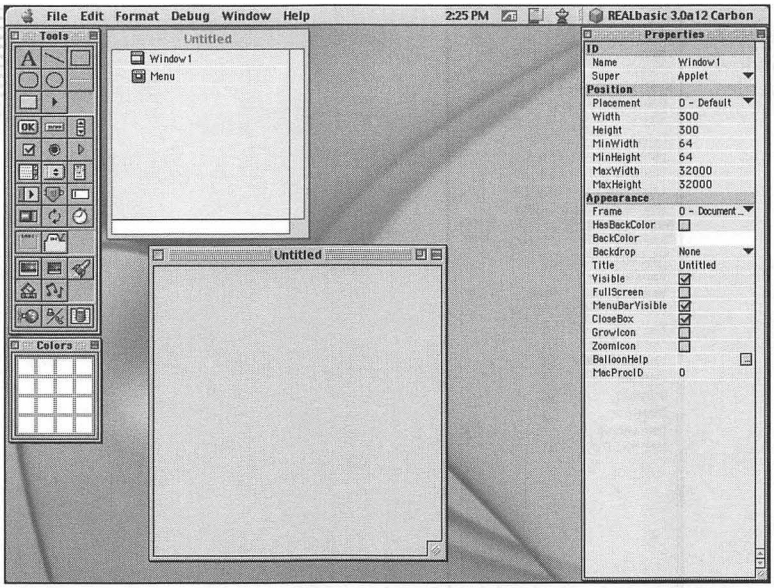



**Figure 10.1**
The REALbasic design environment

- Colors window
- Properties window
- Window Editor

# Adding the Main Window

You should notice right away that REALbasic has created an untitled project for you, containing just a window and a menu object in the Project window. The window object is named *Window1* by default. The first thing we are going to do is to customize this default window object to meet our needs. To change the default settings for Window1 do the following:

1. In the Project window, click on the Window1 object so that its properties are displayed in REALbasic's Properties window as shown in Figure 10.2.
2. In the Properties window, change the name of Window1 to PaintWindow, and press Return. The name of the window should change from Window1 to PaintWindow in REALbasic's Project window as well.

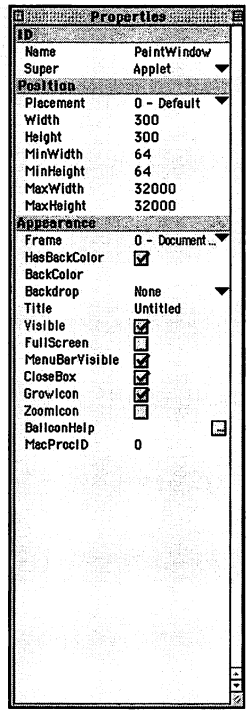



**Figure 10.2**
The PaintWindow properties

3. Click on the HasBackColor property to enable it. This changes the background color of PaintWindow to the default color, which is white.
4. Click on the GrowIcon property to enable it. This will allow the user to resize the PaintWindow object.

# Adding the Paint Canvas

PaintWindow by itself is incapable of acting as a paint canvas. PaintWindow is just a container for other controls. Fortunately there is already a REALbasic control capable of acting as a paint canvas. Coincidentally, it's called the Canvas control. Like other controls, you can add the Canvas control, shown in Figure 10.3, by dragging it from the Tools window to the window in the Window Editor.

To add a Canvas control to PaintWindow, do the following:

1. If PaintWindow is not open in the Window Editor, double click on PaintWindow in the Project window to open it in the Window Editor.

2. Click on the Canvas tool in the Tools window and then drag it onto PaintWindow and drop it anywhere. REALbasic names the Canvas control that you just added *Canvas1* by default. You'll need to change the name and some other default properties of the Canvas control.
3. If the Canvas control, Canvas1, is not currently selected, click on it so that its properties are displayed in the Properties window.
4. In the Properties window, change the name of the Canvas control from Canvas1 to *PaintCanvas* and press the Tab key.

You're now going to be changing the position and size of the canvas control so that it fills the entire PaintWindow window:

1. Under the Position section of the Properties window, change the Left value to –1 and press the Tab key. Notice that the PaintCanvas control has moved all the way to the left-hand side of PaintWindow.

**TIP**

Since you used the Tab key in step 1, the Top value should be selected in the PaintCanvas Properties window. It is easier to use the tab key to change multiple properties than to use the Return key and the mouse to select each property.

2. Change the Top value in the PaintCanvas Properties window to –1, and press the Tab button. Again, notice how the PaintCanvas control has moved to the top of PaintWindow. The PaintCanvas should now look something like the one shown in Figure 10.4.
3. Drag the lower-right resizing handle (the black square in the lower-right corner of the PaintCanvas control) so that it lines up with the

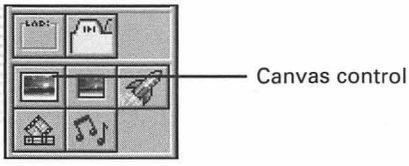



**Figure 10.3**
The Canvas control in the Tools window

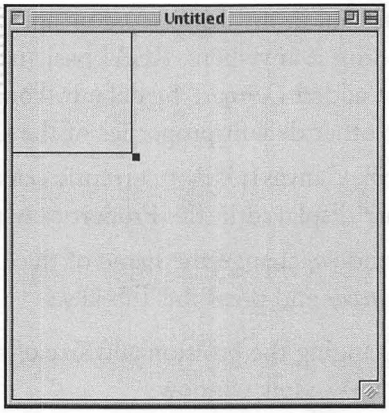



**Figure 10.4**
The PaintCanvas control moved to its new location

PaintWindow's Grow Window control (the rectangular control with the three diagonal lines in the bottom right hand corner of the window). After you've dragged the resizing handle to the proper location, the PaintWindow Window Editor should appear as seen in Figure 10.5.

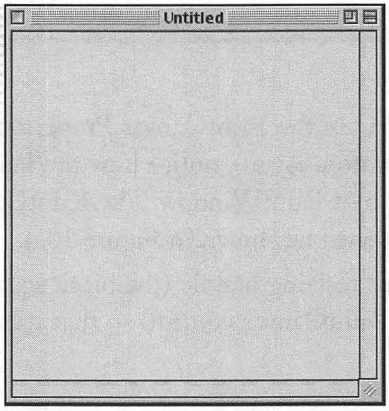
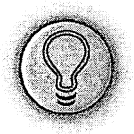



**Figure 10.5**
The resized PaintCanvas control



TIP

As you move the resizing handle near the bottom or right side of PaintWindow, you'll notice that dotted guidelines appear. You'll also notice that PaintCanvas snaps to these guidelines when the resizing handle gets close enough to one of them. Using these guidelines will help to make your control placement a lot simpler and the results will be much neater.

# Testing Your Work

By now you're probably curious to see what your application does. Truthfully, right now it doesn't do a lot. But because you're probably going to want to see for yourself, here's what you'll need to do.

## Saving Your Work in Progress

You should always save your changes before you decide to test your REALbasic application. Even though it's not likely, sometimes errors in your application can cause REALbasic to crash. If this happens, any of your changes will be lost. This can be very frustrating when you've done a lot of work since you last saved your project.

Even though your paint application is very simple at this point, you should get into the habit of at least saving your work every time you test it. The general rule is save twice as often as you think you should.

One nice feature of version 3, and greater, of REALbasic is its built in crash protection. Each time you run your program, REALbasic saves a copy of your project so that, in the case of a crash, you can restore your work the next time you launch REALbasic. Even though this feature works flawlessly, you should still get into the habit of saving your work in progress. If you move to another development tool that doesn't have nice auto-save features like this, you'll be glad you got into the habit ahead of time.

To save your REALbasic project, open the File menu and select the Save command, or press Command+S. Save your project with the filename *My Paint - Step 1*. The title of the Project window will be updated to *My Paint - Step 1*.

**TIP**

Now would be a good time to talk about organizational skills. Many application developers prefer to organize their projects. You may find that creating a *REALbasic Projects* folder, containing folders for each project you're working on, may help you keep things organized. For example, create a *REALbasic Projects* folder somewhere where you'll be able to find it again (in the *REALbasic* folder, or your *Documents* folder). Then create a *My Paint* folder and save all of the My Paint projects within this folder. It may seem like a lot of extra work right now, but later, when you've got a dozen irons in the programming fire, you'll really appreciate the extra organization.

## Testing Your Application

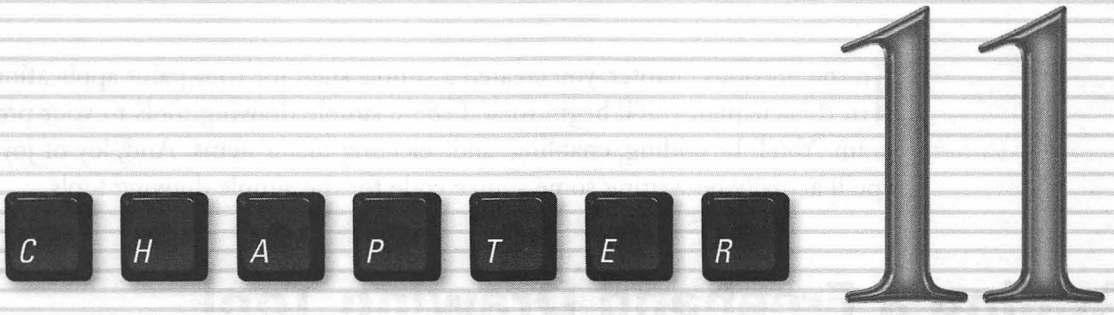
To run your new application, do the following:

1. Open the Debug menu and choose Run, or press Command+R.
2. Play around with your application. Admittedly, there's not much here to do yet. Trust me, things will get more interesting in the next chapter.
3. After you've exhausted yourself playing around with your very limited application, open the File menu and select Quit, or press Command+Q.

When you choose the Run item from the Debug menu or press Command+R to test your application, REALbasic switches to the runtime environment. The runtime environment, sometimes referred to as *debugging mode*, is where you can test and debug your application. When you choose the Quit item from the File menu or press Command+Q to quit your application, REALbasic returns to the design environment.

## Review

In this chapter you've learned how to launch REALbasic and create a simple project. You've also learned how to rename and change the properties of your application's main document window. You learned how to rename and change a control's properties and how to position controls on your application's main document window. In addition, you were shown how to save and test your project.




# Adding Simple Drawing Commands

## In This Chapter

- Adding a freehand drawing tool
- Understanding the Code Editor window
- Adding the drawing code
- Handling window drawing

In the previous chapter, you created a simple shell for your paint application. In this chapter, you'll begin to add some simple drawing tools to your program. You'll be adding, enabling, and selecting menu items. And, joy of joys, you'll actually be writing some source code for the simple drawing tools.

# Adding a Freehand Drawing Tool

OK, first things first, you want to see your application do something, so here we go. The first thing you're going to add to your application is a freehand drawing tool. This will be a tool that will draw on the PaintCanvas when the user clicks and drags the mouse.

To begin, launch REALbasic and open the My Paint - Step 1 project by clicking on the My Paint - Step 1 project icon in the Finder. If REALbasic is already running, you can open the File menu and choose Open to locate and open the project.

## Using the Code Editor Window

You'll need to open the Code Editor window by selecting the PaintWindow object in the Project window and pressing the Option+Tab key combination. The Code Editor window should appear, as shown in Figure 11.1.

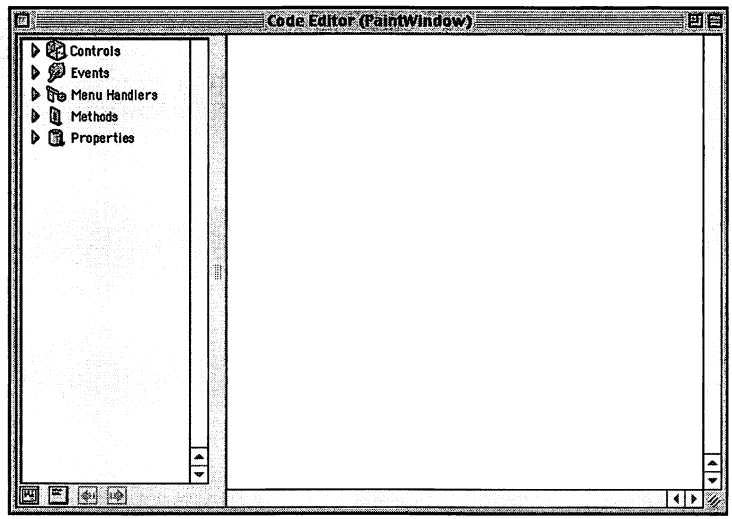



**Figure 11.1**
The REALbasic Code Editor window

The Code Editor window contains two panes. The left pane, called the *Browser pane*, contains a list of all of the objects that are contained within the window being edited. These objects fall into one of the following categories:

- **Controls.** These include any controls that you created by dragging them from the Tools window to your document window.
- **Events.** These include any application events to which the window is capable of responding. *Events* are messages generated by the system whenever the user does something to which your application window needs to respond. For example, a MouseDown event will be sent to your application window whenever the user clicks the mouse while the cursor is within the boundaries of the window.
- **Menu handlers.** These are specific events that are sent to your application window whenever a certain menu item is selected. Your application window can respond to any menu item by adding a menu handler for said menu item. For example, you would add a menu handler for Open, Save, or Save As menu items to respond to the users file requests.
- **Methods.** These are source-code routines that you add to an application window to perform a custom task. For the most part you'll be adding methods to increase source code re-usability and readability. Methods are also referred to as *member functions* because they are functions that are members of a specific window class.
- **Properties.** Like the properties in the REALbasic Properties window, properties in the Code Editor are used to keep track of specific values that relate to the window's state. As is the case with methods, properties are custom values that the developer has decided are needed. Properties are often referred to as *member variables* because they are variables that are members of a specific window class.

The right pane of the Code Editor window, called the *Editor* pane, is where source code for menu handlers and methods is edited. The gray area between the Browser pane and the Editor pane is a resizing column divider. Click and drag this left or right to resize the Browser pane.

# Adding the Drawing Code

The freehand drawing tool is supposed to function as follows: When the user clicks and drags the mouse, the application should draw a line that follows the path of the mouse as it's being dragged. To do this, you'll need your

application to respond to the MouseDown and MouseDrag events. But before you can properly handle these events, you'll need to add a couple of properties to the PaintWindow class.

## Adding the Property Declarations

The properties you'll be adding to the PaintWindow class will be used to track the last-known mouse location. When your application receives the MouseDrag event, it will be able to draw a line from the current mouse location to the last known location of the mouse. It will then update the values of the last-known mouse location so that the next MouseDrag event can draw the next connecting line, and so on.

To add the last mouse location properties to the PaintWindow class, do the following:

1. Open the Edit menu and select the New Property command, or press Option+Command+P to open the New Property dialog box, shown in Figure 11.2.
2. Type the following in the Declaration field of the New Property dialog box:

   ```
   nLastX As Integer
   ```

3. Press Return or click on the OK button to close the New Property dialog box.
4. Repeat steps 1 through 3, but this time add the following property declaration:

   ```
   nLastY As Integer
   ```

To view the properties that were just added, expand the Properties item in the Code Editor Browser pane by either clicking on the triangle next to the Properties icon, or double clicking on the Properties item. The new properties should appear under the Properties item, as shown in Figure 11.3.

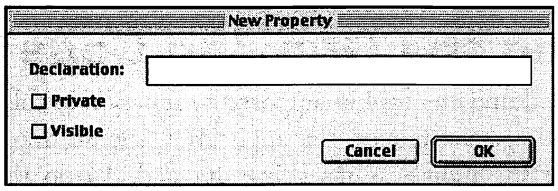



**Figure 11.2**
The New Property dialog box

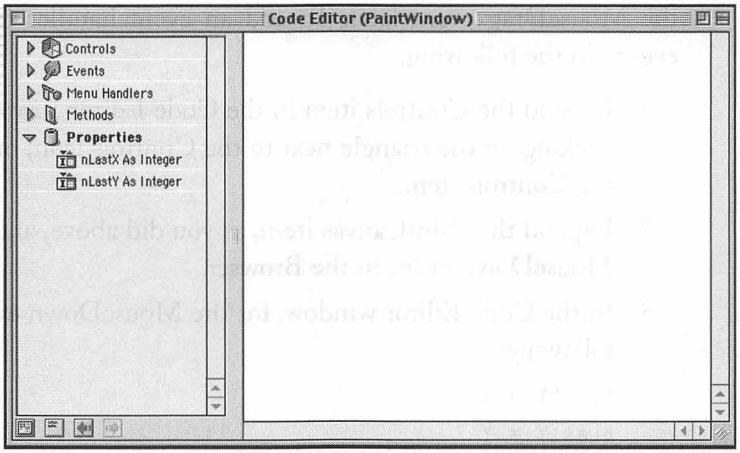



**Figure 11.3**
The new mouse-tracking properties

**The convention used for variable naming throughout this tutorial is referred to as *Hungarian Notation*. Hungarian Notation is a method of naming variables so that the variable type (integer, single, double, string, Boolean, and so on) is immediately obvious to the programmer. For example a variable with the name `bReady` is a Boolean, while `nCount` is an integer, and `strUserName` is a string. Using Hungarian Notation is sometimes difficult for both new and established programmers to get used to, but it is well worth the effort. The amount of time you'll save looking up a variable's type when debugging a lengthy program is well worth the effort. You'll see many examples of Hungarian Notation throughout this tutorial.**

## Adding the Event Handlers

Now that the properties you need have been added, you can work on the event handlers. Remember, events are sent to each window when something happens about which the particular window needs to be informed. In this case, you're interested in the MouseDown and MouseDrag events, because those events are what will be used to make your freehand drawing tool.

### THE MOUSEDOWN EVENT HANDLER

The MouseDown event handler will be used for two purposes. The event handler will set the `nLastX` and `nLastY` properties to their initial value, and will also enable all other mouse events—something you need to do in order to handle

the MouseDrag event later. To add an event handler for the MouseDown event, do the following:

1. Expand the Controls item in the Code Editor Browser pane by either clicking on the triangle next to the Controls icon, or double clicking on the Controls item.
2. Expand the PaintCanvas item, as you did above, and select the MouseDown event in the Browser.
3. In the Code Editor window, for the MouseDown event, type the following:

```
nLastX = X
nLastY = Y
return true
```

4. Returning `true` for the MouseDown event allows your program to respond to the MouseDrag and MouseUp events. The Code Editor window will now appear as shown in Figure 11.4.

### THE DRAGFREEHAND METHOD

The MouseDrag event handler will handle the actual drawing for your freehand drawing tool, along with almost all of the other drawing tools. You could add all of the code for the drawing tools in the MouseDrag event handler, but

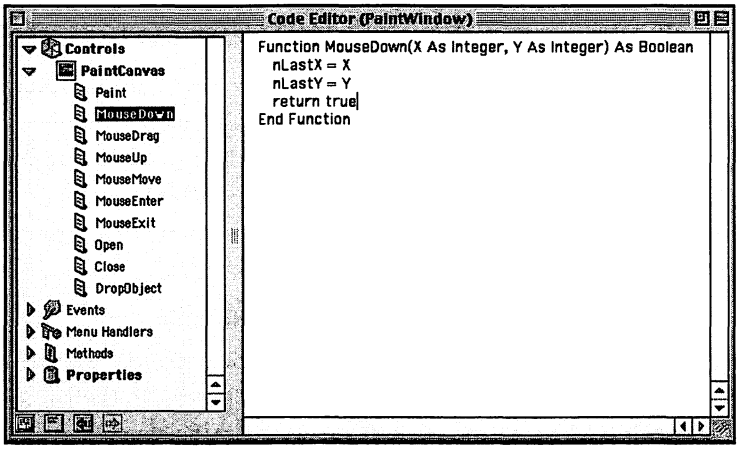



**Figure 11.4**
The Code Editor window for the PaintWindow PaintCanvas MouseDown event

this would get rather messy once all the drawing tools had been added. So, rather than write ugly, hard-to-read code, you're going to create a method, or member function, for each drawing tool.

The freehand drawing tool method will contain the code needed for performing freehand drawing. It will do this by comparing the last known–mouse location properties to the current mouse location. If the values are different, it will draw a line between the two points. It will then copy the current mouse location values to the last known–mouse location properties, so that the next call to this method can repeat the process.

To add the freehand drawing tool method, do the following:

1. Open the Edit menu and select the New Method command, or press Option+Command+M to open the New Method dialog box as shown in Figure 11.5.
2. Type the following in the Method Name field of the New Method dialog box:

   ```
   DragFreeHand
   ```

3. Type the following in the Parameters field of the New Method dialog box:

   ```
   X As Integer, Y As Integer
   ```

4. Leave the new method's return type blank, because this subroutine will not need to return any value.
5. Press Return or click on the OK button to close the New Method dialog box. The Code Editor window will appear as shown in Figure 11.6.

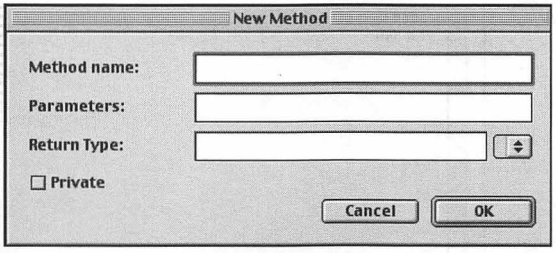



**Figure 11.5**
The New Method dialog box

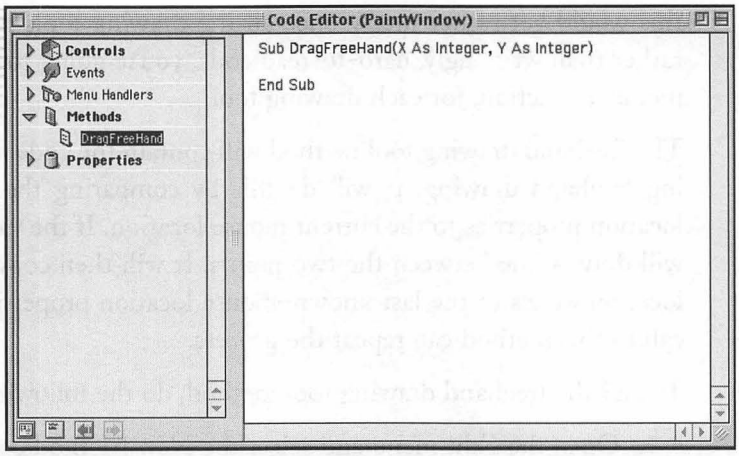



**Figure 11.6**
The new DragFreeHand method in the Code Editor window

6. In the Code Editor window, for the DragFreeHand method, type the following:

```
If ((nLastX <> X) or (nLastY <> Y)) then
    PaintCanvas.Graphics.ForeColor = RGB(0,0,0)
    PaintCanvas.Graphics.DrawLine(nLastX, nLastY, X, Y)
    nLastX = X
    nLastY = Y
  End if
```

The Code Editor window will now appear as shown in Figure 11.7.

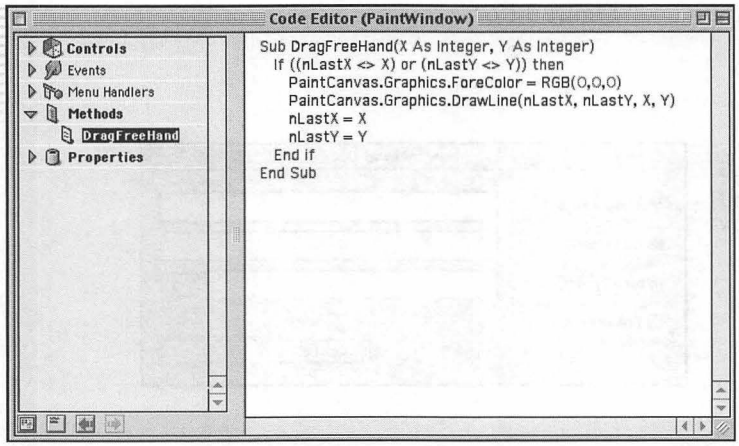



**Figure 11.7**
The Code Editor window for the DragFreeHand method

### THE MOUSEDRAG EVENT HANDLER

Finally, you're ready to enable the MouseDrag event handler so that it can draw a freehand line. All you need to do is add a call to the DragFreeHand method, passing the necessary parameters to the method. To add an event handler for the MouseDrag event, do the following:

1. Select the Controls, PaintCanvas, MouseDrag event in the Code Editor Browser pane.
2. In the Code Editor, for the MouseDrag event, type the following:

```
DragFreeHand(X, Y)
```

   The Code Editor window will now appear as shown in Figure 11.8.

## Testing the Freehand Tool

Now would be a good time to test the freehand drawing tool that you've just added. Don't forget to save your work.

To run the application, open the Debug menu and select the Run command or press Command+R to start the REALbasic runtime environment. If there is anything wrong with your code, REALbasic will display an error message indicating what is wrong. The Code Editor window, containing the offending source code, will be opened, and the cursor will be positioned near your mistakes.

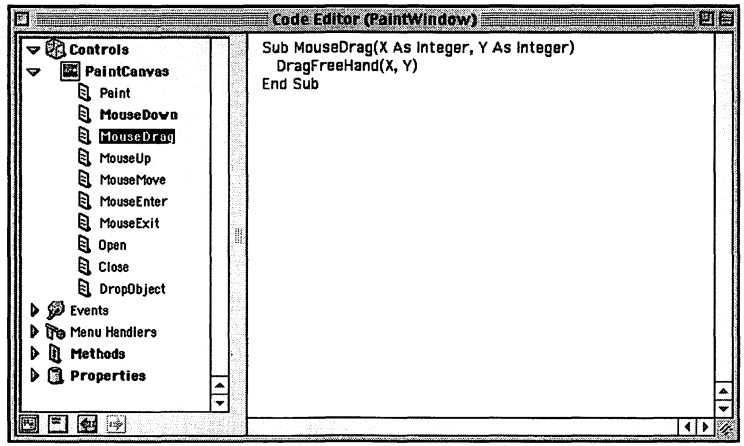



**Figure 11.8**
The Code Editor window for the PaintWindow, PaintCanvas, MouseDrag event

**TIP**

Sometimes the REALbasic error messages can seem a bit hard to understand to the novice programmer. Don't sweat it. Just go back over this section of the tutorial and check your spacing and spelling. Once you've found the error and corrected it, try running the application again.

Experiment with the freehand drawing tool for a while. Click and drag the mouse cursor within the application's window. You should see something similar to Figure 11.9.

When you're done testing the application, Open the File menu and select the Quit command or press Command+Q to return to the REALbasic design environment.

## Handling Window Drawing

You may have noticed that the tutorial application window doesn't redraw if you resize it, or if another window is placed over the application window and then moved away. This is because the paint canvas on which you are drawing does not maintain the picture in memory so that it can be redrawn. What you need to do is program the application to draw the picture in two places: on screen and in memory. When the window is redrawn, you can copy the picture in memory to the screen and refresh the window's contents.

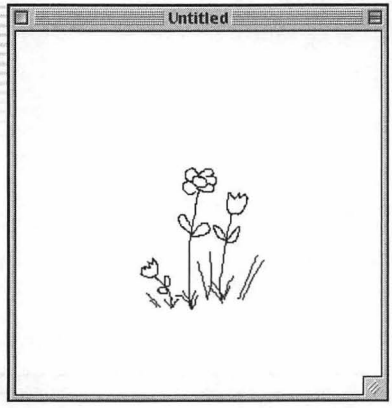



**Figure 11.9**
The application with a crude drawing

## Adding the Picture Buffer Property

To store a copy of the picture in memory, you'll add a Picture property to the PaintWindow class. You'll be painting to both the PaintCanvas control and the Picture property so that the same drawing is being updated on screen and in memory. Whenever the window needs to be repainted, the program will copy the contents of the Picture property to the PaintCanvas control to update the on-screen image.

To add the Picture property, do the following:

1. If the Code Editor window is not open, select the PaintWindow object in the Project window and press the Option+Tab key combination to open the Code Editor window.
2. Open the Edit menu and select the New Property menu item or press Option+Command+P to open the New Property dialog box.
3. Type the following in the Declaration field of the New Property dialog box:

   ```
   picBuffer As Picture
   ```

4. Press Return or click on the OK button to close the New Property dialog box.

## Creating the Picture Buffer Property

Before you can use the new Picture Buffer property, picBuffer, you need to create an instance of picBuffer. The REALbasic command that creates picture objects is `NewPicture`. The parameters passed to the NewPicture command specify the width, height, and number of colors of the picture object.

You only want to create the new picture object once, for every window that is opened. The trick is figuring out where to create the picture object so that it is created only when a new window is opened. Fortunately there is already an event that perfectly suits this purpose. The Window class's Open event is called every time a new window is about to be opened. Any code added in this event handler will be called every time a new window is being opened.

To add the code to create the Picture Buffer property, do the following:

1. Expand the Events item in the Code Editor Browser pane by either clicking on the triangle next to the Events icon, or double clicking on the Events item.

2. Select the Open event in the Browser pane.
3. In the Code Editor window, for the Open event, type the following:

```
dim nColorDepth as integer
nColorDepth = Screen(0).Depth
picBuffer   = NewPicture (640, 480,  nColorDepth)
```

In this code, you used the `Screen(0).Depth` method to determine the number of colors supported by the current monitor. You then use this value to determine the number of colors supported in the `picBuffer` picture object. The Code Editor window will now appear as shown in Figure 11.10.

## Drawing in the picBuffer Object

To ensure that the in-memory copy of the window contents is the same of the on-screen copy, the application has to draw to the in-memory copy as well as the on-screen copy. To program the application to do this, add the following code in the PaintWindow, DragFreeHand method, after the current `DrawLine` command:

```
if (picBuffer <> nil) then
  picBuffer.Graphics.ForeColor = RGB(0,0,0)
  picBuffer.Graphics.DrawLine(nLastX, nLastY, X, Y)
end if
```

This code first ensures that a valid `picBuffer` exists before attempting to access it. It then sets the foreground color and draws a line in `picBuffer`'s memory.

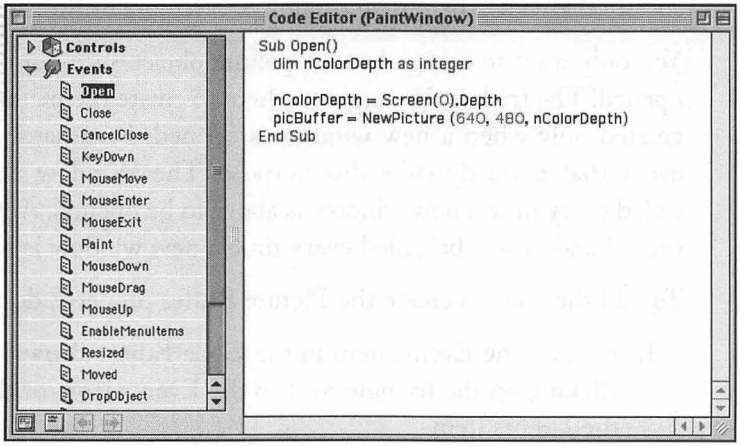



**Figure 11.10**
The Code Editor window for the PaintWindow, Open event

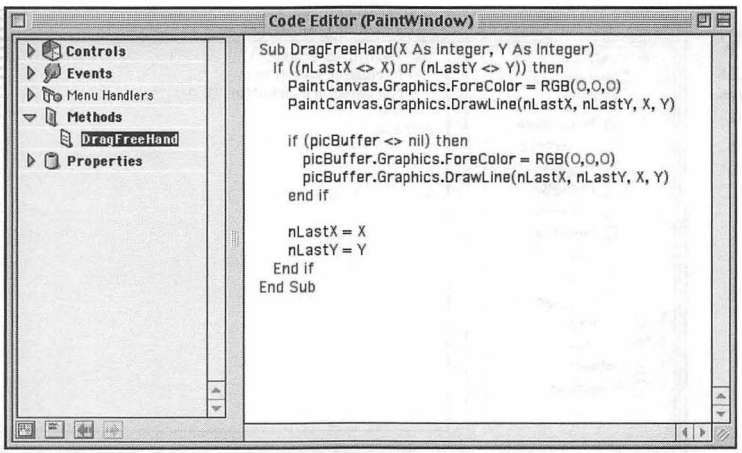



**Figure 11.11**
The Code Editor window for the PaintWindow, DragFreeHand method

After adding the above code, the PaintWindow, DragFreeHand method should appear as shown in Figure 11.11.

## Refreshing PaintCanvas Using the picBuffer Object

The only thing left to do is to update the contents of the on-screen window contents whenever the contents need to be redrawn. The Paint event is called whenever a window, or any of its controls, needs to be repainted. All you have to do is add the following code in the PaintWindow, Controls, PaintCanvas, Paint event:

```
if (picBuffer <> nil) then
  g.drawpicture picBuffer, 0, 0
end if
```

After you add this code, the PaintWindow, PaintCanvas, Paint event should appear as shown in Figure 11.12.

## Testing Your Changes

Now that you've added the code for the window refresh, you should save your work and test the code changes. If you run into any problems, simply review this section of the tutorial and double-check your work.

While testing the application, draw something in the Paint window. Then make sure that when you move the Paint window around, it is updated properly.

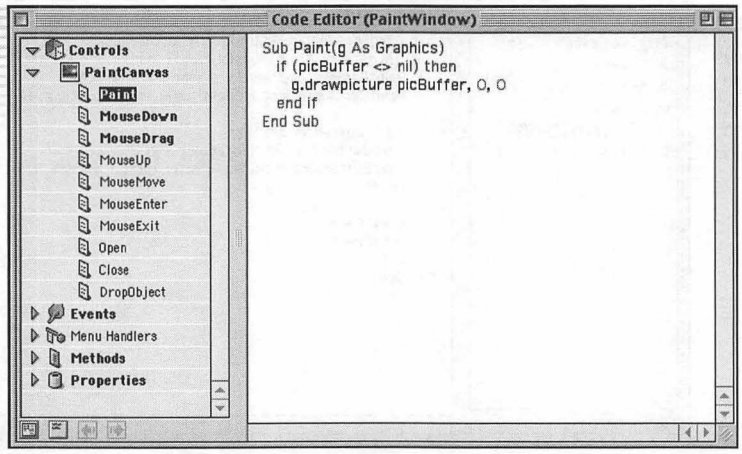



**Figure 11.12**
The Code Editor window for the PaintWindow, PaintCanvas, Paint event

Resize the Paint window to ensure that the entire picture redraws. Also try moving other application windows over the Paint window. When you move them away, the Paint window should update.

Don't forget to open the File menu and select the Quit menu item or press Command+Q to return to the REALbasic design environment after you've finished testing the application.

# Review

In this chapter, you learned how to enable the Code Editor window and the basics of navigating within that window. You learned about the Code Editor Browser and the text-editing sections of the Code Editor window. You also learned how to add event handlers, methods, and properties using the Code Editor window.

In addition, you learned a bit about when various event handlers are called and how you add code to handle the events. You also created a dynamic object, the picBuffer picture object, using the new command.

You should be fairly comfortable navigating the various sections of the Code Editor window by now, so getting around in future chapters of this tutorial should be a lot easier.


# 12

# Adding More Drawing Commands

## In This Chapter

- Adding menu items to select the drawing tools
- Adding a Line Draw tool and updating the Free Hand drawing tool
- Adding two Rectangle and two Oval drawing tools
- Adding a Draw Shape tool

In previous chapters, you created a paint application and added a simple freehand drawing tool. In this chapter, you'll add menu items to enable users to select from multiple drawing tools, and the code to handle these new tools.

# Adding Menu Items for the Selection of Drawing Tools

The first thing you're going to do is add the menus for the new tools you'll be adding: a Line Draw tool, two Rectangle drawing tools, two Oval drawing tools, and a Draw Shape tool. You'll also add a menu selection for the freehand drawing tool you created in the last chapter.

## Understanding the Application Menu Window

In REALbasic, you add menus to your project using the Application Menu window. With this tool, you can add menu items, submenus, and menu separators (the horizontal lines in Mac OS 8 and 9, and the blank spaces in Mac OS X, that separate menu groups). You can define the menu item names, and choose command-key shortcuts for the menu items. Additionally, you can assign Balloon Help and Disabled Balloon Help text, which pops up when the user has enabled the Macintosh help balloons. The Disabled Balloon Help text is displayed when the user hovers over the menu item and it is disabled.

To add your new menu items, do the following:

1. Open the Project window (open REALbasic's Window menu and select the Project command, or press Command+0).
2. Double-click the Menu object.
3. The Application Menu window, shown in Figure 12.1, appears. Click on the blank menu item. The Properties window displays the menu-item properties.
4. Change the Text property to Tools and press Return. The menu item that was formerly blank is now labeled *Tools*, as shown in Figure 12.2.
5. Click on the blank menu item, in the Tools menu. The blank menu item is the dotted line rectangle right under the Tools menu.

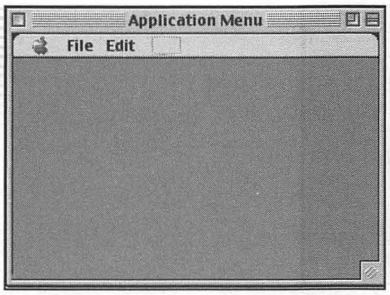



**Figure 12.1**
The REALbasic Application Menu window

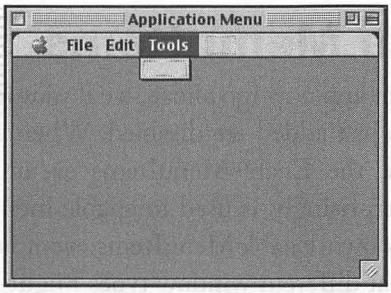



**Figure 12.2**
The Application Menu window with the Tools menu added

The Properties window displays the properties for this menu item. As you can see, there are more options for menu items than menus.

6. Change the Text property to Free Hand and press Return. You'll notice that REALbasic automatically changes the Name property to *ToolsFreeHand* for you.
7. Repeat steps 1–6, adding new menu items with the following names (Figure 12.3 shows the Tools menu with all the items added):
   - Line Draw
   - Draw Shape
   - Rectangle
   - Filled Rectangle
   - Oval
   - Filled Oval

**TIP**

If you click on a menu item and begin typing, the Properties window automatically switches to the Text property. You can also use the cursor keys to switch between menu items. Both of these shortcuts can save you valuable microseconds when you are entering lots of menu items.

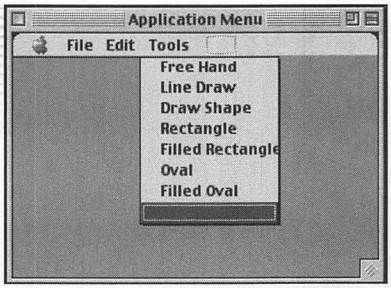



**Figure 12.3**
The Application Menu window with all of the Tools menu items added

# Enabling the Menu Items

If you run/debug the paint app now (go ahead, we'll wait for you), you'll notice that the menu items you just added are disabled. When a REALbasic application creates a window, the EnableMenuItems event for that window is called. This event, not surprisingly, is used to enable menu items. The reason each window type has its own EnableMenuItems event is to support the various valid menu items that different window types might have.

If you create an application class in your project, discussed in later chapters, you also have the ability to enable menu items application wide, meaning that any menu items enabled in the application class will be enabled throughout the entire application.

To enable all your new menu items, follow these steps:

1. Click on the Paint window in the Project window.
2. Press Option+Tab to display the Code Editor window.
3. Expand the Events item (click its disclosure triangle or double-click Events).
4. Select the EnableMenuItems event in the Code Editor window, and enter the following source code (Figure 12.4 shows the results of the entries):

```
ToolsFreeHand.Enable
ToolsLineDraw.Enable
ToolsDrawShape.Enable
ToolsRectangle.Enable
ToolsFilledRectangle.Enable
ToolsOval.Enable
ToolsFilledOval.Enable
```

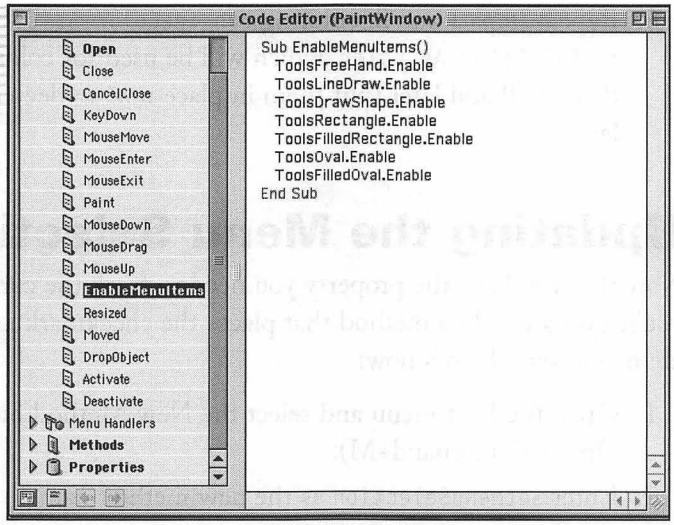



**Figure 12.4**
The Code Editor window for the EnableMenuItems event

Now when you run the app (go ahead, it's cool!), you'll notice the menu items are all enabled. Note that you didn't have to enable the Tools menu. This happens "automagically" when any of the Tools submenu items are enabled.

## Adding Properties for the New Tools

Before you get started on the actual code for the new tools, there are a few properties that you should add. The menus are supposed to function as follows: When the user chooses a tool's menu item, a checkmark is supposed to appear next to it. The checkmark should remain enabled until the user selects a new tool. To facilitate this, you're going to need to maintain the currently selected tool as a property.

You're also going to add a couple other properties at this time, which you'll create now, but won't be using until later.

Here's what to do:

1. Open the Edit menu and select the New Property menu item (or press Option+Command+P).
2. Enter `nCurrentTool As Integer` as the new property.

3. Repeat steps 1 and 2, entering `rgbLineColor As Color` and `rgbFillColor As Color`, which will be used for color-selection tools that you'll add later (put them in place now to decrease coding time later).

# Updating the Menu Selections

Now that you have the property you need to track the currently selected tool, you're going to add a method that places the checkmark next to the appropriate menu item. Here's how:

1. Open the Edit menu and select the New Method menu item (or press Option+Command+M).
2. Enter `SetMenuSelection` as the new method name.
3. Enter `nTool As Integer` as the new method parameters.
4. Leave the return value blank.
5. Click OK and enter the following source code in the SetMenuSelection Code Editor window, as shown in Figure 12.5:

```
nCurrentTool = nTool
ToolsFreeHand.Checked = (nCurrentTool = 1)
ToolsLineDraw.Checked = (nCurrentTool = 2)
ToolsDrawShape.Checked = (nCurrentTool = 3)
ToolsRectangle.Checked = (nCurrentTool = 4)
ToolsFilledRectangle.Checked = (nCurrentTool = 5)
ToolsOval.Checked = (nCurrentTool = 6)
ToolsFilledOval.Checked = (nCurrentTool = 7)
```

The SetMenuSelection does a few things. First, it saves the contents of the `nTool` parameter in the `nCurrentTool` property. You could have let the calling function change the contents of the `nCurrentTool` property, but doing so would violate the object-oriented encapsulation (see Chapter 9 , "Object Oriented Programming") of the `nCurrentTool` property. If you ever changed `nCurrentTool`, and/or how it's used, then every piece of code that uses `nCurrentTool` would have to be changed. So, by passing the new value for `nCurrentTool` into this function, you've centralized the code for what `nCurrentTool` is used for and any future changes will only have to be done in this function. Cool huh?

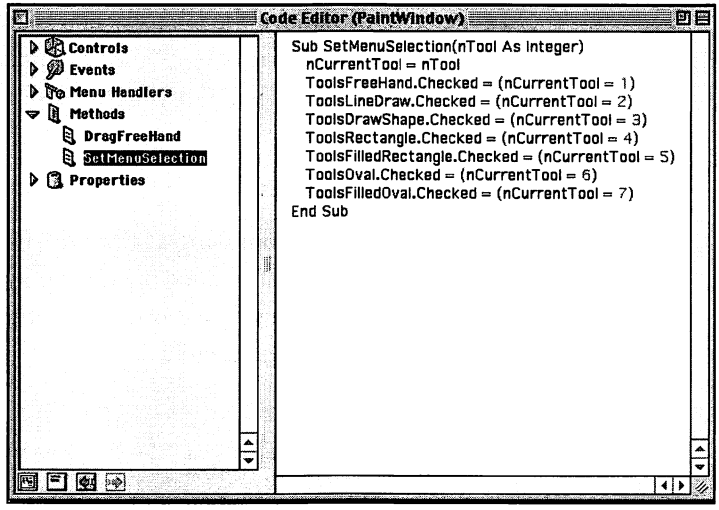



**Figure 12.5**
The Code Editor window for the SetMenuSelection method

The SetMenuSelection method then enables and disables the checkmark for each menu item as is appropriate based on the value of `nCurrentTool`. It does this by using a coding shortcut. You could have enabled and disabled each menu item with code similar to the following:

```
If (nCurrentTool = 1) then
  ToolsFreeHand.Checked = true
Else
  ToolsFreeHand.Checked = false
End if
```

But that's a heck of a lot of code, just to set one Boolean (true/false) value. A more efficient method would be to remove the redundant code, making it look something like this:

```
Dim bFreeHandChecked As Boolean
bFreeHandChecked = (nCurrentTool = 1)
ToolsFreeHand.Checked = bFreeHandChecked
```

This code works because the Equivalence operator (=) returns a Boolean value that indicates whether the values being compared are equal to each other. Even so, this code is inefficient, because it declares a variable that is used only once. Variables should be declared only if you are using them two or more times. It's all about reducing code redundancy.

You can simplify the code to remove the `bFreeHandChecked` variable completely and just store the result of the `nCurrentTool` comparison directly in the `ToolsFreeHand.Checked` property, as was done in the original SetMenuSelection method's code:

```
ToolsFreeHand.Checked = (nCurrentTool = 1)
```

Some developers might find this confusing at first; the use of the Assignment operator on the same line as an Equivalence operator may be what's confusing them. Just remember that an Equivalence operator returns a Boolean value, which can be stored in any Boolean variable.

## Initializing the New Properties

Don't run the application this time. It doesn't do anything new yet. You've still got a couple things to do before you'll notice any new functionality. First, you'll need to initialize the new properties so that they'll be ready when needed. Here's how:

1. Expand the Events item (click its disclosure triangle or double-click Events).
2. Select the Open event, and add the following to the code, as shown in Figure 12.6:

```
rgbLineColor = RGB(0, 0, 0)
rgbFillColor = RGB(0, 0, 0)
SetMenuSelection 1
```

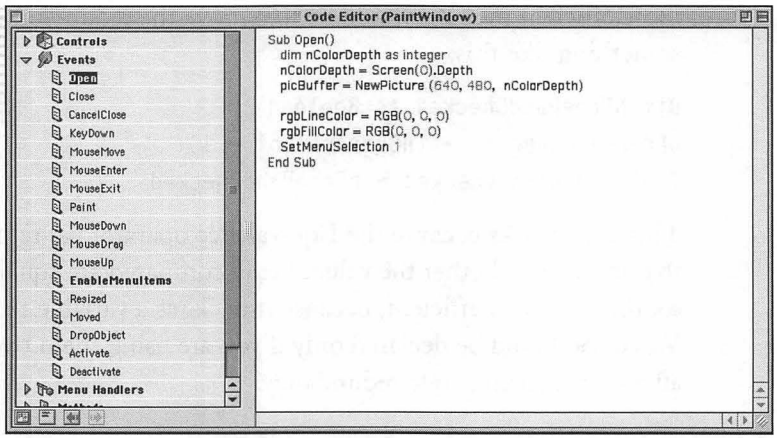



**Figure 12.6**
The Code Editor window for the Open event

If you run the application now, you'll notice that the Free Hand menu item is now enabled by default.

# Selecting Tools with the Menus

Now you're going to add the code that enables the checkmark on each menu item as it is selected. To do this, you must add menu handlers for each of the new menu items you added. To add each menu handler, follow these steps:

1. Open the Edit menu and select the New Menu Handler command (or press Option+Command+H).
2. Select ToolsFreeHand from Menu Item list. The New Menu Handler dialog box, shown in Figure 12.7, will open.
3. Click OK to display the Code Editor window for the ToolsFreeHand menu handler.
4. Enter the following in the Code Editor window for the ToolsFreeHand menu handler, as shown in Figure 12.8:

```
SetMenuSelection 1
```

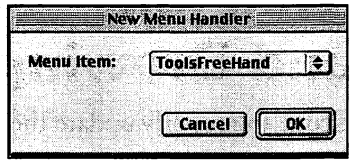



**Figure 12.7**
The REALbasic New Menu Handler dialog box

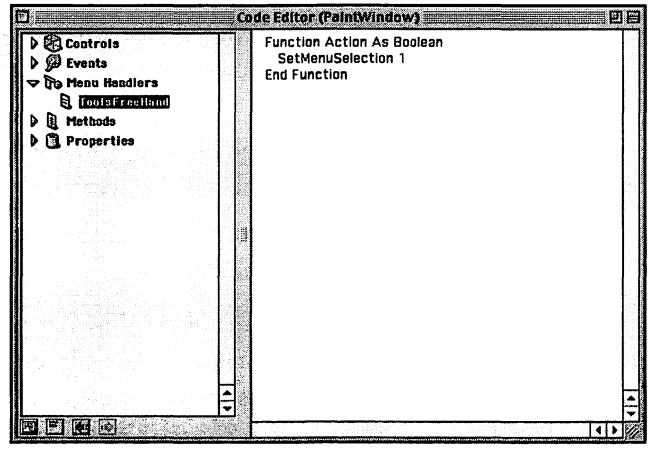



**Figure 12.8**
The Code Editor window for the ToolsFreeHand menu handler

5. Repeat steps 1–3 for the following menus:

| Menu Name | Code to Add in the Menu Handler's Code Editor Window |
|---|---|
| ToolsLineDraw | `SetMenuSelection 2` |
| ToolsDrawShape | `SetMenuSelection 3` |
| ToolsRectangle | `SetMenuSelection 4` |
| ToolsFilledRectangle | `SetMenuSelection 5` |
| ToolsOval | `SetMenuSelection 6` |
| ToolsFilledOval | `SetMenuSelection 7` |

If you run the application now (come on, do it!) you'll notice that all the Tools menu items will be updated when you select them. The checkmark will be displayed next to the menu item that you select. The checkmark will be removed from the last selected menu item when a new one is selected. The drawing commands obviously don't work, however, because no code has been added for each of the drawing commands.

# Adding a Line Draw Tool and Updating the Free Hand Drawing Tool

Now that you've added all the code to properly update the menu items' checkmarks, you can start working on some of the drawing tools. First you're going to work on the new Line Draw tool. Here's how the Line Draw tool works:

1. The user clicks and holds down the mouse button, causing the Line Draw tool to anchor the first end of the line at the location of the mouse click.
2. The user drags the mouse. As she does so, a line will be drawn between the anchored end and the current mouse location.
3. The user releases the mouse button, causing the other end of the line to be anchored to the current mouse location.

You're also going to be updating the Free Hand drawing tool. The current Free Hand drawing tool source code assumes it's the only drawing tool available. You're going to change it so that it fits into the new code design.

# Adding the DragRefresh Method

The first method you're going to add is the DragRefresh method, which will be responsible for refreshing the background of the Paint window when a drawing tool is being dragged across it.

To create the DragRefresh method, do the following:

1. Open the Edit menu and select the New Method menu item (or press Option+Command+M).
2. Enter `DragRefresh` as the new method name.
3. Enter `X1 As Integer, Y1 As Integer, X2 As Integer, Y2 As Integer` as the new method parameters.
4. Leave the return value blank.
5. Click OK and enter the following source code in the DragRefresh Code Editor window, as shown in Figure 12.9.

```
Dim X, Y, nWidth, nHeight As Integer
If (X1 < X2) then
    X = X1
Else
    X = X2
End If
If (Y1 < Y2) then
    Y = Y1
Else
    Y = Y2
End If
nWidth = Abs(X2 - X1) + 1
nHeight = Abs(Y2 - Y1) + 1
PaintCanvas.RefreshRect(X, Y, nWidth, nHeight)
```

Reread the code you just entered. Here's how it works:

1. First, the DragRefresh method declares four variables that are used to locate the top-left coordinates of the area being refreshed. Because the `PaintCanvas.RefreshRect` function doesn't allow for negative heights and widths, the DragRefresh method must determine which X coordinate, `X1` or `X2`, and which Y coordinate is to be used as the upper-left coordinate. That's what the first two `If` statements do.

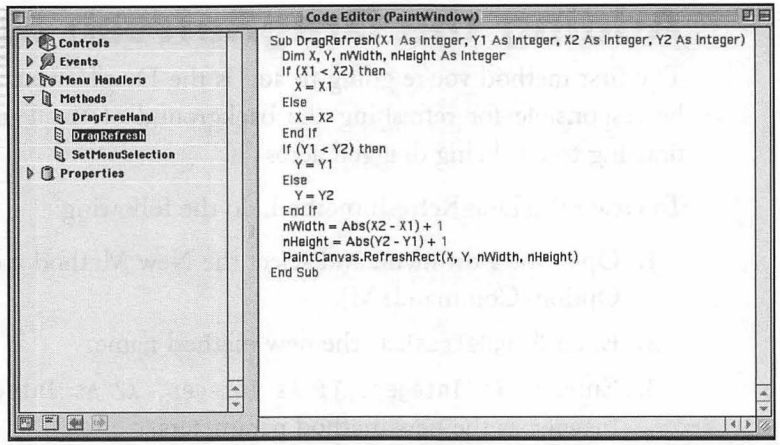



**Figure 12.9**
The Code Editor window for the DragRefresh method

2. The DragRefresh method then calculates the width and height of the refreshed rectangle before calling the `PaintCanvas.RefreshRect` function.

In essence the DragRefresh method determines the proper X and Y coordinates, along with the proper width and height values, and then refreshes the rectangle as needed.

# Adding New End Point Properties

The project already includes two properties used to hold the last-known mouse location: `nLastX` and `nLastY`. For line drawing, two other mouse location properties are needed. When a line is drawn the `nLastX` and `nLastY` properties will hold the location of the first mouse click (the start of the line), and the two new properties will hold the location of the point at which the mouse was released (the end of the line). These new properties, `nLastEndX` and `nLastEndY`, will also be used in the drawing of other shapes, such as rectangles.

Here's how to add these properties:

1. Open the Edit menu and select the New Property command, or press Option+Command+P to open the New Property dialog box.
2. Type the following in the Declaration field of the New Property dialog box:

```
nLastEndX As Integer
```

3. Press Return or click on the OK button to close the New Property dialog box.
4. Repeat steps 1 through 3, but this time add the following property declaration:

```
nLastEndY As Integer
```

You'll see that these properties are used in a few methods, starting next with the DragLineDraw method that refreshes the window.

# Adding the DragLineDraw Method

Now you're going to add a DragLineDraw method that will refresh the rectangular area of the line being drawn, set the proper line color, and then draw a line between the anchored line end and the current mouse position. After all of this, it will update the `LastEndX` and `LastEndY` values so that they can be compared to the mouse position the next time the DragLineDraw method is called. If the current mouse position hasn't changed since the last time DragLineDraw was called, then nothing happens. This helps prevent flickering during redraws.

Follow these steps to add the DragLineDraw method:

1. Open the Edit menu and select the New Method command (or press Option+Command+M).
2. Enter `DragLineDraw` as the new method name.
3. Enter `X As Integer, Y As Integer` as the new method parameters.
4. Leave the return value blank.
5. Click OK and enter the following source code in the DragLineDraw Code Editor window, as shown in Figure 12.10:

```
If ((X <> nLastEndX) or (Y <> nLastEndY)) Then
   DragRefresh(nLastX, nLastY, nLastEndX, nLastEndY)
   PaintCanvas.Graphics.ForeColor = rgbLineColor
   PaintCanvas.Graphics.DrawLine(nLastX, nLastY, X, Y)
   nLastEndX = X
   nLastEndY = Y
End If
```

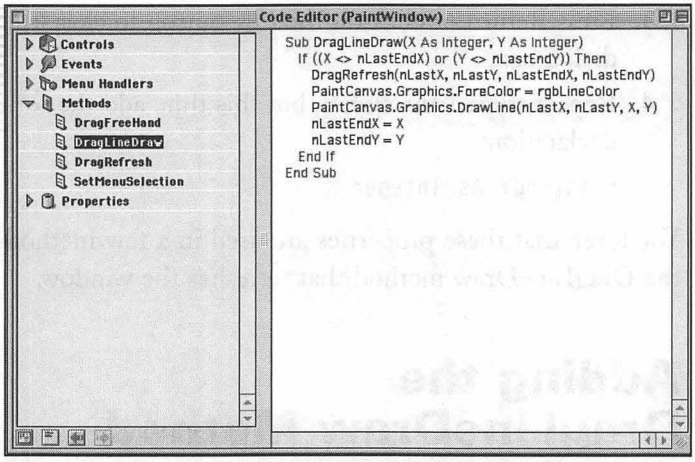



**Figure 12.10**
The Code Editor window for the DragLineDraw method

# Adding the EndLineDraw Method

As we said before, use of the Line Draw tool is completed when the user releases the mouse button. The EndLineDraw method does this by drawing a line between the two anchor points. The line is drawn in the `picBuffer` so that during future refreshing, the line will be maintained.

To add the EndLineDraw method, do the following:

1. Open the Edit menu and select the New Method command (or press Option+Command+M).
2. Enter `EndLineDraw` as the new method name.
3. Leave the parameters and return value blank.
4. Click OK and enter the following source code in the EndLineDraw Code Editor window, as shown in Figure 12.11:

```
If (picBuffer <> nil) then
   picBuffer.Graphics.ForeColor = rgbLineColor
   picBuffer.Graphics.DrawLine(nLastX, nLastY, nLastEndX, ➥
nLastEndY)
End If
```

This code first checks to make sure that `picBuffer` was successfully created. It then sets the proper line color and draws a line in `picBuffer`.

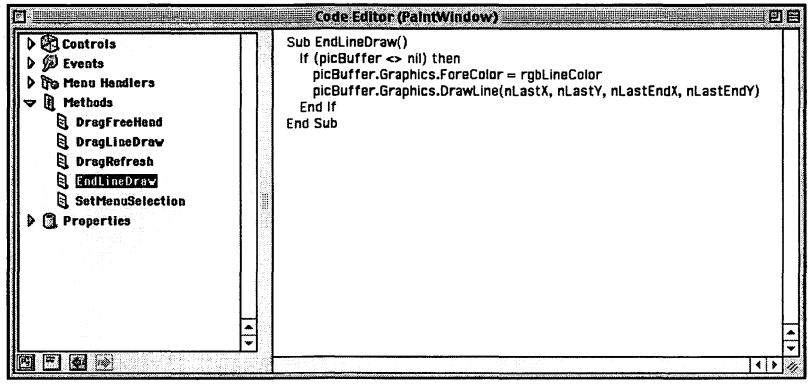



**Figure 12.11**
The Code Editor window for the EndLineDraw method

# Changing the MouseDrag PaintCanvas Event

There's no point in testing the code at this point. You won't see any results yet, but you're getting there. First you need to change PaintCanvas's MouseDrag event so that it can either work with the Free Hand drawing tool or the Line Draw tool.

To update PaintCanvas's MouseDrag event, do the following:

1. Expand the Controls item (click its disclosure triangle or double-click Controls).
2. Expand the PaintCanvas item (click its disclosure triangle or double-click PaintCanvas).
3. Select the MouseDrag event and change the MouseDrag source code to the following, as shown in Figure 12.12:

```
Select Case (nCurrentTool)
Case 1
   DragFreeHand(X, Y)
Case 2
   DragLineDraw(X, Y)
End Select
```

This code represents the first time you've used the Select/Case keywords. As we mentioned in Chapter 6 "Making Your Program Flow," the Select/Case

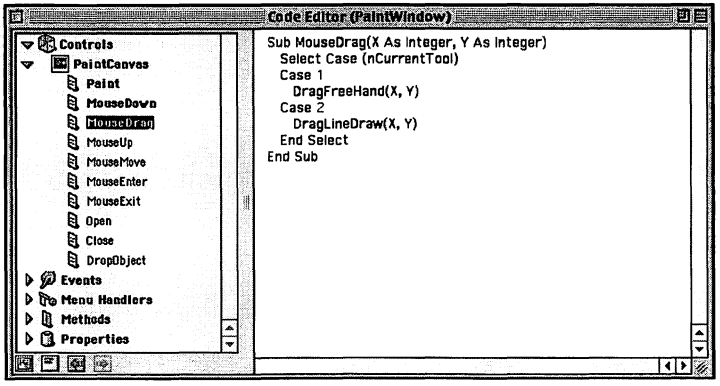



**Figure 12.12**
The Code Editor window for the MouseDrag event

keywords allow you to execute various sections of code similar to multiple If/Else If/Else/End If statements. The MouseDrag code now calls the DragFreeHand method if the Free Hand item is selected from the Tools menu and the DragLineDraw method if the Line Draw menu item is selected from the Tools menu.

# Changing the MouseUp PaintCanvas Event

Last but not least, the MouseUp event needs to be updated just like the MouseDrag event was. You're going to need to call the EndLineDraw method if LineDraw menu item is selected from the Tools menu. There's no need to do anything for the Free Hand drawing tool.

All you need to do to update the code is to select the MouseUp event (it's listed right by the MouseDrag event under the PaintCanvas item) and enter the following source code, as shown in Figure 12.13:

```
Select Case (nCurrentTool)
Case 2
   EndLineDraw
End Select
```

Now, finally you can run the application and see the fruits of your labor. If you run the application now, you'll notice that the Line Draw tool works correctly. The Free Hand drawing tool should still work as well. If either of the

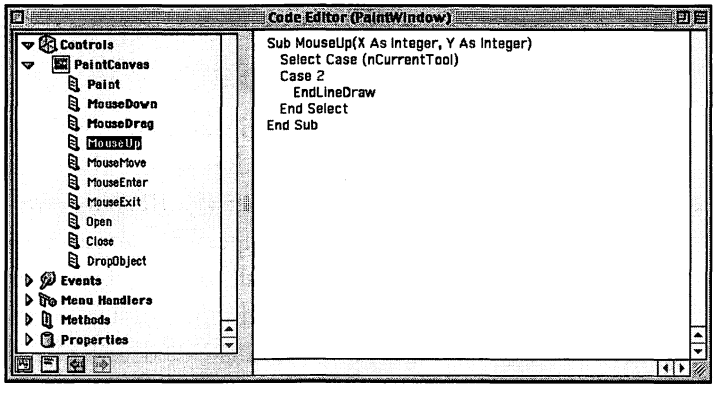



**Figure 12.13**
The Code Editor window for the MouseUp event

tools works incorrectly, don't be too concerned. Simply recheck your code and try again.

# Adding Rectangle and Oval Drawing Tools

The Rectangle and Oval tools work in similar fashion. You use both by clicking and dragging to determine the shape and size of the object being drawn. Both have optional filled-in versions, and both require new ToolDrag, ToolEnd, MouseDrag, and MouseUp methods.

Here's how these tools work:

1. The user clicks and holds down the mouse button, causing the Rectangle or Oval tool to anchor the first corner of the shape at the location of the mouse click.
2. The user drags the mouse. As she does so, a rectangle or oval will be drawn between the anchored corner and the current mouse location. If the user has selected the filled version of one of these tools, the area is filled in with the selected fill color.
3. The user releases the mouse button, causing the other corner of the shape to be anchored to the current mouse location.

Even though they operate the same way, the code is different enough to work on these tools separately.

# Adding the DragRectangle and DragOval Methods

First, you must add the DragRectangle and DragOval methods. Both of these methods first refresh the area being drawn, and then check whether the user selected the filled-in versions of either the Rectangle or the Oval tool. If so, the object is first drawn as a filled shape. The proper line color is selected, and the shape outline is drawn. Last of all, the `LastEndX` and `LastEndY` values are updated so that they can be compared to the mouse position the next time the DragRectangle or DragOval method is called. DragRectangle and DragOval do nothing if the mouse hasn't changed location since the last time these methods were called.

Follow these steps to add the DragRectangle method:

1. Open the Edit menu and select the New Method command (or press Option+Command+M).
2. Enter `DragRectangle` as the new method name.
3. Enter `X As Integer, Y As Integer, bFill As Boolean` as the new method parameters.
4. Leave the return value blank.
5. Click OK and enter the following source code in the DragRectangle Code Editor window:

```
If ((X <> nLastEndX) or (Y <> nLastEndY)) Then
  DragRefresh(nLastX, nLastY, nLastEndX, nLastEndY)
  If (bFill) Then
    PaintCanvas.Graphics.ForeColor = rgbFillColor
    PaintCanvas.Graphics.FillRect(nLastX, nLastY, X-nLastX, ➥
    Y-nLastY)
  End If
  PaintCanvas.Graphics.ForeColor = rgbLineColor
  PaintCanvas.Graphics.DrawRect(nLastX, nLastY, X-nLastX, Y-nLastY)
  nLastEndX = X
  nLastEndY = Y
End If
```

Follow these steps to add the DragOval method:

1. Open the Edit menu and select the New Method command (or press Option+Command+M).

2. Enter `DragOval` as the new method name.
3. Enter `X As Integer, Y As Integer, bFill As Boolean` as the new method parameters.
4. Leave the return value blank.
5. Click OK and enter the following source code in the DragOval Code Editor window:

```
If ((X <> nLastEndX) or (Y <> nLastEndY)) Then
  DragRefresh(nLastX, nLastY, nLastEndX, nLastEndY)
  If (bFill) Then
    PaintCanvas.Graphics.ForeColor = rgbFillColor
    PaintCanvas.Graphics.FillOval(nLastX, nLastY, X-nLastX, ➥
    Y-nLastY)
  End If
  PaintCanvas.Graphics.ForeColor = rgbLineColor
  PaintCanvas.Graphics.DrawOval(nLastX, nLastY, X-nLastX, ➥
    Y-nLastY)
  nLastEndX = X
  nLastEndY = Y
End If
```

When we discussed the DragRefresh code earlier in this chapter, we mentioned that the RefreshRect method doesn't support negative height and width values. You therefore needed to choose the proper corner for the X and Y coordinates and calculate the appropriate height and width. You would think that the `FillRect`, `DrawRect`, `FillOval` and `DrawOval` functions would work the same way, but both of these functions use the X and Y coordinates of the upper-left and lower-right corners of the shape to specify the region of the shape being drawn. Therefore, the proper coordinates don't need to be chosen and the height and width are not calculated either.

# Adding the EndRectangle and EndOval Methods

As was explained previously, use of the Rectangle and Oval tools (and their filled-in versions) is completed when the user releases the mouse button. If the user chose the filled versions of these tools, the EndRectangle and EndOval methods fill in the shape, using the proper fill color, in `picBuffer`. Last, the methods draw the shape outline, using the proper line color, in `picBuffer`.

Recall that `picBuffer` is updated so that future screen refreshes will display the proper graphic.

To create the EndRectangle method, do the following:

1. Open the Edit menu and select the New Method command (or press Option+Command+M).
2. Enter `EndRectangle` as the new method name.
3. Enter `bFill As Boolean` as the new method parameters.
4. Leave the return value blank.
5. Click OK and enter the following source code in the EndRectangle Code Editor window:

```
If (picBuffer <> nil) then
  If (bFill) Then
    picBuffer.Graphics.ForeColor = rgbFillColor
    picBuffer.Graphics.FillRect(nLastX, nLastY, ➥
nLastEndX-nLastX, nLastEndY-nLastY)
  End If
  picBuffer.Graphics.ForeColor = rgbLineColor
  picBuffer.Graphics.DrawRect(nLastX, nLastY, ➥
nLastEndX-nLastX, nLastEndY-nLastY)
End If
```

You create the EndOval method by following these steps:

1. Open the Edit menu and select the New Method command (or press Option+Command+M).
2. Enter `EndOval` as the new method name.
3. Enter `bFill As Boolean` as the new method parameters.
4. Leave the return value blank.
5. Click OK and enter the following source code in the EndOval Code Editor window:

```
If (picBuffer <> nil) then
  If (bFill) Then
    picBuffer.Graphics.ForeColor = rgbFillColor
    picBuffer.Graphics.FillOval(nLastX, nLastY, ➥
nLastEndX-nLastX, nLastEndY-nLastY)
  End If
```

```
    picBuffer.Graphics.ForeColor = rgbLineColor
    picBuffer.Graphics.DrawOval(nLastX, nLastY, ➥
nLastEndX-nLastX, nLastEndY-nLastY)
End If
```

Now that you've added the code for the tools, all you need to do is call these methods where appropriate.

# Changing the MouseDrag PaintCanvas Event

As with the Line Draw tool, the Rectangle and Oval tools' "drag" methods need to be called from the MouseDrag event. To do this, you'll be adding cases to the select statement. Follow these steps to call these methods where appropriate:

1. Expand the Controls item (click its disclosure triangle or double-click Controls).
2. Expand the PaintCanvas item (click its disclosure triangle or double-click PaintCanvas).
3. Select the MouseDrag event and add the following to the end of the Select Case statement in the MouseDrag source code:

```
Case 4
  DragRectangle(X, Y, false)
Case 5
  DragRectangle(X, Y, true)
Case 6
  DragOval(X, Y, false)
Case 7
  DragOval(X, Y, true)
```

Each of the Drag methods is called with the `bFill` parameter set to either `false` or `true`, to handle the fact that the Rectangle and Oval drawing tools can be used in filled or unfilled mode.

# Changing the MouseUp PaintCanvas Event

Last of all, the MouseUp event needs to be changed to call the proper End drawing tool methods. As with the MouseDrag event, the Select statement of

the MouseUp event needs to be extended to include calls to the proper End tool method. To add the new cases to the select statement, select the MouseUp event (located right by the MouseDrag event under the PaintCanvas item) and enter the following source code:

```
Case 4
  EndRectangle(false)
Case 5
  EndRectangle(true)
Case 6
  EndOval(false)
Case 7
  EndOval(true)
```

If you run the application now, you'll notice that both of the Rectangle and Oval drawing tools function properly. Go ahead and run the app, and try the new tools by selecting each one, clicking and dragging in the drawing window, and releasing the mouse button once the shape is the size you want it. Make sure you move the drawing window around on the screen and resize it to test the DragRefresh function to insure that it's working properly.

# Adding a Draw Shape Tool

The Draw Shape tool is a bit different from the other tools you've worked on. They all follow the click-drag-release method of operation. The Draw Shape tool, however, works like this:

1. The user clicks and releases the mouse button, locking one end of a line-drawing tool. The system "remembers" this location for future reference.
2. The user moves the mouse; the application draws a line from the locked end of the line to the current mouse position.
3. The user clicks and releases the mouse button again, locking the other end of the line, which is then committed to the paint canvas. A new line is started, with its end locked at the same location as the end of the first line.
4. The user can then move the mouse to a new location and repeatedly single-click, drawing one line attached to the end of the previous line.

5. When the user wants to stop the shape-drawing process, she double-clicks the mouse. At this point, the system draws a line from the end of the last line drawn to the beginning of the first line.

So, as you can see, this tool is just a bit more complicated.

## Adding Properties for the New Tools

First you're going to add some new properties that will be used to track the location of the mouse click that started the process. You're also going to add a property to save the time that the mouse button was released. This will be used to calculate the difference between the time of current mouse click and the last time the mouse was released. This helps you to determine whether a double-click occurred.

Now, let's add those properties:

1. Open the Edit menu and select the New Property command (or press Option+Command+P).
2. Enter the following new properties:

```
bDrawingShape As Boolean
nShapeStartX As Integer
nShapeStartY As Integer
dMouseUpTime As Double
```

Now that you've got these new properties in place, you're going to need to modify three events to make the new tool work: MouseUp, MouseMove, and MouseDown.

## Changing the MouseUp PaintCanvas Event

The only thing new thing that the MouseUp event has to do is keep track of the last time the mouse button was released. The Microseconds function returns the number of microseconds since the computer was lasted started up. By calculating the difference between two Microseconds return values, the application can determine the duration between two events—in your case, the MouseUp event and the MouseDown event.

Here's how to change the MouseUp event:

1. Expand the Controls item (click its disclosure triangle or double-click Controls).
2. Expand the PaintCanvas item (click its disclosure triangle or double-click PaintCanvas).
3. Select the MouseUp event and add the following to the top of MouseUp source code, as shown in Figure 12.14:

```
dMouseUpTime = Microseconds
```

# Changing the MouseMove PaintCanvas Event

The MouseMove event only has to draw a line from the last anchor point of the current mouse position. Fortunately for us, the line drawing can be accomplished through the use of the existing DragLineDraw method, so you won't need to do a lot to satisfy the needs of the MouseMove event.

Here's what you need to do:

1. Expand the Controls item (click its disclosure triangle or double-click Controls).
2. Expand the PaintCanvas item (click its disclosure triangle or double-click PaintCanvas).

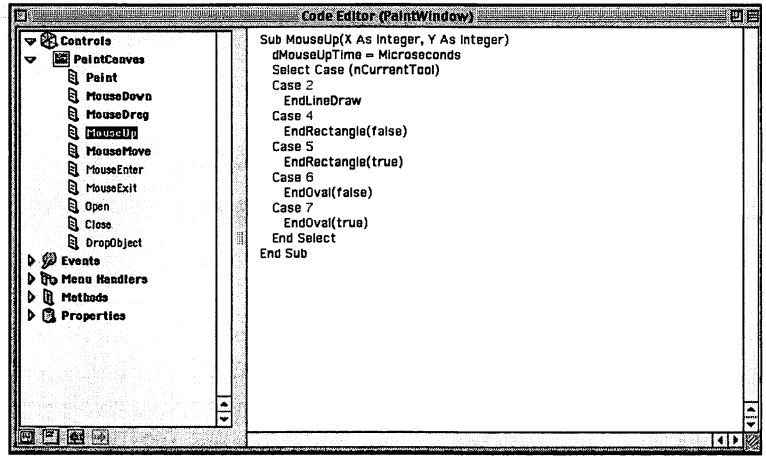



**Figure 12.14**
The Code Editor window for the MouseUp event

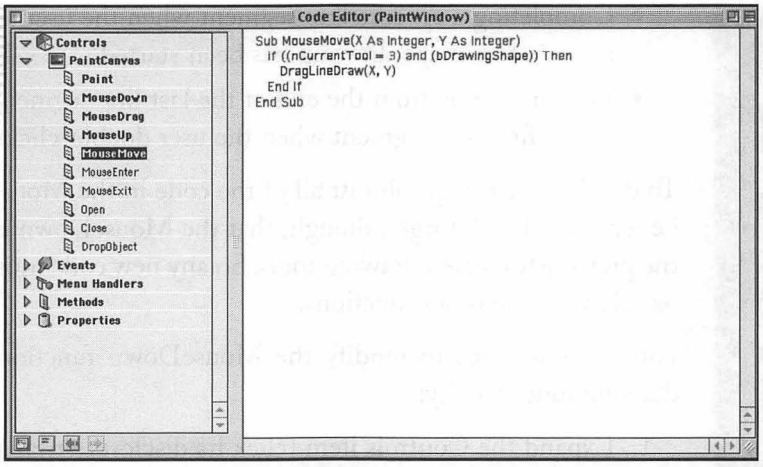



**Figure 12.15**
The Code Editor window for the MouseMove event

3. Select the MouseMove event and enter the following source code, as shown in Figure 12.15:

```
if ((nCurrentTool = 3) and (bDrawingShape)) Then
    DragLineDraw(X, Y)
End If
```

The MouseMove event code only calls the DragLineDraw method if the Draw Shape tool is selected, which you can determine by checking whether `nCurrentTool` is equal to a value of `3`. Also, the `bDrawingShape` Boolean variable must contain a value of true. If both of these conditions are satisfied, then the line can be drawn.

The reason you use the MouseMove event, rather than the MouseDrag event as with the other tools, is that the requirements of the tool don't call for click-drag-release functionality. The Draw Shape tool draws the line while the mouse cursor is moving, not while it's being dragged.

## Changing the MouseDown PaintCanvas Event

The real powerhouse of the Draw Shape tool is the MouseDown event, which is responsible for

- Recording the mouse click that starts a new shape-drawing process and saving this value.

*Backward compatibility* refers to the capability of an application to work with an older version of the application. You might have heard the term used to refer to the capability of a word-processing application to open files from a previous version of the app. The term *backward compatibility* can also be used in reference to sections of your code.

- Completing a specific line segment when the user single-clicks the mouse after shape drawing has been started.
- Drawing a line from the end of the last line segment to the beginning of the first line segment when the user double-clicks the mouse.

To do all these things, almost all of the code in the MouseDown event has to be replaced. Don't forget, though, that the MouseDown event is also used for the previously created drawing tools. So any new code must be backward compatible with the other functions.

Follow these steps to modify the MouseDown function to add the shape-drawing functionality:

1. Expand the Controls item (click its disclosure triangle or double-click Controls).
2. Expand the PaintCanvas item (click its disclosure triangle or double-click PaintCanvas).
3. Select the MouseDown event and change the entire MouseDown source code to the following, as shown in Figure 12.16:

```
If (nCurrentTool = 3) Then
  If (bDrawingShape) Then
    If ((Microseconds - dMouseUpTime) < 400000) Then
      DragLineDraw(nShapeStartX, nShapeStartY)
      bDrawingShape = false
```

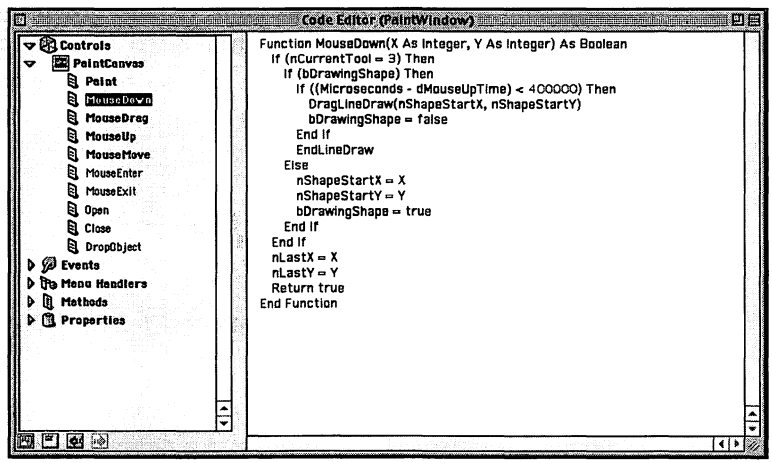



**Figure 12.16**
The Code Editor window for the MouseDown event

```
        End If
        EndLineDraw
      Else
        nShapeStartX = X
        nShapeStartY = Y
        bDrawingShape = true
      End If
    End If
    nLastX = X
    nLastY = Y
    Return true
```

The new code for the MouseDown event does the following:

- If the Draw Shape tool is selected (`nCurrentTool = 3`), and a shape is currently being drawn (`bDrawingShape`), the time since the last MouseUp event is calculated to determine whether a double-click has occurred. If the user double-clicked, then a line is drawn to the coordinates where the shape-drawing command was started. Whether the user clicked or double-clicked, the MouseDown event code calls the EndLineDraw method to complete the current line segment.
- If the Draw Shape tool is selected and a shape is not currently being drawn, the EndLineDraw method records the location of the mouse click, which is used to close the shape when the user double-clicks. It also sets the `bDrawingShape` Boolean variable to `true`, so that future mouse clicks will be handled as above.
- The coordinates of the last mouse click are recorded and the MouseDown event handler returns `true`, indicating that the application is handling the MouseDown event. Alert programmers will realize that this last step is the same as the original MouseDown event handler. So, the original tools should function with no modifications.

Whew! You just gotta love those coding marathon sessions. Anyway, now would be a great time to save and test the project. Test the new Draw Shape tool. Select the tool, and then single-click in the Paint window to start the Draw Shape tool. Move the mouse—don't click and drag—to another location. A line should be drawn between the first click location and the current mouse location. When you single-click again, a new line should begin at the end of the first line.

Repeat the steps above, clicking the mouse and moving to a new location, a few times, and then double-click the mouse. A line should be drawn from the end of the last line to the beginning of the first line.

Test it a few more times, resizing and moving the window to ensure that the DragRefresh functions are working properly. If everything is properly coded, then it should be working without a hitch. If you're having problems, double-check your work, fix any problems, and try again.

# Review

By now you should have a pretty good feel for events, methods, and variables. You've added some of each and should be comfortable with these concepts.

When working on your own projects in the future, you will find that you rely on the REALbasic Language Reference. All the events you've worked with are documented in the Language Reference. Now might be a good time to take a look at some of the objects and events you've used so far. Open the REALbasic Window menu and select Language Reference (or press Command+0) to display the Language Reference window.

Browse a bit in the Language Reference. To search for information about the canvas, type `canvas` in the edit field, and click the Search button; then read up on the object you've been working with for the last few chapters.

If you're so inclined, browse around a bit more and see what you can pick up. Don't be discouraged if you don't understand a lot of what you're reading for now. Understanding will come in time. For now, just get a feel for the layout of the reference.

In later chapters, you'll be adding file operations (such as Save and Open), clipboard operations (such as Cut, Edit, and Paste) and some color and line-width selection tools. Purely for exercise, try to find some information about these subjects in the Language Reference. The practice will do you good.


# 13

# File Operations

## In This Chapter

- The new menu Items
- Closing and creating windows
- Saving to a file
- Opening an existing file
- Printing your pictures

You have a simple drawing program. But what if you want to save those wonderful works of black-and-white art so that you can retrieve them later? You need to add some file operations so that you can save and restore your work.

**NOTE**

Before you get started with the modifications in this chapter, open the last version of the program source code and save it as *My Paint - Step 4*.

# The New Menu Items

Almost all programs that perform some type of editing—be they word processors or paint programs—support at least minimal file operations: New, Open, Close, Save, Save As, and Print. Table 13.1 illustrates what these file menu items are supposed to do.

Clever readers may have noticed something about the menu item descriptions above: Some of the menu items are available whether or not a Paint window is open, while others are available only when one or more Paint windows is open. How does one accomplish this?

Your first instinct might be to try to write some kind of code in the Paint Window EnableMenuItems event to show only those menu items that you need, but you'd be running down the wrong rabbit hole with that approach. Menu items defined in the Paint Window class are automatically disabled when all the Paint windows are closed. Don't believe me? Give it a try. Run the My Paint application as is. Close the one and only Paint window, and check out the Tools menu. All the menu items are disabled, as shown in Figure 13.1.

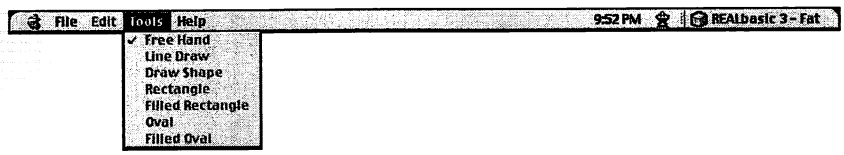



**Figure 13.1**
The My Paint menu with the disabled Tools menu items

**TABLE 13.1 FILE MENU OPERATIONS**

| Menu Item | Behavior |
|---|---|
| New | The menu handler for the New menu item creates a new Paint window, allowing the user to work in more than one window at a time. The New menu item should be available whether or not a Paint window is open. |
| Open | The menu handler for the Open menu item enables the user to select and open a previously saved My Paint file using a standard File Open dialog box. The Open menu item should be available whether or not a Paint window is open. |
| Close | The menu handler for the Close menu item allows the user to close the currently active Paint window. The Close menu item should only be available only if one or more Paint windows is open. |
| Save | The menu handler for the Save menu item enables the user to save the contents of the currently selected Paint window. If the contents of the Paint window have never before been saved, a standard File Save dialog box must be displayed, allowing the user to specify a filename. If the contents have already been saved at least once, then selecting the Save menu item simply updates the saved file to reflect the changes made to that file since the last time it was saved. The Save menu item should be available only if one or more Paint windows is open. |
| Save As | The menu handler for the Save As menu item enables the user to save the contents of the currently selected Paint window. Unlike the Save menu handler, the Save As menu handler will always display the standard Save File dialog box, allowing the user to change the name of the file before it is saved. The Save As menu item should be available only if one or more Paint windows is open. |
| Page Setup | The menu handler for the Page Setup menu item enables the user to change the printer settings for the currently selected Paint window. The Page Setup menu item should be available only if one or more Paint windows is open. |
| Print | The menu handler for the Print menu item enables the user to print the contents of the currently selected Paint window. The Print menu item should be available only if one or more Paint windows is open. |

Adding menu items to the Paint window won't do you any good; they'll all be disabled when the Paint windows are closed. The New and Open menu items need to be available even when there are no open Paint windows. If you can't put the menu items and menu handlers in the Paint Window class, however, you have to put them somewhere else—but where?

## Application-Wide Menu Items

What you need is another class that remains instantiated for the entire life of the application. That means that if the application is running, this object should exist—unlike the Paint Window object for which an instance exists only when at least one Paint window is open on the screen.

You need to create a new class and then change its Super property (its class type) to Application. Creating a class of Super type Application ensures that the instance of that class will always exist when the application is running. Here's how to create that class:

1. Open the Project window (open REALbasic's Window menu and select Project, or press Command+0).
2. Open the REALbasic File menu and select New Class. REALbasic will add a new class to the Project window, with a class name of *Class1*.
3. The new class, Class1, should already be selected, but if it's not, click it to select it and change the following items in the Properties window:
   - Change the Name property to *theApp*.
   - Change the Super property to *Application*.

The Project window and Properties window should appear as shown in Figure 13.2.

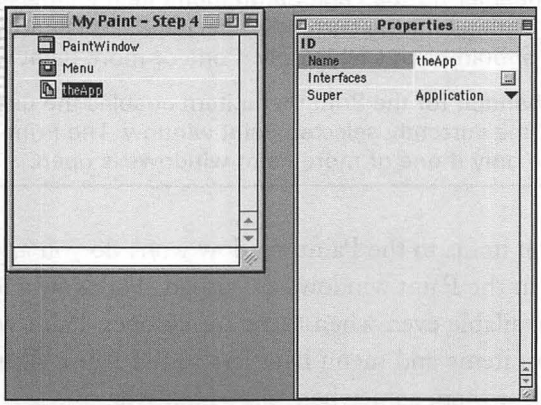



**Figure 13.2**
The new theApp class in the Project window and its associated Properties window

# Today's Menu Items Are . . .

Now that you've created the new Application class, in which the My Paint application's File/New and File/Open menu handlers will reside, all that is left to do for now is to create the menu items. (You'll create the menu handlers later, some in the Paint Window class and some in the Application class.)

To create the new menu items, do the following:

1. Open the Project window (open REALbasic's Window menu and select Project, or press Command+0).
2. Double-click the Menu object. The Application Menu window appears.
3. Click on the File menu item to open it to see the results.
4. Click on the blank menu item and type `New`. Notice that when you begin typing, the cursor moves to the Text field of the Properties window.
5. Tab down to the Command Key field of the Properties window and change its value to `N`. This will enable the user to select the File/New menu item by pressing the Command+N key combination when the My Paint application is running.
6. Repeat steps 3, 4, and 5 for the following menu items:
   - Create an Open . . . menu item with a CommandKey property of `O`.

**NOTE**

Note the ellipsis (. . .) in the Open menu item text; this lets the user know that a dialog box will be displayed when this menu item is selected.

   - Create a Close menu item with a CommandKey property of `W`.
   - Create a Save menu item with a CommandKey property of `S`.
   - Create a Save As . . . menu item with a blank CommandKey property.
   - Create a Page Setup . . . menu item with a blank CommandKey property.
   - Create a Print . . . menu item with a CommandKey property of `P`.

- Click on the blank menu item, type a dash (-), and press the Return or Enter key. This will create a menu separator (the horizontal line that you often see in menus, which is used to group together related menu items).
- Again, click on the blank menu item, type a dash (-), and press the Return or Enter key to create another menu separator.

7. One by one, click on each of the new menu items and drag them up or down the menu to position them in the following order:
   - New
   - Open . . .
   - Close
   - Save
   - Save As . . .
   - Menu separator
   - Page Setup . . .
   - Print . . .
   - Menu separator
   - Quit

After you've rearranged the menu items they should appear as shown in Figure 13.3.

Now that you've created all the new menu items, you probably want to see what they look like. Save your project and run the My Paint application as is. You'll see the new menu items in the File menu, but none of them are activated. We'll take care of that next.

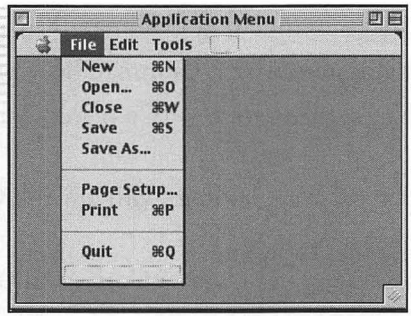
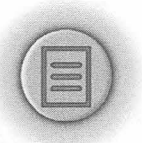



**Figure 13.3**
The REALbasic Application Menu window with the new File menu items arranged in their proper order.



NOTE

Mac OS X automatically hides the Quit item you created, and it inserts a Quit item as the last item in the application menu.

## Enabling the New Menu Items

To enable the various menu items, you add code to the EnableMenuItems event—but which one? If you have already dug around in the Application class, you probably noticed that there is an EnableMenuItems event in that class as well as the one in the PaintWindow class.

Remember when we mentioned that some menu items (Close, Save, Save As, and Print) should be enabled only when a Paint window is open, while others (New and Open) should always be enabled? This comes into play here. You'll enable the first four menu items in PaintWindow's EnableMenuItems event, and the other two in Application's EnableMenuItems event, like so:

1. Click on the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window.
3. Expand the Events item (click its disclosure triangle or double-click Events).
4. Select the EnableMenuItems event and add the following source code to the end of the event's code, as shown in Figure 13.4:

```
FileSave.Enabled = bPictureHasChanged
FileSaveAs.Enable
FileClose.Enable
FilePageSetup.Enable
FilePrint.Enable
```

5. Click on the Application class in the Project window.
6. Press Option+Tab to display the Code Editor window.
7. Expand the Events item (click its disclosure triangle or double-click Events).
8. Select the EnableMenuItems event and enter the following source code, as shown in Figure 13.5:

```
FileNew.Enable
FileOpen.Enable
```

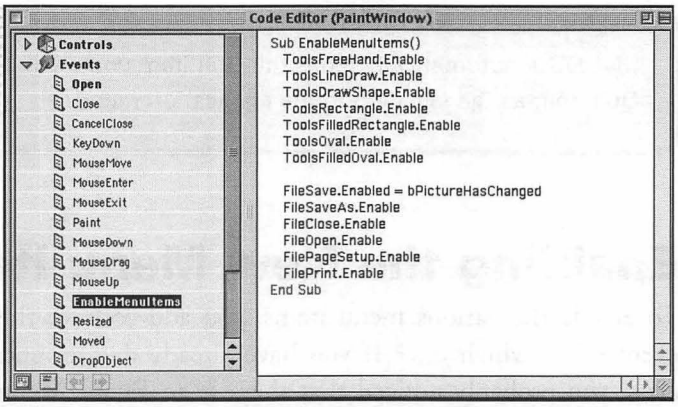



**Figure 13.4**
The Code Editor window for the PaintWindow class's EnableMenuItems event

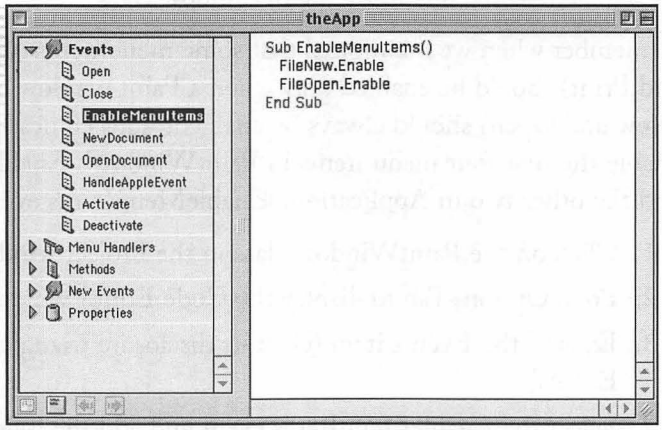



**Figure 13.5**
The Code Editor window for the Application class's EnableMenuItems event

You probably noticed in the first section of the preceding code that the File/Save menu item is enabled based on the value contained in the `bPictureHasChanged` property. You'll need to add this property to the PaintWindow class:

1. Click on the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window.
3. Open the Edit menu and select the New Property command (or press Option+Command+P).
4. In the New Property dialog box, enter `bPictureHasChanged As Boolean` as the new property, as shown in Figure 13.6.
5. Click the OK button to close the New Property dialog box and save the new property.

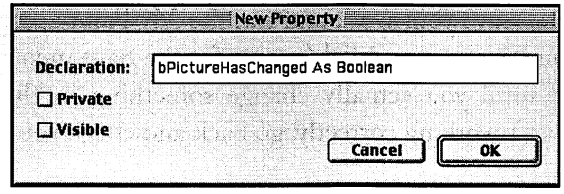



**Figure 13.6**
The New Property dialog box with the `bPictureHas Changed` property definition

You don't need to initialize the value of the `bPictureHasChanged` property. Recall that Boolean variables are automatically given an initial value of `false`, which just so happens to be the value to which you want it to be initially set. Having a value of `false` will cause the Save menu item to be disabled by default each time a new PaintWindow class is created.

That said, you will need to change the value of `bPictureHasChanged` when the contents of the PaintWindow window have changed:

1. Expand the Controls item (click its disclosure triangle or double-click Controls).
2. Expand the PaintCanvas item (click its disclosure triangle or double-click PaintCanvas).
3. Select the MouseDrag event.
4. And add the following source code to the end of the MouseDrag event's source code, as shown in Figure 13.7:

```
bPictureHasChanged = ((nCurrentTool > 0) and (nCurrentTool < 8))
```

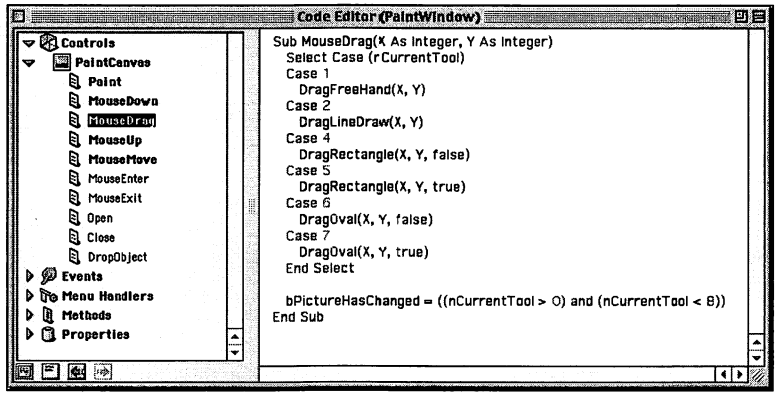



**Figure 13.7**
The Code Editor window for the MouseDrag event

Go ahead and save your work and run the My Paint application to make sure all the menu items are enabled (except for the Save menu item, which should be disabled until you actually change something in the Paint window). If something isn't working correctly, go back and check your work and try again.

# Closing and Creating Windows

Closing windows (with the File/Close menu item) and creating new windows (with the File/New menu item) are related tasks, so let's tackle them together. The code for closing windows is absurdly simple so we'll do that first, just to get it out of the way. After completing the last task, the Code Editor for the PaintWindow should still be open. If it's not, click on the PaintWindow class in the Project window and press Option+Tab. Now do the following:

1. Open the Edit menu and select the New Menu Handler command (or press Option+Command+H).
2. Select FileClose from the Menu Item list and click OK.
3. Enter the following in the Code Editor window for the FileClose menu handler, as shown in Figure 13.8:

```
Close
```

All that happens with the FileClose menu handler, for now, is that the PaintWindow class Close method is called. From there, the PaintWindow class takes over. You'll add to this function later, but this is all it needs to do for now.

The code for creating a new window is only a little more complicated than the code for closing windows. You added the code for closing windows in

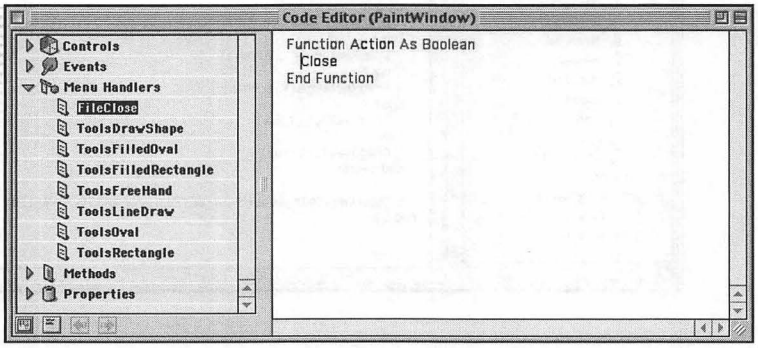



**Figure 13.8**
The Code Editor window for the incredibly simple FileClose menu handler

the PaintWindow class, because the Close menu handler is needed only when a window is open. You add the code for creating new windows in the Application class, because you need to be able to create new windows even if no other windows are open. To add the code for creating new windows, do the following:

1. Click on the Application class in the Project window.
2. Press Option+Tab to display the Code Editor window.
3. Open the Edit menu and select the New Menu Handler command (or press Option+Command+H).
4. Select FileNew from the Menu Item list and click OK.
5. Enter the following in the Code Editor window for the FileNew menu handler, as shown in Figure 13.9:

```
Dim w As PaintWindow
w = New PaintWindow
```

Even though the preceding code looks simple, what happens behind the scenes is what really matters:

- The first statement defines a new variable, which is a reference to a PaintWindow object. When the variable is defined, it has an initial value of nil, meaning that the variable is not yet pointing to an instance of the PaintWindow class.
- The second statement creates a new instance of the PaintWindow object and saves a reference to the object in the variable that was defined in the first statement.

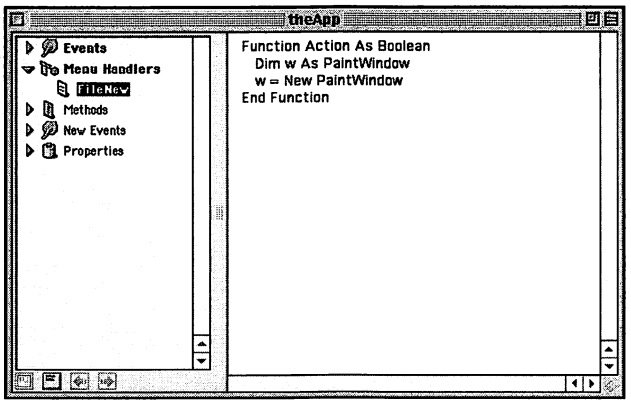



**Figure 13.9**
The Code Editor window for the Application class's FileNew menu handler

When the new instance of the PaintWindow object is created, its Open event handler is called automatically and the window is displayed. These two simple lines do so much work!

Go ahead and save your work and test the changes you just made. Run the My Paint application. You should be able to create new windows by using the File/New menu item or by pressing Command+N. (Don't create too many windows, or the app will run out of memory. Just create one or two and then close them.) You should be able to close windows by using the File/Close menu, by clicking the window's Close button to the left of the title bar, or by pressing Command+W.

If you ever have problems debugging a REALbasic application due to memory constraints in an operating system other than Mac OS X, you can always increase the memory allocated to the REALbasic application. To do so, follow these steps:

1. Locate the REALbasic application icon and click it.
2. Open the File menu, select Get Info, and choose the Memory command, or press Command+I and select Memory from the Show pop-up menu.
3. Increase the value of the Preferred Size field (see Figure 13.10) until the REALbasic application has enough memory to debug your code.

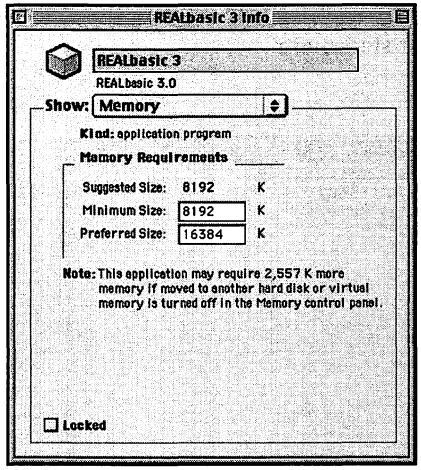



**Figure 13.10**
The REALbasic Info window's memory settings

# Saving to a File

All these modifications do a fat lot of good if you still can't save your paintings to disk. To do so, you'll need to add the code to the FileSave menu handler which will save the contents of the picBuffer property to a file. You'll also need to add code to the FileSaveAs menu handler to allow the user of the application to save the file with a new name.

## Adding Supported File Types

Before you do anything, you must first add the supported *file types* to your REALbasic application. File types help the Finder determine what kind of document is in use in the Mac OS and which application should be used to open it. On a Windows PC, this function is normally handled using the file extension at the end of the file's name, such as .doc for a Microsoft Word document, or .jpg for a JPEG graphic file. In the Mac OS, file type information is stored in the file's resource fork.

My Paint can open, modify, and save files in PICT, the graphic file format supported directly by Mac OS 9. You're probably familiar with it; PICT files are created when you take a picture of the Mac OS screen using Command+Shift+3. We'll need to add PICT to the list of file types supported by your app. Do the following:

1. Open the Edit menu and select the File Types command. The File Types dialog box, shown in Figure 13.11, opens.

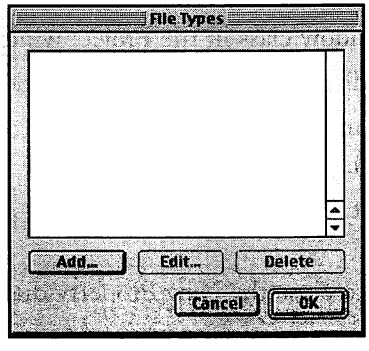



**Figure 13.11**
The File Types dialog box

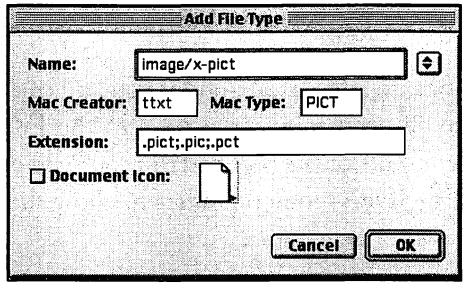



**Figure 13.12**
The Add File Type dialog box with the new file-type definition

2. Click the Add button to open the Add File Type dialog box.
3. Choose image/pict from the Name drop-down list, as shown in Figure 13.12.
4. Click the OK button to close the Add File Type dialog box.
5. Click the OK button to close the File Types dialog box and to save the new file type.

# The Filename Property

Ready to add that code yet? Hold on. Before you add the code to save to a file, you need to add a property to the PaintWindow class that will contain the filename and location. This property will be used by the FileSave menu handler to determine whether the document has been saved, and to set the default filename during a Save As operation. This will become more clear to you when you see how the code works.

First, let's add that property:

1. Click on the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window.
3. Open the Edit menu and select the New Property command (or press Option+Command+P).
4. Enter `fiPaintDocument as FolderItem` as the new property, as shown in Figure 13.13.
5. Click the OK button to close the New Property dialog box and save the new property.

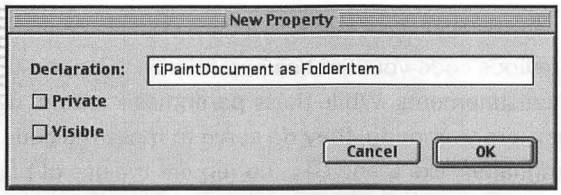



**Figure 13.13**
The New Property dialog box with the `fiPaintDocument` property definition

# The FileSave Menu Handler

Now that you've taken care of the filename property, you can create the FileSave menu handler. It's not very complicated code, so just go ahead and create it now; we'll explain how it works after you've coded the handler. Follow these steps to create the FileSave menu handler:

1. Click on the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window.
3. Open the Edit menu and select the New Menu Handler command (or press Option+Command+H).
4. Select FileSave from the Menu Item list and click OK.
5. Enter the following in the Code Editor window for the FileSave menu handler, as shown in Figure 13.14:

```
dim fiSaveLocation as FolderItem

if (fiPaintDocument = nil) then
  fiSaveLocation = GetSaveFolderItem ("image/pict", self.Title)
  if (fiSaveLocation <> nil) then
    self.Title = fiSaveLocation.Name
    fiPaintDocument = fiSaveLocation
  End if
End if

if (fiPaintDocument <> nil) then
  fiPaintDocument.SaveAsPicture picBuffer
  bPictureHasChanged = false
End if
```

**TIP**

In the previous code you'll notice parenthesis () around the conditions in the If/Then statements. While these parenthesis are not needed in REALbasic programming, they do serve to make the code easier to read. Other languages, like C and C++, do require the use of parenthesis. So as to not get into religious programming debates about the merits of using or not using parenthesis, let it be said that some programmers prefer to use them, and others do not. Choose whichever method you're most comfortable with. Whichever method you choose, try to be consistent. Consistent coding, above all, will make your code much easier to read.

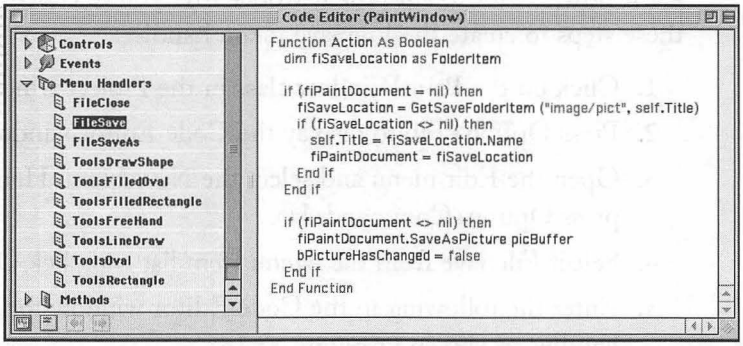



**Figure 13.14**
The Code Editor window for the FileSave menu handler

This code first creates a new FolderItem object, `fiSaveLocation`, to be used later. If the document has never before been saved (that is, if the `fiPaintDocument` object is nil), then a standard File Save dialog box is displayed. When the File Save dialog box is closed, the file save location is copied to the `fiSaveLocation` object and, as long as the user doesn't cancel the save, the PaintWindow's title is updated and the file save location is copied to the `fiPaintDocument` property.

Last, but definitely not least, the `fiPaintDocument` property is checked to see if it's nil. If it's not nil, it's because it wasn't nil when the menu handler started, or because it was changed when the File Save dialog box was displayed. In either case, the contents of the `picBuffer` property are saved to the file location specified by the `fiPaintDocument` property. The `bPictureHasChanged` property is then set to a value of `false`, so that the app knows the file has not changed since the last time it was saved.

Simple huh? Well, don't sweat it too much if you don't get it. Now would be a good time to save your work and test the app. Go ahead and do so before we move on.

# The FileSaveAs Menu Handler

Did all of your changes above work? If so, good. If not, double-check your code and try again. Hopefully you've got the hang of what's going on. If so, you can probably predict what the FileSaveAs menu handler is going to look like. Make sure that the Code Editor for the PaintWindow is open (click on the PaintWindow class in the Project window and press Option+Tab) and then follow these steps to create the FileSaveAs menu handler:

1. Open the Edit menu and select the New Menu Handler command (or press Option+Command+H).
2. Select FileSaveAs from the list and click OK.
3. Enter the following in the Code Editor window for the FileSaveAs menu handler, as shown in Figure 13.15:

```
dim fiSaveLocation as FolderItem

fiSaveLocation = GetSaveFolderItem ("image/pict", self.Title)
if (fiSaveLocation <> nil) then
  Title = fiSaveLocation.Name
  fiPaintDocument = fiSaveLocation

  fiPaintDocument.SaveAsPicture picBuffer
  bPictureHasChanged = false
End if
```

This code is almost identical to that of the FileSave menu handler, except that the original filename isn't checked first to determine if the File Save dialog box should be displayed. The File Save dialog box is always displayed so that the user can save the current file with a different filename.

Go ahead and save the REALbasic project and test the File/Save As modifications that you just made. You should be able to save your painting with the File/Save and the File/Save As menu items. When you use the File/Save menu item, you should be prompted for the filename only the first time the file is saved. You should be prompted for the filename every time you use the File/Save As menu item.

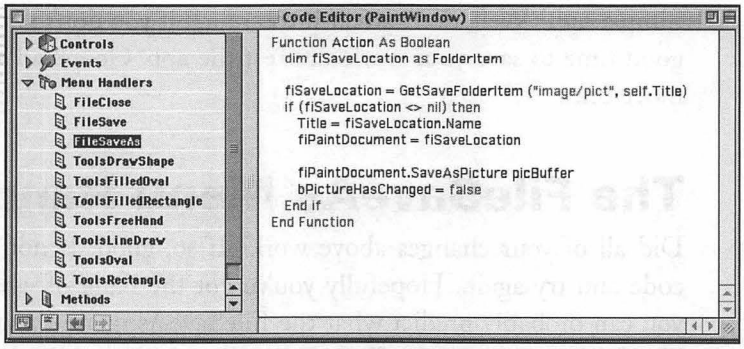



**Figure 13.15**
The Code Editor window for the FileSaveAs menu handler

Now that all that file-saving stuff is behind you, let's move on to the File/Open menu item so that you can retrieve the documents you've been saving.

# Opening an Existing File

The process of opening an existing file is simple enough: Prompt the user for a filename and location, and open the file. The code to do so is pretty simple, but keep in mind that the user should be able to open a file, regardless of whether a Paint window is open. Do the following:

1. Click on the Application class in the Project window.
2. Press Option+Tab to display the Code Editor window.
3. Open the Edit menu and select the New Menu Handler command (or press Option+Command+H).
4. Select FileOpen from the Menu Item list and click OK.
5. Enter the following in the Code Editor window for the FileOpen menu handler, as shown in Figure 13.16:

```
Dim fiOpenDocument as FolderItem
Dim pwNewWindow as PaintWindow
fiOpenDocument = GetOpenFolderItem ("image/pict")
If (fiOpenDocument <> nil) then
  pwNewWindow = new PaintWindow
  pwNewWindow.picBuffer = fiOpenDocument.OpenAsPicture
  pwNewWindow.Title = fiOpenDocument.Name
End if
```

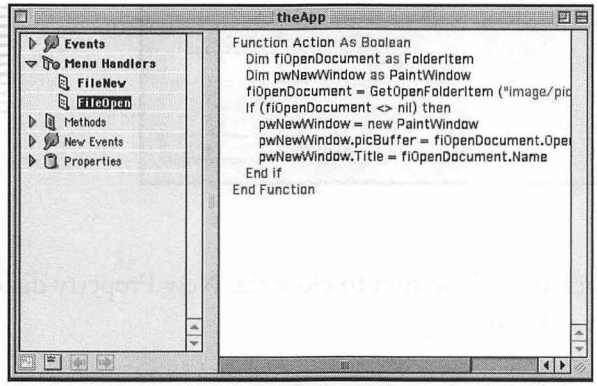



**Figure 13.16**
The Code Editor window for the FileOpen menu handler

This code simply prompts the user for an existing document to open. If the user chooses one, a new PaintWindow object is created, the contents of the specified file are copied into it, and the new Paint window's title is updated.

Like always, save your work and try out the code.

# Printing Your Pictures

Printing a document is a bit more complicated than the rest of the file operations. First off, you have to allow for the user to choose a specific page setup for each document he or she prints.

## Adding the PageSetup Property

When the user opens the File menu and chooses the Page Setup command, their page-setup selections must be saved for use when the File/Print menu item is chosen. The property you'll add here is used to save these page-setup properties. Do the following:

1. Click on the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window.
3. Open the Edit menu and select the New Property command (or press Option+Command+P).
4. Enter `strPageSetup as String` as the new property, as shown in Figure 13.17.

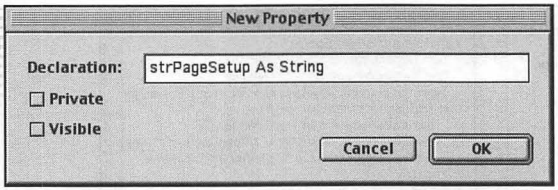



**Figure 13.17**
The New Property dialog box with the `strPageSetup As String` property definition

5. Click the OK button to close the New Property dialog box and save the new property.

## Adding the PageSetup Menu Handler

Now that the page-setup property exists, you'll need to create the Page Setup menu handler. With the Code Editor for the PaintWindow open, here's what to do:

1. Open the Edit menu and select the New Menu Handler command (or press Option+Command+H).
2. Select FilePageSetup from the Menu Item list.
3. Enter the following in the Code Editor window for the FilePageSetup menu handler, as shown in Figure 13.18:

```
Dim psPrinterSetup As PrinterSetup

psPrinterSetup = new PrinterSetup
if (strPageSetup <> "") then
  psPrinterSetup.SetupString = strPageSetup
End if
if (psPrinterSetup.PageSetupDialog) then
  strPageSetup = psPrinterSetup.SetupString
End if
```

The PageSetup menu handler first creates a new `PrinterSetup` object. If a previous printer setup has already been specified, then this value, `strPageSetup`, is copied to the instance of the `PrinterSetup` object's `SetupString` property.

The menu handler then displays the Page Setup dialog box . If the user clicks the OK button, then the new page-setup preferences are copied to the `strPageSetup` dialog, so that they can be used later by the Print menu handler.

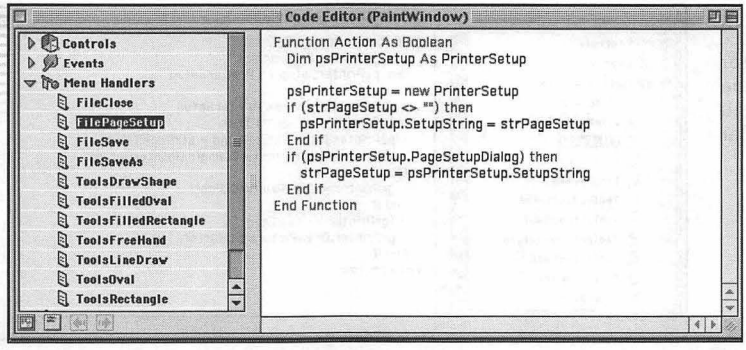



**Figure 13.18**
The Code Editor window for the FilePageSetup menu handler

# Adding the Print Menu Handler

Now that you've set down the building blocks, you're ready to add the actual Print menu handler. While still working in the PaintWindow Code Editor, follow these instructions:

1. Open the Edit menu and select the New Menu Handler command (or press Option+Command+H).
2. Select FilePrint from the list.
3. Enter the following in the Code Editor window for the FilePrint menu handler, as shown in Figure 13.19:

```
Dim grPrinter As Graphics
Dim psPrinterSetup As PrinterSetup

psPrinterSetup = new PrinterSetup
if (strPageSetup <> "") then
  psPrinterSetup.SetupString = strPageSetup
  grPrinter = OpenPrinterDialog(psPrinterSetup)
else
  grPrinter = OpenPrinterDialog()
end if
if (grPrinter <> nil) then
  grPrinter.DrawPicture picBuffer, 18, 18
  grPrinter.NextPage
end if
```

The Print menu handler builds upon all the previous work to create the ability to print your documents. It begins by defining `Graphics` and `PrinterSetup`

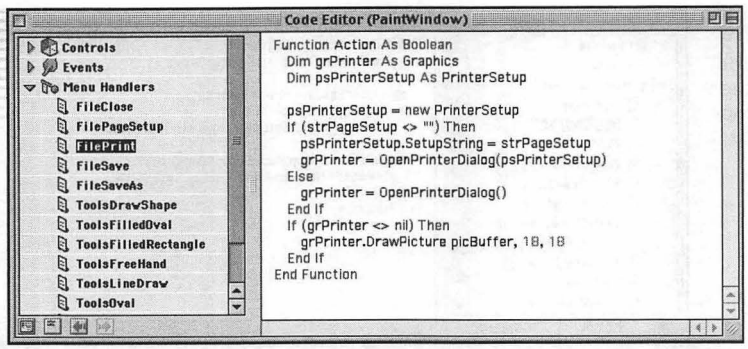



**Figure 13.19**
The Code Editor window for the FilePrint menu handler

objects, which are used later in the menu handler. The menu handler creates an instance of a `PrinterSetup` object; if a previous page-setup string, `strPageSetup`, exists, it copies the previous setup values to the current printer setup. It then creates the instance of the `grPrinter` graphics object, using the current printer setup values. If a previous printer setup wasn't selected by the user, an instance of the `grPrinter` graphics object is created, using the default printer setup values.

If either method of creating the instance of the `grPrinter` graphics object succeeds, then the contents `picBuffer` will be drawn to the `grPrinter` graphics object.

Now would be a good time to save that work, as always, and test your modifications. Make sure everything works, and double-check your work if you have any problems.

# Review

In this chapter you learned how to add menu items that are smart enough to appear and disable themselves as needed so users won't be confused as they open, save, and print their work. These are common to virtually every Macintosh application so return to this chapter again as you build your own applications to refresh your memory.

Some developers have created variations on "Save As," such as "Save a Copy." You may find your app may have a need for such a command if you want to ensure that a user's document will not be changed accidentally or should you want to provide other file management options. Just remember: In a Macintosh application, less is more. Don't overdo the features of your application.


# 14

# Editing Operations

## In This Chapter

- Working with the clipboard
- The new source code
- The paste feature
- The copy feature
- The clear and cut features

Your little paint program is coming along quite well. You have controls for different drawing tools, and can even save, open, and print files. Although what you have is a perfectly serviceable application, there is one important common feature that you need to consider: editing.

Almost all editing programs, whether they allow the user to edit text or graphics, have the capability to copy, cut, and paste information to and from the clipboard. The *clipboard*, in case you're not aware, is a portion of the computer's memory that has been set aside to allow programs to copy information to other locations within the same document, multiple documents within the same application, or documents within multiple applications. Figure 14.1 illustrates the use of the clipboard.

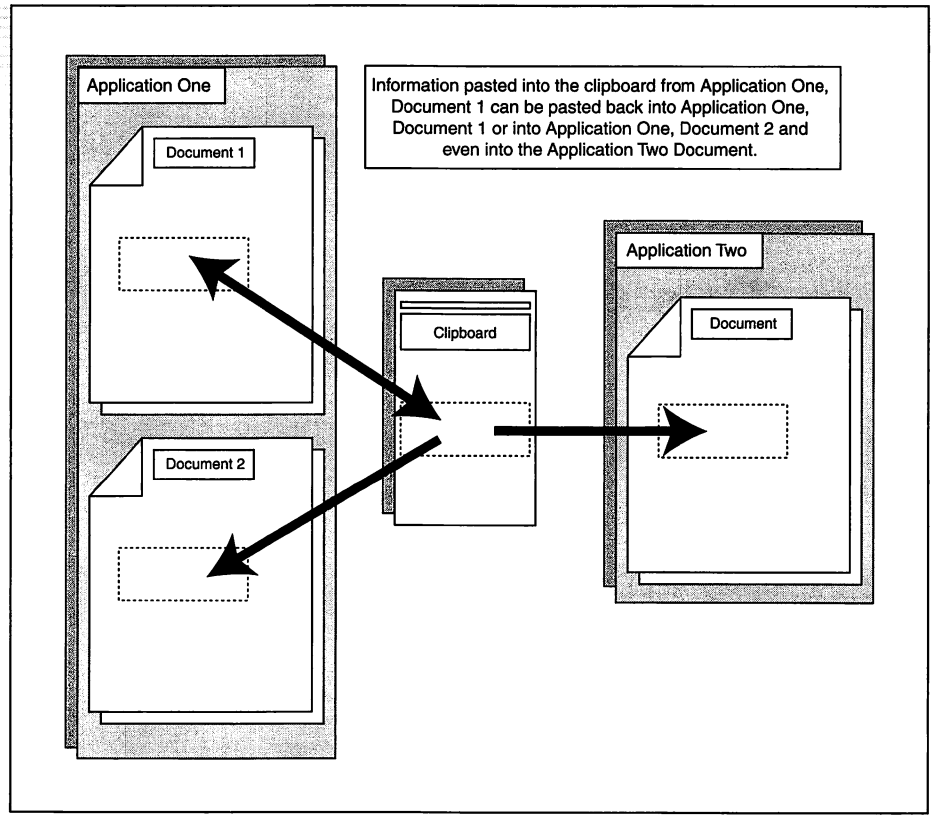



**Figure 14.1**
A simple representation of the clipboard operation

# Working with the Clipboard

Any type of data can be copied and pasted to and from the clipboard, but the two most common types are text and graphics. REALbasic allows access to the clipboard using the clipboard class, shown here:

```
Dim c as Clipboard
c = new Clipboard
```

When an instance of a clipboard object is created, the application can then call members of this class to perform various operations related to the clipboard. For example, to copy text to the clipboard, you'd do something like the following:

```
Dim c as Clipboard
c = new Clipboard
If (c <> nil) Then  // the memory for the class might not have ➥
been available
  c.text = "All your base are belong to us."
c.close  // you have to close the clipboard, or an error will occur
End If
```

This code first attempts to create an instance of the clipboard class and makes sure it was created successfully. If the clipboard object was created, then the value of the clipboard's text property is set to the string specified. The clipboard is then closed. Closing the clipboard actually copies the data from the clipboard class to the actual system clipboard. If you don't close the clipboard within the method or event handler in which the clipboard object was instantiated, an error will occur.

If you want to check for data in the clipboard, and copy the data to a variable within your application, you'd do something like this:

```
Dim c as Clipboard
Dim str as String
c = new Clipboard
If (c <> nil) Then  // the memory for the class might not have ➥
been available
  If (c.TextAvailable) Then
    str = c.text
```

```
    End If
    c.close  // you have to close the clipboard, or an error will ➥
    occur
End If
```

As you can see, copying data to and from the clipboard in REALbasic is relatively easy. The REALbasic clipboard class also provides methods for copying picture data as well. Using the PictureAvailable and Picture properties allows the application to copy pictures to and from the clipboard just as easily as text.

# The Edit Menu Items

Fortunately, you don't really need to do anything when it comes to the Edit menu items, which are shown in Figure 14.2. The REALbasic project editor already includes all the standard menus needed to perform all the basic editing functions. You might want to double-check the menu items, however, just to be sure that they're still there, in case you accidentally deleted one or two of them.

If any of the Edit menu items is missing, or out of order, correct the problem now before moving on. To edit the menu items, double-click the Menu icon on the REALbasic Project window. From there, just drag and drop property settings. If the Properties window isn't visible, enable it by opening the Window menu and selecting Show Properties, or by pressing Command+Option+2.

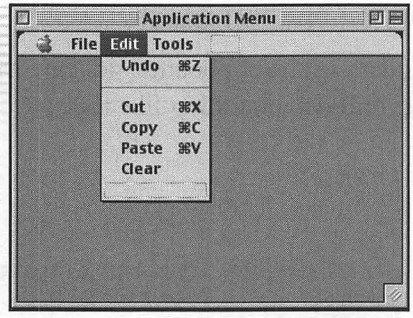



**Figure 14.2**
The standard Edit menu items

# The New Source Code

The code you'll be adding for the menu items needs to support the Edit menu items: Copy, Clear, Cut (which is just a copy followed by a clear), and Paste. Adding an Undo option is something we'll leave for you to figure out on your own.

You'll also add a new control to the Paint window, which will be used to handle pasting a picture into the window. Adding a control will enable you to drag the pasted-in picture around before the application copies it to the Paint window. You'll be adding event handlers to the new control to allow for the dragging feature to work.

As far as changes to the PaintWindow class itself, you'll be enabling the Edit menu items, adding methods for selecting areas of the paint window, copying to the clipboard, clearing a portion of the picture, and pasting from the clipboard. You'll also be modifying some existing methods and event handlers. Lastly, you'll be adding the menu handlers for the Edit menu items.

Before you get started, open the Step 4 version of the My Paint REALbasic project, and save it as *Step 5* before moving on.

# The New Properties

First, add some properties to the PaintWindow class, which will be used by the new methods, menu handlers, and event handlers that you'll be adding to the PaintWindow class.

To add the menu items, select PaintWindow in the Project window, and press Option+Tab to display the Code Editor window. Then open the Edit menu and select the New Property item, or press Command+Option+P to add each of the new properties described in Table 14.1. Figure 14.3 shows the resulting Code Editor window.

**TABLE 14.1 THE NEW PROPERTIES FOR THE EDIT FUNCTIONS**

| Property Definition | Description |
|---|---|
| `nPasteDragLastX As Integer` | Used during the paste functions while dragging the pasted image. This, along with the `nPastDragLastY` property, will be used as the location where the pasted-in image should be dropped. |
| `nPasteDragLastY As Integer` | Also used during the paste dragging functions. |
| `nLastMouseX As Integer` | Used to keep track of the last mouse position, so that you know where to begin pasting the image in the clipboard. |
| `nLastMouseY As Integer` | Used along with the `nLastMouseX` property. |
| `bSelectionAvailable As Boolean` | This property will contain a value of `true` when the user has selected an area of the picture using the selection tool. |

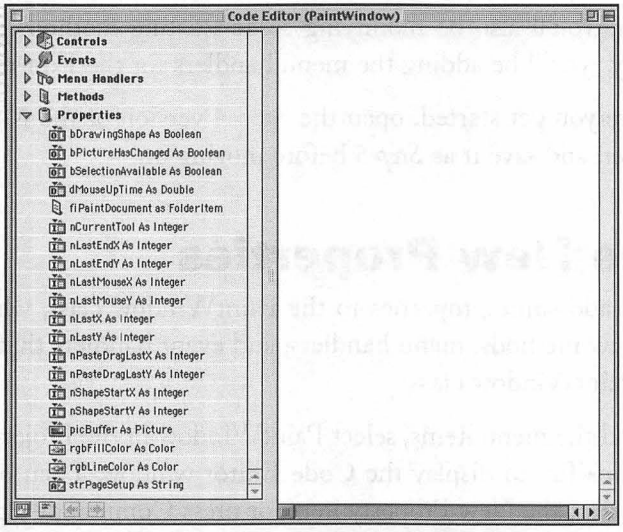



**Figure 14.3**
Look at all the properties! The five new properties have added to the ever growing list of properties for the PaintWindow class.

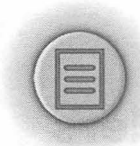
# The Paste Feature

We briefly mentioned that you'd need to add a new control to handle the capability to drag the pasted-in pictures around on the Paint window before copying the pasted-in data to the Paint window. This new control, called PasteCanvas, will act as the repository of the data that is pasted in.

# Adding the PasteCanvas Control

To add the PasteCanvas control, do the following:

1. If the REALbasic project Window isn't visible, open it by opening the Window menu and selecting the Project item, or by pressing Command+0 (numeral zero).
2. Double-click the PaintWindow class to display the window editor.
3. Drag a paint-canvas control (the control that has a picture of a blue sky on it) from the Tools palette to the Paint window, as shown in Figure 14.4.



**NOTE**

You can put the paint-canvas control anywhere on the Paint window you like; its size and position doesn't matter because it will be resized and moved depending on what's being pasted.

4. Change the name of the control to *PasteCanvas* and disable the Visible property. Figure 14.5 shows the properties of the PasteCanvas control.

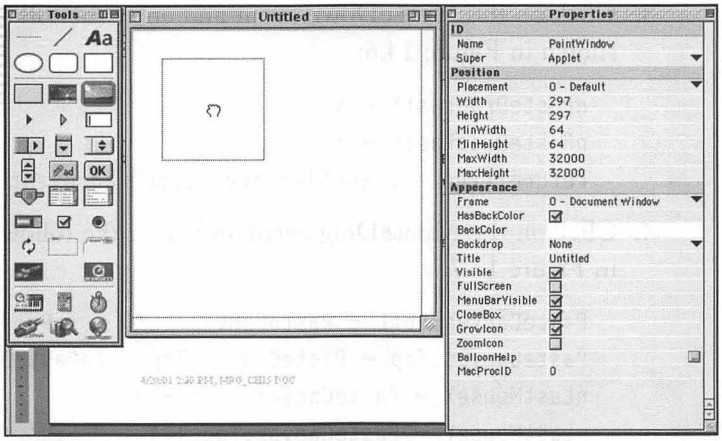



**Figure 14.4**
The new paint-canvas control being dragged onto the PaintWindow Window Editor.

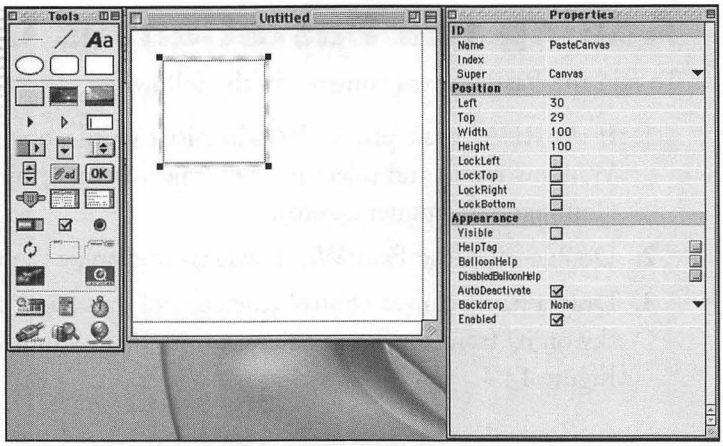



**Figure 14.5**
The properties of the PasteCanvas control

# The PasteCanvas Event Handlers

Whenever a picture is pasted into the My Paint application, it will first be copied into the PasteCanvas control. That way, the user will be able to drag the picture around to position it where she wants it to be. That means you must add the code for the event handlers that allow the PaintCanvas window to be dragged around. To do so, follow these steps:

1. If it is not already selected, click on the PasteCanvas control to select it.
2. Press Option+Tab to display the Code Editor window for the PasteCanvas control.
3. Click on the MouseDown event and enter the following code, as shown in Figure 14.6:

```
nPasteDragLastX = X
nPasteDragLastY = Y
return true  // Enables the MouseDrag event
```

4. Click on the MouseDrag event and enter the following code, as shown in Figure 14.7:

```
PasteCanvas.Left = PasteCanvas.Left - (nPasteDragLastX-X)
PasteCanvas.Top = PasteCanvas.Top - (nPasteDragLastY-Y)
nLastMouseX = PasteCanvas.Left + X
nLastMouseY = PasteCanvas.Top + Y
```

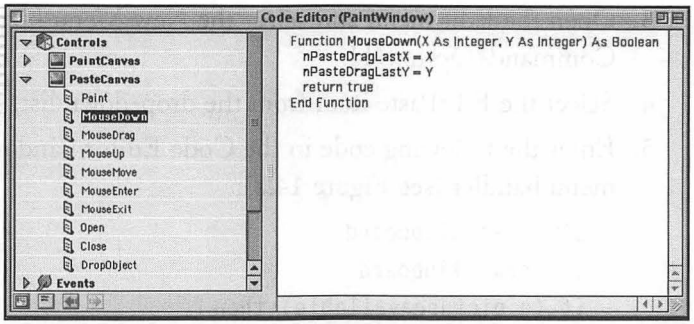



**Figure 14.6**
The code for the PasteCanvas control's MouseDown event

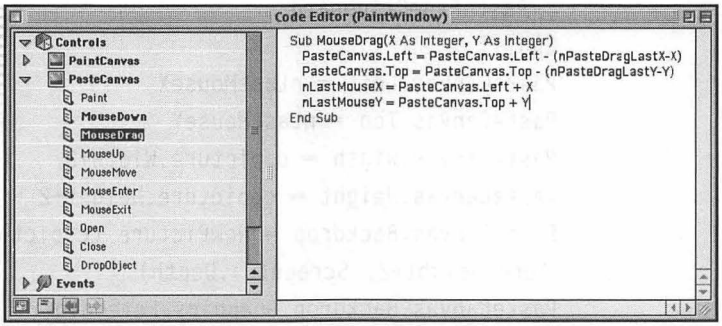



**Figure 14.7**
The code for the PasteCanvas control's MouseDrag event

# The Edit/Paste Menu Handler

As we said above, the Edit/Paste menu handler doesn't actually paste an image from the clipboard into the PaintCanvas. Instead, it pastes the image from the clipboard into another control, the PasteCanvas control, and makes the control visible so the user can drag the pasted image around before actually pasting it to the PaintCanvas control.

To make this possible, you'll need to add code that will first check whether there is an image available in the clipboard. If an image is available on the clipboard, the size, location, and content of the PasteCanvas control will be changed to match that of the clipboard before making the control visible.

To add the new Edit/Paste menu handler, do the following:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.

3. Open the Edit menu and select the New Menu Handler item, or press Command+Option+H.
4. Select the EditPaste item from the drop-down list, and click OK.
5. Enter the following code in the Code Editor window for the EditPaste menu handler (see Figure 14.8):

```
dim c as clipboard
c = new Clipboard
if (c.pictureavailable) then
  if (PasteCanvas.Visible) then
    PasteFromClipboard
  End if
  PasteCanvas.Left = nLastMouseX
  PasteCanvas.Top = nLastMouseY
  PasteCanvas.Width = c.picture.Width+2
  PasteCanvas.Height = c.picture.Height+2
  PasteCanvas.Backdrop = NewPicture (c.picture.Width+2, ➥
c.picture.Height+2, Screen(0).Depth)
  PasteCanvas.Backdrop.Graphics.ForeColor = RGB(192,192,192)
  PasteCanvas.Backdrop.Graphics.DrawRect 0, 0, ➥
c.picture.Width+1, c.picture.Height+1
  PasteCanvas.Backdrop.Graphics.DrawPicture c.picture, 1, 1
  PasteCanvas.Visible = true
  SetMenuSelection (9)
end if
c.close
```

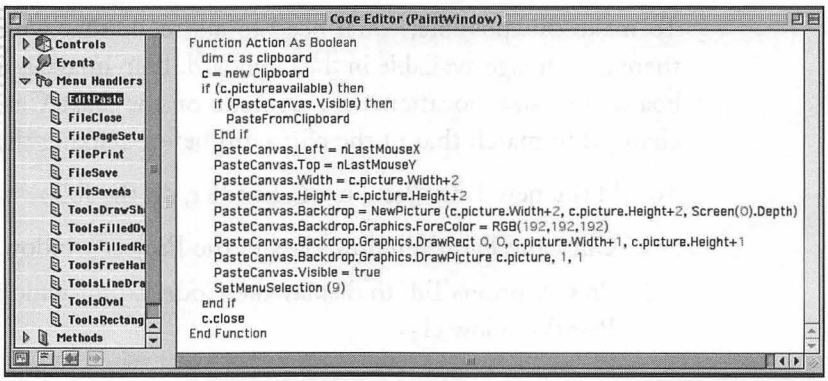



**Figure 14.8**
The code for the PaintWindow/EditPaste menu handler

# The PasteFromClipboard Method

It is in the PasteFromClipboard method that the clipboard image is actually copied from the PasteCanvas control to the PaintCanvas control. To add the code for this method, do the following:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Open the Edit menu and select the New Method item, or press Command+Option+M.
4. Enter `PasteFromClipboard` as the new method name, and click the OK button.
5. Enter the following code in the Code Editor window for the PasteFromClipboard method, as shown in Figure 14.9:

```
    dim nPasteX,nPasteY,nPasteW,nPasteH as integer
    PasteCanvas.Visible = false
    nPasteX = PasteCanvas.Left+1
    nPasteY = PasteCanvas.Top+1
    nPasteW = PasteCanvas.Width-2
    nPasteH = PasteCanvas.Height-2
    picBuffer.Graphics.DrawPicture PasteCanvas.Backdrop, ➥
nPasteX, nPasteY, nPasteW, nPasteH, 1, 1, nPasteW, nPasteH
    PaintCanvas.refreshRect(0, 0, paintCanvas.width, ➥
paintCanvas.height)
```

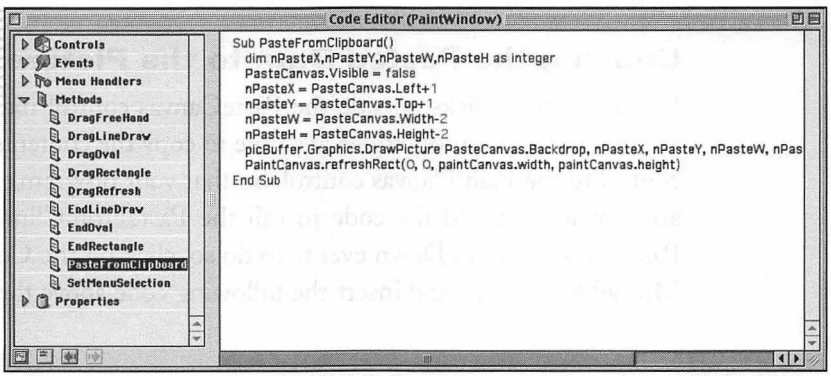



**Figure 14.9**
The code for thePaintWindow/ PasteFromClipboard method

# The PaintCanvas Paste Events

To actually paste information into the picture, the application has to handle three events:

- It must keep track of the last-known mouse position so that it knows where to paste into the picture.
- It must have a handler for the mouse-click event (in the PaintCanvas control outside of the PasteCanvas control) to know when to actually copy the data to the picture.
- It needs to be able to enable the Edit menu items as necessary.

## Keeping Track of the Last-Known Mouse Position

So that it knows where the user is going to attempt to paste, your application must keep track of the last-known position of the mouse. This will allow the program to paste the image into the picture right where the mouse is when the user presses the Command+V shortcut for paste. All you need to do is add code to the PaintWindow class's MouseMove event handler:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Expand the Controls item and the PaintCanvas item.
4. Click on the Controls/PaintCanvas/MouseMove event.
5. Add the following code, as shown in Figure 14.10:

```
nLastMouseX = X
nLastMouseY = Y
```

## Copying the Pasted Data to the Picture

When the user clicks outside the PasteCanvas control that was being dragged around, the application knows it's time to copy the contents of the PasteCanvas control to the PaintCanvas control. So that your program can handle this operation, you must add the code to call the PasteFromClipboard method in the PaintCanvas/MouseDown event. To do so, click on the Controls/PaintCanvas/MouseDown event and insert the following code above the last `End If`:

```
elseif (nCurrentTool = 8) then
  paintCanvas.refreshRect(0, 0, paintCanvas.width, ➥
  paintCanvas.height)
elseif (nCurrentTool = 9) then
  PasteFromClipboard
  nCurrentTool = 8
```

After you add this code, the code for the PaintWindow/PaintCanvas/MouseDown event handler will look like the code in Figure 14.11.

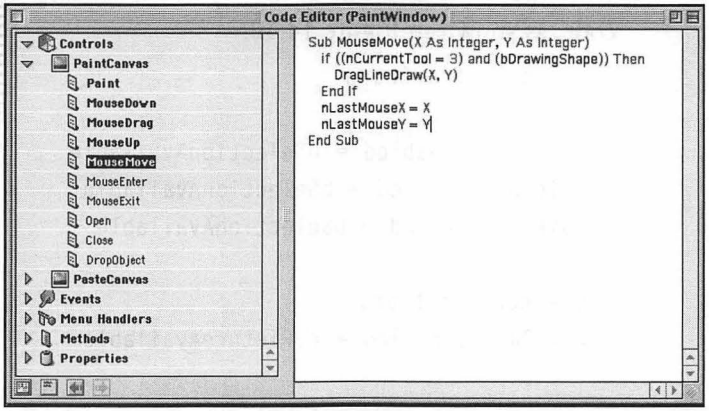



**Figure 14.10**
The code for the PaintWindow/PaintCanvas/Mouse Move event

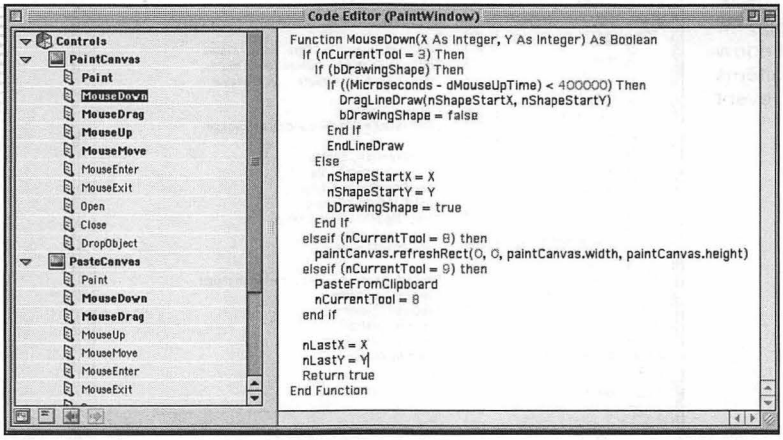



**Figure 14.11**
The code for the PaintWindow/PaintCanvas/MouseDown event

## Enabling the Menu Items

Now it's time to add the code to enable all the menu items. More importantly, you'll be enabling the Paste menu item, so that you can test it after the rest of its coding is complete. The Edit/Paste menu item is enabled only if there is picture data available in the clipboard, which is what the PictureAvailable check in the following code is for. Do the following:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Expand the Events item and click on the EnableMenuItems event and enter the following code at the top of the EnableMenuItems source code, as shown in Figure 14.12:

```
dim c as Clipboard

EditClear.Enabled = bSelectionAvailable
EditCopy.Enabled = bSelectionAvailable
EditCut.Enabled = bSelectionAvailable

c = new Clipboard
EditPaste.Enabled = c.PictureAvailable
```

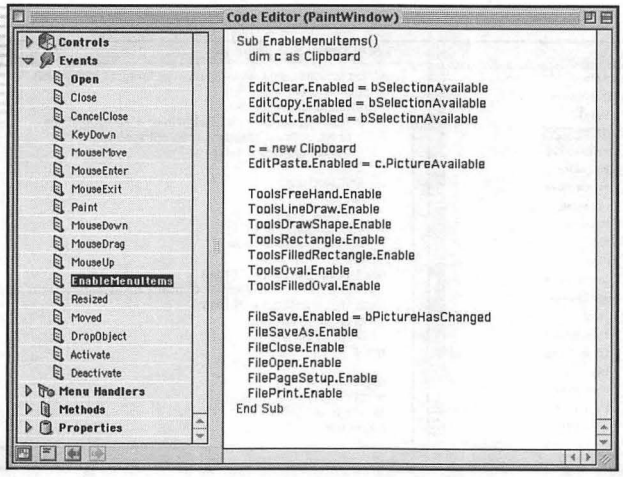



**Figure 14.12**
The new code for the PaintWindow EnableMenuItems event

## Testing the Paste Function

All the coding for pasting into the My Paint program is complete, so you should now be able to save and test your changes. If you have access to some other paint program, then copy a portion of a picture to the clipboard in the other program, and try to paste it into the My Paint application. If you don't have another paint program, you'll have to test the Paste functions after you write the Copy functions in the My Paint tutorial.

# The Copy Feature

The Paste feature has been coded, and should be working correctly (assuming, of course, that you were able to test and debug it). Now it's time to code the Copy feature.

Before users of your program can copy to the clipboard, you'll need to add a few things. First and foremost, you need to add a Selection tool, which the user can use to select an area of the screen to be copied. You'll then need to add the code for the Edit/Copy menu handler.

## The Selection Tool

The Selection tool is simply a drawing tool that draws a rectangle around the area that the user wants to select for copying (or cutting) to the clipboard. In most drawing and painting programs, the Selection tool draws an animated line of dashes, usually called a *marquee* but otherwise known as "marching ants," around the selected area. Although it's possible to code for marching ants in REALbasic, it tends to get a bit complicated, so instead, you'll add code that enables users to draw a gray rectangle around the selection rectangle.

The addition of the Selection tool, as with all other tools you've added, is accomplished in four parts:

1. Creating the menu item
2. Adding the code to enable the new menu item
3. Adding the code to add a checkmark to the active Tools menu item
4. Creating the menu handler for the new menu item

## Creating the Selection Tool Menu Item

The first thing you need to do is add the new Selection Tool menu item to the My Paint application's Tools menu. You should be fairly familiar with this process by now, but if not, here's a refresher for you:

1. Double-click the Menu icon on the REALbasic Project window.
2. Select the Tools menu in the Application Menu window.
3. Add a Selection Tool entry in the last slot of the Tools menu, as shown in Figure 14.13.

## Enabling the New Menu Item

The new menu item can't be used until it's enabled. To enable it, use the EnableMenuItems event handler:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Expand the Events item and click on the EnableMenuItems event.
4. Enter the following code below the source-code lines that enable the other Tools menu items, as shown in Figure 14.14:

```
ToolsSelectionTool.Enable
```

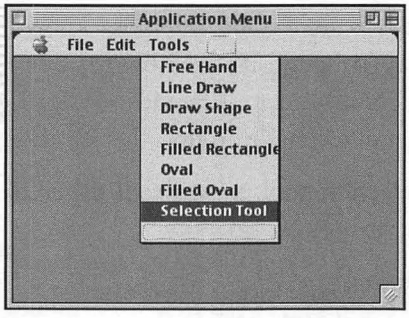



**Figure 14.13**
The new menu item for the Selection tool

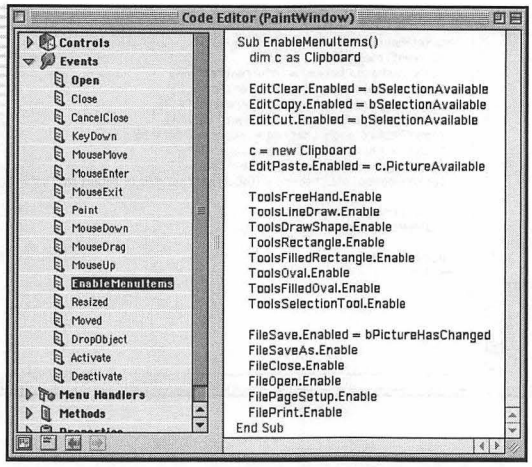



**Figure 14.14**
The new PaintWindow/EnableMenuItems event handler after the addition of the code to enable the Selection Tool menu item

## Updating the New Menu Item

Your Tools menu has a checkmark beside the last selected tool to make it obvious to the user which tool is currently in use. You'll need to modify the SetMenuSelection method to allow for the new Selection Tool menu item that you just added:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Expand the Methods item of the Code Editor window.
4. Select the SetMenuSelection method from the list.
5. Enter the following code before the `PaintCanvas.Refresh` line of code in the SetMenuSelection method, as shown in Figure 14.15:

```
ToolsSelectionTool.Checked = ((nCurrentTool= 8) or ➥
(nCurrentTool= 9))

bSelectionAvailable = false
EnableMenuItems
```

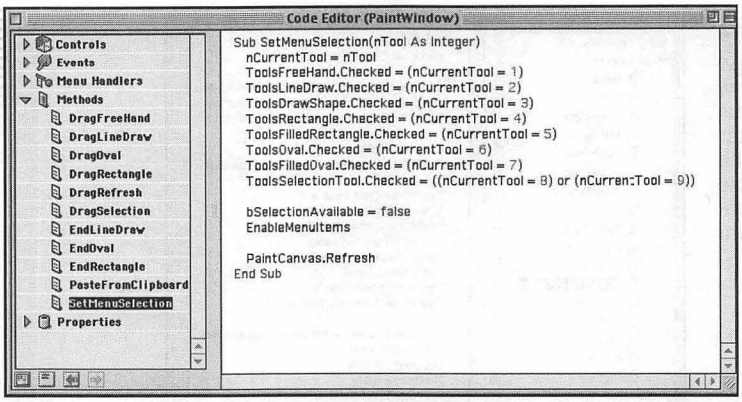



**Figure 14.15**
The code for the SetMenuSelection method after the addition of the Copy and Paste features

You might be wondering why, in the preceding code, you're setting the checkmark on the Selection Tool menu item if the `nCurrentTool` variable contains a value of `8` or `9`. If you recall, the Paste code you entered earlier uses a value of `9` when you're in the middle of a paste operation. Because there is really no tool for this operation *per se*, you'll mark the Selection Tool as the current tool when a paste operation is in progress.

Also notice that you're setting the `bSelectionAvailable` property to a value of `false` every time a Tools menu selection changes. This is so that the selection goes away, and the Edit menu items are disabled, if the selected tool changes.

## The Selection Tool Menu Handler

Finally you can add the code to the Selection tool's menu handler so that the Selection tool is activated when the menu item is selected. It's a one-liner, so without further ado:

1. Click on the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Open the Edit menu and select the New Menu Handler item, or press Command+Option+H.
4. Select the ToolsSelectionTool item from the drop-down list, and click OK.

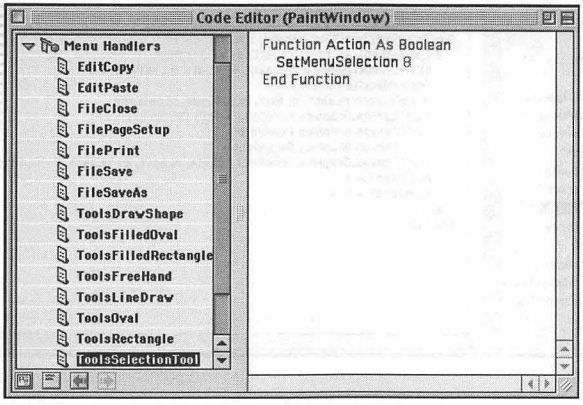



**Figure 14.16**
The code for the ToolsSelectionTool menu handler

5. Enter the following code in the ToolsSelectionTool menu handler, as shown in Figure 14.16:

```
SetMenuSelection 8
```

# Adding the DragSelection Method

Now that the Selection tool menu stuff is done, you can move on to the guts of the code. The first thing you're going to tackle is the method for drawing the selection rectangle in the PaintWindow. This code, not surprisingly, looks a lot like the DragRectangle code, which is used for drawing a rectangle with the Rectangle tool. After all, both do the same type of thing: They draw a rectangle. The major differences are that the color, pen height, pen width, and background of the rectangle are hard-coded to a specific value. Let's go ahead and create the method, and you'll see what we mean:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Open the Edit menu and select the New Method item, or press Command+Option+M.
4. Enter `DragSelection` as the new method name.
5. Enter `X As Integer, Y As Integer` as the parameters.

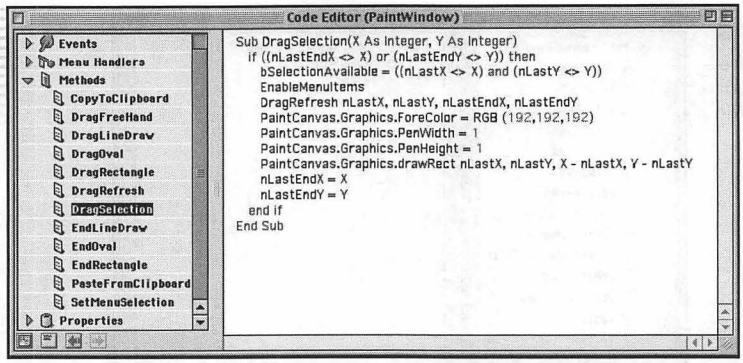



**Figure 14.17**
The code for the new DragSelection method

6. Leave the return type blank and click OK.
7. Enter the following code in the DragSelection method, as shown in Figure 14.17:

```
if ((nLastEndX <> X) or (nLastEndY <> Y)) then
  bSelectionAvailable = ((nLastX <> X) and (nLastY <> Y))
  EnableMenuItems
  DragRefresh nLastX, nLastY, nLastEndX, nLastEndY
  PaintCanvas.Graphics.ForeColor = RGB (192,192,192)
  PaintCanvas.Graphics.PenWidth = 1
  PaintCanvas.Graphics.PenHeight = 1
  PaintCanvas.Graphics.drawRect nLastX, nLastY, X - nLastX, ➥
  Y - nLastY
  nLastEndX = X
  nLastEndY = Y
end if
```

## The MouseDrag Event Handler Changes

Just as with the other tools, you'll need to modify the PaintCanvas/MouseDrag event handler so that it calls the appropriate drag method depending on the currently selected tool—in this case, the DragSelection method:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.

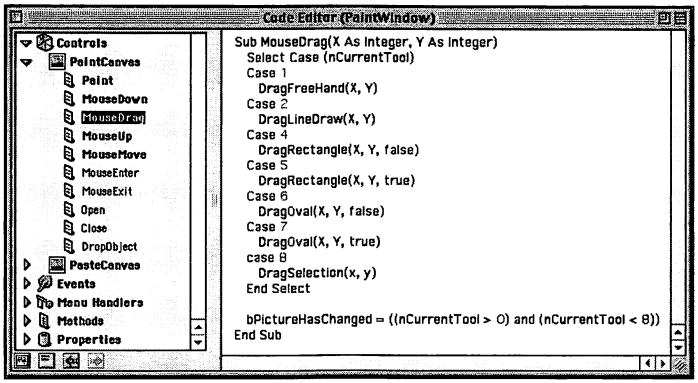



**Figure 14.18**
The code for the MouseDrag event handler, after adding the case for `DragSelection`

3. Expand the Controls and PaintCanvas items in the Code Editor window.
4. Select the MouseDrag event handler and insert the following code above the `End Select` line of code, as shown in Figure 14.18:

```
case 8
    DragSelection(x, y)
```

# The CopyToClipboard Method

You're almost there, almost ready to actually add the EditCopy menu handler, but not quite. You need to create a method that will be used to copy data from the PaintCanvas, based on the current selection rectangle, to the clipboard. You could just put the following code in the EditCopy menu handler, but because you want to be able to reuse the code elsewhere, you'll create a new method instead:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Open the Edit menu and select the New Method item, or press Command+Option+M.
4. Enter `CopyToClipboard` as the new method name, and click the OK button.
5. Enter the following code in the CopyToClipboard method, as shown in Figure 14.19:

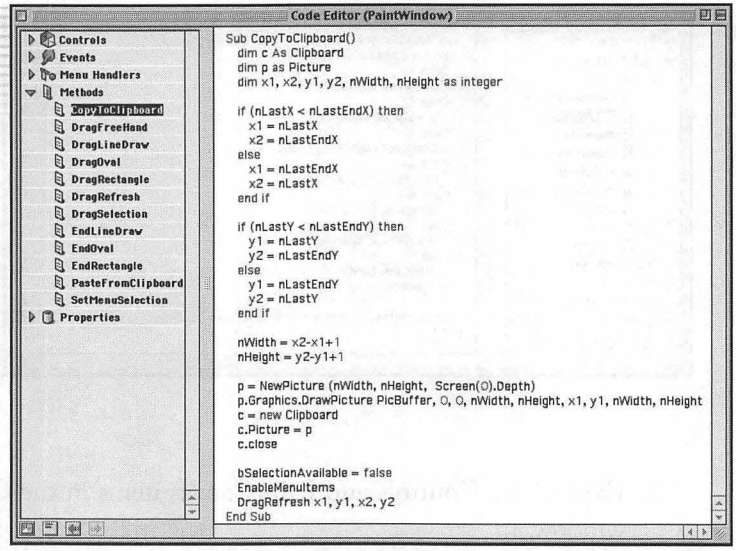



**Figure 14.19**
The code for the new CopyToClipboard method

```
dim c As Clipboard
dim p as Picture
dim x1, x2, y1, y2, nWidth, nHeight as integer

if (nLastX < nLastEndX) then
  x1 = nLastX
  x2 = nLastEndX
else
  x1 = nLastEndX
  x2 = nLastX
end if

if (nLastY < nLastEndY) then
  y1 = nLastY
  y2 = nLastEndY
else
  y1 = nLastEndY
  y2 = nLastY
end if

nWidth = x2-x1+1
```

```
    nHeight = y2-y1+1

    p = NewPicture (nWidth, nHeight,  Screen(0).Depth)
    p.Graphics.DrawPicture PicBuffer, 0, 0, nWidth, nHeight, ➥
x1, y1, nWidth, nHeight
    c = new Clipboard
    c.Picture = p
    c.close

    bSelectionAvailable = false
    EnableMenuItems
    DragRefresh x1, y1, x2, y2
```

This code looks pretty complex, but it really isn't. It starts by defining a few variables, the most important of which is the clipboard variable. It then figures out what the proper upper-left and lower-right corners of the rectangle should be, and calculates the width and height based on these values. The contents of the `PicBuffer` are copied to a new picture, based on all these calculated values. The contents of the new picture are then placed in the clipboard. At this point, the Edit menu item status is refreshed and the selection rectangle is removed from the screen.

# The New EditCopy Menu Handler

Now that all the behind-the-scenes work is done, you can add the EditCopy menu handler. The code for this menu handler is amazingly simple, because all the work is being done in other methods:

1. Click on the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Open the Edit menu and select the New Menu Handler item, or press Command+Option+H.
4. Select the EditCopy item from the Menu Item drop-down list, and click OK.
5. Enter the following code in the EditCopy menu handler, as shown in Figure 14.20:

```
    CopyToClipboard
```

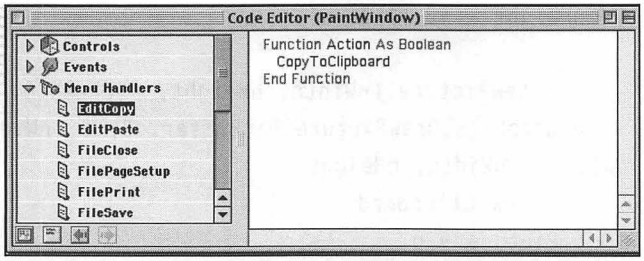



**Figure 14.20**
The deceptively simple code for the EditCopy menu handler

The code for the menu handler might be simple—one line and no parameters—but what happens behind the scenes is where the complexity comes in. All the supporting methods and properties used by this simple menu handler are fairly complex, but when broken into smaller chunks, they become much more manageable.

# The Clear and Cut Features

The last two Edit menu items you'll be working on are the Clear and Cut items; both are similar in function. The Clear menu item clears the area of the window selected with the Selection tool. The Cut menu item does the same thing, after copying the contents of the selected rectangle to the clipboard.

## The New ClearSelection Method

The ClearSelection method is used when the user chooses the Edit/Clear menu item, and is also used by the Edit/Cut menu item. By creating a ClearSelection method rather than putting this code in the EditClear menu handler, you can take advantage of code reusability. Here's how it's done:

1. Click the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Open the Edit menu and select the New Method item, or press Command+Option+M.
4. Enter `ClearSelection` as the new method name, and click the OK button.

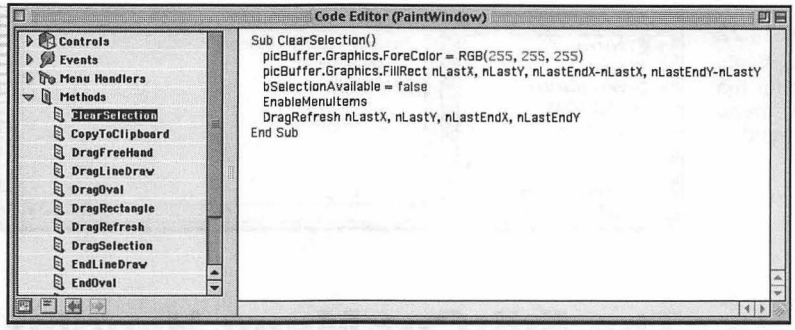



**Figure 14.21**
The code for the new ClearSelection method, which will shortly make your life much simpler

5. Enter the following code in the ClearSelection method, as shown in Figure 14.21:

```
picBuffer.Graphics.ForeColor = RGB(255, 255, 255)
picBuffer.Graphics.FillRect nLastX, nLastY, ➥
nLastEndX-nLastX, nLastEndY-nLastY
bSelectionAvailable = false
EnableMenuItems
DragRefresh nLastX, nLastY, nLastEndX, nLastEndY
```

# The EditClear Menu Handler

As we said earlier, the EditClear menu handler will use the code in the ClearSelection method; as a result, the code for this menu handler is amazingly simple. All it needs to do is clear the selected area of the Paint window using the ClearSelection method:

1. Click on the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Open the Edit menu and select the New Menu Handler item, or press Command+Option+H.
4. Select the EditClear item from the Menu Item drop-down list, and click OK.
5. Enter the following code in the EditClear menu handler, as shown in Figure 14.22:

```
ClearSelection
```

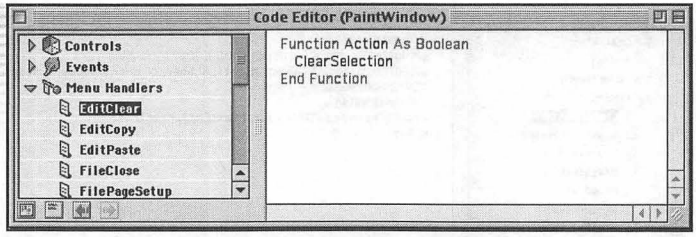



**Figure 14.22**
The deceptively simple code for the EditClear menu handler

# The EditCut Menu Handler

Just like the EditClear menu handler, the EditCut menu handler uses pre-existing code, so the actual menu-handler code is also very simple. All it needs to do is copy the selection to the clipboard and then clear the selected area of the Paint window:

1. Click on the PaintWindow class in the Project window.
2. Press Option+Tab to display the Code Editor window for the PaintWindow class.
3. Open the Edit menu and select the New Menu Handler item, or press Command+Option+H.
4. Select the EditCut item from the Menu Item drop-down list, and click OK.
5. Enter the following code in the EditCut menu handler, as shown in Figure 14.23:

```
CopyToClipboard
ClearSelection
```

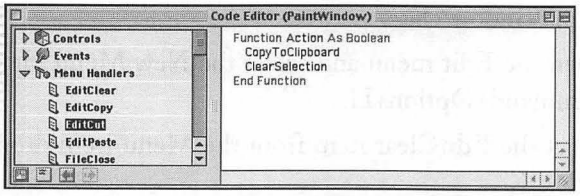



**Figure 14.23**
The even-more deceptively simple code for the EditCut menu handler

The last two code examples aptly demonstrate the power of code reusability. It is possible to write code that is very readable, but that at the same time does a lot of work. You should always keep in mind, however, that code reusability comes at the expense of performance. Don't overuse code reusability just for the sake of simplifying code. If the code you're writing has performance considerations, you may want to avoid, or at least keep to a minimum, the amount of reusable code you write.

# Review

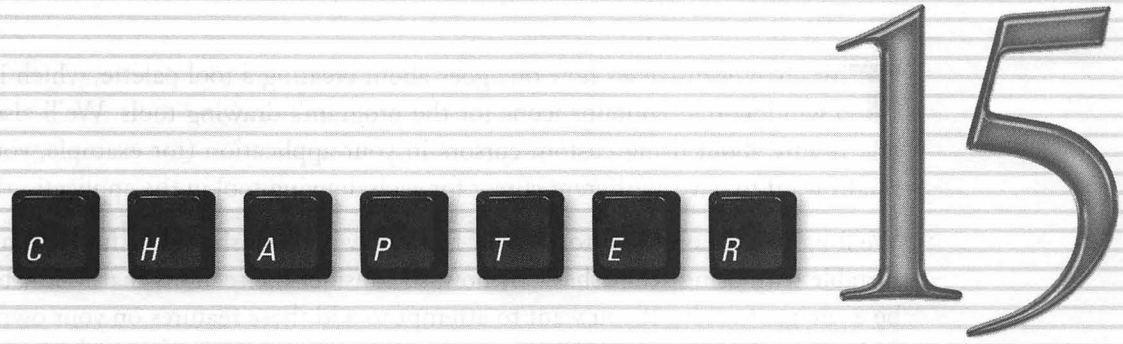
What started out sounding fairly simple—copying data to and from the clipboard—turned into quite a task! Sometimes, what sounds like a simple programming task may in fact balloon into a much more complicated one. Always consider the user-interface elements and the structure of your program when assessing the difficulty of a task. What seems like a simple change might require you to work with many parts of your program. You can save time by sitting down in advance and figuring out how you're going to achieve any goal, rather than just jumping in and doing the coding.

After reading the chapter, you should be comfortable adding new controls to a window and hiding them in response to conditions in your program. You could do this with buttons, text, and just about every other window control as well. Controls in windows don't have to be static items; they can respond to and change based on the conditions of your application.

You began by learning about the clipboard functions, and about how data is moved to and from the clipboard, concentrating on the functions for copying text to and from the clipboard. You then learned about the Paste feature, starting with the PasteCanvas control. You added code for pasting from the clipboard into the control, and then copying the data from the control to the window.

For the Copy feature, you added a new menu item and the methods used by the new tool, and modified its menu handlers and event handlers. Lastly, you added the code to copy the data from the clipboard to the PaintCanvas control.

Finally, you coded the Clear and Cut functionality. The Clear feature required new coding, but you created the Cut feature by combining the Clear and Copy features, enabling you to reuse code. Although code reusability can be a great thing, it can negatively affect program performance.




# Tool Palettes and Cursors

## In This Chapter

- Creating tool palette icons
- Creating a tool palette window
- Mapping the tools to the menu items
- Creating the tool cursors
- Using the appropriate cursors

This chapter discusses how one goes about creating a tool palette, which is a window that contains icons for the program's drawing tools. We'll also talk briefly about using custom cursors in your application (for example, you might want to have a different cursor for each drawing tool in the application).

We'll lighten up on the tutorial stuff from now on. You should be fairly comfortable with using REALbasic by now, so consider the sections that follow to be exercises for you. If you want to attempt to add these features on your own, feel free to do so.

A lot of the material in this chapter relies on the ability to create and edit resource files. If you're working on pre–Mac OS X applications, you'll want to download and learn how to use a tool like Apple's ResEdit or Mathemaesthetics' Resourcer application. You can download any tools you'll need at the Apple Developers Connection Tools Web site (http://developer.apple.com/tools). We aren't going to get into how to use these tools; you'll find plenty of documentation about these tools on the Apple Developers Connection site.

# Creating Tool Palette Icons

When you create tool palette icons, you'll add them to a resource file created using one of the tools we talked about in the preceding section. After you've created your icons, save the resource file, naming it *Resource*, and place it in the same folder as your REALbasic application.

The file must be named *Resources* so that you can drag and drop it into the REALbasic project window. For example, suppose you've created a resource file in ResEdit with four icons (which, by the way, are created via ResEdit's Resource/Create New Resource menu item, shown below in the "cicns" window). It would look something like the one in Figure 15.1.

**NOTE**

Notice that there are icons that represent each button before it has been selected, as well as icons that represent each button after it has been selected. You'll want to do this for all of your tool palette buttons.

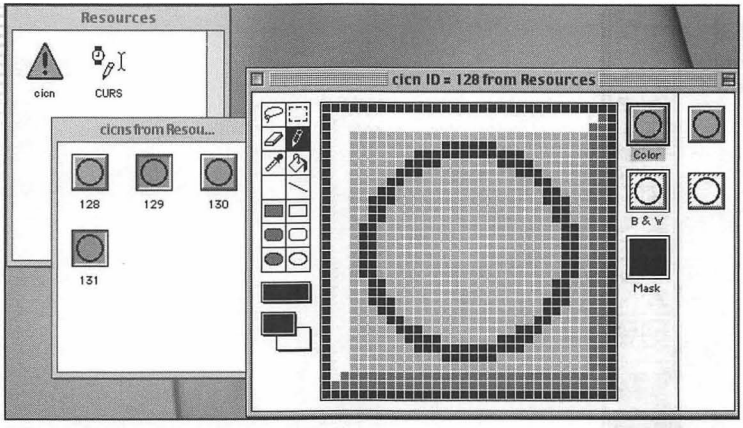



**Figure 15.1**
Four icons defined in a ResEdit resource file.

Now that you've created a resource file, all you need to do is drag it into the REALbasic Project window of your application, as shown in Figure 15.2, and voilà! You'll be able to use the icons in your REALbasic application.

# Creating a Tool Palette Window

If you've ever used a Macintosh drawing or painting program, you're familiar with tool palettes. As a matter of fact, REALbasic itself contains a tool palette. As mentioned before, tool palettes are those windows with the thin title bars that contain icons used to select the various application tools. A typical tool palette window is shown in Figure 15.3. To create a tool palette window in REALbasic, do the following:

1. Open the File menu and select the New Window item.
2. Change the window name to `ToolPalette`.

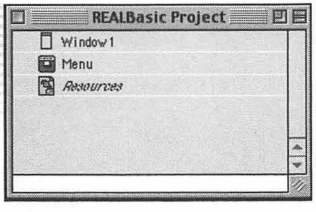



**Figure 15.2**
The resource file in the REALbasic project


**Figure 15.3**
A typical tool palette window

3. Change the window frame to *3 - Floating Window*. This enables you to create a window with a narrow title bar, like you normally see in palette windows.
4. Add one 32 × 32 image-well control to the tool palette window for each tool that you want to add.

Figure 15.4 shows a blank tool palette window created in REALbasic.

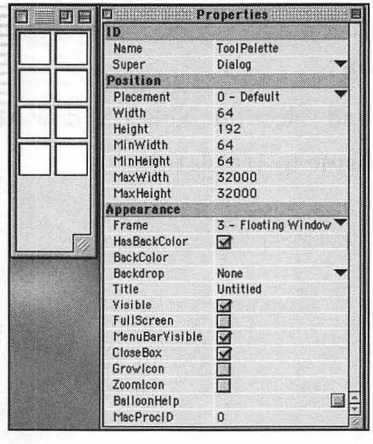



**Figure 15.4**
A blank tool palette window created in REALbasic

# Mapping the Tools to the Menu Items

Simply creating a window, dropping some controls on it, and setting the contents of those controls to some icons isn't enough to make them work the same as your Tools menu items. You'll need to add MouseDown event handlers to each of the image-well controls to change the icon to the selected tool icon, and then call the `PaintWindow.SetMenuSelection` method to choose the appropriate tool. You could do this in each image-well control's MouseDown method, but it would be better to create a few new methods, which would be called from each MouseDown event:

```
Sub SetToolIcon (iwImageWell As ImageWell, nBaseIcon As Integer, ➥
bSelected As Boolean)
  ' Set the contents of an image-well control to one of two icons
  Dim nIcon As Integer
  If (bSelected) Then
    nIcon = nBaseIcon + 1
  Else
    nIcon = nBaseIcon
  End If
  iwImageWell.image = app.ResourceFork.Getcicn(nIcon)
End Sub

Sub SetToolSelection(nTool As Integer)
  ' Set each tool icon, depending on what the currently selected ➥
tool is
  SetToolIcon(ImageWell1, 128, nTool = 1)
  SetToolIcon(ImageWell2, 130, nTool = 2)
  SetToolIcon(ImageWell3, 132, nTool = 3)
  SetToolIcon(ImageWell4, 134, nTool = 4)
  SetToolIcon(ImageWell5, 136, nTool = 5)
  SetToolIcon(ImageWell6, 138, nTool = 6)
  SetToolIcon(ImageWell7, 140, nTool = 7)
  SetToolIcon(ImageWell8, 142, nTool = 8)
  'PaintWindow.SetMenuSelection(nTool) me.image = ➥
app.ResourceFork.Getcicn(129)
  PaintWindow.SetMenuSelection(6)
End Sub
```

```
Function MouseDown(X As Integer, Y As Integer) As Boolean
' Since this image-well control was clicked, change the tool ➥
selection
  SetToolSelection (1)
End Sub
```

After making all these modifications, you want to change the original menu handlers of the PaintWindow class to use the new `ToolPalette.SetToolSelection` method so that you can select tools by selecting the menu items or by selecting the icons in the tool palette. When running, the new tool palette will look similar to the one shown in Figure 15.5 (but, obviously, with more icons).

# Creating the Tool Cursors

If you've used other drawing programs before, you're no doubt aware that when you select different drawing tools, the mouse cursor changes to reflect the selected tool. Just as with the icons you created earlier, you'll create your cursors in your Resource file, using the appropriate resource editor for your operating system. You create cursors by adding CURS resources to the resource file, like the ones shown in Figure 15.6.

# Using the Appropriate Cursors

To swap to the appropriate cursor, you use the PaintWindow.SetMenuSelection method to change the active cursor. If you create a new cursor for each tool, with resource IDs beginning at 128, then all you need to do is add the following line of code to the SetMenuSelection method:

```
PaintCanvas.MouseCursor = app.ResourceFork.GetCursor(127+nTool)
```

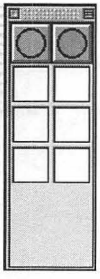


**Figure 15.5**
A tool palette window taking shape in REALbasic.

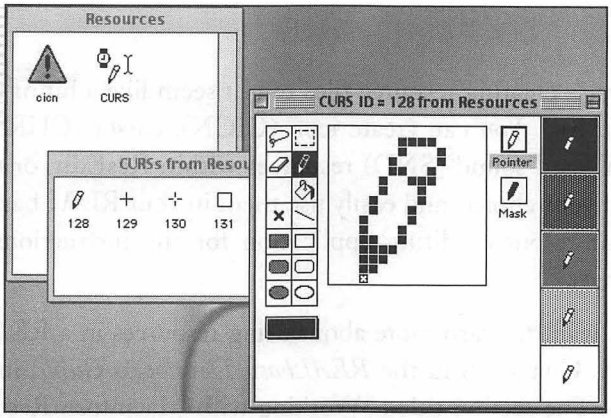



**Figure 15.6**
A few cursors defined in a ResEdit resource file

This sets the mouse cursor for the PaintCanvas control to the specified cursor in the Resource file. If you remember, each tool is specified as tool number 1, tool number 2, tool number 3, and so on. These tool values are added to 127, giving values of 128, 129, 130, and so on, which are the values of the CURS resources you'll be using.

In Figure 15.7, the cursor is displayed as the free-hand pencil cursor. If you drag the mouse cursor off the canvas, as shown in Figure 15.8, it reverts to the standard arrow cursor.

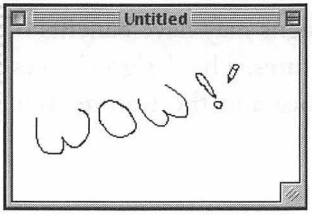



**Figure 15.7**
The PaintWindow window, with the free-hand pencil mouse cursor

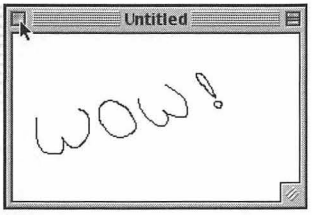



**Figure 15.8**
The PaintWindow window, with the standard mouse cursor

# Review

Although creating resource files might seem like a bit of a pain, it's really not all that bad. You can create icon (CICN), cursor (CURS), and even picture (PICT) and sound (SND) resources within ResEdit, or the resource-editing tool of your choice, and easily use them in your REALbasic application. Refer to your resource-editing application for the instructions on how to create resources.

If you want to learn more about using resources in a REALbasic application, refer to Chapter 8 in the *REALbasic Developers Guide*, included on the CD-ROM. The section titled "Working with Macintosh Resources" will give you all the information you need on using cursor, icon, picture, and sound resources in your apps, along with other custom resources.

This chapter is intended only to point you in the right direction to add these new features to your application. We can't give you all the answers and source code this time. The implementation of these features is left as an exercise for you. If you get stuck, refer to the *REALbasic Developers Guide* and *Language Reference* documents included in the *Open Me For REALbasic* folder on the CD-ROM that comes with this book.

By the way, don't get overly hung up on the "right way" to write your new code. There are countless ways to write program code, all of which will yield the same results. It's like walking through a forest with many paths. Each path might have different advantages and pitfalls, but all lead to your goal, the other side of the forest. Different designs may have benefits of performance, application size, ease of use, and features. The design choices you make should be based on your level of experience and the features you most value for your application.


# 16

# Finishing Touches

## In This Chapter

- Adding color selection tools
- Adding line-width selection tools
- The About box: patting yourself on the back

About the only things left to wrap up in the tutorial section of this book are some of the finishing touches of the application. As with just about any other project, there are always some features that you'd classify as not necessary, but nice to have—in other words, fluff. That said, fluff can be the difference between a mediocre product and one that's, well, still mediocre, but that looks better.

Seriously though, touchy-feely features often sell a product. If you had the choice between two applications with the exact same "useful" features, but one of them threw in some additional "purely-for-fun" features, you'd probably pick the one with the additional features, even if they weren't something you were looking for.

Don't get me wrong. Fluff isn't what you should be concentrating on when designing a product. Always work on the features that you deem necessary first, and then work on the fluff later. Fluff is like dessert—you don't have it until you've eaten your main course. The same goes for your applications. You've got to worry about the meat of your apps before worrying about the frills.

Along the same lines as fluff is "feature creep." Quite often, developers add features to a product just because designing and implementing those features is fun for the developer. Whether something is feature creep, or as some put it, "creature feep," is sometimes hard to judge. This is why, quite often, a lot of software-development companies leave decisions about features to someone other than the software developers. Sometimes marketing is best suited to make feature decisions.

Conversely, don't assume that just because something is fun to develop, it's feature creep. If the feature you're adding has any measurable benefit to the user of the application and doesn't severely affect your development schedule, then by all means add the feature. It's good to get into the practice of honestly asking yourself whether the feature is absolutely necessary, or whether you are adding it just because it would be fun/neat/cool to work on. If you can objectively state that you're adding features based on their merits, then they pass the test and should be included in your application's feature set.

# Adding Color-Selection Tools

So maybe being able to paint and draw in color isn't quite a fluff feature. Of course, you could argue that your application is just a sketching program in

which the only tools available to the artist are pencils. In reality, however, just about everyone expects a paint program to support color.

## Adding and Enabling the New Menu Items

First, you need to add the Color Selection menu as a submenu of the Tools menu. To do so, simply select the blank item in the Tools menu, change the menu's Name property to *Color Selection*, and enable the SubMenu property, as shown in Figure 16.1.

Next, add the menu items for the Color Selection submenu, just as you would with any normal menu, so that they look like those in Figure 16.2.

You need to add the menu handlers for these new menus to enable users to select the line or fill color, but before you do, you should add a method. That's because the code for the menu handlers is almost identical. By adding a new method, you can reduce redundant code. Besides, you'll be using the SetColorSelection method again later, so it's worth the effort. Add a new method named `SetColorSelection` to the PaintWindow, giving it a parameter

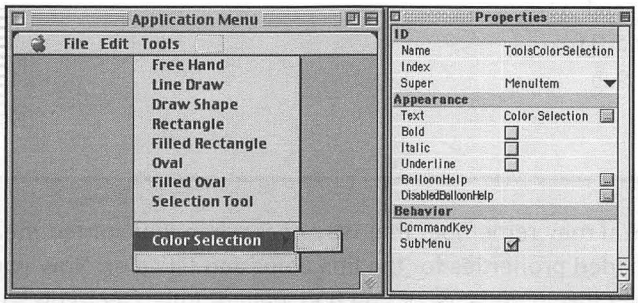



**Figure 16.1** The Color Selection menu

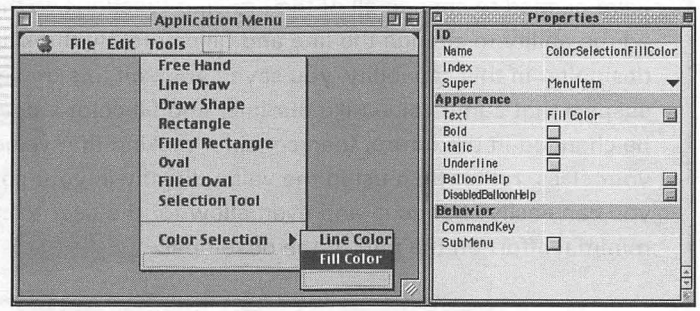



**Figure 16.2** The menu items in the Color Selection submenu

of `bIsFillColor As Boolean`. Your new SetColorSelection method code should look something like this:

```
Sub SetColorSelection (bIsFillColor As Boolean)
   Dim rbgSelectedColor As Color
   Dim strSelectionPrompt As String

  strSelectionPrompt = "Select the "

  If (bIsFillColor) Then
    rbgSelectedColor = rgbFillColor
    strSelectionPrompt = strSelectionPrompt + "fill color."
  Else
    rbgSelectedColor = rgbLineColor
    strSelectionPrompt = strSelectionPrompt + "line color."
  End If

  if (SelectColor(rbgSelectedColor, strSelectionPrompt)) then
    If (bIsFillColor) Then
      rgbFillColor = rbgSelectedColor
    Else
      rgbLineColor = rbgSelectedColor
    End If
  End If
End Sub
```

**NOTE**

You may remember that way back when you started the tutorial, you added properties for the line color and fill color. Now you see why you did this in advance. If you had waited until now to do so, you would have needed to change all of the drawing functions so that they supported the ability to change the line and fill colors. By thinking ahead and designing in this capability, you saved yourself lots of work. If you ever suspect that some value (like our line- and fill-color values) may need to be changed in the future, then consider making that value a property of your class, rather than using the value directly in your code. This way you can easily change it, and even allow for the user to change it, with minimal effort on the part of the developer.

The `SelectColor` function is used in REALbasic to select a color and return it as one of the parameters of the function. The `SelectColor` function returns `true` or `false`, depending on whether the user clicks the OK or Cancel buttons on the Color Picker dialogs. The function supports all the color-selection methods available to the operations system. If your Macintosh is like mine, you can use the CMYK, Crayon, HLS, HSV, HTML, or RGB color-selection tools that you see in Figures 16.3–16.8. The nice thing is that you can use any of these cool color-selection tools in your applications without having to write a stitch of code for them. They're part of the Mac OS and every application gets to use them.

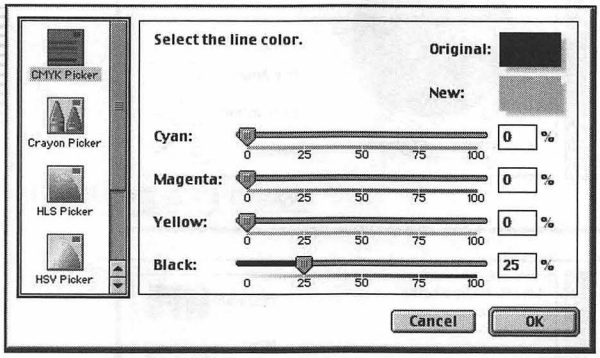



**Figure 16.3**
The CMYK Picker dialog box

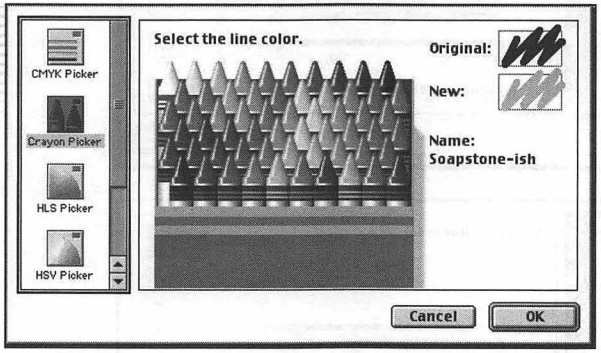



**Figure 16.4**
The Crayon Picker dialog box

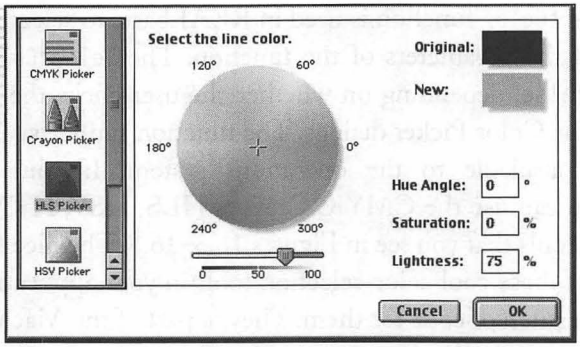



**Figure 16.5**
The HLS Picker dialog box

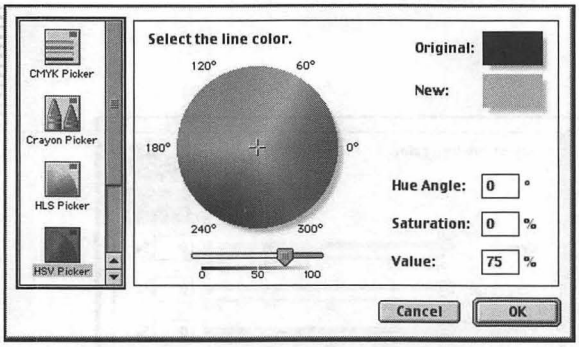



**Figure 16.6**
The HSV Picker dialog box

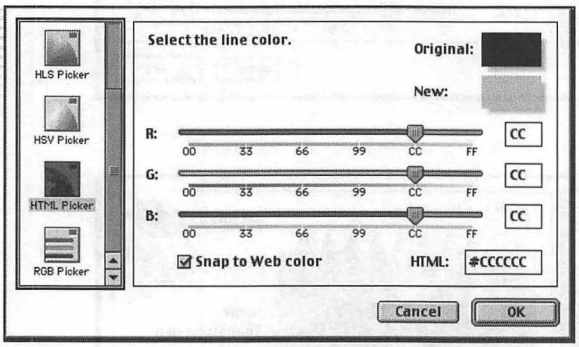



**Figure 16.7**
The HTML Picker dialog box

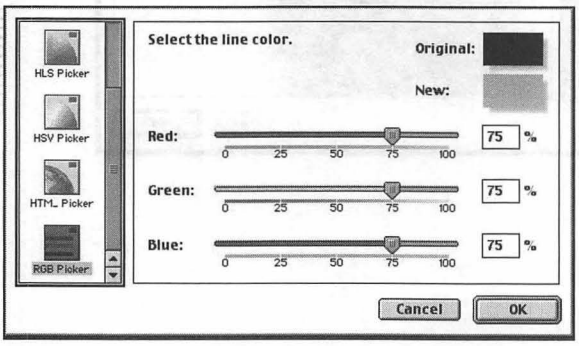



**Figure 16.8**
The RGB Picker dialog box

You can now add the menu handlers for those new color-selection menu items. Because you added a method previously to do the actual color-selection itself, the code for the menu handlers will be extremely simple. For the Fill Color menu item add a new ColorSelectionFillColor menu handler for the PaintWindow. Here's what the code for the Fill Color menu item looks like:

```
Function Action as Boolean
  ' The ColorSelectionFillColor Menu Handler
  SetColorSelection(true)
End Function
```

For the Line Color menu item add a new `ColorSelectionLineColor` menu handler for the PaintWindow. And here's the code for the Line Color menu item:

```
Function Action as Boolean
  ' The ColorSelectionLineColor Menu Handler
  SetColorSelection(false)
End Function
```

Don't forget that you have to add code to enable the new menu items. Add the following code to the EnableMenuItems event handler of the PaintWindow and you should be able to test the changes:

```
ToolsColorSelection.Enable
ColorSelectionFillColor.Enable
ColorSelectionLineColor.Enable
```

Go ahead and compile and then test the MyPaint application. You should be able to select any color for the line or fill color and use any of the drawing tools with those colors. If you find that you have any problems with the drawing tools using the correct color, double-check the Drag . . . method for that drawing tool to make sure it's using the `rgbFillColor` and `rgbFillColor` properties correctly.

**TIP**

We'll give you a hint: There is at least one method that was intentionally coded incorrectly with respect to its use of the selected colors. We know—what a DRAG! But think of it like this, you're FREE, practice your debugging skills, finding the error on your own, without us giving you a HAND.

# Adding Color-Selection Tools to the Tool Palette

Before you start adding the controls to handle color selection via the tool palette, you'll need to add some properties to the application class to track which PaintWindow window is active, so that the tool palette displays the proper colors for the active PaintWindow window.

## Keeping Track of the Active PaintWindow

First you need to add the following properties to the Application class (theApp):

```
ActivePaintWindow As PaintWindow
wToolPalette As ToolPalette
```

Then, in the Application class's Open event handler, add the following code to create the ToolPalette window:

```
wToolPalette = New ToolPalette
```

At this point, you need to add the following code to the PaintWindow's Activate event handler, which will set the Application class's `ActivePaintWindow` property whenever a new PaintWindow is created or when the user swaps PaintWindows:

```
app.ActivePaintWindow = me
app.wToolPalette.UpdateColors(rgbLineColor, rgbFillColor)
```

We'll explain what the `UpdateColors` function does in just a bit; just go ahead and code it for now. Also, you'll want to add the same function-call code to the `PaintWindow.SetColorSelection` method so that the colors are updated whenever the user chooses a new color:

```
app.wToolPalette.UpdateColors(rgbLineColor, rgbFillColor)
```

Go ahead and add the following code to the PaintWindow's Close event handler, which will clear the Application class's `ActivePaintWindow` property whenever a PaintWindow is closed:

```
app.ActivePaintWindow = nil
```

The addition of this code will help you later to prevent an error when attempting to change the selected colors if no windows are open.

## The Actual Tool Palette Work

Like always, you'll add some properties before you do any code work. First, add the following properties to the ToolPalette dialog class, which will be used to store the line and fill colors for the last-selected PaintWindow:

- `rgbCurrentFillColor As Color`
- `rgbCurrentLineColor As Color`

To add the Color Selection tool to the tool palette, you simply add two Canvas controls to the ToolPalette dialog window: one shaped like a line, and one shaped like a box. These tools will correspond to the Line Color and Fill Color menu items. Your Tool Palette window might look something like the one shown in Figure 16.9 when you finish.

Now that you've added the Canvas controls, you'll want them to show the actual color selections. You can do this by changing the Paint event handler for these controls to include a `FillRect` call. For the Line Color Selection Canvas control, double-click on the control itself, then enter the code in the Paint event handler so that that handler looks like this:

```
Sub Paint (g as Graphics)
  g.ForeColor = rgbCurrentLineColor
  g.FillRect(0,0,Width,Height)
End Sub
```

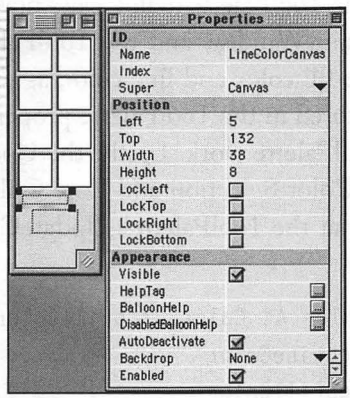



**Figure 16.9**
The Tool Palette dialog with the Canvas controls added

The Fill Color Selection canvas control's Paint event handler would look something like this:

```
Sub Paint (g as Graphics)
  g.ForeColor = rgbCurrentFillColor
  g.FillRect(0,0,Width,Height)
End Sub
```

Now you'll need to add code to the MouseDown event handlers for these two Canvas controls. The FillColorCanvas control's MouseDown event handler code should be

```
If (app.ActivePaintWindow <> nil) Then
  app.ActivePaintWindow.SetColorSelection(true)
End If
And the LineColorCanvas control's MouseDown event handler should be
If (app.ActivePaintWindow <> nil) Then
  app.ActivePaintWindow.SetColorSelection(false)
End If
```

This code calls the SetColorSelection method of the currently active PaintWindow (if there *is* a currently active paint window). When the user clicks these color-selection controls on the paint canvas, the SetColorSelection method for the proper PaintWindow is called, displaying the proper Color Selection dialog.

Remember that mysterious UpdateColors method we mentioned earlier? You're going to add this to the ToolPalette dialog class now. This is where you magically tie together the PaintWindow and the ToolPalette windows. This method takes two colors, the fill color and line color, as parameters. The values in these parameters are stored in the ToolPalette properties that you added when you started all this ToolPalette work. Lastly, the UpdateColors method forces the ToolPalette's Fill Color Selection and Line Color Selection canvas controls to update. Here's what the ToolPalette's UpdateColors method looks like:

```
Sub UpdateColors(rgbLineColor As Color, rgbFillColor As Color)
  rgbCurrentLineColor = rgbLineColor
  rgbCurrentFillColor = rgbFillColor
  LineColorCanvas.Refresh
  FillColorCanvas.Refresh
End Sub
```

That may seem like a lot of work, but it's worth the effort. You should now be able to click on the color-selection controls on your tool palette and bring up the Color Selection dialogs. Also, when the color selection changes, be it via clicking on the control in the tool palette or selecting the menu item, the color of the respective control in the tool palette should be updated. Finally, you should be able open more than one PaintWindow and change the fill and line color in each window independent of the other windows. Swapping between all the windows should change the colors in the ToolPalette dialog window when you change from one window to the other.

# Adding Line-Width Selection Tools

You could argue that the capability to select the width of the drawing tool's line is not a necessary feature, but, as we said before, most users expect a paint application to have this feature. We'll show you how to add it here.

## Adding and Enabling the New Menu Items

As with the color-selection tools, you must add menu items and tool-palette controls for line-width selection. You are, however, going to do something a little different for the menu items: you'll be adding the menu items to the menu dynamically.

When we talk about adding menu items "dynamically," we mean that you're not going to predefine the menu items. Instead, you'll add them to the menu when the application runs. They're *dynamic* in the sense that they aren't defined statically as a menu-item definition in the REALbasic application's project.

Still, you need to define *something* in the REALbasic project to let it know that you're going to be adding items to the menu dynamically. You do this by adding an indexed menu item to the menu that contains the dynamic menu items.

For example, add a Line Width submenu to the Tools menu as you did earlier for the color-selection tools. Then, as the first and only item of the submenu, add an indexed menu item by adding a Points menu item as a marker and then entering an index property of `0` in the menu item's Properties window (see Figure 16.10).

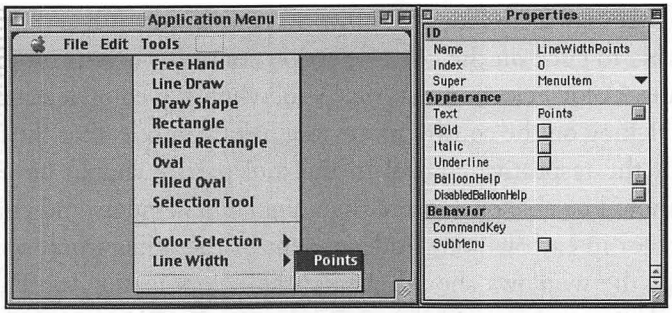



**Figure 16.10**
The new Line Width submenu

You'll need to add the dynamic menu items at this point, so change the Application class's Open event handler so that it looks like the following:

```
Sub Open()
  Dim i As Integer
  Dim m As MenuItem

  wToolPalette = New ToolPalette

  LineWidthPoints(0).Text = "1 Point"
  For i = 1 To 7
    m = New LineWidthPoints
    m.Text = str(i+1) + " Points"
  Next
End Sub
```

The cool thing is that you only need to add one menu handler for all the menus you just added. Because the menus are indexed, your one menu handler will be used for all these menus. You add menu handlers for indexed menus the same way as for other menus: by opening REALbasic's Edit menu and selecting the New Menu Handler item. Almost identical menu-handler code will be added to your project window, the only difference being that an Index parameter has been added. You'll use this parameter to determine which of the dynamically created menus was actually chosen.

For example, say you add the LineWidthPoints menu handler to your application. All you need to do to respond to the various menus being clicked is to

create a menu handler for the LineWidthPoints menu that looks like the following:

```
Sub Action(Index As Integer) As Boolean
  ' The LineWidthPoints Menu Handler
  nLineWidthIndex = Index - 1
End Sub
```

That's it! Because all you're doing is changing the line width, you don't have much other than that to do.

### Adding an Other . . . Menu

You might also want to add an Other . . . menu item to the end of the menu items above. You'd change the code that creates the menu items to include the following line at the end of the subroutine:

```
m = new LineWidthPoints
m.Text = "Other..."
```

We leave it to you to implement handling the Other . . . menu item, if you so choose. Here's a hint: Create a new dialog window containing a single-line text-edit control, which allows the user to type in the value to use. When the user selects the Other . . . menu item (index 9), you'll want to display the dialog and use the value typed in as the line width. It's really not too terribly difficult, give it a shot. Look through the examples in the *REALbasic Developers Guide* and see if you can figure it out. If not, don't be too concerned; just move on to the next step and come back to it later. Maybe you'll have an epiphany in the meantime.

## Adding a Line-Width Selection to the Tool Palette

There are many ways that you could enable the user to choose the line width via the tool palette. We're going to discuss a fairly simple method of doing so: adding a single-line text-edit control to the Tool Palette dialog, which will enable the user to simply type the line width that she wants to use. (Sound familiar?) After you add the text-edit control, the Tool Palette dialog window should look something like the one shown in Figure 16.11.

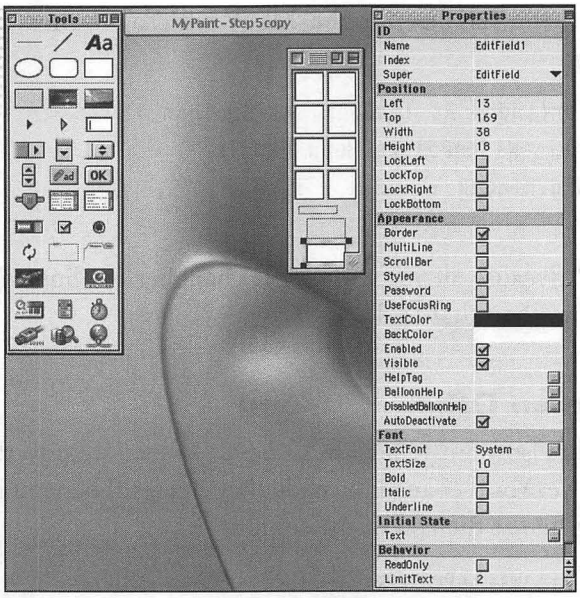



**Figure 16.11**
The Tool Palette dialog with the Line Width control added

All that's left to do is to update the line width using the contents of the selection tool. You can do this in the text-edit control's TextUpdate event handler. Simply modify the ToolPalette.EditField1.TextUpdate event handler to look like the following:

```
Sub TextChange
  app.ActivePaintWindow.nLineWidthIndex = val(me.Text)
End Sub
```

You might want to change the LineWidthPoints menu handler at this point, so that it looks like this:

```
Sub Action(Index As Integer) As Boolean
  ' The LineWidthPoints Menu Handler
  nLineWidthIndex = Index - 1
  app.wToolPalette.EditField1.Text = str(Index+1)
End Sub
```

This change will update the contents of the tool palette's line-width text-edit control so that when the user chooses a line width using the menu, the tool palette will be updated to show the new line width.

NOTE

Admittedly, there are other ways to enable the user to choose the line width on the tool palette. Think about some other ways to select the line width on the tool palette, and see if you can figure out how to make them work.

# The About Box: Patting Yourself on the Back

If you've used your Macintosh for any length of time, you've probably noticed that just about every application adds at least one menu item to the Macintosh's Apple menu—at the very least, an About menu item.

NOTE

The standard location for the About menu item on a Windows application is not the Apple menu—after all, Windows applications don't even *have* an Apple menu. Instead, in Windows applications, the About menu item is usually located in the Help menu, typically the last item in the menu, after a menu separator. To save you some work, REALbasic automatically places the About menu item in the Help menu of your Windows version of the application

When the user selects the About menu item, the program usually responds by displaying a dialog box (most developers call it the "About box") containing information. Contents of the About box might include information about the application and the company that created it, as well as trademarks, copyrights, patent numbers, company contact information (e-mail addresses or URLs), the company or product logo, and so on.

There is no standard for what an application's About box should look like. Some are simple dialog boxes, with a single OK button, that contain only the product name and copyright information, as shown in Figure 16.12. You can create About boxes such as these simply by using the MsgBox function to display the desired effect. Other About boxes, like the one shown in Figure 16.13,


**Figure 16.12**
A simple About box

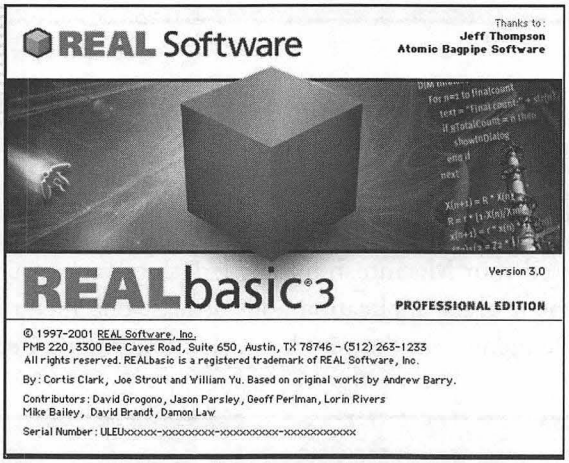



**Figure 16.13**
A more-complex About box

include bitmaps, icons, logos, or other information, like a scrolling list of credits, which requires the design of a new window with multiple controls.

Regardless of how complicated you want to be, you should always include an About box in your application. At the very least, include an About menu item under the Apple menu, enable it, and add a menu handler in your Application class. Then, add a MsgBox function call in the menu handler—something like this:

```
Function Action As Boolean
  ' AppleAboutMyPaint Menu Handler
  MsgBox "MyPaint Tutorial Application"+chr(13)+"Copyright, © 2001"
End Function
```

This way, you've at least claimed a copyright on your product, and people can check the date to see how old your application is. By the way, the `chr(13)` in the preceding code adds a line feed to the text so that the MsgBox displays two lines of text in the About box. Also note that if you're using Mac OS X, the About menu item will appear under the Application menu rather than under the Apple menu.

# Review

Adding these two simple features actually involved a lot of code, even though they weren't really necessary to the application. You may find that certain features don't merit the effort involved in adding them. Use your best judgment to determine how much effort is acceptable for adding a specific feature.

Even though this part of the tutorial might have seemed a bit painful, you learned about some important things: SubMenu items, the SelectColor dialog, how to create dynamic menus, and how to effectively use and change properties in one window from another window. All are important skills that you'll probably use in more than one application that you create in the future.

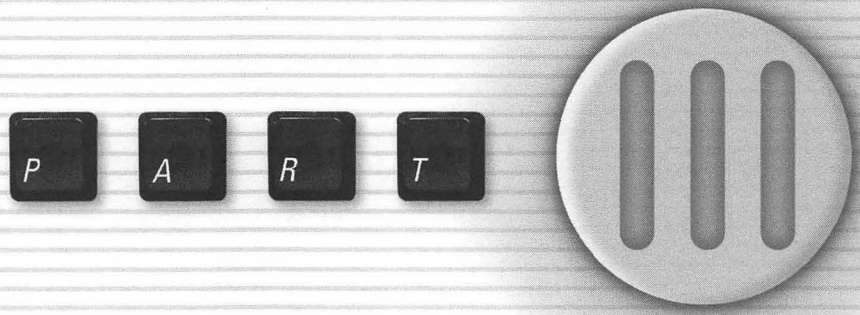
Finishing touches such as the ones discussed in this chapter often add just the right amount of polish to your apps, which can distinguish your tools from others. Remember, though, that you can't get bogged down fiddling around with the cosmetics of your program. If you are always cleaning it up, adding touchy-feely features, and working in new cute ideas, then you'll never release your program. A program that no one ever sees, no matter how nice it looks or what cute features it has, won't ever make a dime. Eventually, you have to stop programming and release it already!

On the other hand, an application that is feature complete but lacks a sufficient amount of spit and polish probably won't be purchased; it will sit on the shelf unnoticed. Knowing when enough is enough can be a tough decision, and one that will take experience to nail down. A good marketing person can make this decision a little easier if he has a better understanding of your potential customers than you do.

Designing, developing, and marketing your application isn't an exact science. As with all artistic endeavors, you'll get better at it as you gain more experience. Don't let the fear of making mistakes keep you from proceeding. Sure, mistakes can be both emotionally and financially painful, but making mistakes can be the best way to learn.

# Beginning Mac®

PROGRAMMING




# The Age of Mac OS X


# 17

# Enter the World of Aqua

## In This Chapter

- In the beginning
- Aqua is more than a pretty face
- A quickie tour of Mac OS X features
- Apple interface guidelines

What makes a Macintosh a Macintosh is the simplicity of its graphical interface. Despite the improvements that Microsoft Windows has brought to PCs, the Mac OS has led in the ease-of-use department. Others in the computing industry have complained of the rather ancient speed and performance of the original Mac OS, but few will deny the success of the Mac OS interface.

It's important that you get acquainted with the successor to Mac OS 9 because, eventually, your applications must run on it. We thought it important to give you a digest on the new operating system so you won't have to go out and buy another book for now. As a programmer, you'll need to know more about the new operating system under the hood.

# In the Beginning . . .

Mac OS X is the latest generation of an evolving operating system developed by NeXT Computer, the company Steve Jobs created after his expulsion from Apple in 1987 as Chairman.

The very advanced NeXT Cube computer used NEXTSTEP, yet another UNIX–derived operating system. After NeXT stopped further production of the Cubes, NEXTSTEP continued to grow and change under the new name of OpenStep, Mac OS X's grandfather.

Apple, struggling to make a revised version of the Mac OS, purchased NeXT in 1996 in hopes of adapting the OpenStep technologies to a new Mac OS. Apple's first attempt, the Rhapsody project, essentially adapted most of OpenStep's features and programming schemes outright. This did not please Mac OS developers because Rhapsody would force them to spend too much time and energy in rewriting applications designed for the original Mac OS. Worse, Apple told the developers that they could only use Objective C, a version of the somewhat scary C programming language. Most developers (until REALbasic developers take over the world, that is) use C++ as their development language.

Rhapsody was renamed as Mac OS X Server 1.0, a powerful, but still complicated server OS. But Apple still had to take the lessons learned from Rhapsody, OpenStep, and irate developers and form something useful to replace Mac OS 8.

### Mac OS X for Intel?

For a time, Rhapsody developer releases included an *Intel-compatible version* as well as a PowerPC version. So far, despite a grass-roots effort on the Web in support of an Intel-compatible version, Apple has not announced any plans for Mac OS X for Intel. Apple has, however, released the core operating system of Mac OS X as open-source software known as *Darwin*. If you're interested in helping the Darwin project move toward a Mac OS X–like experience on other computer hardware, visit Apple's official Darwin Web site at http://www.darwin.org.

Apple went back to the drawing board and fashioned portions of OpenStep and Rhapsody's technologies so that developers could rewrite only a portion of their original Mac OS applications to make them work in a new Mac OS. They also created the necessary programming tools to develop applications designed especially for the new operating system. Developers were again given Objective C as one development environment, but were also given Java, an impressive programming language created by Sun Microsystems. Java gave developers new directions, as Java code can be created once and then *recompiled* (converted from program code to an application for another computer) for any other computer that can use Java. That means you could write a Cocoa application for Mac OS X using Java, then port the code to Windows or Linux and have the application run on these operating systems with very little modification.

We'll talk more about Cocoa and Java in Chapter 20, "The Cocoa Environment."

## Aqua is More than a Pretty Face

All versions of UNIX use an application called a *window manager*. And, almost all members of the UNIX family use *X Windows* for their window manager. As you can guess, a window manager provides the mechanics necessary to draw a basic graphical interface on a computer display.

X Windows provides a very simple interface called *twm*, and it's not a very pretty sight. Fortunately, X Windows is highly flexible and allows other interfaces to be grafted to it. Linux, the UNIX clone that's popular in the open-source operating system arena, offers two very popular graphical shells for X Windows: KDE and GNOME. Other versions of UNIX used for graphics professionals, such as IRIX from Silicon Graphics, have their own graphical shell.

## View (and Print) Different

Apple, never being a company to follow the status quo, chose something else for the Mac OS X window manager. (See Figure 17.1.) They decided on combining three technologies to make a stunning new interface environment called *Quartz*.

Like X Windows, Apple divided their Quartz plan into two components: the window manager and the graphical shell library. The *Core Graphics Server*, as Mac OS X's window manager is called, handles events and manages services from the operating system and the graphical shell.

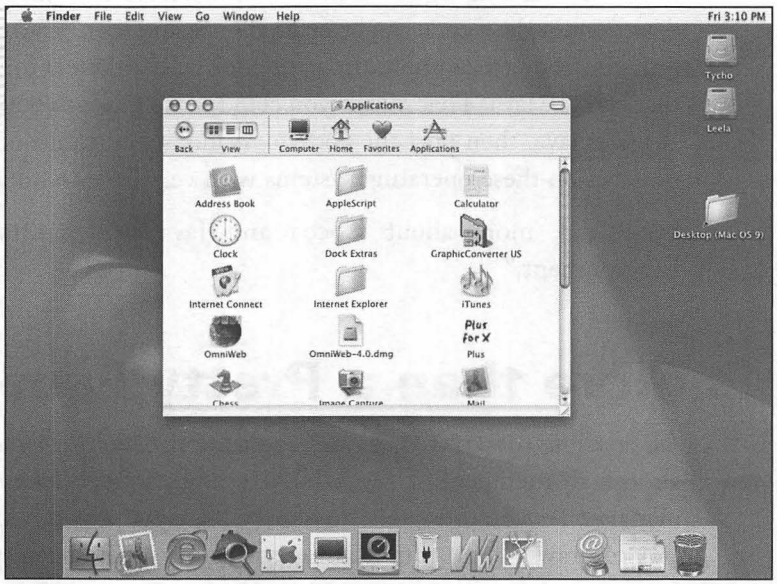



**Figure 17.1**
Mac OS X's Aqua interface is created through a series of technologies collectively known as "Quartz."

The *Core Graphics Rendering* component of Quartz was given the task of providing the tools needed to make the 2D elements on a display. For this, Apple would employ three additional graphic libraries.

The first element wasn't hard for Apple to consider. OpenStep enjoyed a rather revolutionary graphic engine named Display PostScript. It's an Adobe technology that allows extremely accurate reproductions of text and graphics on a printed page. Display PostScript simply reversed that process and gave OpenStep a smooth, detailed interface.

For the second element, Apple looked to an improved kind of Display PostScript. The answer was another Adobe technology: Portable Document Format, or PDF. You've used this technology in Adobe Acrobat Reader to view documents that can be read on any computer with an app that can open PDF files. Like a PostScript–printed page, PDF gives you a what-you-see-is-what-you-get experience whether you print or view the document.

The PDF engine in Quartz not only provides the precision display that PostScript generation creates, but also gives a bonus for Mac OS X in the form of a new printing system. The Mac OS print system supports PDF as a native element of the operating system where all applications (except those in the Classic environment) can save their document information as PDF documents when possible. The print system supports PostScript printers as well as inkjets and other non-PostScript printers, and handles print previews for all Carbon and Cocoa applications. The Classic environment isn't supported, however; Mac OS 9 must use its LaserWriter or other printer drivers.

OpenGL, the third element of the Core Graphics Rendering component, was a no–brainer for Apple. When it comes to 3D graphics such as modeling or monsters you'd find in the latest realistic computer games such as Quake 3, Apple's continued support of OpenGL was critical to present and future applications available on other systems that don't depend on proprietary graphic APIs such as Microsoft DirectX.

The fourth element of Apple's graphics technology plan was QuickTime, Apple's popular multimedia software used for creating and viewing video, animation, music, and the like.

Much of what you would use from Quartz, as a beginning REALbasic developer, is transparent to you since REALbasic provides you with all the necessary interface tools built in Mac OS X.

# A Quickie Tour of Mac OS X Interface Features

When a traditional Mac OS user views Aqua for the first time, they are typically enchanted and confused at the same time. The Mac OS X interface is familiar but has many new features that users must adapt to, and developers must learn to utilize in a Carbon or Cocoa application. There are many books on learning Mac OS X available right now, so we won't get into a detailed analysis on all the features—just the differences you should remember when creating an app that will work in Mac OS 9 as well as Mac OS X.

## Windows, the Finder, and the Dock

For the most part, the Mac OS X desktop isn't extremely different from previous Mac OS versions, but it returns to the desolation of the first Mac OS of 1984. The Finder remains a separate application, and its distinction from the desktop is more apparent in Mac OS X.

Finder windows can only be moved by their title bar, unlike in Mac OS 8 and later. Finder window contents can be shown in three different configurations: icon view, list view, and the browser view (see Figure 17.2). The three buttons on the left of the title bar are, from left to right, close, minimize, and maximize. A toolbar containing various Finder settings can be shown in a Finder window and can be toggled on and off by the clear button on the right side of the title bar.

The WindowShade feature has been replaced by the minimize effect, which moves the window in a dramatic "genie" effect from the desktop to the right side of the Dock. With this change, pop–up windows in the Mac OS are a thing of the past.

The Dock combines the functions of application switching with a launcher. Application icons can be dragged to the left side of the border where the Dock makes an alias of the application for launching. The right side of the Dock holds document windows from these applications. The Dock's appearance can be dynamically different from user to user by changing the Dock's size and icon magnification settings from the Apple Menu's Dock settings. Officially, Apple does not allow the Dock to be moved from its bottom center location in Mac OS X version 10.0.3, however, third-party utilities can reactivate these

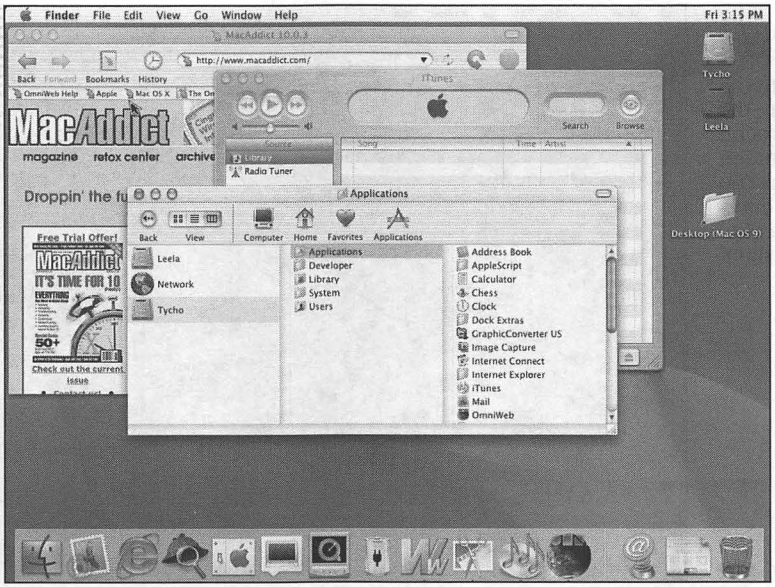



**Figure 17.2**
The Mac OS X desktop in action. Note the Dock location, the Trash icon, and the relative size of icons in the Finder window. Items are being shown in the new Browser view.

dormant features, allowing the Dock to float on any edge of the screen. Version 10.1 of Mac OS X removes the stationary limitation of the Dock. The Trash resides in the Dock and cannot be moved to the desktop.

The Finder's preferences are changed from the Preferences command under the Application menu when the Finder is the active application. Finder Preferences allow you to change the desktop picture (no Mac OS 9 appearance themes are directly supported in Mac OS X) and change a couple of additional settings. With these preferences, it is possible for a user to hide any mounted disks from the desktop. If the user also hides the Dock, running applications, and closes all Finder windows, there will be a Mac OS X desktop with absolutely nothing on it but the menu bar. (Figure 17.3) Keep this in mind as you develop and document applications that make assumptions about a user's desktop appearance.

# Menu Changes

When developing for REALbasic, most of the menu changes are transparent to you when you compile your project as a Mac OS X/Carbon application. Keep in mind, however, key changes in application and Finder menus in Mac OS X.

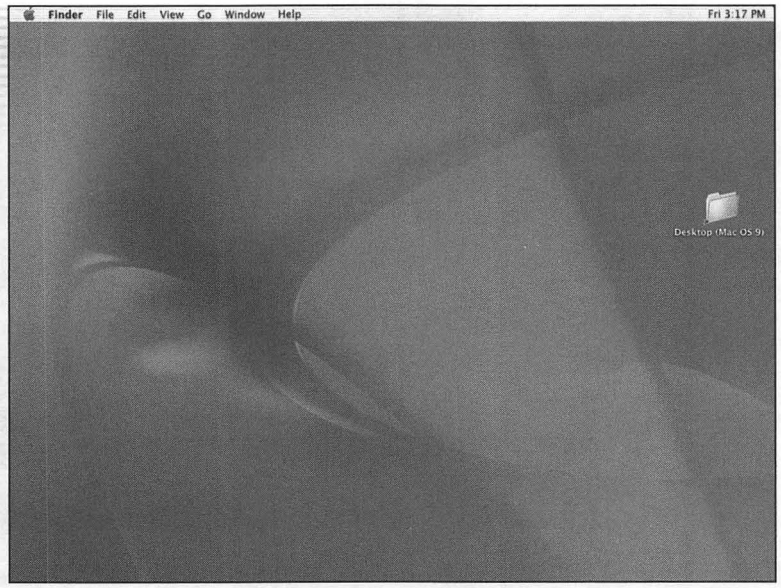



**Figure 17.3**
The Mac OS X desktop, with the Dock hidden, the Show Disks on desktop setting turned off, and no Finder windows open, is a very unusual sight.

The Application menu is located immediately right of the Apple menu. This menu should be the location of your application's Preferences command, if you happen to have any preferences available to the user. The Application menu no longer handles switching between applications; that's now the job of the Dock. (See the following section, "The Dock" for more information.)

The Apple menu in Mac OS X combines the commands of the Special menu in Mac OS 9 with some features available from the old Apple Menu Options control panel. Recent Items shows recently opened documents and applications (but not servers). The Restart, Sleep, and Shut Down commands now reside here in addition to several new commands, including Force Quit, and Logout. Unlike Mac OS 9, the contents of Mac OS X's Apple menu cannot be modified directly by users.

## Same Stuff, Different Places

Mac OS 9 allowed users and developers to place all kinds of things in practically any location. In Mac OS X, this is not allowed to prevent unwanted and potentially disastrous changes to Mac OS X's system files or other user's data.

Following the UNIX tradition of separate documents and workspaces, each Mac OS X user has a login and password which allows them access to their Home folder. The Home folder name is based on a condensed version, or *short name*, of their full name created when the computer administrator (the owner) creates their login account. Each account stores different Finder, Dock, desktop picture, and system settings.

Each Home folder has a built–in set of folders as illustrated in Figure 17.4. Users can add additional folders here and in subfolders. This feature can be advantageous to developers who develop only Mac OS X applications since the Mac OS provides a consistent location for user documents.

As in other forms of UNIX, Mac OS X is very strict about access to folders without proper authorization. Keep in mind that your applications can only change what the user's permission level allows them to change. For instance, each user has a Library folder, which works much like the Preferences folder in the Mac OS 9 System Folder. The Library/Preferences folder is user–customizable, as is the user's Library/Fonts folder. However, Mac OS X also has a Library folder that's restricted in most cases, and other user accounts and their folders cannot be altered without administrative privileges.

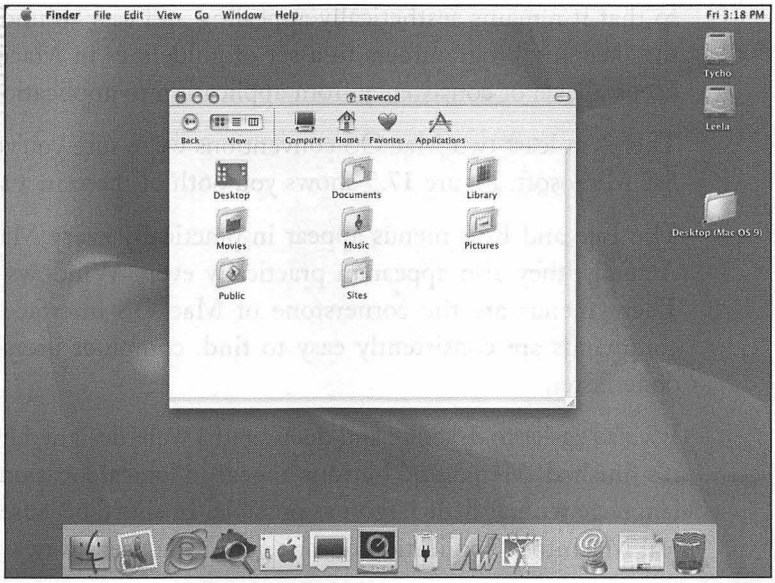



**Figure 17.4**
The Home folder is a Mac OS X user's storage spot for their documents. All applications are stored in the Applications folder and are available to all.

System extensions as used in Mac OS 9 no longer exist. Your applications in Mac OS X are self–sufficient entities that don't leech off any common system files, for the most part. Preferences that were adjusted by Mac OS 9's control panels are now available through the System Preferences application from the Apple menu. As with files, applications and users must have administrator privileges to change some preference settings, such as network and startup disk settings. If you happen to be porting an older REALbasic application that might have created Mac OS 9 extensions and control panels, you will have to integrate that functionality directly into the application or seek another alternative.

# Apple Interface Guidelines

Ever look at a typical application from the Microsoft Office suite? It's filled with all sorts of controls and menus. To us, Microsoft Word seems more like the cockpit of the Space Shuttle than a word processor. We wanted a *computer* program, not a space program!

Why do many computer users prefer the Macintosh interface and applications over those found in Microsoft Windows? The answer is obvious: simplicity and ease of use. Since 1984, when the Mac OS was created (it was known simply as the *System* back then), Apple has taken the time to refine the Mac OS so that it remains aesthetically appealing and easy to understand. Apple feels that keeping programmers to a set of guidelines in Macintosh programming keeps a level of consistency from application to application.

There's at least two Mac OS conventions we're sure you've noticed. In fact, so has Microsoft. Figure 17.5 shows you both of them in a single shot.

The File and Edit menus appear in practically every Macintosh application. Actually, they also appear in practically every Windows application as well. These menus are the cornerstone of Mac OS interface consistency. When commands are consistently easy to find, computer users can get their work done faster.

It's also easier to describe and document a well–designed Mac application once it's finished. Menus and buttons appear in logical locations. Labels use simple language with as little jargon as possible. In short, because of its design, a typical Macintosh application doesn't get in the user's way.

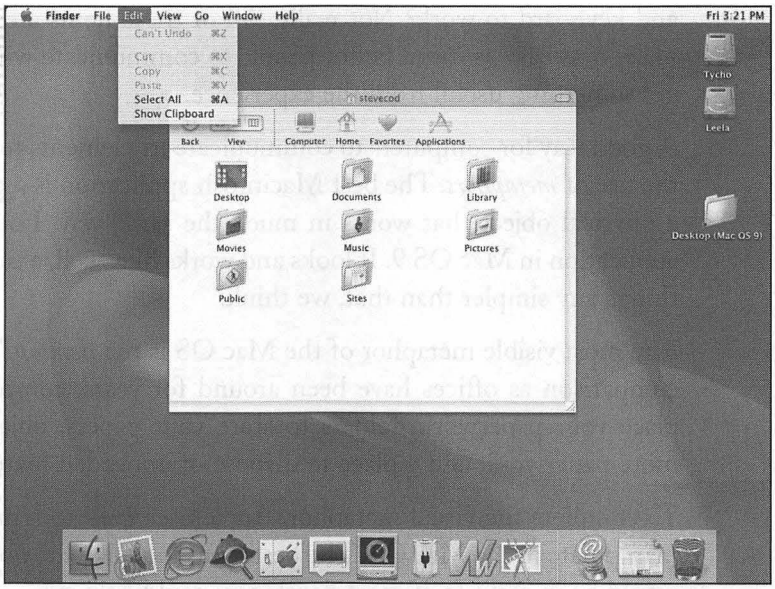



**Figure 17.5**
The File menu, and Edit menu, with its Cut, Copy and Paste commands, are always where you expect to find them in almost every application.

If you're considering the idea of selling your finished application, you should remember that there are plenty of Macintosh users out there that love a great application. Quite a few of those users are also critics that won't hesitate to tear you and your application design apart if it fails to be logical, consistent, or understandable. Oh yeah—Mac users will certainly berate you for making a buggy app, too. It pays to do your homework in design.

Apple has extensive documentation on human interface design, which are available free from Apple Developer Connection on the Web at http://developer.apple.com. (Remember that you need to register on the site to have access to these and all other documentation. Registration for online access is free.) Let's highlight some of Apple's interface guidelines.

## Rule 1: Stick to Metaphors in Your Application

Apple describes Aqua as a *human* interface, not a computer interface. That makes sense. After all, does a computer really need buttons, windows, a mouse

and keyboard to work? Not really. All of these items are devices intended to make it as easy as it can be for people to communicate with the computer and get something useful from the experience.

A good way for computers to communicate its elements to a human is through the use of *metaphors*. The best Macintosh application is a perfect metaphor for a physical object that works in much the same way. Look at the Calculator application in Mac OS 9. It looks and works like, well, a calculator! Can't make things any simpler than that, we think.

The most visible metaphor of the Mac OS is the *desktop*. This was a powerful comparison as offices have been around for years, complete with a place to place your paperwork, folders to store your papers, objects to use to create more paperwork, and a place to dispose of unneeded materials.

To complete the visual metaphor, Apple uses *icons* to represent the items you manipulate in the Mac OS. The best icons look like the object or function they would be in real life, if most developers could help it.

Over the years, the Apple desktop metaphor became strained. Dozens of improvements by third-parties and adjustments by Apple were made to the Mac OS to improve a thing here and there. In the 1990s, Apple began adjusting the Mac OS to appear more like Microsoft Windows so that Windows users who make the move to a Macintosh would feel more comfortable. In the end of the original Mac OS development, starting with Mac OS 9, the complication of the desktop became readily apparent to Apple.

Mac OS X returns to the simplicity of the first Macintosh interfaces to make it easier for users to understand their desktop once more. Aqua's introduction raised much praise and criticism from the media and Apple's faithful. But Aqua is really a blast to the past. Our computers have become so complicated that it seems to take more time for an application to explain how it works than a user will actually spend using the application. In case you've never realized it, Figure 17.6 shows you what System 6.0, the Mac OS of 1988, looks like. Even if you've never used this older Mac OS yourself, we bet that you could use this version without any training. The Mac OS desktop, uncluttered in System 6.0, and uncluttered in Mac OS X with Aqua, works like your bicycle: you'll never forget how to ride it.

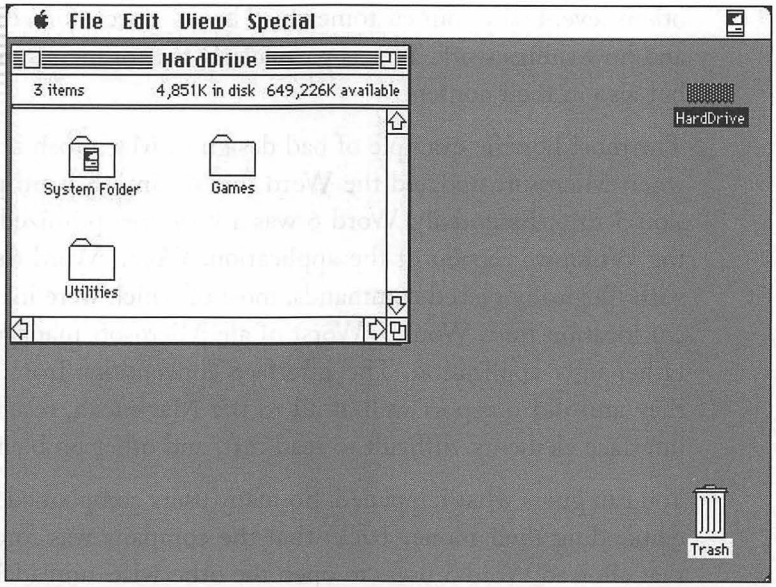



**Figure 17.6**
System 6.0 was, for many Mac OS users, their first bicycle. Mac OS X is merely a flashier bicycle with extra horsepower and better paint.

## Rule 2: Keep a Logical Design with Aesthetic Consistency

Nothing ticked off more Macintosh users about Mac OS X Public Beta than the disappearance of the Apple menu. The Apple menu of Mac OS 9 was a highly customizable way to configure the Mac OS and launch applications. Without this simple menu, many Mac OS X Public Beta users suddenly found it very difficult to maneuver about.

Apple eventually returned an Apple menu to Mac OS X, although it is not customizable as its Mac OS 9 predecessor. Apple managed to maintain its goal of simplifying the Mac OS but also realized that moving such a powerful tool that has been available since the beginning of the Mac OS was tantamount to changing around the controls of a car.

As you create and evolve your application, remember Apple's lessons from Mac OS X. Think of a logical or aesthetically pleasing design for your application's menus and components. Then, stick to it. If your program is useful to

others, eventually your customers will become accustomed to where things are and how things work. This is particularly true of menus, not only in location, but also in their content.

The most horrific example of bad design in Macintosh history came in 1995 when Microsoft updated the Word for Macintosh word processor from version 5 to 6. Essentially, Word 6 was a very un–optimized and buggy port of the Windows version of the application. Worse, Word 6's menus were filled with illogically placed commands, most of which were in a completely different location from Word 5. Worst of all, Microsoft managed to make Word a rather ugly application. The interface conventions from Windows (such as they are) did not port well at all to the Macintosh, resulting in overlapping interface elements, difficult to read text, and other problems.

You can guess what happened. So many users complained to Microsoft (some demanding their money back) that the company was forced to write a filter that allowed Word 5 users to open the otherwise–unreadable Word 6 files.

A positive example of good menu maintenance falls again to the Mac OS, in this case, the Finder. Over the past 17 years, Apple has added only two menus: the application menu in System 7, and the Window menu in the Mac OS 9.1 update for later Mac OS X compatibility once a user installed the new operating system. Some of these menu's contents changed slightly to meet the times. We bet you barely noticed. Hopefully, any future menu changes you offer in your updates of your application will go just as unnoticed until needed.

## Rule 3: Forgive Mistakes and Allow Reversal

One thing that many Windows applications seem to do is punish a user for a mistake in a command selection. We're sure that's not intentional, but it shows the distinction between Microsoft and Apple interface guidelines.

If you're like most Macintosh users, you like to play with your new applications before you put them to use. In the case of games, you literally play with them. In any case, the last thing you want to do is accidentally change a setting in the application that completely rearranges the work you spent hours creating.

Exploration is a crucial component to learning a new tool and determining how it meets your needs. One of the most powerful commands in a Macintosh

application is *Undo*. As you know, the Undo command will reverse the last action you performed in an application. Some programs, such as the well-written Microsoft Word 2001 for Macintosh, will undo a series of actions.

As you create your application, provide your users the ability to reverse their selections or options. Undo options provide users some comfort in knowing that their actions are not completely etched in stone. Whenever possible, alert the users using message boxes when they are about to perform an action that cannot be undone.

Be careful of overdoing this guideline, however. Microsoft Word allows repeated undos by using temporary files containing the present changes. Worse, with its "Allow Fast Saves" feature enabled, a document is not fully saved until you quit the application. In an application crash or if the user has poor document management, data could be lost. Be sure that your application reasonably ensures the user's data or provides warnings or notes to inform the user.

## Rule 4: Use Dialogs Wisely

Language plays a proper role as well in your application. Emotionally charged words or computer jargon appear in the worst applications by far. Your job is to help simplify information while keeping your app's visual appeal.

Informing or instructing the user to make changes to documents or to your application as a whole must be clear and concise without being overly technical. Apple also wanted to eliminate the practical hijacking of a computer when error or open/save messages appeared on the screen. Mac OS X has refined their concept of message windows that appear to inform or direct a user to a particular course of action. These windows are known collectively as *dialogs*.

There are actually three kinds of dialogs in Mac OS X. The first type are *document modal* dialogs, also known as *sheets* (see Figure 17.7), which swish out from the bottom of a document title bar when a user performs an action that requires a decision to be made. Similar to its Classic counterpart, document modals prevent further changes to the document and require the user to handle an action in regards to the entire document. For instance, attempting to close a document window in a Mac OS X application brings up a familiar "Do you want to save changes" sheet at the top of the document window, but unlike original Mac OS applications, you can ignore the message indefinitely while opening other applications, documents and the like.

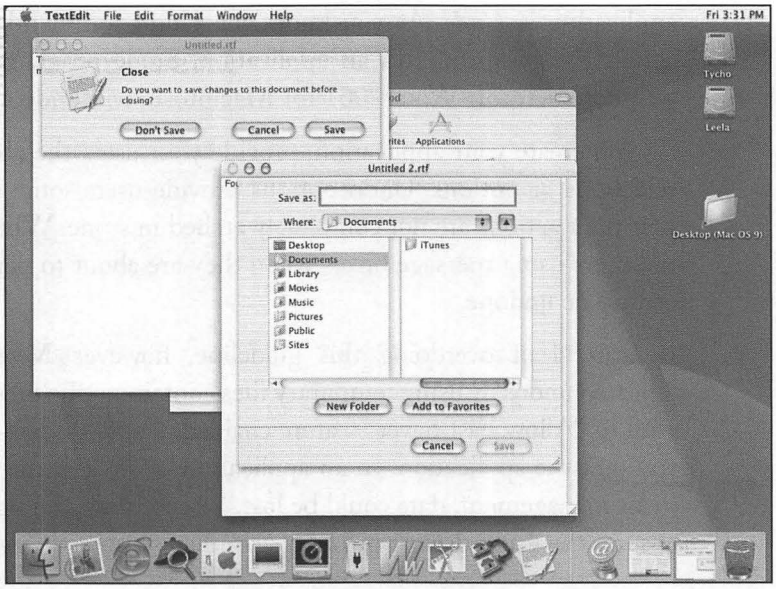



**Figure 17.7**
Sheets hold alert or dialog boxes in Mac OS X applications, both shown here. They're modal only to the document window, so you can move to other windows or applications.

Remember, sheets are a feature of a *Carbonized* application running natively in Mac OS X. Applications running in Classic still use the old-style dialogs and will still prevent a user from switching to another Classic application, but the user can still switch to or launch other Mac OS X applications.

Open and Save dialogs also appear as sheets that drop from a document window. Normally these windows are minimized to show only the most pertinent information: the name of the saved file, the location where the file will be saved, and the appropriate Save and Cancel buttons.

The second type of dialog, an *application modal* dialog, prevents further use of an application until an action is chosen. Application modal dialogs commonly appear as *alerts*, floating windows which immediately ask a user for a critical decision. In Mac OS 9, users routinely encounter Save alerts that appear when a user attempts to close a document window before they have saved their work with the Save command. Figure 17.8 shows a typical one from Microsoft Word running in Mac OS 9.

We have two problems with alerts in Mac OS 9. The first problem is that Microsoft applications display alerts with generally useless error information, such as "Error in writing file to disk." Such a message is not only uninformative, but typically causes a user to, well, freak out because they can't understand

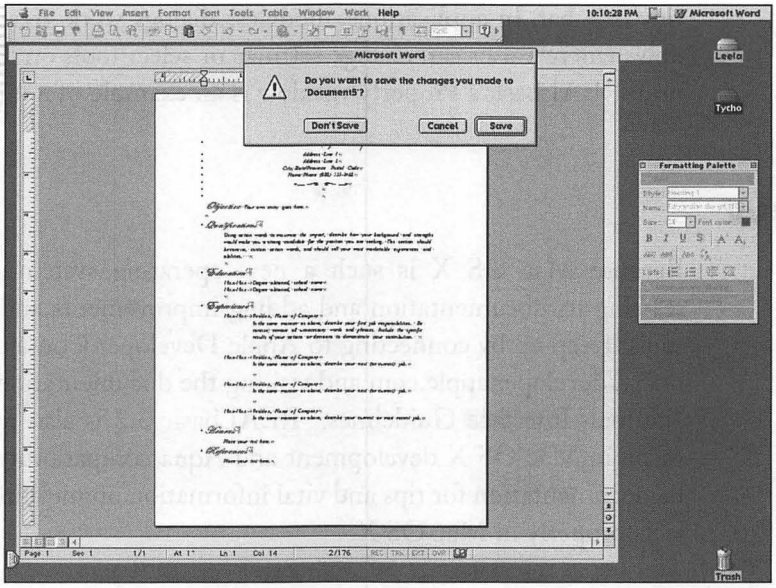



**Figure 17.8**
Closing a document window in Word 2001 shows you this alert, which prevents further use of the application until a decision is chosen from the alert.

what the application is telling them. Most Macintosh applications will report the problem and also suggest a course of action to fix the problem. Windows applications can do this as well; the problem is not the code but the documentation that developers write within the application. That means, in short, that alerts in your app should explain *what* is going on, *why* it might be occurring, and *how* to fix the issue.

The second problem was that alerts in Mac OS 9 would also stop the user from doing any other action with the computer until the alert box was dismissed. S*ystem* modal behavior such as this is discouraged in Mac OS X. Alerts in Mac OS X applications allow users to move from application to application while the alert remains "attached" to the application's view.

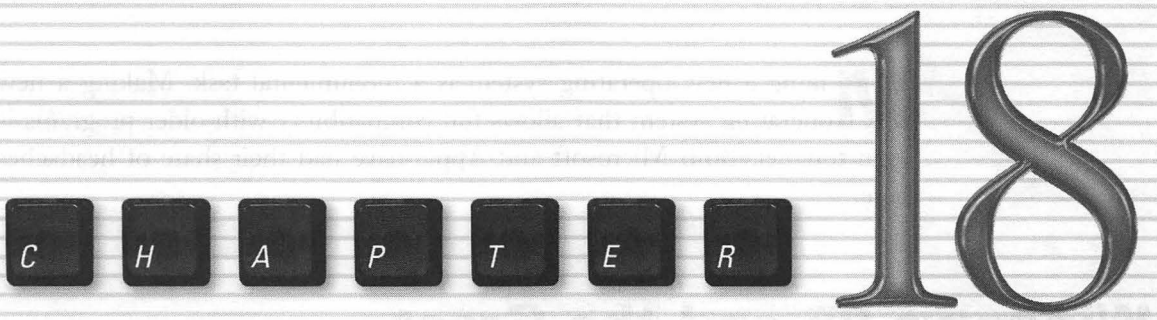
Apple encourages the generous use of white space around your alert information to minimize clutter. You can also place more explanatory text in a smaller font below the primary message. Apple encourages developers to use your application's icon in most alerts. In rare instances where a serious problem may occur in an application, Apple advises that you use the Caution icon in place of the application's icon in an alert.

The third type of dialog is the modeless dialog. Generally, a modeless dialog is a smaller window, complete with minimize, zoom, and close controls on a

tiny title bar. In applications, modeless dialogs work great as preference windows where a user can change settings or select tools on the fly in an application. REALbasic's Property window is an example of a modeless dialog.

# Review

Because Mac OS X is such a new operating system, Apple is routinely revising its documentation and adding improvements and new programming tools. Keep up by connecting to Apple Developer Connection on the Web at http://developer.apple.com and visiting the documentation section. Search for "Human Interface Guidelines." REALbasic 3.2 is also relatively new to the scene in Mac OS X development and Aqua compatibility, so be sure to read its documentation for tips and vital information on meshing your Carbon projects properly in Mac OS X.




# The Classic Environment

## In This Chapter

- Windows and the great compatibility problem
- You can't play vinyl records in your compact disc player
- The 16-bit egg and the 32-bit chicken
- Apple's turn

Making a new operating system is a monumental task. Making a new operating system that allows full compatibility with older programs is even tougher. Both Microsoft and Apple have had their share of headaches here.

# Windows 95 and the Great Compatibility Problem

In 1994, Microsoft was midway through developing Windows 95, its next-generation PC operating system. Windows 95 promised fast performance and a level of crash-proof behavior unlike any other operating system made for home and business users.

To do that, Microsoft had a programming challenge to overcome. Windows 3.1, the previous version, was merely an interface over an old operating system, MS-DOS. For its time, MS-DOS was reliable, but nothing like the Mac OS that used a graphical interface—icons, buttons, windows, and so on. Windows 3.1 simply draped some windows and icons over MS-DOS but added no improvements.

MS-DOS was a 16-bit operating system. *Bits* are elements of a computer program or memory. The more bits that a computer can manipulate, the more realistic and powerful the computer and the applications become. Remember the old computer video games of the 1970s? The edges of the pictures on the game screen were really jaggy, with few colors. Compare that with 32-bit games like Quake 3 Arena with dramatically realistic shapes and colors.

Being only a 16-bit operating system meant that the MS-DOS programs made for it weren't as versatile, suffered from instability, and couldn't take advantage of advanced operating-system features. MS-DOS was designed to use only 640KB (that's kilobytes, or about 1000 bytes) of memory. Today's computers require 64MB (megabytes) or more of RAM to operate.

Also, MS-DOS and Windows 3.1 couldn't handle the larger hard-disk drives that began to appear in the market. Although there were a few utilities here and there in the computing world that could help MS-DOS over one of its many limitations, Microsoft knew that Windows 3.1 was limited until it could be modernized.

## You Can't Play Vinyl Records in Your Compact-Disc Player

A challenge to making a new computer and its operating system is designing the computer in such a way that it can run programs used on a previous computer or operating system.

Remember the original Nintendo computer game console? It used square, flat game cartridges for its games. When Nintendo introduced the Super Nintendo game console, game players were dismayed to find that their game cartridges wouldn't work in the more advanced Super Nintendo's game system. A few years later, Nintendo completely skirted the compatibility issue again by making Super Nintendo cartridges incompatible with the Nintendo 64 game console.

Compare this to Sony Corporation's original PlayStation game console. Games for this machine were stored on CD-ROMs much like the CD-ROMs used in computers. Recently, Sony introduced the PlayStation 2 (or PS2), a game console with many advancements over its predecessor. Since CD-ROM drives use the same disk sizes as DVD-ROM drives (which the PlayStation 2 uses), Sony's PS2 accepts old PlayStation games in addition to PS2 games.

In the mid-1980s, the compact disc appeared and changed the way we listened to music forever. The machines you needed to play them replaced the turntables for vinyl records and albums. The record industry skirted the compatibility issue here as well, because trying to make vinyl albums (which played music from an *analog*, or non-digital format) work in a digital compact disc player was more trouble than it was worth. Luckily for the recording industry, the quality and advantages of audio CDs over vinyl made it worthwhile to buy new, somewhat expensive CD music players.

## The 16-Bit Egg and the 32-Bit Chicken

Microsoft couldn't skirt the compatibility problem. Many homes and businesses relied on the MS-DOS programs they used in Windows 3.1; that meant Microsoft had to design Windows 95 in such a way that MS-DOS could operate as part of Windows 95 without affecting Windows 95's speed

and reliability. Even so, a chicken-and-egg problem was hatched. Many MS-DOS programs made long before Windows 95 would not be able to handle the existence of Windows 95.

When Windows 95 arrived in August 1995, it was praised for its new appearance and greater stability. Many users of MS-DOS programs, however, soon found problems with the new operating system. Sometimes, running MS-DOS programs would make Windows 95 slower. In a few cases, the MS-DOS program would not run at all.

Microsoft tacitly admitted that, in order for Windows 95 to meet its shipping deadline, and to ensure that the new operating system as a whole would function, it had to cut a few corners. Basically, Windows 95's version of MS-DOS was still fairly integrated within the new 32-bit parts of Windows 95 itself. Since the new version of MS-DOS wasn't really walled off in the way Microsoft preferred it, Windows 95's stability and speed did suffer for a time.

It wasn't until Windows Millennium Edition (Windows Me) arrived in 2000 that the last parts of MS-DOS were removed in an attempt to improve speed and performance in the consumer version of Windows. Sadly, however, no visible performance increases ever came from the consumer Windows client. Perhaps Windows XP, an upcoming version based primarily on the refined Windows NT code, will prove itself as a compatibility winner.

# Apple's Turn

Apple had enough operating system–project failures by the mid-1990s to make people in the computing community worry that Apple would never create another operating system. When Apple bought Steve Job's NeXT computer company in 1996 and took in its OpenStep operating-system technologies, it was a breath of fresh air. Work soon began on another possible successor to the original Mac OS, code-named *Rhapsody*.

In OpenStep, Apple found a strong system kernel (the heart of an operating system) that could keep up with many computing tasks at once. Apple decided that one of these tasks could be to run programs designed for the original Mac OS while the operating system performed its business.

## It's Virtually Simple

Apple decided that the best way to get the original Mac OS to work with Rhapsody was to wall off the old operating system from the new. The plan would still allow the old operating system full access to the hardware as if the original Mac OS was the only thing running. This programming principle is known as a *virtual machine*. A virtual machine would run at near-native computer speeds. If the virtual machine was designed well, few people could tell it from a computer running the original OS single-handedly.

Apple first created its virtual machine for the Rhapsody project, which was later named *Mac OS X Server 1.0*. The "Blue Box," as Apple called its original Mac OS compatibility environment, ran a version of Mac OS 8. The Blue Box wasn't without its faults and limitations, but it successfully kept to its business without affecting Mac OS X Server.

## The Blue Box Goes Classic

Rhapsody/Mac OS X Server wasn't a big hit for programmers. New programs designed for the new OS had to be written using Objective-C, a version of the C programming language that few people knew. In addition, Mac OS developers practically had to rewrite existing programs so they could run in Rhapsody.

As a result, few developers wrote much Mac OS X Server 1.0 software. It would take too much time and energy. So Apple spun off Rhapsody to form the Mac OS X Server project, then proceeded to take some of the lessons learned to build yet another new OpenStep-inspired OS.

As the pieces of Mac OS X came together, the original Mac OS–compatibility issue came up again. The Rhapsody Blue Box would be needed once more, with a few refinements, and a name change: the *Classic environment*.

## Installing Mac OS X for Classic

During the end of Mac OS X's completion, Apple realized that, as with the PowerPC chip transition in the early 1990's, few applications would be available on Mac OS X's release that would run natively in Mac OS X. With each copy of Mac OS X, you receive a copy of Mac OS 9.1 to install on your computer to run traditionally, or as the basis of Classic.

## Virtual Machines Versus Emulation

A virtual machine works almost as quickly as an operating system running in full control of the computer's hardware. Virtual machines send the same instructions to a computer processor in the same format as the dominant operating system. When Mac OS X is running, the Classic environment allows the Mac OS 9 installation to behave as if it is the only operating system in place, and allows the user to interact with OS 9 on almost all levels. Most importantly, Classic runs at near-normal speeds since it sends the same kind of processor instructions as Mac OS X would to the computer processor. No translation or conversion of instructions from a virtual machine is needed.

*Emulation* simulates hardware and some software elements of a computer. For example, Connectix's Virtual PC works as a Macintosh application that simulates the hardware and system responses of an actual Pentium-style PC (see Figure 18.1), complete with sound card, network card, processor, and video card. Virtual PC is slower than a virtual machine or an actual PC because it must take the Pentium processor instructions created in the simulated PC and Windows environment and convert them into something that a Macintosh's PowerPC processor can understand. Likewise, Virtual PC must translate instructions from the PowerPC processor back into Pentium instructions for the emulation to complete its work.

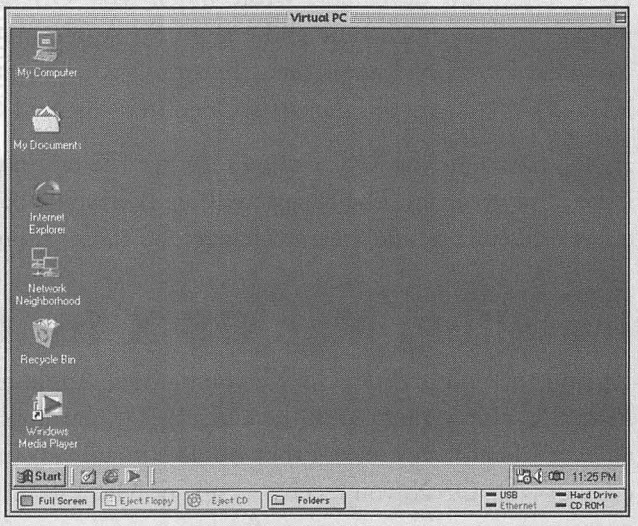



**Figure 18.1**
Virtual PC running on a Macintosh. Yes, it's always a weird sight to see Windows running on a Macintosh. Thank goodness you can turn it off.

Emulators almost always have some performance hit. From our experience, Virtual PC is about three times slower than the Macintosh it runs on. On a 500MHz Power Mac G4, Virtual PC works around166MHz when running most PC applications, give or take some programs that have lots of sound or graphics. Remember that emulators such as Virtual PC have programming that simulates a graphics card and sound card, too. With a really complex application such as a game, so many parts of Virtual PC are translating that the whole application begins to slow down dramatically.

Not all emulator applications have issues, however. Because today's computers are much faster than earlier ones, there are many game-console and computer emulators available for Macintoshes and PCs that play old computer games such as Pac-Man at the same speed and quality as the original. The hard part in making these emulators is trying to keep them from being too fast—computing power today would make Pac-Man zip around the screen at hundreds of miles an hour!

Apple recommends that you install Mac OS 9.1 before installing Mac OS X. You must use Mac OS 9.1 or later if you intend to run applications in Mac OS X that aren't designed with Mac OS X in mind. Earlier versions of the Mac OS will not be accepted as a Classic environment. You can use the Mac OS 9.1 installer CD to update your hard disk if you are using Mac OS 9.0.4 or earlier.

Mac OS 9.1 will rearrange previously created Mac OS folders found at the root of your hard disk to make it compatible with Mac OS X's folder structure. If you have Mac OS 9.1 installed on a separate partition, this is only a minor nuisance. If you install Mac OS X in the same location as Mac OS 9.1, the Mac OS X installation moves the entire Mac OS 9.1 disk installation into its own folder on the disk. We've read more than our share of reports on problems with Mac OS X and 9 installations on a single volume. We recommend installing Mac OS 9 and X on separate partitions, as shown in Figure 18.2.

Classic remains a virtual machine in Mac OS X. When you start up any application designed for Mac OS 9 and earlier, Mac OS X will activate the Classic environment. When you start up Classic for the first time, Mac OS X asks you if it can add a few new components inside Mac OS 9's System Folder to make

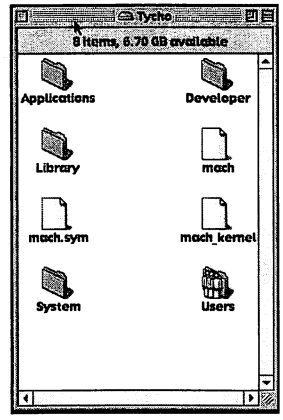



**Figure 18.2**
It's a good idea to have two separate hard disks or divide a single hard disk into two or more partitions for Mac OS X installation. Here, my installation of Mac OS X resides on the drive partition named *Tycho*.

## Separate Drives Are Probably Best for Installation

The Public Beta of Mac OS X made more radical changes to a single-disk installation on one of our computers. After a time, a problem occurred on the Mac OS 9 side that prevented the computer from starting up in Mac OS 9. Because the installation was combined, there wasn't much of a choice other than to reformat the drive and reinstall Mac OS 9 and Mac OS X—this time on separate partitions. Yep, it was pretty painful. It helps to be a certified Apple Service Technician in these moments.

In the PC world, one of us met a similar fate on a Windows 3.1 PC upgraded to Windows 95, which recommended updating the C drive where Windows 3.1 existed. Even after Windows 95 was removed and the C drive downgraded to Windows 3.1, the PC never seemed quite the same.

Because the release version of Mac OS X is such a radically different operating system, it's probably best to install it to a separate hard disk or partition. But no matter how you install Mac OS X, the Macintosh hardware still recognizes that there are two operating systems through Open Firmware—the boot software in the logic board, unlike Windows 95, which completely assumed control of a PC, leaving previous operating systems that were installed dormant.

it work in the Classic environment, as shown in Figure 18.3. On startup, a temporary window, normally closed, appears. Open it, and you see the old face of Mac OS starting up in a window that's basically a monitor window within the monitor. Everything operates in the same way as when Mac OS 9 works alone, right down to the icons for extensions appearing as they load into Classic.

Once startup is complete, the Classic startup window disappears, and moments later, your Classic application appears. The wild thing about Classic is that Mac OS 9's menus, buttons, and windows are active when you are using a Classic application, as shown in Figure 18.4. When you switch to the Mac OS X desktop, document windows in Classic move to the foreground, but the Mac OS X menu bar returns.

# What Classic Means to Developers

The age of Mac OS X means that the original Mac OS lifetime is coming to an end, although not in the near future. Any useful applications available today will eventually have to be *ported* to (that is, converted to a form useable within)

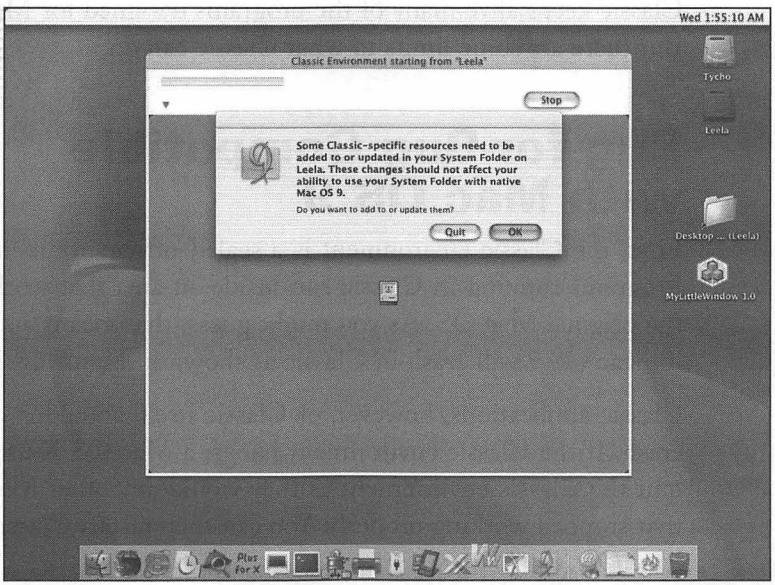



**Figure 18.3**
When you double-click a program written for the original Mac OS, the Classic environment turns on. Mac OS X requires you to add a few new parts to make your Mac OS 9 installation also work as a virtual machine to start applications in Classic.

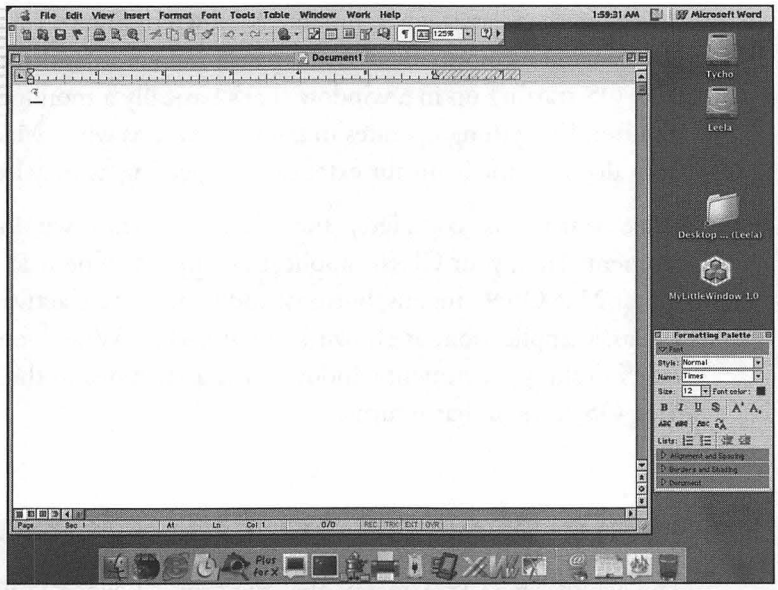



**Figure 18.4**
Mac OS Classic programs use their old windows and menus, but portions of the Aqua interface do show through.

Mac OS X. Being able to run programs in Classic slows the clock a little so developers have time to make the switch, but that clock is ticking away.

Classic keeps alive many of the programs designed for Mac OS 9 and earlier. But there are some limits to what Classic can do.

## Bug-for-Bug Compatible with Mac OS 9

First, the Classic environment is a sealed-off environment from Mac OS X. Programs running in Classic run inside an area that works and acts just like the original Mac OS. As you might guess, that means applications that crash in Mac OS 9 will crash in Classic as shown in Figure 18.5.

Classic applications, however, or Classic itself, shouldn't cause Mac OS X to crash. If the Classic environment hangs, a Mac OS X user can simply force-quit the Classic environment as they would any other Mac OS X application that stopped working properly. You can then restart Classic without restarting Mac OS X.

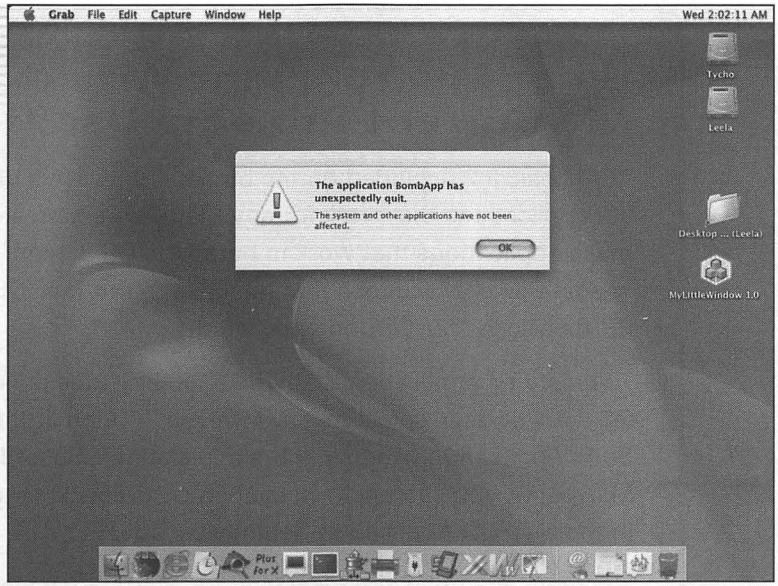



**Figure 18.5**
Whoops! Fortunately, crashes that happen in Classic don't affect Mac OS X.

## Classic Applications Use Mac OS 9 Only

Applications designed for Mac OS 9 aren't written to take advantage of the new features in Mac OS X. All Classic applications are isolated from Mac OS X. That means OS 9 applications offer no pre-emptive multitasking, no advanced memory management, or any other Mac OS X buzzword you can think of.

To use Mac OS X features with an application, the programmer must make adjustments to the program in a process Apple calls *Carbonizing* (more about this in the next chapter).

## No Direct Hardware Access

In the early days of computer programming, many hardware features weren't as easily accessible to a developer because the operating system didn't provide a practical way to use the new components. Fortunately, some savvy programmers learned to use their programs to directly talk to the hardware parts they needed for their work.

## What Environment Would My Current Projects Use?

REALbasic 3.2 (included on the book's CD-ROM) is designed to create applications that work for any Mac OS version since 7.6.1. To make applications that work in both Mac OS 9.1 and Mac OS X, you need a Power Macintosh that qualifies to run Mac OS X, and REALbasic 3.2 for Carbon applications.

To create an application that works on any Macintosh running Mac OS 7.6.1 or later, select the "Macintosh" option at the top of the Build Application window. If the application is intended for Power Macs only, you can uncheck the 68K code checkbox under "Include" in the Mac OS Application Settings section.

To create an application designed for Mac OS 9.x and Mac OS X, select the "Mac OS X/Carbon" build selection. When you select this, all options to change what code to include are greyed out—only PowerPC code is used in a Carbon application.

If you choose to port your application to Windows, see Chapter 23 first for helpful information. When you're ready to build the application, check the Windows build option at the top of the Build Application window.

As more advanced operating systems such as the Mac OS and Windows appeared and evolved, direct access to computer hardware caused many problems, particularly if one application improperly used a hardware component while another properly written program became confused because it could use the hardware at the same time. Such conflicts usually cause the original Mac OS to hang or crash.

The whole idea of an operating system is to manage what applications can use on the computer without crashes or poor system performance. With that in mind, Mac OS X is designed to ignore any attempts by Classic applications to directly communicate with hardware such as CD-ROM burners, video cards,

and the like. Many software companies have announced forthcoming releases of applications that can use hardware in the manner Apple dictates in Mac OS X.

# What Classic Can and Can't Do

Using Classic has a few advantages and several pitfalls, at least as of the Public Beta. Hopefully, many of these issues will have workarounds in the official release of Mac OS X by the time you read this.

Personally, we're hoping that the companies and people who created the applications that have problems in Classic will just Carbonize their applications to run natively in Mac OS X and save all of us the trouble of dealing with Classic's shortcomings.

- **Classic cannot mount Windows NT file server volumes.** This is a bit bizarre because AppleTalk still survives in Classic. To confirm this on a Macintosh running Mac OS X with Classic active, open the Chooser. You'll see AppleTalk NT servers in the Chooser, and can log into any of them provided that your Mac OS 9 System Folder has a Microsoft networking information file, a *user authentication module*, installed. When you try to access the mounted volume from Mac OS X (remember, the Classic side doesn't have a desktop where volume icons appear) you realize that there aren't any volumes on the Mac OS X desktop. Whoops.
- **Classic can't save to PDF format without help.** If your Classic/Mac OS 9 installation has the full Adobe Acrobat application installed, you can create Portable Document Format documents (OK, it's redundant, but I didn't name the thing). PDF documents are based on Adobe's high-quality PostScript technology with two major advantages. First, PDFs retain the high-quality graphics and font styles from the original document. Second, a PDF document made on a PC can be read on a Macintosh without any adjustments. Mac OS X includes *Quartz*, a graphics engine based on PDF technology. Quartz runs the Aqua interface, providing the rich details and colors on that interface. Quartz also allows any Mac OS X application to print graphics or text as PDF documents. Because the original Mac OS running in the Classic environment wasn't designed with PDF technology built-in, Classic applications can't create PDFs without Acrobat.

- **Classic uses only Mac OS 9's printing and page-preview support.** Classic applications use the same LaserWriter and inkjet printer support as found in Mac OS 9, but Classic doesn't get any further benefits from Mac OS X's presence. In Mac OS X, print previews are actually PDF documents.
- **Classic is the only place where non–Power Macintosh applications will operate.** If you have to use a really, really old Macintosh application designed expressly for Macintosh II family computers, Classic is your only avenue. Mac OS X can't run original Mac OS applications, and certainly not 68K code designed originally for Quadras and other older Macintoshes, without Classic. The only condition to running old applications in Classic is that the application was initially compatible with Mac OS 9.
- **TCP-IP has limits in Classic.** Certain features of TCP/IP, such as access to AppleShare IP servers, don't work if you select servers from the Chooser in Classic. You need to use Mac OS X's Connect To Server command in the Finder's Go menu to handle these connections.

## Review

Classic is a great Apple solution for running otherwise incompatible Mac OS software in Mac OS X. Classic has frustrated some users who have become used to a particular way of performing tasks in the original Mac OS, but this problem should subside as more applications are carbonized to work only with Mac OS X's interface and abilities. You should write and build any REALbasic applications with Mac OS X in mind.


# 19

# The Carbon Environment

## In This Chapter

- It's tool time
- A few small repairs
- Carbon: good for your programming diet
- Carbonized applications can use Aqua
- How REALbasic uses Carbon

When Apple announced the Rhapsody project, programmers' hopes for a powerful new operating system were finally realized. Then, the other shoe dropped. Rhapsody required developers to spend days or weeks rewriting their existing Mac OS programs, and they had to do it using Objective-C, a variant of the C programming language that few programmers knew. Most programmers preferred C++, a different variant.

After shoving the Rhapsody project to the side as a consideration for the new Mac OS replacement, Apple realized then that they had to figure out another way for existing developers to port their old programs to the new Mac OS X without causing a lot of pain. Further, Apple had to allow the ported programs to take advantage of most of Mac OS X's features.

## It's Tool Time

While developing the first version of the Mac OS in 1983, Apple realized that it would be terrible for programmers to write the code needed to draw windows, buttons, and icons to the desktop. Even if the programmers could do that much work, it's likely that each element of the Mac OS interface would look and work differently because no single programmer would write things the same way as another. (Some of us would say that the applications would be as cluttered and ugly as the Windows interface, but that's another argument!)

For this reason, Apple provided a set of *application programming interfaces*, or *APIs*, which were, essentially, canned Mac OS programming for every graphical element that a programmer could use in a Macintosh application. Instead of having to write a section of code that said, "draw a window using this procedure," a developer would only need to write in the name of the subroutine used for drawing a window.

Apple gave a nickname to this collection of programming subroutines: the *Toolbox*.

## A Few Small Repairs

The Mac OS original Toolbox became cluttered with obsolete and redundant subroutines over the 15 years of the original Mac OS. The Toolbox was still

### It's All in a Name

Why the name Carbon? Steve Jobs, during a Macworld Expo keynote speech, said with a smile that everything good evolves from it.

The joke, in case it flew right over you, is that humans and other life forms are based on the element carbon, and that life evolved from those carbon building blocks.

versatile, but was bogged down with code from discontinued versions of the now-ancient original Mac OS architecture.

Apple had to allow original Mac OS applications to work with their new Mac OS X operating system. To do this, Apple realized that, both Mac OS X and Mac OS 9 could use a refined version of the Toolbox APIs. That way, moving a Mac OS 9 application to Mac OS X would take much less time to do.

Apple essentially threw out many old and obsolete Toolbox APIs, then added and modified others. About 70 percent of the original Toolbox routines were left. Most of the things that were chucked were Toolbox routines for 68000-type processors used in the old Macintosh II family and Quadra systems. The result was a new set of APIs for use in Mac OS X programming. Apple retired the name *Toolbox* for its API set and christened the new set *Carbon*.

## Carbon: Good for Your Programming Diet

Carbon APIs allow you to design programs you create to work in original Mac OS or Mac OS X. When you convert a Mac OS 9 application to also work in Mac OS X using the Carbon APIs, you've *Carbonized* it. A Carbonized application running in Mac OS 9.1 and later will take the windows, menus, and button appearances of the original Mac OS.

To use Carbonized applications, Mac OS 9.1 and later must have a system extension, *CarbonLib*, installed in the System Folder. CarbonLib adds the new routines necessary for Carbonized applications to operate properly.

If history has told us anything, it's this: Software development forces additional hardware and software upgrades. In other words, as a Mac OS developer, it's a good idea for you not to bet on the idea that Carbon applications for Mac OS X will be supported in anything earlier than Mac OS 8.6. Write for Mac OS 9 and Mac OS X.

Keep an eye out on the Apple Developer Connection Web site at http://www.apple.com/developer. Once you sign up for a free membership, you'll have access to many software development kits (also known as *SDKs*) and development tools. The downloadable Carbon SDK contains the latest developmental version. Most importantly, Apple will indicate what original Mac OS version is supported in later versions of CarbonLib.

The SDKs deal primarily with C/C++ programming techniques, but the information can be useful with some issues in REALbasic programming.

# Carbonized Applications Can Use Aqua

Applications designed with Carbon APIs take on the interface styles of Mac OS 9 when running in that environment as shown in Figure 19.1. You don't get any Mac OS X benefits from a Carbonized application running in Mac OS 9, except perhaps the likelihood that it may have better stability than an application created with the original Toolbox. Know why?

The original Toolbox APIs included many obsolete routines not designed with Power Macintosh systems or Mac OS 8 and 9, such as routines for Macintosh Quadras, Macintosh II systems, and other computers without PowerPC processors. Those routines, if activated by mistake on PowerPC systems, can cause odd behavior or crashes. So a Carbonized application has all-PowerPC routines that aren't as likely to destabilize Mac OS 9.

When you double-click a Carbonized application in Mac OS X, it operates within the Mac OS X environment, not in the Classic environment, as shown in Figure 19.2. A Carbonized application inherits the Aqua look and feel, from the windows to the warm pulsating buttons. It's not necessary for Mac OS 9 to be installed to run any Carbonized application with Mac OS X.

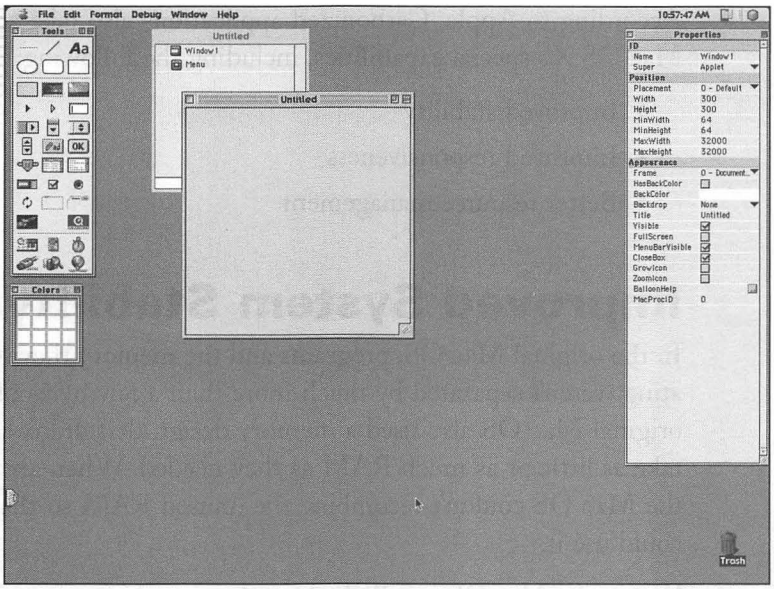



**Figure 19.1**
REALbasic 3.2 running in Mac OS 9.

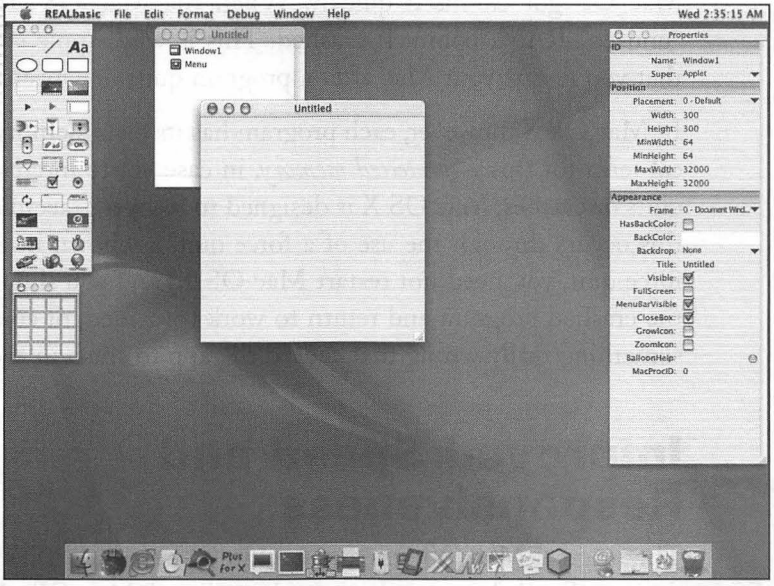



**Figure 19.2**
REALbasic 3.0 running in Mac OS X. Windows and buttons use Aqua's interface features.

According to Apple, Carbonized applications also take advantage of most of Mac OS X's special capabilities, including the following:

- Improved stability
- Improved responsiveness
- Better resource management

## Improved System Stability

In the original Mac OS, programs and the memory they reside in when operating weren't separated by much more than a few bytes of empty RAM. The original Mac OS also used a memory design that didn't allow applications to take as little or as much RAM as they needed. When an application crashed, the Mac OS couldn't recombine the unused RAM so that other applications could use it.

Worse, the Mac OS usually had no idea that the memory is unused. So, if you continued to do work in the Mac OS after an application had crashed, the Mac OS got confused quickly. When the Mac OS attempted to use the "unused" RAM, boom. It crashed. This is why Apple sternly recommended that you restart your Mac after a program quits unexpectedly.

In Mac OS X, however, each program has its own memory space. The technical term for this is *protected memory*, in case you're interested. If the program quits or crashes, Mac OS X is designed to recover gracefully without crashing; this occurs through the use of a force quit, as shown in Figure 19.3. After a force quit, you need not restart Mac OS X, and you should be able to restart the crashed program and return to work (provided, of course, that there's not something really wrong that caused the app to crash in the first place).

## Improved Speed and Responsiveness

The original Mac OS was a marvel in its heyday, but it's never been a speed demon. That's because the way the original Mac OS allowed programs to share time with the processor—called *cooperative multitasking*—wasn't very efficient.

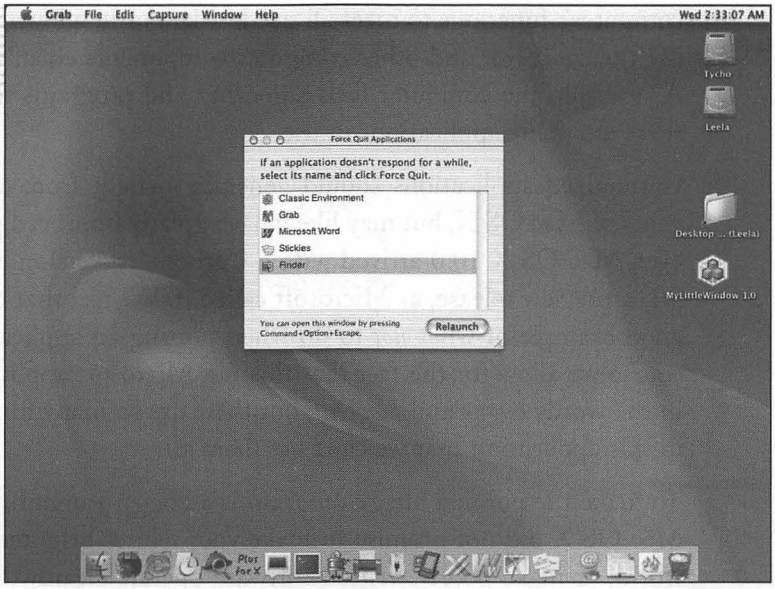



**Figure 19.3**
Applications that aren't running properly or at all can be force quit in Mac OS X.

Imagine you're a Mac OS program, trying to tell the PowerPC processor to handle a task. Now imagine you're one of several Mac OS programs, all asking for time with the processor as well. In cooperative multitasking, the processor would ask other applications to give up their time so that the processor could tend to the needs of one program. From there, the processor would look to the next program in line and handle its needs while temporarily ignoring the other applications' needs. If that particular Mac OS app hung or crashed during its time with the processor, the processor couldn't switch to other apps, and your Mac was frozen.

Here's one way you can watch the pitfalls of cooperative multitasking. Start up your Macintosh, then run a QuickTime movie and click on any menu on the menu bar. Notice that the movie no longer plays while the menu is open? That's cooperative multitasking. The Mac OS cannot play the movie and handle your menu selections at the same time. Once you select a menu command and release the mouse button, however, the movie resumes play.

Try the same trick in Mac OS X. Notice how the movie keeps playing while you select any menu command? That's *preemptive multitasking*. It controls the

computer's functions to meet all needs asked of it, allowing all applications to use the processor and other computer components equally, for the most part. As a result, the computer feels smoother and programs feel agile even when there are many applications in use.

Carbonized applications should generally benefit from at least the stability gains of Mac OS X, but may likely have greatly improved performance. However, Mac OS X 10.0 arrived with a Carbonized version of Internet Explorer 5.1, a preview release, as Microsoft called it. We feel that this applications is a good example of what *not* to do when porting an application to Carbon. Of course we allow for the fact that this *is* a Microsoft application (you can read in the words *buggy* and *bloated*), but there are so many fundamental problems in this application that we can't list them all.

To make the point: a buggy program is a buggy program, Carbonized or not. Mac OS X did not promise to make your whites brighter, bring peace to the world, or make poorly written software operate cleanly. It's up to developers like yourself to ensure that your Carbon apps work properly.

Apple initially claimed early on during Mac OS X development that Cocoa applications would benefit best in Mac OS X. Today, Apple appears to have improved overall operating system performance sufficiently to claim that Carbon and Cocoa apps work equally well. Writing an application using the Cocoa frameworks is a labor of love, but usually just a labor. We'll touch on Cocoa applications a bit in Chapter 20, "The Cocoa Environment."

## Better Resource Management

To allocate more RAM to a Mac application today, you have to open the application's Get Info window and change the number of bytes of RAM assigned by the program's maker. But many programmers tend to overestimate the actual amount of memory a program needs. Or, when a program misbehaves, some people simply allocate more memory in hopes of bringing things back in order. As a result, programs running under the original Mac OS become a bit greedy. Your computer hasn't sufficient resources to do what it's asked to do.

Carbonized applications in Mac OS X are assigned as little or as much memory and other resources as it requires. Precious RAM and other energies aren't wasted.

Carbon APIs are, by Apple's design, a convenient and fast way to develop new Mac OS X applications. By using Carbon, you'll take advantage of the current methods and tools to develop original Mac OS programs so your learning curve is less steep.

# How REALbasic Uses Carbon

There are actually several versions of REALbasic 3.2 included on the CD-ROM in this book. All of them function identically in terms of developing. However, the 68K version of REALbasic cannot create applications for Carbon or PowerPC. Do note that any Power Mac can create Carbon apps with REALbasic 3.2, however, only Power Macintosh systems capable of running Mac OS X can test the results in that operating system.

REALbasic 3.2 takes full advantage of the Carbon APIs as you develop. When it comes to developing for Mac OS 9.1 and X, you can't find an easier to use programming tool.

When you're ready to build your application, all that's necessary to Carbonize it is to choose the Mac OS X/Carbon option in the Build Application window, shown in Figure 19.4.

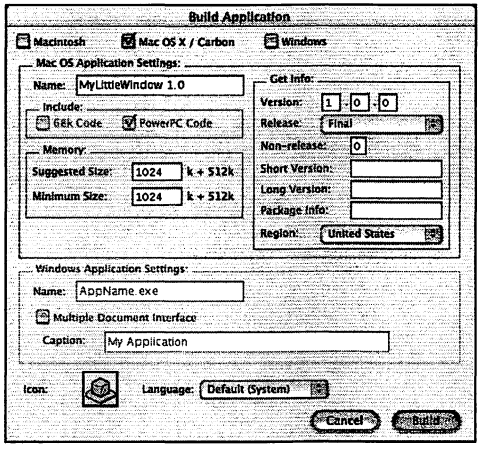



**Figure 19.4**
REALbasic 3.2's Build Application window has three options for building your Macintosh application. Most of the time, Mac OS X/Carbon will work for your needs.

Using REALbasic over other programming environments offers you a huge advantage. For the most part, Carbonizing an application doesn't require you to change your processes for building a project. Figure 19.5 shows a basic Carbonized application running under Mac OS X. However, you should pay attention to the interface differences between Mac OS 9 and Mac OS X. Apple Developer Connection offers comprehensive interface guidelines for Aqua. These guidelines may undergo some final tweaking by Apple as Mac OS X becomes a finished product, so be sure to drop in and download the latest information. See also Chapter 17, "Enter the World of Aqua."

# Review

Thanks to the abilities found in REALbasic 3.2, creating Carbonized applications can't be much easier. However, sloppy coding will not magically work when Carbonized. Remember that building a Carbon application allows you to use it on Mac OS 9.x as well as Mac OS X. Carbon apps should run better in Mac OS X due to its better multitasking and memory protection features.

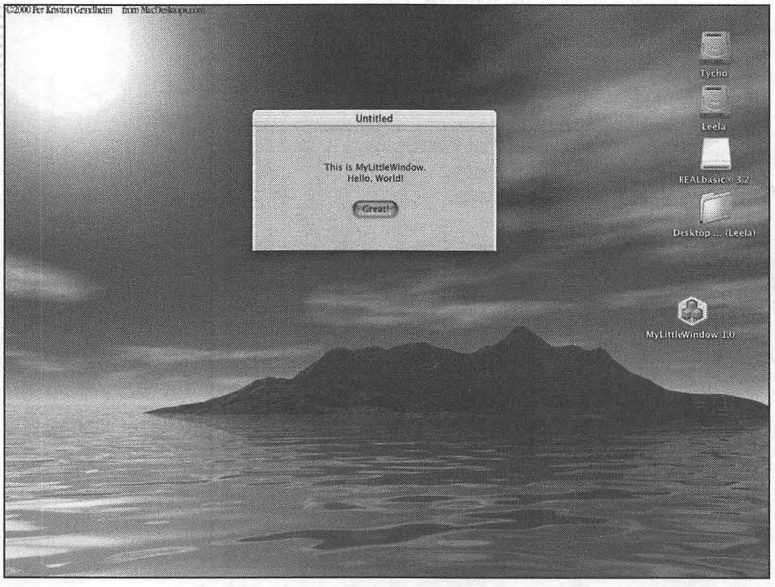
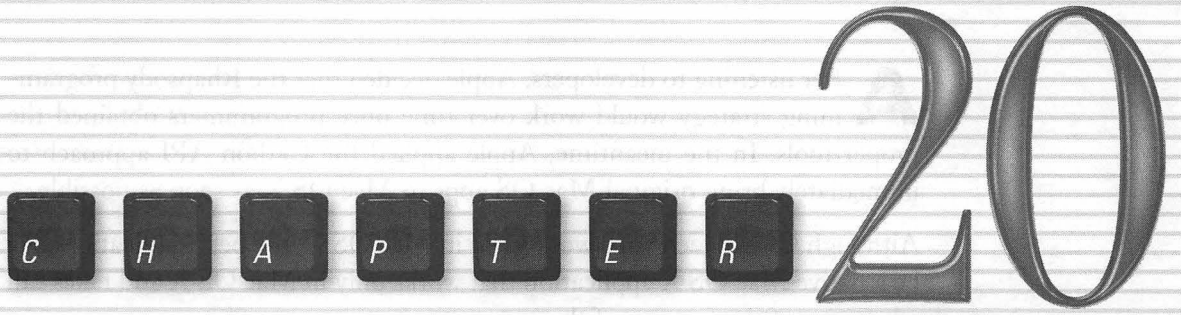



**Figure 19.5**
MyLittleWindow, a Carbonized Mac OS X application.




# The Cocoa Environment

## In This Chapter

- Have some hot Cocoa
- Java: it's not just for Web pages anymore
- What you need to begin Cocoa development
- For more information

After listening to developers, Apple decided that the Rhapsody programming strategy would work over time once programmers obtained the proper tools. In the meantime, Apple applied the Carbon API approach to immediately bring original Mac OS apps to Mac OS X as soon as possible.

Although Carbonized apps work great in Mac OS X, the best application may be a native Mac OS X application that works only in Mac OS X and takes full advantage of the features of the new operating system.

In Rhapsody, the native APIs were known as the "Yellow Box." Today, in Mac OS X, you'll know them as *Cocoa*.

# Have Some Hot Cocoa

Cocoa is an advanced set of programming frameworks for developing native Mac OS X applications. *Frameworks* are application-programming interfaces that replace Carbon and previous Mac OS programming interfaces. With this, you would design a new application that can run natively in Mac OS X.

Cocoa (implemented by Apple in Objective-C) projects can make use of existing C and C++ libraries, and can even use Java routines. Yes, *that* Java—the same programming language that adds so many features and enhancements to Web sites, among other things.

### Why Cocoa?

Cocoa's name likely originated from the blend of programming languages and APIs available. In particular, the Java programming language spawned more than enough coffee and hot beverage metaphors within the computer industry. Apple wanted to take its stab at it.

# Java: It's Not Just for Web Pages Anymore

Java was introduced in 1994 by Sun Microsystems. Back in 1991, Sun began a project to develop a way to make software for many kinds of household electronics such as TVs and VCRs. Imagine a smart toaster that makes the bread and bakes it, too! The whole idea behind the various implementations of the Oak Project (as it was known then) was to make an efficient way to program many kinds of consumer-electronic products, including computers.

Java works by separating the idea that a computer's programming must be natively created, or *compiled*, for the computer on which it would run. For example, say that you were writing a new word-processing application. You want the application to work on both Macintosh and Microsoft Windows operating systems.

If you're not using REALbasic to write your Macintosh application (that is, you're using CodeWarrior or some other IDE), your first (or, to be more accurate, second or third) challenge after writing an application is to compile a version of the code to work on Intel-compatible systems. After the bugs were removed and additional Windows conventions added, hopefully, you would have two versions of the same application.

Because we're so pleasantly used to REALbasic, we oversimplified our example. Unfortunately, very few development tools or procedures are around to help you with cross-platform development. REALbasic's abilities are among these few exceptions. To make a cross-platform application, your IDE has to be able to compile the application code into instructions that the computer's processor can understand. That's not an easy task when you're dealing with such diverse chips as Intel's Pentium and the PowerPC processors used in IBM mainframes and Apple Macintoshes.

So Sun changed the paradigm of traditional computer programming. As we've said, normally the process of compiling the code for an application is processor dependent because the IDE's compiler you use is designed to create application code only for one kind of processor. So, Sun thought, why not simulate the computer hardware using another specialized application for Java on each

computer platform? This way, in theory, the unique parts of Java's operation that require an understanding of the unique abilities of a particular computer is built in each application that simulates a Java computer. This kind of application is called a *virtual machine.*

Once you slam out some Java code, it needs to be converted to a file that can run on those virtual machines. Sun's answer to that dilemma (because you still can't run Intel instructions through a PowerPC or other chip without help) was to create a compiled form of Java that runs through the virtual machines on any platform. When you compile Java code from your computer, you create a *bytecode* file. Run the bytecode through your Java virtual machine, and out comes the completed Java application, running happily and doing its job. Well, that was the idea, anyway.

When Sun introduced Java, most of the newest programs available came in the form of *applets*, tiny Java applications that were called up within a Web page using a Web browser that could run Java code through a virtual machine built in the browser. For the most part, most of us have only had this kind of exposure to Java. Most of us probably haven't seen *HotJava*, an experimental Web browser developed by Sun that could run Java applets. Later on, however, a tiny Internet startup company named Netscape modified its Web browser to run Java applets, and the Java craze took off.

For a time, hundreds of Web sites happily announced their use of Java applets. However, most sites presented rather simple and mostly useless applets that served more as novelties and programming examples than anything else. I mean, you could only appreciate Java clocks and tic-tac-toe games so much.

Another problem introduced itself that re-ignited the computer-software/platform-compatibility issues. When Java first arrived on the scene for personal computers, it seemed more equal for some computers than others. Although a Java software development kit for Windows was available almost immediately, it was some time before versions for the Mac OS appeared. Worse, there appeared to be glaring incompatibility issues that hobbled or crippled Java functionality under the Mac OS, version 7.5 at that time.

The biggest issue involved Java's maturity. Although showing a lot of promise, Java was a rich programming language that required a lot of horsepower from both operating system and computer hardware. Because of this, running Java outside of a Web browser wasn't consistent between most com-

puter platforms. Windows Java tended to run as much as twice as fast as the Mac OS implementation.

It wasn't until version 1.1 or so of Java that greater features and stability appeared for the Mac OS. Available for Mac OS 7.6.1, Macintosh Runtime for Java, or MRJ for short, included a stronger Java virtual machine that other applications on your Macintosh could use for running Java applications. In fact, you probably use MRJ and don't know it; Internet Explorer for Macintosh, the Web browser from Microsoft, allows you to use the MRJ for running Java applets in that browser.

Today, with the hype subsided, Java's still mostly on Web pages, used for those many clocks, forms, and some online applications. But Apple has led the forefront in adapting Java as a major player in Mac OS X by providing tools you can use to develop large, complex Mac OS X applications with Java. No other operating system to date has so fully integrated Java as part of its development.

Sun, as we told you earlier, created Java from elements of C and C++. Their Web site contains a comprehensive assortment of information on learning and using Java that you can try out. Visit the official Java home page at http://java.sun.com for information on Java development, as well as comprehensive tutorials for many computer platforms, including the one probably closest to your mind, the Mac OS.

## About Objective-C

Objective-C first came to light as part of NeXT's OpenStep operating system. Objective-C code is a cross between C and Smalltalk, an object-oriented programming language of the 1980s. Objective-C is fully compatible with conventional C programming environments, so if you or a friend happened to have some source code to a C-based application you were doodling with, you could port it to Objective-C and Mac OS X with little trouble.

If you're familiar with C or C++, you can adapt your code to Objective-C so it can operate as a native Mac OS X application. Objective-C, as the name implies, is a subset of C that adds object-oriented programming compatibility to the programming language. While it's beyond the scope of this book to teach you how to program with Objective-C, there are some resources available from the Internet and elsewhere that you can study for more information.

Swarm.org (http://www.swarm.org/resources-objc.html) is a great starting place for more Objective-C information. An Objective-C information page compiled by Steve Dekorte (http://www.dekorte.com/Objective-C/) has more links on the language.

# What You Need to Begin Cocoa Development

A caution or two: Eventually, Apple would like every developer to program using the Cocoa environment, and better sooner than later. Keep in mind that programming in Objective-C, C++, or Java isn't the easiest thing in the world.

We hope you learn all you need to know about object-oriented programming concepts in our book as a stepping stone to Cocoa development. This chapter is intended to introduce you to Cocoa programming and the available tools, but isn't a comprehensive guide in any way. (You *did* want to keep your first steps in Mac OS programming *easy*, right?) We won't be going too deep, but will give you a taste of what you can explore later as you venture deeper into Mac OS programming.

The best thing about Cocoa development is that Apple provides you with all the tools you need to create the programming elements as well as the graphic interface elements for the time it takes you to download them from their Web site.

Or, better yet, why download the tools when you can get them free when you buy your copy of Mac OS X? That's right—one of the three CD-ROMs included in your Mac OS X purchase contains all the developer software you require to build Carbon or Cocoa applications. Although practically all other members of the UNIX family include their tools as integral applications when the operating system is installed, Apple thought it best to keep the tools separate. Nevertheless, Apple upholds the UNIX family tradition of providing the needed software to build more applications.

Once you install the tools, your next step should be to subscribe free to Apple Developer Connection, Apple's official Web home for Mac OS development, shown in Figure 20.1. The free subscription allows you access to any downloadable tool or documentation. Apple Developer Central is located on the Web at http://developer.apple.com.


**Figure 20.1** The Apple Developer Connection can provide you with a plethora of Mac OS X information and software.

## Project Builder and Interface Builder

Apple provides two powerful programming tools you can use immediately to create a Cocoa application. The first is, appropriately named, Project Builder, and is shown in Figure 20.2.

When you open Project Builder and create a new project, the New Project window asks you to select a project type. As you can see in Figure 20.3, Project Builder can design Objective-C and Java applications for any variety of Mac OS. It can even create Carbon applications, which, as you know now, allow you to support older Mac OS 9 systems.

Where Project Builder helps you create an application, it doesn't provide the graphical elements for the Aqua interface. Logically, that's where Interface Builder comes in. Its name, in typical Apple style, describes what it does precisely.

Interface Builder serves as, you guessed it, the graphic interface construction kit for a program. This program, shown in Figure 20.4, might seem a bit familiar to you. In what we felt was a sincere bit of imitation as flattery, Interface Builder also works much like REALbasic in terms of accessing interface

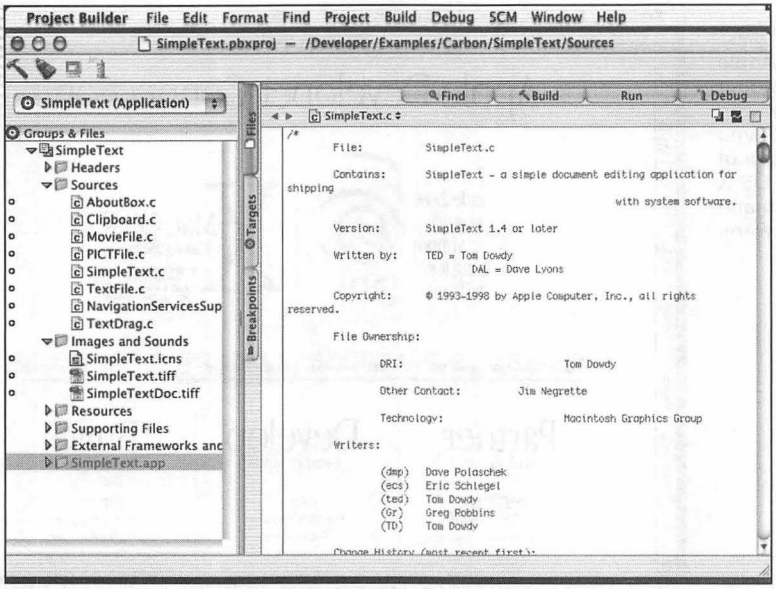



**Figure 20.2**
Project Builder is designed so you can slam out great Macintosh code for Cocoa as well as Carbon applications.

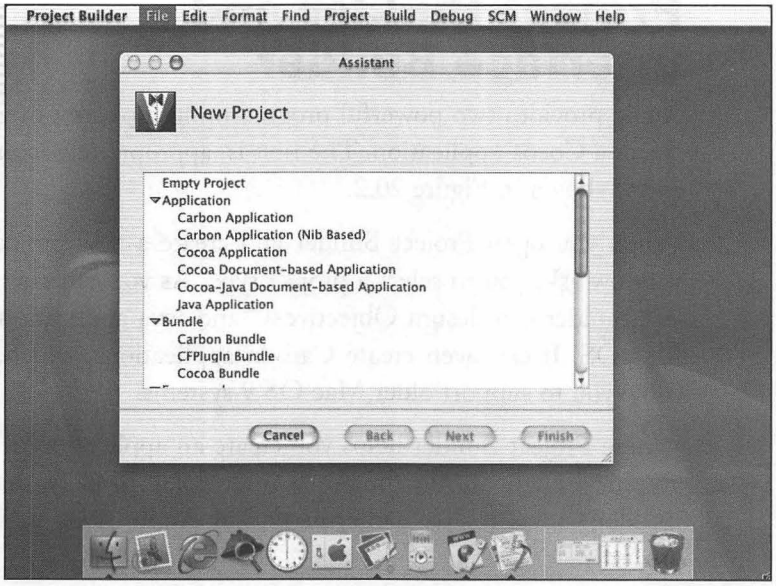



**Figure 20.3**
Starting a new Project Builder project.

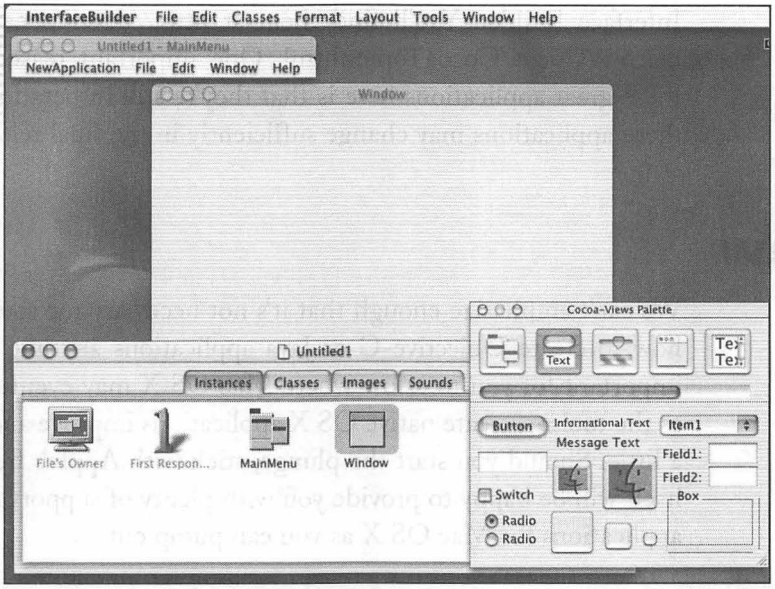



**Figure 20.4**
Interface Builder adds the necessary Aqua finishes for a native Mac OS X project.

and control elements from floating palettes. From there, however, the similarities start to disappear. For the most part, this is an application specifically designed to integrate your code with the Aqua GUI. Interface Builder not only provides the visual controls needed for any interface, but also aids you with making your project conform to Apple's interface standards as you develop it.

# For More Information

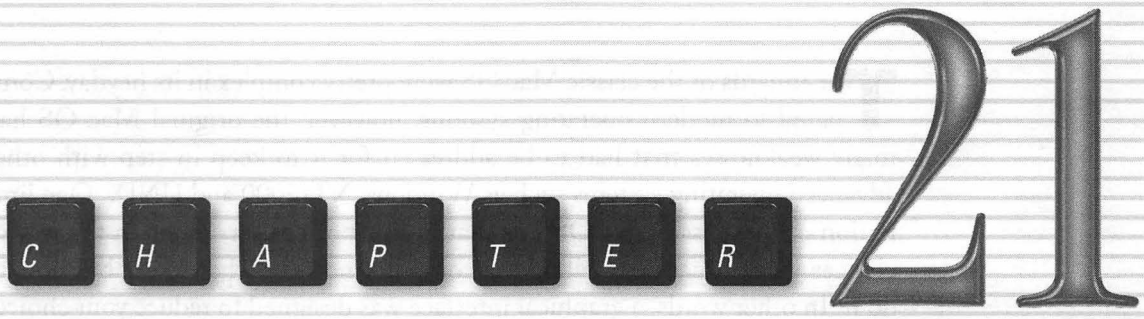
Java and other programming languages that Project Builder and Interface Builder support are a bit above the curve of what we're trying to teach you in this book. There are plenty of books on C, C++, and Java that will be of help to you to develop in Cocoa as you gain more experience in object-oriented programming with REALbasic.

We've found several great resources on the Internet for Java and Objective-C information. The best ones are from Apple itself, in the form of tutorials for creating an Objective-C and a Java application using Project Builder and

Interface Builder. You'll find them at http://developer.apple.com/techpubs/macosx/Cocoa/CocoaTopics.html. One important reason we can't get into these great applications here is that they're still in beta form. The features of these applications may change sufficiently in the final release.

# Review

We can't emphasize enough that it's not necessary for you to absolutely know how to create Objective-C or Java applications anytime soon. However, it's important for you to know where Mac OS X may eventually go, particularly as the tools to create native OS X applications improve and hopefully simplify a little. Should you start the plunge, stick with Apple's free tools since Apple itself will be happy to provide you with plenty of support to generate as many applications for Mac OS X as you can pump out.




# UNIX: A Shell Surrounding a Tasty Kernel

## In This Chapter

- Forward to the past: the command line
- She sells C shells by the C shore
- The Terminal application
- A summary of useful Terminal commands

The innards of the classic Mac OS were pretty complex in its heyday. Compared to modern operating systems, however, the original Mac OS had many weaknesses that had to be addressed for it to keep in step with other advanced operating systems such as Windows NT/2000 and UNIX. One limitation of the classic Mac OS was that it didn't offer users much in the way of choices when it came to telling the computer not just what to do, but how to do it. In other words, a graphical interface was designed to reduce your choices and so reduce your operating system's relative power or efficiency. Despite the advantages of graphical interfaces, the most powerful way to direct a computer in its fundamental tasks of file management, systems control, and user management, is a *command-line interface.*

# Forward to the Past: The Command Line

The BSD/Mach kernel fusion that forms the underpinnings of Mac OS X brings something brand new to the graphical world of the Mac OS: a native command-line interface, or *CLI.* As many of you know, the original Mac OS was among the first personal-computer operating systems without a CLI. To move documents from one disk to another, you moved icons on the screen that represented files to and from windows that represented the contents of a disk or folder.

The greatest advantage of a graphic interface is that commands sent to the computer are greatly simplified. Suppose you have stored a text file, MYWORK.TXT, in an MS-DOS subdirectory named C:\MYDIR\MYFILES, and that you need to copy MYWORK.TXT to a subdirectory on a floppy drive named A:\BACKUP\MYFILES. To copy the file from the C drive to the A drive, you would need to type in the following command at the MS-DOS prompt:

```
COPY C:\MYDIR\MYFILES\MYWORK.TXT A:\BACKUP\MYFILES
```

As some of you who've used MS-DOS can remember, this command will fail if any of the characters within it are mistyped. Worse, there are at least two other ways to complete this command. For instance, you could use the command:

```
D \MYDIR\MYFILES
COPY MYWORK.TXT A:\BACKUP\MYFILES
```

or perhaps this command:

```
A:
CD \BACKUP\MYFILES
COPY C:\MYDIR\MYFILES\MYWORK.TXT
```

Life's stimulating enough without having to practically learn a new language just to operate a computer, don't you think? It's no wonder that the Mac OS and, later, Microsoft Windows, became the preferred way for most computer users to interact with their applications and data. Still, a command-line interface is inherently more powerful because of the complex ways you can adjust your instructions.

Graphical user interfaces like the one used by Mac OS were oft referred to by CLI enthusiasts by this disparaging acronym: WIMP (windows, icons, menus, and pointers). Although the simplicity of GUIs had its advantages, a command-line interface allowed faster access to the power of an operating system. As you can guess, however, using a command line has a much steeper learning curve. There are hundreds of commands available in some UNIX versions, and Mac OS X, underneath that beautiful Aqua exterior, is no different.

# She Sells C Shells by the C Shore

One advantage of using UNIX is that there are many command interpreters available. A *command interpreter* is a program that translates the commands you type into instructions upon which the operating system can act. In conversation, UNIX-savvy individuals simply call them *shells*.

To most Macintosh users, shells are foreign entities. If you've used the old Apple security/interface called At Ease, however, you've already used a type of shell. Shells provide flexibility in the operating system by allowing the user to modify the complexity of input and output from the computer. Graphical shells like At Ease were designed to *reduce* the options available to you to run the computer or change its settings. In short, Macintosh "shells" simplified the Mac OS even further, giving extra security from, for instance, crazed, file-trashing youngsters. Or, you use Mac shells to simplify things beyond what you can do in the Finder.

There are many UNIX shells out there, although a few aren't free and can't be distributed. About four of them are pretty common and can be found on most UNIX versions. The second most-popular command shell is the terminal-based C shell, or *tsch*. (We haven't found a pronunciation that sounds quite right, although we once heard a Linux guy pronounce it "*tee-cee-shell.*") The tsch shell is a variation of the C shell, the most popular shell. The tsch shell happens to be the default shell when you install your copy of Mac OS X.

In the case of UNIX, a shell interprets the barely English–like instructions from a command-line prompt into binary instructions that the computer kernel reads and executes.

Many typical UNIX and Linux users mix and match the convenience of a graphical interface with a command-line interface using a graphical-interface generator and interpreter known as *X Windows*. X Windows interacts with the kernel and whatever graphical interface you choose to use. That probably sounds weird to you if you're a Mac user—unless you've used the shareware program Kaleidoscope, which patches Mac OS 9's interface with snazzy window and button changes. The one important difference is that most Linux and UNIX graphical shells don't make your life easier. These GUIs simply tend to be graphical representations of the gazillion possible commands you would encounter from a typical command-line shell. KDE and GNOME are popular examples of graphical shells used with X Windows.

If you're thinking that Aqua is a shell of sorts, you're right. Aqua is a graphical interface like KDE and GNOME. The similarity continues in that the X Windows and Quartz display engines provide the necessary resources for KDE or Aqua, respectively, to display, send, and receive instructions from the kernel. The main difference—and this is an important one that makes a Mac the easiest computer to use—is that most X Windows interfaces maintain a similar complexity as the command-line shells they supplement. Yes, that's right, *supplement*, not *replace*. Mac OS X's interface is designed to fully control the UNIX underpinnings without having to use a terminal window or other non-Mac OS component.

Like all other forms of UNIX, however, you could strip the interface (Aqua) and the window manager (Quartz) away and still have a useful (but dry as a martini) operating system. Figure 21.1 shows the onion that is Mac OS X from an interface standpoint. Even without Quartz and Aqua, you can still happily operate Mac OS X's innards from a command line.

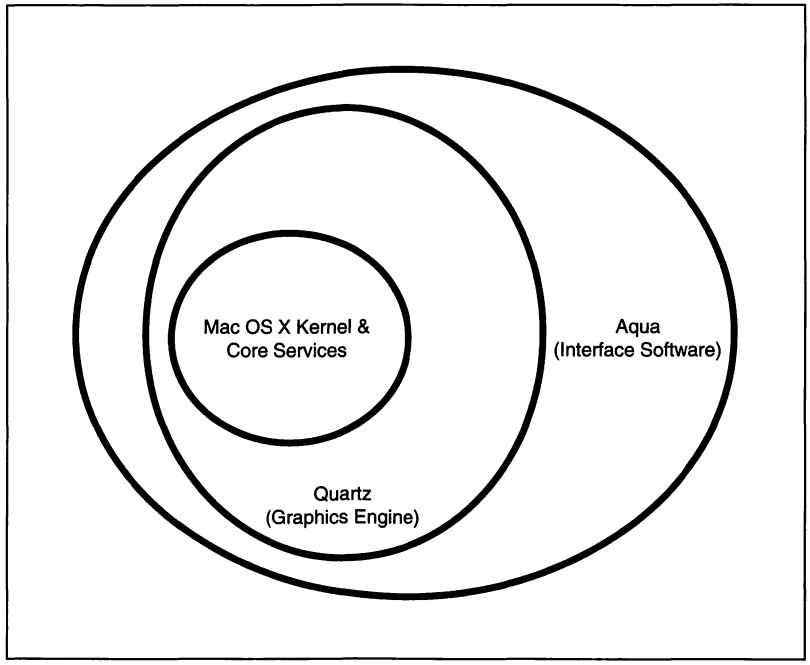



**Figure 21.1**
Aqua is a graphical shell over the Quartz graphics engine, which communicates with the Mac OS X heart.

With so many shells comes the number of commands available in each one. Nobody can remember all those commands, but fortunately, some shells have similar commands to each other (such as tsch, which is derived from csh). Because the C shell is by far the most popular shell in the UNIX world, most of the commands you may run into later may work fine in *Terminal*, the command line application included in Mac OS X.

Keep in mind that shells *aren't* the actual operating system. A shell is merely one way of sending instructions to components that the operating system controls. That's why the Aqua interface has as much basic control over the computer as a typical shell, but can't compare to the complexity and detail of instructions you can give the computer from a command line. Command-line interfaces are known more for complicating the process of using your computer—but in a good way.

Most importantly, keep in mind that UNIX itself understands only one thing: the binary information that a command interpreter (be it terminal shell or graphic interface shell like X Windows or Aqua) translates from mouse clicks or phrases you type. Because a typical UNIX kernel is extremely flexible,

the more complex the command interpreter used, the more dynamic your computer can be.

Before we get into how powerful UNIX's command-line interfaces can be, perhaps you should get acquainted with Terminal, the Mac OS X application that allows you access to the UNIX command line.

# The Terminal Application

You'll find the Terminal application in the Applications folder, inside the Utilities folder. Double-click the Terminal application, and you'll be greeted with the window shown in Figure 21.2.

If you've used a computer terminal, MS-DOS, or older online services like CompuServe before 1996 or so, Terminal will seem a bit familiar. It's nothing more than a text interface in which you enter UNIX commands at the command prompt.

But why use Terminal? Aside from the sheer geek chic of using a command line in the Mac OS, you'll need Terminal if you want to recompile an application that's compatible with Mac OS X's BSD UNIX heart for use on your computer.

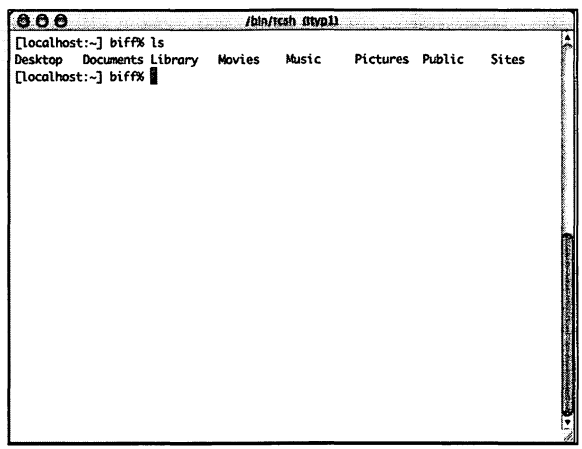



**Figure 21.2**
It's new! It's bland! It's Terminal! There's not much to it, but it's a powerful way to use Mac OS X's hidden powerful features.

## Prompts, Lists, and Permissions

One thing that's cool about shells is how much direct power and information is available from just a few keystrokes. Now, we know the Macintosh argument about this: "But the mouse makes using a computer so easy." Easy, yes. Powerful? It depends on what you're doing. Are you drawing something complex and colorful with layers and gradients and all that other graphic mumbo-jumbo? Okay, then, a mouse is better. Are you trying to copy many different files from many different locations to another spot on your computer? Or are you attempting to install software you've found that requires you to launch an application that doesn't have a graphical interface? Ah, that's the beauty of working with the command line.

The only hard part of using a command line, especially with shells, is the sheer number of commands and their variations that are possible. Let's look at a few tsch commands and let you explore a new world. If you happen to have Mac OS X installed on your computer, just locate Terminal in Applications/Utilities and launch it. For those of you who don't have Mac OS X installed yet, just follow along with the screenshots.

When you start up Terminal, all that greets you is a single window with a prompt, as shown in Figure 21.3. (It was nice of Apple to make the format as black text on a white background. That makes it easier to see and less harsh on the eyes if you have to stare at it for a while.)

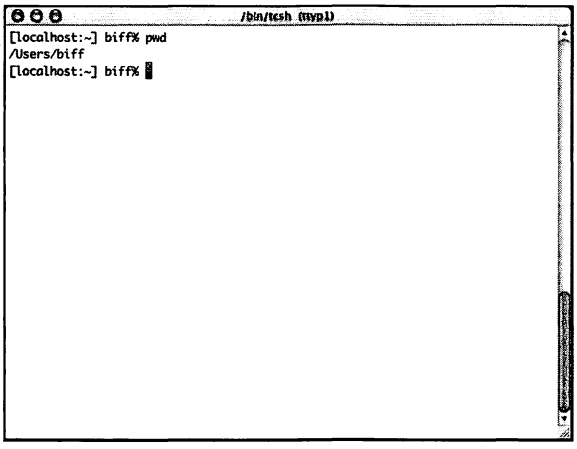



**Figure 21.3**
This prompt indicates the name you gave the computer when you installed Mac OS X. The pwd command can tell you exactly where you are.

The hardest part about using a command line is the fact that the commands available to you usually depend on where you are. Well, that is, not where *you* are (computers care little about your geography), but the directory to which your shell is pointing. The prompt itself tells you where you are when you open Terminal. By default, Terminal places you in your own user directory. You'll see your directory name represented as your login short name. The default name of a Mac OS X computer (from a UNIX point of view in terms of networking, not AppleTalk) is *localhost*. Unless you changed the name during Setup Assistant (like we did), that's what you'll see in your prompt.

To find out what working directory you're using at any time, type in `pwd`. (This command stands for "present working directory." Don't confuse this with other UNIX commands that allow you to change your login password.)

Let's try a simple command. Type `ls` at the command prompt, and press Return. The Terminal will list all directories and files available from a folder.

Like the MS-DOS command named *dir*, the ls command lists the contents of a directory. Typing in ls by itself shows the contents of your current directory. By default, Terminal points to your home folder for the login you're using.

When you or someone who administers your Mac OS X system creates a new user, a folder is created in the Users folder on the Mac OS X hard drive. Each user folder contains the following folders: Desktop, Documents, Library, Movies, Music, Public, and Sites. Many of these folders work like their old Mac OS 9 counterparts and are essentially places to store your stuff.

In the case of Mac OS X programming, you can take advantage of the fact that all users have this basic folder structure. You can use this information to build your applications to search, add, or remove items from these folders—with the

### Uh, What's a Directory?

A *directory* is a location where files and applications are stored. You may know them by a simpler name: *folder*. When we mention directory or subdirectory, just think "folder" or "subfolder." And yes, the term "directory" came from the UNIX world.

proper system administrative access, of course. Mac OS X's UNIX security requires Administrator access for changing or accessing the most critical system functions. In some cases, your application or the user who installs your application may require superuser access, or *root* access to the computer. Generally, you shouldn't need to use root, and it's a good practice to keep from changing files in Mac OS X that you aren't meant to change.

One thing you should note is that Mac OS X's ls command does not show certain files and directories; this is for the sake of security and simplification. To show every item in a directory, you could type in `ll`, the command for a long list. This shows files in a table (see Figure 21.4).

Notice that ll gives you far more information about the directory's contents than the simple ls command, including hidden system files such as the ones with periods at the beginning of their names. Some of the more useful information involves the *permissions* for each folder or directory.

## What? No Root Account?

In all other UNIX-family operating systems, including Linux, the "root" administrator account is the one you, as the computer owner and administrator, would be asked to create on setup of your computer. The root is the superuser. A person with root privileges can make changes to all files of the computer. If you're not savvy in UNIX administration, this could be a bad thing.

Apple, knowing how users of the classic Mac OS love to tweak things in their System Folder, disabled the root account in Mac OS X. This is a good thing. Change the wrong thing in OS X, and your computer turns into a paperweight.

If you really need root access—for example, to install a piece of software—you can open the NetInfo Manager application in the Applications/Utilities folder. From the Domain menu, choose Security, and then Enable Root User. You'll need to be an administrator of the computer to make this change.

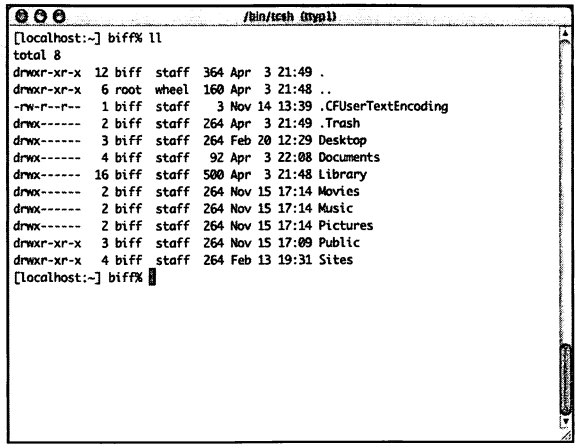



**Figure 21.4**
Use the ll command to list all directories and files available from a folder. This view is more like the MS-DOS dir command.

Permissions in Mac OS X work much like the privileges you could place on a folder in Mac OS 9, except you can place restrictions on individual files in Mac OS X, whereas sharing permissions were only for folders and disks, not files, in Mac OS 9.

First, let's get some terminology down. When you create a file in Mac OS X, it assigns your account name to the file as the *owner* of the file. As owner of the file, you can read the file, change (or write to) the file, or execute the file (if the file is an application).

Files can also be available to groups. A *group* consists of a list, maintained by Mac OS X, of users who have identical permissions for particular items. As with owners, groups also have read, write, and execute permissions for files. Most applications and files are assigned to one or more groups so that members of those groups can share the applications and files, and work on them collectively.

Lastly, there are files and directories, where everyone shares access. In the UNIX universe, files and directories that everyone can use are given *world* permissions.

So, what do owner, group, and world permissions have to do with the funny code to the left of each file and directory name? Let's break it down.

The first character simply indicates whether an item is a file or a directory. If there's a `d`, it's a directory. If there's a hyphen (-), it's a file.

Next are three groups of letters, each with a series of characters: r (read), w (write), and/or x (execute). These permissions work similarly to the ones you're used to in Mac OS 9 and earlier. The first group of letters indicates *owner permissions*. For example, if you're logged in with your account, as the owner of your Public folder, you have full access to look in, change the contents of, or open any item in that folder.

The second group of letters indicates group permissions. Unlike in Mac OS 9, Apple hasn't yet provided a defined way to create groups for users, but Mac OS X comes with a few built-in groups. For now, note that "staff" is a general group where users with Admin and User privileges reside. A root administrator has permissions in the "wheel" group.

World permissions, the third set of three letters, are generally set to read–only access or no access at all for most files, since a typical UNIX installation (including Mac OS X) tries to keep inquisitive minds out of places and files that don't belong to them. This permission is equivalent to using the "Everyone" privilege in Mac OS 9 File Sharing. Items that you place in the Public folder for file sharing on a network might have certain World permissions for users to read and launch the files in the folder.

To remember which permissions are which, try assigning a number to each letter in each permission. Break down the permission letters into its three groups (ignore the first character that shows a file or directory designation). Next, assign a 4 to each *r*, a 2 to each *w* and a 1 to each *x*. Now look again at Figure 21.4. It shows the permissions normally given by Mac OS X to a Public folder for any user. Assign the necessary numbers for the owner, group, and world permission letters, then add the three numbers in each group together. In this case, the owner has rwx permissions, or 7; the group permissions are rw, or 5; and the world permissions are rw, or 5. For this reason, this type of permission is referred to as "755" access—the owner of the directory can do anything he or she wants, while groups and all others can see that the directory exists and can open anything in it, but can't change or delete items from it.

Any files you happen to come by with 700 access (the owner has rwx permissions, but the group and world have no permissions at all) are for a superuser, or root access only. Generally, that includes all Mac OS X system software components. Don't mess with these.

# A Few Basics in Terminal

Let's move about from directory to directory. Right now, because you just opened Terminal, you should be pointed in your home directory. In the Terminal window, type the following and press Return.

```
cd ..
```

You'll see a screen like the one shown in Figure 21.5.

Congratulations. You just moved yourself to the Users directory. This command, cd, is short for "change directory." The two periods tell the shell to go backward one directory level. Type ls, and you'll see all the various home folders for any other users of your Macintosh.

Now, time to move back to your home directory. Type *cd homefolder* and press Return, where *homefolder* represents your home folder's short name (hint: it's the last part of your Terminal prompt's name).

Ta-da. You're back in your home folder.

Notice something about the folder's name: it's typed in all capitals, even though the folder name is actually lowercase. Those of you familiar with other versions of UNIX may think, "Hey! They didn't type the folder name in the proper case!" That's right, we didn't. In practically all other UNIX versions and clones, *Applications*, *applications*, and *APPLICATIONS* are three different file or directory names. Not so in Mac OS X. We entered *applications* as the direc-

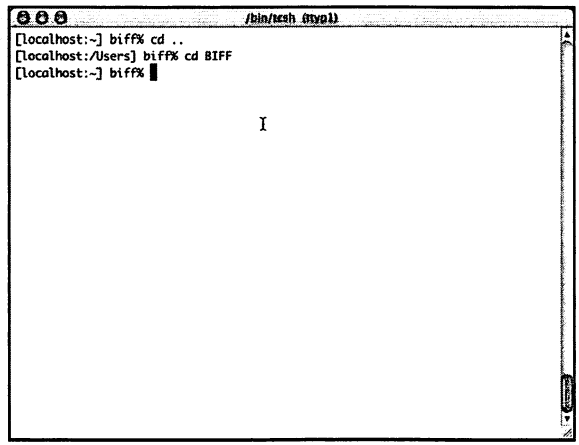
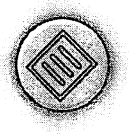



**Figure 21.5** Changing directories doesn't show an obvious result until you view the changed information within the prompt.

tory path, and Terminal dutifully moved us to Applications. Apple decided to change this UNIX convention to match the classic Mac OS convention.



**CAUTION**

Although moving about in Mac OS X should be a breeze with this change, you should remain aware of this when porting UNIX applications to Mac OS X or when creating Mac OS X application code that you can port to other UNIX operating systems. Most other versions of UNIX must receive its directory and file names precisely, or your applications or commands will get treated poorly.

If you wanted to jump to any directory, you could type in the full directory path. Alternatively, if, for example, you wanted to go the Applications folder from your home folder, you can simply type the following and press Return:

```
cd /Applications
```

Probably the most useful command for UNIX newbies is man. No, not "man" as in "man and woman," but the command that's short for "manual." (In case you haven't noticed, the UNIX world not only loves making obscure abbreviations, but ones that seem to have very little basis in the English language or is part of some geek humor.) All versions of UNIX have built-in help available through a terminal window. To access it, you simply type `man` and the command about which you want more information, as in the following:

```
man top
```

After typing this command, you'll see the text of the manual page for the top command, as shown in Figure 21.6. Most of the pages are based on BSD UNIX commands and usually show "BSD Experimental" and the date the man page was last updated. But, as with any UNIX version, some man pages you'll see are clearly revised for your benefit.

**NOTE**

Most of the manual pages are too long to view in a single window. To scroll a line at a time, use the up or down arrow keys. To scroll a page at a time, press the spacebar.

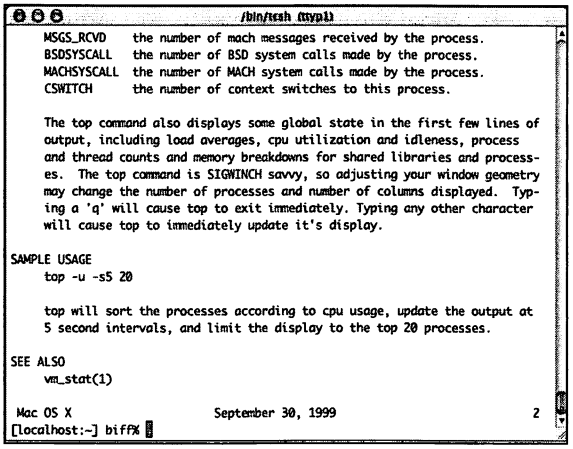



**Figure 21.6**
This last page from the man top command shows at the bottom of the page that it was updated just for you, the Mac OS X user.

# A Summary of Useful Terminal Commands

So as not to bore you with more text screens than you (or the publisher) would care to see, we've compiled a list, shown in Table 21.1, of some of the more useful commands in Terminal. You can use these commands to speed up your work or to access information not easily seen through the Finder. Just remember that Mac OS X, like all good members of the UNIX family, evolves constantly, so commands will appear and change over time. Be sure to use the man command for more information on how these commands can be adjusted to serve you.

**NOTE**

I think the coolest command you can show a Windows user is *uptime*. Type this in your Terminal and let your PC-loving friends feast their eyes on how long your computer has been running without restarting. In UNIX, the time between OS restarts could be months or even years! Not even Windows 2000 can make this claim.

**TABLE 21.1 GOOD COMMANDS TO KNOW IN TERMINAL**

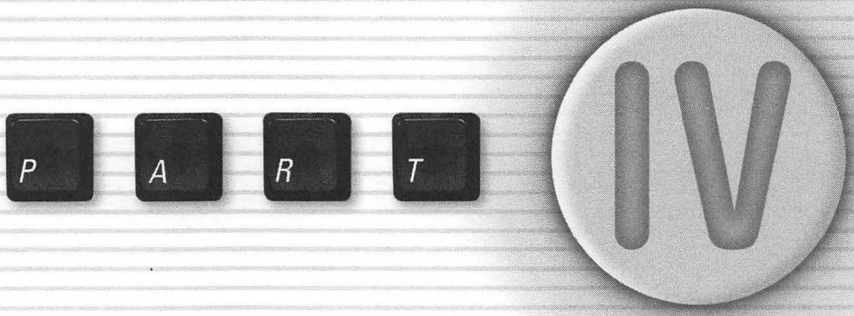
| Command | What It Does in Mac OS X |
|---|---|
| ls | Displays a list of the contents of the current directory. |
| ll | Displays a detailed list of a directory's contents. |
| man *command* | Displays a manual page for *command.* |
| whereis | A quickie search for files in a few nearby directories. |
| find *arguments* | Can extensively search by many, many different variables. For instance, the command find / -name apache -print would locate all instances of "Apache" in all directories. |
| cp *file file* | Copies files from one location to another. |
| mv *file file* | Moves files from one location to another. |
| top | Shows all processes running in Mac OS X. (You can see the same information with the Process Manager application. This is just cooler.) |
| open *filename* | Opens a file or application. |
| kill *process ID* | Force-quits an application. The *process ID* is the number next to each process listed by the top command or in the Process Manager application. |
| zip *filename* | Compresses ZIP archive files made on a Windows PC (Yay! a ZIP command that's finally built-in!). |
| unzip *filename* | Decompresses ZIP archive files made on a Windows PC. |
| mkdir *directory* | Creates a new directory. |
| rm *filename* | Removes a file. |
| rmdir *directory* | Removes a directory. |
| pwd | Shows your present working directory. |

# Review

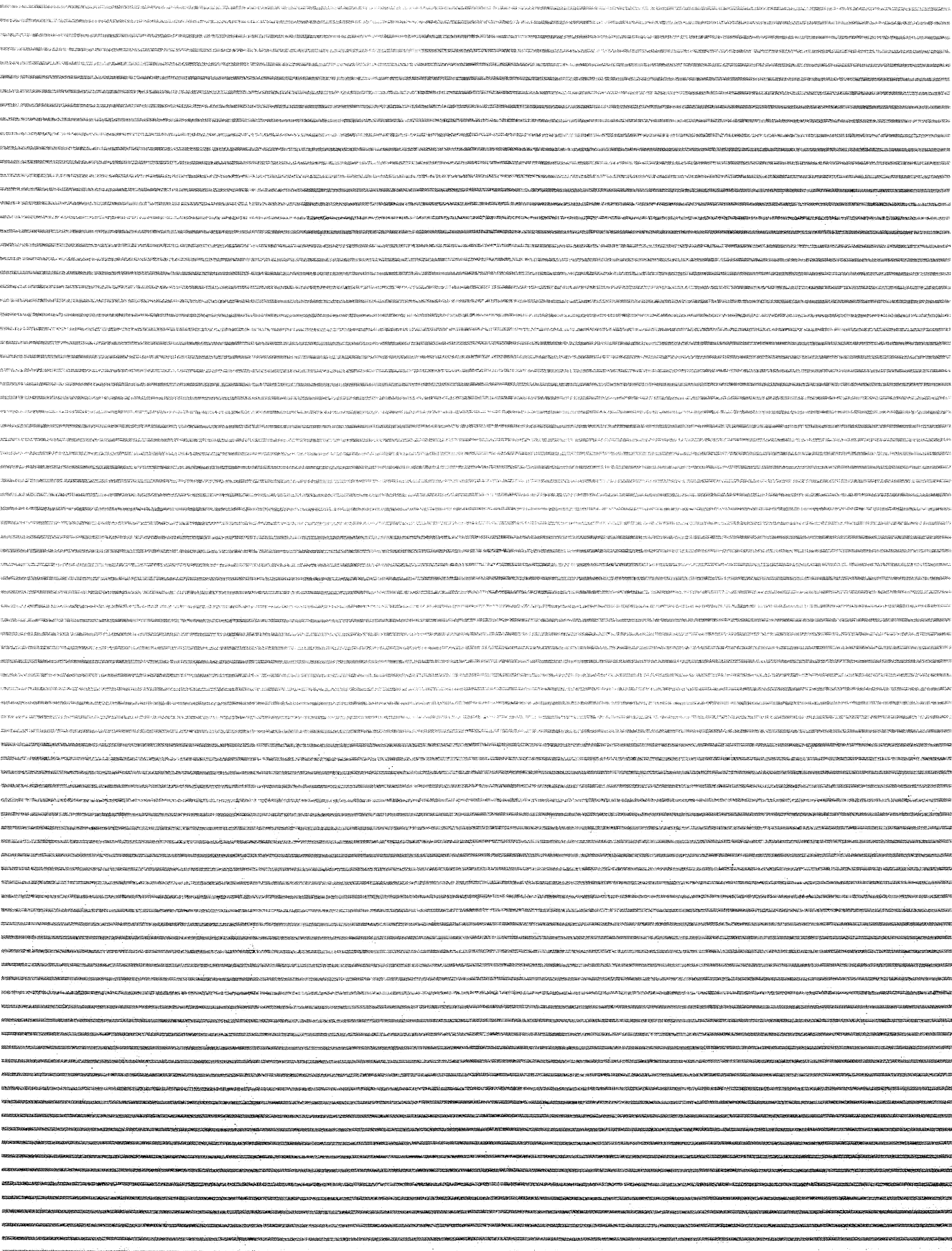
We've only scratched the surface of Mac OS X's powerful BSD interior. Programming while using Mac OS X as your operating system offer abilities not possible in Mac OS 9, such as shell scripting (like MS-DOS batch files) containing commands to execute in Mac OS X automatically when you launch an application. For now, you can relish at the ability to rename files all at once, or perform other tasks that seem to take forever under a graphical interface.

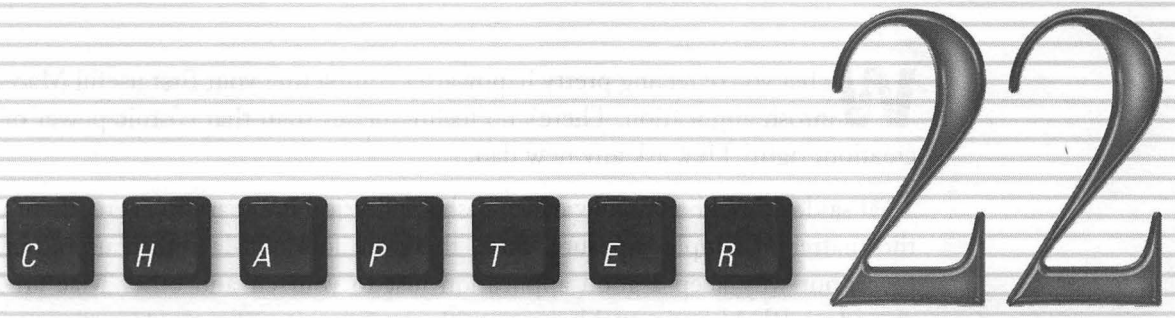
Beginning Mac®

PROGRAMMING




# Advanced Things to Do






# Porting Applications to Microsoft Windows

## In This Chapter

- Start with a Macintosh application
- Handling path names
- Watch out for conventions
- Compile only the code required for the ported application
- Porting Visual Basic code

We bet you're feeling pretty happy after completing your first useful Macintosh application. There's no feeling better than that cosmic power of creation, right? Hey, ask any new dad.

That said, although your new Macintosh application is a sparkling gem, there's the matter of simple exposure. Sure, there are lots of Macintosh users out there, but why keep a good application just to yourself and other Mac aficionados? REALbasic enables you, with the selection of a single option, to build a version of your application that runs in Microsoft Windows. That's a great feeling, and, depending on what you build, it could be a profitable feeling, too.

To get the most out of your creation, you can add some REALbasic code that optimizes the application to perform efficiently on a particular platform. In addition, you'll need to adjust your code for Mac OS features that aren't immediately available or allowed in a Microsoft Windows version of your completed applications.

# Start with a Macintosh Application

Your first REALbasic applications probably won't require extensive modifications just to make them work under Windows. The parts you have to worry about involve the different ways that Windows users operate their applications as compared to Macintosh users, as well as special Macintosh-only code you should isolate so that only specific code will be compiled for the particular platform you're using.

Our first words of advice involve *when* to build your Windows version. It's a good idea to build your application first for Macintosh only. After you're satisfied that the application works well under the Mac OS, you can revisit your project, save a version of it, and then add the modifications to make it work satisfactorily under Microsoft Windows.

# Handling Path Names

One thing to watch out for during modifications are path names. A *path* is an address of sorts for a particular folder or file on a hard drive or network volume. A *path* works much like a street address; the difference is that although

a street address begins with the item you want to find and the area surrounding it outward, a path begins with the surrounding area and works its way inward. If we were to write a street address in the way that paths are written, it would look like this:

MN\Metropolis\Broadway\5120

On a Macintosh, hard-disk names are whatever you want them to be, up to 32 characters. (That's Classic Mac OS, as in Mac OS 9. Your mileage may vary with Mac OS X, which plans support for many more characters using the Unicode standard.) In Windows, the various hard drives and volumes are not assigned names; they are assigned letters. Typically, a Windows system with a single floppy drive (how quaint), a single hard drive, and a CD-ROM drive will have volumes named *A*, *C*, and *D*, respectively (*B* is skipped, but would be used if the system had a second floppy-disk drive).

These letter assignments aren't set in stone, however. The CD-ROM drive letter might be different if the hard drive has multiple partitions. For example, Windows normally assigns drive letters after *A* and *B* to each hard drive or partition it detects. So if Windows finds three hard-drive partitions, it names them *C*, *D*, and *E*, and the CD-ROM drive is assigned *F*. The CD-ROM

## Who's Idea Was It to Use Drive Letters in Windows, Anyway?

The use of drive letters instead of volume names in Windows is a holdover from an old, old operating system named *CPM*. I'll give 100 quatloos to anyone who remembers this OS fossil. Better yet, I'll give 1,000 quatloos to anyone who gets my joke about "quatloos." (If you don't, turn in your computer-geek card—you don't watch enough TV, particularly science fiction!)

In CPM, drives were labeled as numbers: *0*, *1*, and so on. IBM borrowed this idea from CPM, but changed the numbers to letters to make it simple—well, as simple as it could be until we mount that 27th hard drive.

drive letter could still be *D* with the other hard-drive partitions named *E* and *F*. Fun, huh? No wonder Windows users are always losing files.

Path names in Mac OS and Windows are separated by various characters, which delineate a component (drive, directory, or file). The Mac OS, which rarely uses path names because of the simplicity of getting around, separates its items by colons, like this:

```
Macintosh HD:Applications:Classic Arcade Games:Tron
```

In Windows, a backslash is used to separate drives, folders (also called *directories*), and files. For example, a Microsoft Word application is typically available at

```
C:\Program Files\Microsoft Office\winword.exe
```

To confuse things even more, Windows uses a different method of specifying where an item is on a network server. Two backslashes are added before the network volume's name, like so:

```
\\Computer Name\Shared Directory Name\Directory Name\Another
Directory Name\File Name
```

Here's an example:

```
\\jeffserver\backup drive\embarrassing\babypictures.jpg
```

REALbasic's FolderItem class (in combination with other tools for cross-platform development, which we'll cover shortly) is the key to handling paths and the items within them.

**CAUTION**

People are people, and they love to change things. So, absolute paths are a bad thing unless you are installing items to a place that you can safely assume exists, like the System Folder, or \WINDOWS\SYSTEM.

# Watch out for Conventions

There are other internal differences and conventions you need to keep in mind if you choose to make your REALbasic application in Macintosh and Windows versions.

## Window, Window, Who's Got the Window?

One significant difference between Windows and Macintosh applications involves documents displayed by an application. Take Microsoft Word, for instance. When you open that application on a Macintosh, there's the menu bar at the top of the screen, the Word toolbars, and a document window. Now view the same application in Microsoft Windows. The application resides in a window itself, floating on the desktop. Document windows float inside the application's window. The menu bar rests at the top of the application window. Odd, to say the least.

This format is what REALbasic calls a *multiple document interface*. The application window is also known as a *frame*. In REALbasic 2.1.2, the completed Windows application's frame size can't be changed, so you'll need to ensure that the application works under larger or smaller screen resolutions, or that the user can see the content of your application's window. If you don't choose the multiple document interface option in the Build Application window, then your Windows application will be compiled with a *single document interface*. That is, your app has a single window, and any content within adjusts to the window. You should test out your builds of applications with both options to see which configuration is most appropriate.

## Take Note of OS-Specific Folder Items

Both the Mac OS and Microsoft Windows have common areas you can take advantage of when developing your application for cross-platform use.

Table 22.1 contains the REALbasic functions you can use for getting to key Mac OS system folders, Finder folder items, or their Microsoft Windows counterparts. Make a note of the ones that you can't use in Microsoft Windows (ControlPanelsFolder, ShutDownItemsFolder, StartupItemsFolder, and TrashFolder). In all cases, you should use TargetWin32 or TargetMacOS to determine what code should be used. We'll talk about the Target constants in the following section, "Compile Only The Code Required for the Application."

**TABLE 22.1 SPECIAL FUNCTIONS FOR ACCESSING SPECIAL LOCATIONS IN MAC OS AND MICROSOFT WINDOWS**

| Function | What It Does in Mac OS | What It Does in Windows |
|---|---|---|
| AppleMenuFolder | Accesses the Apple Menu Items folder | Accesses the Programs folder in the Start Menu folder |
| ControlPanelsFolder | Accesses the Control Panels folder | Returns Nil (not available for Windows) |
| DesktopFolder | Accesses items in the Desktop Folder (that is, items on the desktop) | Accesses the Desktop folder |
| ExtensionsFolder | Accesses the Extensions folder | Accesses the Windows\System folder |
| FontsFolder | Accesses the Fonts folder | Accesses the Windows\Fonts folder |
| PreferencesFolder | Accesses the Preferences folder | Accesses the Windows folder |
| ShutDownItemsFolder | Accesses the Shutdown Items | Returns Nil (not available for Windows)folder |
| StartupItemsFolder | Accesses the Startup Items | Returns Nil (not available for Windows) folder |
| SystemFolder | Accesses the Mac OS System folder | Accesses the Windows\System Folder |
| TemporaryFolder | Accesses the invisible Temporary Items folder in Mac OS. | Accesses the Windows\Temp folder |
| TrashFolder | Accesses the Trash folder | Returns Nil (not available for Windows) |

# Adding Hot Keys for Windows

Not to get into the whole Windows-versus-Macintosh thing, but there's a distinct philosophical difference in navigation between the operating systems. When the Mac OS was created, the mouse was chosen as the primary way to initiate commands and manipulate stuff, period. A few keyboard shortcuts (also known as *accelerators* in REALbasic or *hot keys* in other circles) were available to help users avoid unnecessary repetition, such as the Clipboard commands: Cut (Command+X), Copy (Command+C), Paste (Command+V), and sometimes Select All (Command+A).

In Microsoft Windows, however, things get a little more complicated. Windows has never fully shaken its roots in MS-DOS, where its command-line interface brought confusion—not to mention tears of frustration—to many users. The first commercially successful version, Windows 3.1, included keyboard shortcuts for most commands in the Program Manager (the desktop-manager application, which is similar to the Macintosh Finder), and for most applications as well. In fact, there are so many keyboard shortcuts in Windows that it's possible to operate it almost completely without a mouse. Figure 22.1 shows a simple Windows application, Notepad. Virtually every command in every menu has a keyboard shortcut.

Most of us who've sampled the Mac OS took a liking to the simplicity created by using the mouse for practically everything. In fact, the Mac OS's hardware and applications are designed so that having a mouse is virtually mandatory; you can't be very productive on a Macintosh without one. The mouse makes the Mac easier to use than any other operating system. But because the Mac OS is so mouse dependent, there are rarely more than two ways of performing any command from the Finder level.

In REALbasic, keyboard shortcuts are handled differently with Mac OS than with Windows. In Mac OS, you can add keyboard shortcuts through a menu-item property, but in Windows you can add keyboard shortcuts for both menus and menu-item properties. If you are developing an application, you can add the underlined shortcuts for the Windows version of that application

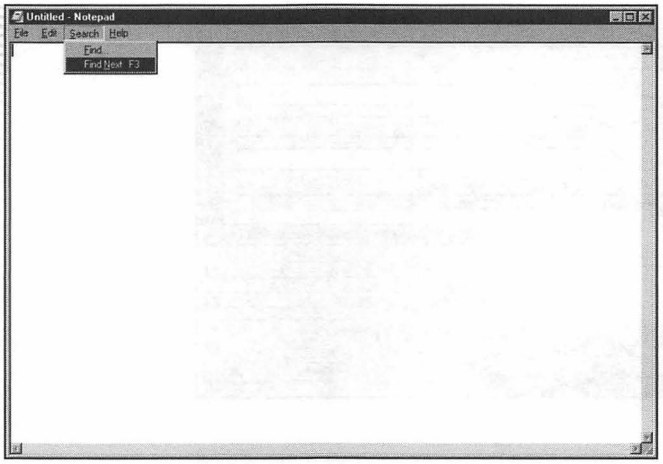



**Figure 22.1**
In Windows, keyboard shortcuts, or *accelerators* (like the ones in Notepad's menus), make for less mousing about.

through the use of *modules*, which can store a constant that adds the shortcut in a Windows version of your application. To do this, you create a new constant in a module in your application. Here's how:

1. Select the project window, and choose New Module from the File menu.
2. With the newly created module window open, choose New Constant from the Edit menu.
3 In the Value field of the New Constant window, type the name of the menu item for the Macintosh side of the application.
3. Click the Add button in the New Constant window to bring up a new dialog for entering a different constant value.
4. Choose Windows from the Platform selection.
5. In the Value field, type the menu item name. It should be identical to the corresponding Macintosh command, but with an added ampersand (&) character immediately before the letter in the command name that should have the shortcut underline (see Figure 22.2).

Another way to add certain constants in captions for buttons and tab controls is simply by adding an ampersand before the letter you want to set as the keyboard shortcut. To a Macintosh application, the ampersand is invisible. To a Windows application, the letter following the ampersand will have a keyboard shortcut, making it easier for Windows users to move about using the Tab key, which moves the button focus from one keyboard shortcut to another. To actually show an ampersand, place two ampersand characters.

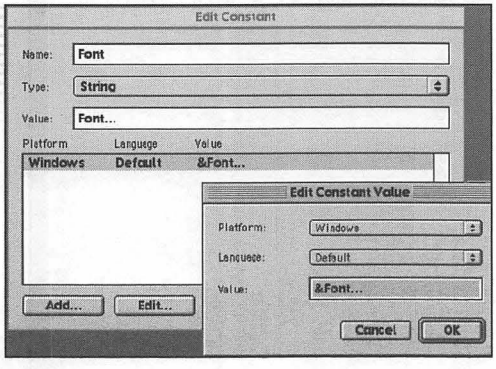



**Figure 22.2**
In your application, create a new module, then add a new constant for each keyboard shortcut.

# Compile Only the Code Required for the Ported Application

There may be Windows-specific or Mac-specific items you need to access, or code that you'd rather not include in your app if it's not needed, which could cause the whole app to misbehave or halt. For instance, a problem you might encounter is in testing the application during debugging as you're running the app in the runtime environment. Goodness knows you don't want any code you've added for Microsoft Windows–specific events to attempt to activate while running there. So, you'll need to isolate platform-specific code by identifying the code that's running in the application.

REALbasic code should make a distinction between Power Macintosh systems with PowerPC processors and older Macintosh systems using 68000-style processors, like Macintosh II and Quadra computers. There's always a chance that someone who gets your application will run it on older hardware. When you build your application, the Build Application window gives you the option of compiling for Mac 68K or Mac PPC. Although an app designed for 68K will run (slowly) on a Power Mac, the reverse isn't true—a PPC app won't run at all on most 68K systems.

Rather than use the rather generic TargetMacOS flag, try using the TargetPPC, TargetCarbon or Target68K constants instead to weed out unneeded or undesirable code when you compile.

There are six constants you can use to prevent compilation of unneeded code:

- TargetWin32
- TargetCarbon
- TargetMacOS
- TargetPPC
- Target68K
- DebugBuild

All of these target flags are Boolean constants that allow you to isolate code not needed for a specific build of your application for a specific platform. Of these, the one you might use the most is the DebugBuild constant. This flag is useful when preventing snippets of code from running while in the runtime

environment during debugging. If you're making an über-app that runs on everything, the DebugBuild flag can be a godsend by preventing the non-Mac OS code from killing a test run.

The TargetWin32 constant, to give a quick example, assigns a Boolean result to any variable you assign to it. Suppose you've created a variable named checkTarget. You could use the following to indicate code that should or shouldn't be included in the application built for Microsoft Windows:

```
checkTarget=TargetWin32
```

The variable would return true or false in response to the check. The other flags do the same thing but, logically, respond to the presence of the platform it detects.

The target flags are most useful with the #If/#Else/#EndIf statement. It works much like the conventional If statement, but allows you to isolate the code after the #If statement if the platform check was true. If the platform check from the Target constant is false, the code after #Else will be compiled into the built application.

Suppose your application creates documents that allow the user to save her work. (What a concept, huh?) On the Macintosh, you've instructed the application to open a Save dialog box and provide a default name for the document, *Untitled*. In Microsoft Windows, you need to ensure that a file extension is added to the default file name. You could use the #If statement in combination with the TargetMacOS (or TargetPPC or TargetCarbon, if the app you created is meant for PowerPC systems running Mac OS 9.1 or Mac OS X) like this:

```
#If TargetMacOS then
    Dim workfile As FolderItem
    Dim unsavedFile as TextOutputStream
    workfile=GetSaveFolderItem ("","Untitled")
    //...more file saving code...
#Else
    //Windows-specific instructions for the file save
#EndIf
```

You should use #If not only for Macintosh-specific instructions and unique features not available on Microsoft Windows (such as AppleEvents, AppleScript and Toolbox items), but also for Windows-specific calls to features

available only in Windows that could make your cross-platform application stronger when run on that platform.

# Porting Visual Basic Code

If you happen to have written an application in Visual Basic (or know someone who wants to give you free code), an integrated programming environment for Microsoft Windows, you can import those applications into REALbasic and create a Macintosh version. This isn't a completely perfect port—there are a few incompatibilities, but some can be repaired rather quickly. REALbasic automatically re-creates all the VB controls, event handlers, and methods from the imported code.

A quick and dirty way to import Visual Basic form files (files with the .frm file extension) is to drag and drop the .frm files in a Project window of a new REALbasic project. However, this process leaves much to be desired in terms of locating and correcting the syntactically different or incompatible code in a Visual Basic form.

A better way to start the import process is to use a utility included with REALbasic called *VBCleaner*. This utility examines and modifies the various classes, forms, and projects in your VB code and prepares them so that importing them into a REALbasic application is less time consuming. Processed VB class files are saved as VB forms and not classes, however, so you'll need to import those classes manually into REALbasic later.

### VBCleaner 2.0 Isn't Perfect

Avoid taking everything that VBCleaner does as gospel. Version 2.0 (reviewed here) was designed to correct code to work with REALbasic 1.0. Although REALbasic 2 and later may accept most commands from its previous incarnation, there are certain to have been a few changes in REALbasic syntax. Don't port code and assume that you need to correct only the VB issues—you may have REALbasic 1.*x*-to-2.*x* or -3.*x* issues, too.

The great news about cleaning up your code from a Visual Basic import is that REALbasic and Visual Basic share the old BASIC language as an ancestor. Also, both IDEs are modern, object-oriented programming environments. Best of all to you, there are plenty of sites to visit for VB code that you can port to create a Macintosh version of the application. (This is where you cheer the crowd of Microsoft Visual Basic developers for their work!) It didn't take us long to find some free VB code snippets on the Internet. One place was http://www.freevbcode.com/, shown in Figure 22.3, where we found samples for just about any application or task (for Microsoft Windows, of course).

**TIP**

All the sample code you find here will be compressed in the ZIP archive format, which means you'll need StuffIt Expander 5.5 or greater to properly extract the file's contents. Mac OS 9 and recent versions include a copy of StuffIt Expander 5.5, but you can download the latest version from Aladdin Systems at http://www.aladdinsys.com.

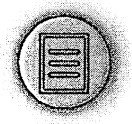
As a simple example, we downloaded the free source code for a Visual Basic application called *Graphic Browser*, made by a group called LazersDesign. The


**Figure 22.3**
FreeVBCode is one of many Web sites with loads of code samples. Of course, they're Greek to you until you get REALbasic to import them.

code files, once decompressed, are essentially text files readable by any text editor or word processor.

After installing VBCleaner from the REALbasic CD and downloading the Graphic Browser code, it was time to let the Little App That Could do its work:

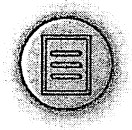
1. First, we opened the folder containing the VB code and identified the VB project file. Typically, VB project file names have the Windows file extension .VBP, which stands for (obviously enough) *Visual Basic Project*.



**NOTE**

In case your Windows understanding is less than stellar, a *file extension* is a three-letter identifier that tells Microsoft Windows what type of file you are using. The file extension (which is not the same as Mac OS extensions in any way) is separated by the file name with a period. Files with .EXE as their extension are Windows applications. A file with a .DOC extension is probably a Microsoft Word document. (We say "probably" because file extensions are a bit arbitrary and sometimes Windows mistakenly uses the wrong application to open a particular file. Keep this in mind when you create Windows versions of your applications.)

2. Next, we opened VBCleaner's File menu and clicked the Open command.
3. We selected the file Bmpbrowser.vbp.



**NOTE**

Although VBCleaner can process individual files, it's easiest to open the .VBP project file, which will automatically process other files in the same location as the project file.

4. VBCleaner's default settings exhaustingly detailed each step of the process, starting with a confirmation asking whether the file we selected was really the one we wanted to process (see Figure 22.4).

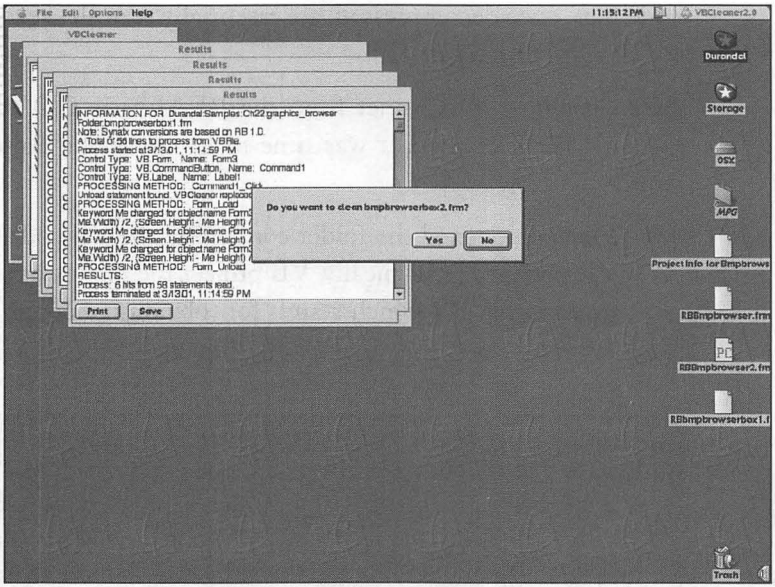



**Figure 22.4**
VBCleaner is very hard at work. The VB code we downloaded doesn't look particularly special. Note the Windows file extensions on each file.

5. After each file we processed (involving yet more confirmation dialog boxes), a summary window appeared that detailed what was found and modified in each file.

The upside to this blitzkrieg of windows is that you can save or print each summary for later review. For example, you might find a change that wasn't as useful or clear as it should be. Or there might have been a bug in the original code that VBCleaner changed from annoying to outright nasty.

VBCleaner doesn't alter the original code files; rather, it creates a new copy and deposits it without ceremony on your Desktop by default. In the version of VBCleaner we used (2.0), we couldn't change the default folder for some reason, so perhaps you can color-code your Desktop icons one color using the Finder's Label command (you do this by selecting the files, Ctrl+clicking one of them, and choosing a color from the Label menu context). When VBCleaner's finished, the files it processes will be on your Desktop in a different color, and you can move the files into a new folder somewhere on your computer.

With the files processed, the next logical step would be to go through the summary reports and begin the process of correcting code that VBCleaner can't modify—and that's exactly what you should do. But, being us, we wanted to see how well the VB code was translated. We fired up REALbasic, opened the

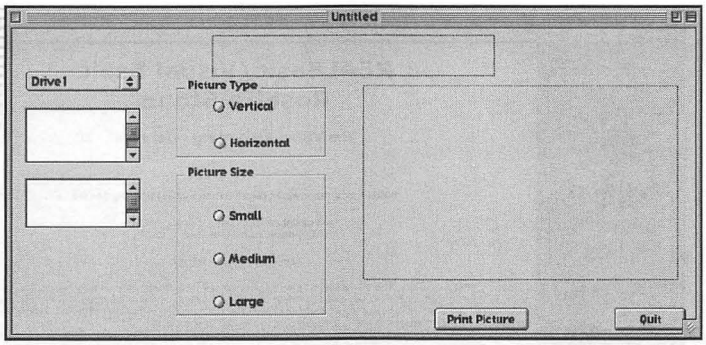



**Figure 22.5**
Although the Graphic Browser interface isn't much to look at and needs a lot of additional work, remember that this was originally designed for use only on Microsoft Windows.

File menu and selected the Import command, and selected the .VBP project file. Figure 22.5 shows the promising result. REALbasic dutifully displayed a section of the ported application's interface, but not much else. To add the remaining code, you'll need to cut and paste the snippets of code into the appropriate parts of your project. The great thing is that, thanks to VBCleaner, most of the code is already properly formatted in the correct syntax.

The next stages of porting are among the hardest. It may feel harder since it's difficult at this point for us to anticipate what code you will encounter in the wild that warrants a port. This is where your growing programming experience must pay off in identifying what you see. Try running the code to spot the errors first. You'll definitely see quite a few problems. To begin, watch for things such as

- Variable names that attempt to define the variable's type using the "%" and "$" characters, which stand for string and integer variables.
- Code relating to the opening of files. Windows file and folder name structure handles this a bit differently.
- Any kind of code that appears to be related to ActiveX and Visual Basic Scripting. These items aren't supported in the Mac OS or REALbasic since they utilize quite a few APIs from Microsoft that the company reserves for use only in Microsoft Windows applications.

A program such as this is probably fairly easy to re-construct from here on a Macintosh because of its graphical nature. The real challenges for you come when you try to port larger, complex applications. Fortunately for you, you can use many online resources to help—one you should jump to right away is the Visual Basic/REALbasic Rosetta Stone at http://kode-fu.com/rosetta/

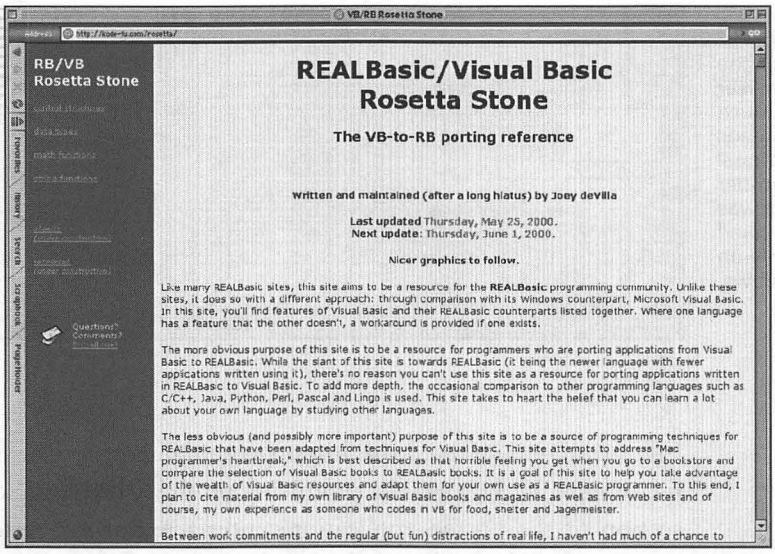



**Figure 22.6**
The VB/REALbasic Rosetta Stone Web site should be useful for correcting errors in translation that VBCleaner doesn't catch.

(see Figure 22.6). This site contains information about differences between certain commands in Visual Basic and its REALbasic counterpart, if one exists.

Should your taste in porting VB applications become unquenchable, we suggest that you drop in on your local bookstore and find a few books on VB to get a bit more familiar with it. REALbasic and Visual Basic make for good counterparts and a powerful alternative to the conventional C++ programmers.

# Review

REALbasic and Visual Basic have a few common roots that can make it a worthwhile experience to port useful Windows applications for Macintosh users. The double-whammy in porting with REALbasic is that applications can also be Carbonized for Mac OS X as well.

Porting applications is among the hardest programming tasks. This process will take time for you to learn the nuances of the source code, and rearrange the components in REALbasic to make a good fit. In addition to online Web resources, take advantage of REALbasic's mailing lists. Don't feel afraid to write other developers for more information on their code.


# 23

# A Word about Advanced Programming

## In This Chapter

- Let's C what develops
- Macintosh C++ development
- The Apple Developer Connection Web site

The great thing about using REALbasic to create Macintosh (and Windows) applications is how easy REALbasic is to use compared to other application-programming environments. The challenging thing is that the majority of source code and programming tools are based on the most popular programming languages, some of which can be a bit of a challenge for beginning and intermediate programmers.

Although REALbasic is a very powerful development tool, keep in mind that other advanced programming and development tools usually have more features and are more versatile in certain situations. Try as you might, you'll be hard pressed to use just one programming tool to write all the types of applications and programs you might ever want. Some tools are more suited for specific tasks than others. As you gain more experience with programming and development, you're probably going to run into these other languages and development tools.

# Let's C What Develops

The C programming language, like the UNIX operating system, was created at Bell Labs in the early 1970s. To discuss the history of C, you have to discuss the history of UNIX. In a way, they both created each other.

## In the Beginning . . .

Way back in 1969, hippies, long hair, incense, and peppermints weren't the only thing people were thinking about. Although some were tuning in, turning on, and dropping out, Ken Thompson (no relation to the co-author of this book) was working on a computer-programming research project for Bell Labs in Murray Hill, New Jersey. He was developing computer programs that would be used to write other applications. (Talk about your chicken-and-egg problems, this guy was writing software to write software.) Ken's work was related to a project that Bell Labs had been working on in conjunction with MIT and General Electric. The Big Three were working on the development of the Multiplexed Information and Computing Service, or, for those who didn't want to deal with that tongue twister, the MULTICS operating system. Bell Labs decided to drop out of the MULTICS project and develop their own operating system.

The name *UNIX* was a backhanded play on words. Because a single group was developing UNIX, instead of the previous three that were developing MULTICS, they decided to drop the *MULTI*, meaning "many," and replace it with *UNI*, meaning "one." The *CS* was replaced with *X*—probably because it looked cooler.

Not long after Ken started working on his research, Dennis Ritchie stepped in to assist. The two of them transmogrified Ken's initial work into an operating system, which was the seed of the operating system that would eventually become UNIX.

## Writing the Programs to Write UNIX

One of the main goals of the UNIX development project was to write the operating system in a *high-level language* instead of in *machine language*. Writing in machine language usually means that your programs won't work on multiple computer platforms, and the creators of UNIX wanted to be able to use the operating system they were developing on all types of computers. Using a higher-level language, one that looks more like English, would allow them to compile the source code for multiple computing platforms with very few changes. This is probably the very first use of the term *cross-platform compatibility*, a term near and dear to the hearts of Macintosh users, who have to work in a world dominated by PCs.

A high–level computer language, among other things, is based closer to an English syntax, making it easier for humans to create and debug their programming work. C++, BASIC, and FORTRAN are examples of high–level languages. On the other hand, machine language is generally a complex string of either binary or hexadecimal numbers. Binary is sometimes (but not often) called base 2; hexadecimal counts up to 16 numbers, so additional "numbers" in base 16 are represented by the letters A through F.

For example, we know the number of fingers on both (human) hands is normally represented in decimal (base 10) as the number 10. In binary, there are only two numbers, 0 and 1, as opposed to base 10, which has 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. So, to represent the number of fingers on your hand in binary notation, you would have to see the decimal number 10 as 2 + 2 + 2 + 2 +2. Then, knowing that 2 in decimal notation is 10 in binary, you add things up, remembering to carry your "1" for each time you count to 10 base 2, or 10. Without belaboring this further, 1010 is the binary notation for decimal 10. Confusing? Now try writing decimal 20 in hexadecimal notation. (We'll let you figure this one out yourself.)

Although you're certain to have to learn how to count and translate numbers from binary or hexadecimal as part of some computer class, we suggest that you grab your sibling's or child's math book to see why coding a computer in this way is far more trouble than it's worth today.

In order to write UNIX in a high-level language, the folks at Bell Labs had two choices: they could use an existing language or write their own. They chose the latter, because there really wasn't a cross-platform language available at the time. They began working with a computer-programming language that was written by Ken Thompson; this language was based on Martin Richard's BCPL. The name Ken gave his computer-programming language was *B*. Dennis Ritchie improved upon Thompson's language, adding features that became necessary as the UNIX project progressed. Dennis, in a brilliant flash of insight, decided on the name *C* for his improved version of Ken's B programming language.

## UNIX, C and Beyond

In 1973, the fruits of the labors of Ken Thompson, Dennis Ritchie, and many other unsung programming heroes at Bell Labs were released to colleges and universities worldwide. UNIX became so popular that a mere six years later, UNIX Version 7 was released, which was written almost entirely in C, removing parts of the OS that were, of necessity, still written in machine language.

The first versions of C were available only with UNIX installations. C was included with UNIX so that programmers and developers could expand the capabilities of UNIX using the same language in which UNIX itself was written. Because C was such a powerful language, allowing the programmer to work very closely with the computer hardware and operating system, it wasn't long before non-UNIX programmers started clamoring for their very own version of C.

With the advent of personal computers in the early 1980s, it wasn't long before home-computer users could become fledgling computer programmers. Although most beginning programmers didn't opt for C as their language of choice, the software companies, which wrote the programs used by home-computer users, demanded it. Soon, countless versions of C were available for microcomputers, and C development caught on like wildfire.

## An Object-Oriented Revolution

About the same time that C was gaining a foothold in the personal-computer market, a change was occurring in the computer-programming community. Previously, most computer programming followed the standard structured methodology, in which the programmer decided on the tasks that the program must perform and wrote many separate routines to perform those tasks. This is often referred to as *top-down programming* because, for the most part, you can read the source code from the first line to the last, top to bottom, and glean from it an understanding of the tasks performed.

A bunch of wildcat, think-outside-the-box, do-things-differently kinds of programmers decided that there must be a better way. They wanted their code to be organized into chunks of functions, all of which operated on specific tasks or areas of the applications they were developing. Not only would this make it easier to locate specific code, but these chunks of related code, which they referred to as *objects*, could be used in more than one application, supporting code reuse and thus reducing duplication of effort.

Surprisingly, this object-centric view of programming, called *object-oriented programming*, wasn't really such a new idea. Even though most object-oriented languages didn't come into existence until the 1980s, the concepts of object-oriented programming had been around for quite some time. Some of the first object-oriented languages pre-date the existence of C itself.

Not wanting to give up the low-level hardware and operating-system control of C, developers began working on their very own object-oriented version of C, known as *C++*. C++ can be thought of as a bigger, better version of C. It can do everything C can, plus a whole lot more.

## The Once and Future King

Sure, there may be a few Java and Visual Basic holdouts in the development world, and of course, we REALbasic developers count, too. But C and its variant languages are still the dominant force in the programming world, and many tools are based on it. When new languages are developed, their syntax is often compared to C, and the more C-like they are, the more popular they seem to become.

Although other languages have gained in popularity, good old C, and its more feature-rich descendant C++, are going to be around for a long time. It's worth a bit of your time to become familiar with the de facto standard of C and C++ programming on the Macintosh platform, Metrowerks CodeWarrior.

# Macintosh C++ Development

Those not-too-faint-of-heart developers who wish to jump into C or C++ development have a quite few options to choose from. As with most software, there are both commercial and shareware tools for C and C++ development. The commercial versions usually offer more features, but they're pricier, and the features you gain might not make up for the difference in price. Commercial versions usually have better support, but shareware versions often make up for this with online support forums or discussion groups where you can converse with other users about the development tool you have chosen. Whether you choose to use a shareware or commercial package is up to you. It's really about what you're most comfortable with and your budget.

If your budget is a major issue, then first, thanks for setting aside a small portion of your budget on this book. Second, there's another option that should be attractive to you. Unlike other computer platforms, there are actually some free C and C++ compilers available for Macintosh development—yet another reason why owning a Macintosh is so cool!

## Metrowerks CodeWarrior

When it comes to Macintosh C++ development, Metrowerks CodeWarrior stands head and shoulders above the competition. Founded in 1985, Metrowerks makes many development tools for many platforms. Not only does Metrowerks have a Macintosh C++ compiler, it also has compilers for Windows, Solaris, and Linux, not to mention Java compilers and tools for Nintendo, PlayStation, and PlayStation2 development. It even has tools for developing software for the Palm handheld organizer. But enough of that, what we're interested in is their C++ development tool, CodeWarrior.

Examined on a large scale, CodeWarrior, along with most other development tools, is a lot like REALbasic. It has a Project window, a Code Editor window, tools to edit user interface elements, and so on (see Figure 23.1).

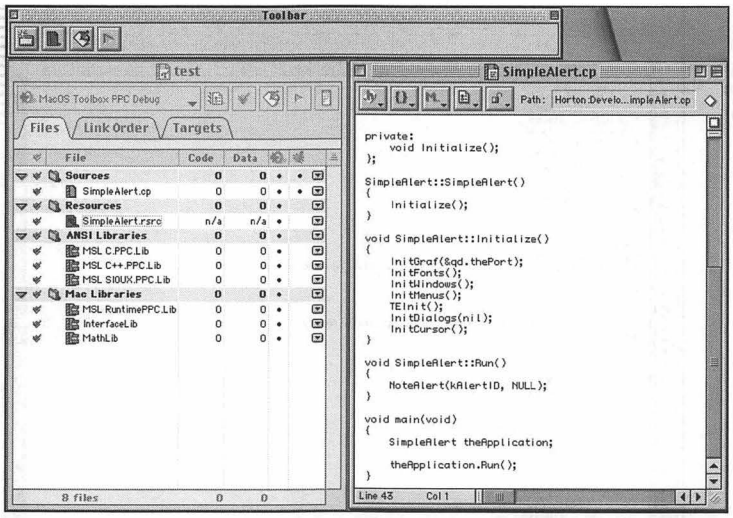



**Figure 23.1**
Metrowerks CodeWarrior's integrated development environment

Metrowerks offers a Learning Edition version of CodeWarrior, which can do darned near everything that the commercial version of CodeWarrior can do. The only real limitation of the CodeWarrior Learning Edition is that you can't use it to develop applications that are released to other users.

The benefit of using the Learning Edition (besides being cheaper than the full-blown version of CodeWarrior) is that it gives you a chance to try out an advanced Macintosh programming tool without shelling out a large chunk of money. If you decide that CodeWarrior is the way you want to go, you can always upgrade to the full-blown commercial version. Metrowerks will even give you a discount on the full version to help make up for the cost of the Learning Edition.

Although the Code Warrior path to enlightenment is cheap, there are cheaper ways to go. How does free sound?

## Macintosh Programmers' Workshop (MPW)

Free you say? Yep, that's right. Like any good computer hardware company, Apple wants third-party developers to build applications for its platform—so much so that Apple is willing to give away the tools to develop applications for

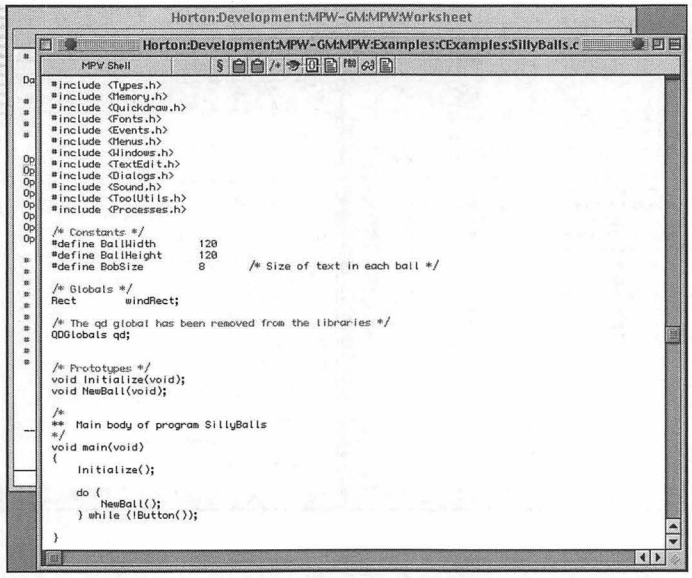



**Figure 23.2**
The Macintosh Programmers' Workshop

Apple products. One major development tool offered by Apple is the Macintosh Programmers' Workshop, or MPW, shown in Figure 23.2.

You can download MPW from the developers section of the Apple Web site if you'd like to check it out, but be prepared for your brain to hurt. The first thing you'll notice is that the only real tool that exists in the MPW is a text editor. You use the text editor to edit your source code, to edit the project definition files, and to build your application. It's only about one step above a UNIX, or MS-DOS, command-line editor and code compiler.

There's an old saying about anything that's free: You get what you pay for. Although it's possible to develop applications using the Macintosh Programmers' Workshop it's not something to attempt unless you've got a lot of time and patience. Developing applications using the Programmers' Workshop isn't impossible, but it can be painful. Don't get us wrong: we're sure there are developers out there who are big fans of the MPW, and can do great things with it—we're just not them.

We recommend that anyone who finds modern Macintosh application development to be difficult should download the MPW. You'll gain a whole new level of appreciation for the other development tools.

# Project Builder and Mac OS X

When it comes to developing applications for Mac OS X, one of the best-known tools is Apple's Project Builder, shown in Figure 23.3. Using Project Builder, you can build applications for OS X in one of many development environments. With Project Builder, you can create applications using Carbon and Cocoa, which are written in C, C++, Objective C, and Java.

Project Builder sounds like a great tool for developing Mac OS X applications—but you're probably wondering how much it costs and where you can get it. Well, good news! Every Mac OS X box comes with the Developer Tools for Mac OS X 10.0 CD. That's right, it's free! Everyone who buys Mac OS X gets a copy of the Developer Tools CD! Not only does the Developer Tools CD include Project Builder, it also includes other development, design, and debugging tools, all of which integrate with Project Builder to provide you with everything you need to build OS X applications.

So if Project Builder is so great, why is this book about REALbasic instead of Project Builder? Well, Project Builder, like all advanced development tools, can be very hard for beginning and intermediate programmers to wrap their brains around. It's not that REALbasic is less powerful than these advanced

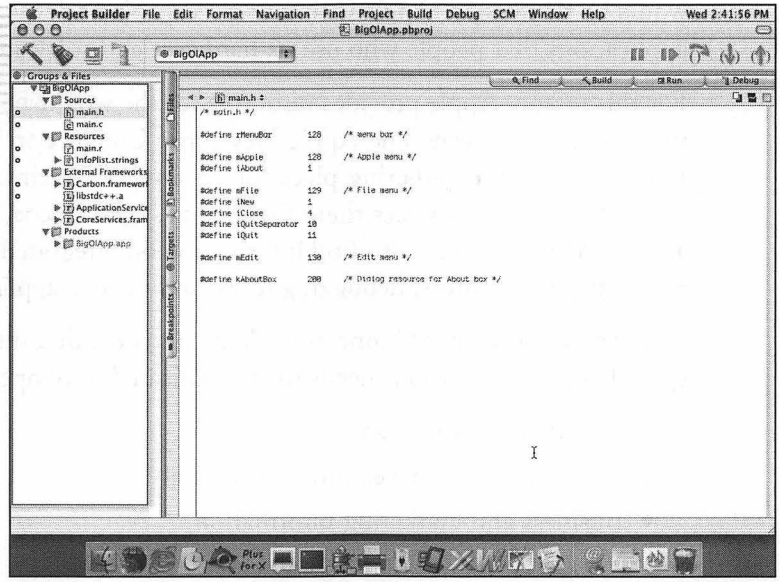
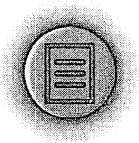



**Figure 23.3**
Project Builder and Mac OS X

tools, it's just a lot easier to pick up and start using. Plenty of applications written in REALbasic compete with applications written using more-advanced tools.



**NOTE**

Some time in the future, you may find that you've exceeded REALbasic's capabilities, but we doubt it. REALbasic continues to improve, with new features being added all the time. If you find that your current version of REALbasic can't do what you want, check the latest version—it might just support your needs.

# The Apple Developer Connection Web Site

Apple Computer, being the clever hardware and software company that it is, is well aware of the fact that developers are the lifeblood of its survival. Without third-party developers to write the applications, utilities, productivity tools, and games that computer users demand, the very survival of Apple would be threatened. Apple is simply not capable of developing all the various applications that people who use its hardware and software require.

For this reason, Apple provides resources for the would-be software developer to get him on his way. The Apple Developer Connection Web site, shown in Figure 23.4, is the gathering place for all sorts of Macintosh software developers. You'll find resources there for C, C++, Java, Cocoa, FORTRAN, Lisp, and REALbasic. You'll also find information on integrated development environments, code editors, debugging tools, and various application frameworks.

The Apple Developer Connection Web site is divided into three main areas specializing in the various needs of the dedicated developer:

- The Partners Program
- Technology and development resources
- Business and marketing information


**Figure 23.4**
The Apple Developer Connection Web site

## Grab Your Partner: The Partners Program

The first and foremost of the ADC areas is the Partners Program area, shown in Figure 23.5. In this area, you have access to the online version of the ADC newsletter and can get information about various ADC programs and products. You can also visit technical-support areas and the Macintosh Product Guide, which can provide much-needed information and assistance regarding the thousands of existing Macintosh products.

One of the nicest features of the ADC Partners area of the Web site is that it offers you the ability to join various Apple Developer Connection programs, geared toward both large and small development shops. There are various levels of membership:

- The online program
- The student program
- The select program
- The premier program


**Figure 23.5**
The Apple Developer Connection Partners Program section

## The Online Program

When you join the free ADC online program, you can download various free development tools and receive weekly updates via the Apple Developers Connection newsletter. Just about anyone who is considering becoming even a part-time author of Macintosh shareware—or if you're just interested in learning about Mac programming—should consider joining the online program. You can't beat the price.

## The Student Program

Another notable membership program is the student program. For a small annual fee, you get all the benefits of the online program, plus discounts on select third-party tools and conferences. This level is geared toward college students, so most of the benefits are geared toward discount programs. A student developer can more than make up the cost of membership with the various discounts she receives. So think of this as a coupon book that will pay for itself after the first few purchases.

## The Select Program

The select program gives semi-professional developers the most bang for their buck. At this level, you receive the benefits associated with the online and stu-

## The ADC Developer Mailing Option

At the online and student levels, you have the option of adding the ADC Developer Mailing feature, a 12-month subscription of CDs that contain updates on Macintosh technical and marketing information. You'll also get at least one edition of the Developer CD Series, which includes the following:

- System software. This includes the latest versions of Macintosh Operating System software, including foreign-language versions. Having access to the foreign-language versions of the OS will allow you to develop versions of your applications for other countries.
- Mac OS Software Development Kits. These are development kits that allow you to develop tools that work with the various Macintosh operating systems and applications. You'll get all the system software, programming interfaces, libraries, sample code, and technical documentation you need to develop applications, that work with QuickTime, AppleScript, and the like.
- Tool Chest. This includes development tools, utilities, sample source code, and documentation.
- Reference Library. This contains a complete set of the core Apple development documentation series, and includes the *Macintosh Human Interface Guidelines, Inside Macintosh, Developer Tech Notes,* and *Develop Magazine*—all on CD.

dent programs. In addition, you get monthly updates of system software, development tools, and technical documentation—all part of the ADC Developer Mailing option, which is included in with the select program. As a member of the select program, you'll become part of the Mac Seeding program, which allows developers to get pre-release versions of Macintosh application and system software. You'll also get two free non-warranty technical-support calls, allowing you to get help straight from the horse's mouth without having to pay the cost normally associated with calls made on non-warranty products.

### The Premier Program

The ADC Premier program, costing $3,500, is the end-all, be-all level of ADC membership. At this level, you get everything the previous levels get, plus the ADC premier mailing. You also get free access to various Apple conferences, eight tech-support calls, software discounts, and more. This is the level of membership that the serious commercial application developer should consider purchasing. You get access to more information and discounts than at any other level. It's the priciest level of membership, but the benefits make it worthwhile.

## Development Resources

The second most useful of the ADC areas is the Technology and Development Resources section, shown in Figure 23.6.

The Technology and Development Resources section is where you can go to get information on various Apple developer products. You can download Apple development tools, such as the Macintosh Programmer's Workshop and Project Builder. You can even download sample code and technical documentation to help you learn more about Macintosh software development.


**Figure 23.6.** The Apple Developer Connection Technology section

The Getting Started section is extremely valuable for the beginning to intermediate developer who wants to learn how to use the various Apple development tools. In this section, you'll find step-by-step tutorials and sample code to lead you through creating applications in Carbon, Cocoa, and Java.

## Grow Your Business: The Business and Marketing Section

The last of the ADC areas is the Business and Marketing section, shown in Figure 23.7.

This under-appreciated section of the Apple Developer Connection Web site will come in particularly handy when you decide to turn your small programming company into a larger professional company. You'll find resources in this area to help you understand Apple's various market advantages in consumer, education, creativity, and small-business markets. You'll learn how to best target your applications for each of these markets to improve your chances of delivering a successful product.

As an Apple user, you're aware—or will be shortly—of others' resistance to using Macintosh products. The documentation in this area will help you


**Figure 23.7.** The Apple Developer Connection Business and Marketing section

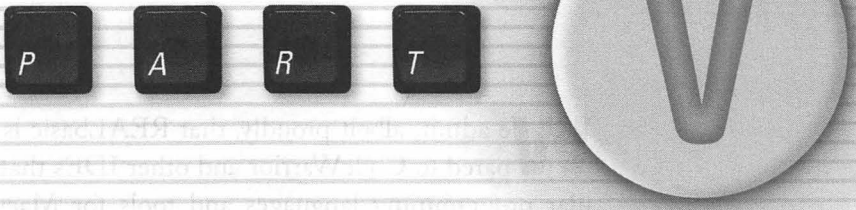
convince these Nervous Nellies that not only is the Macintosh a stable, safe, and affordable computing platform, it's just plain more fun than using IBM-compatible PCs. You can check into various market-research studies published by Apple and read various Apple business cases to help you strengthen your arguments when trying to convince that reluctant business owner that Apple (and your applications) is the best way to go.

# Review

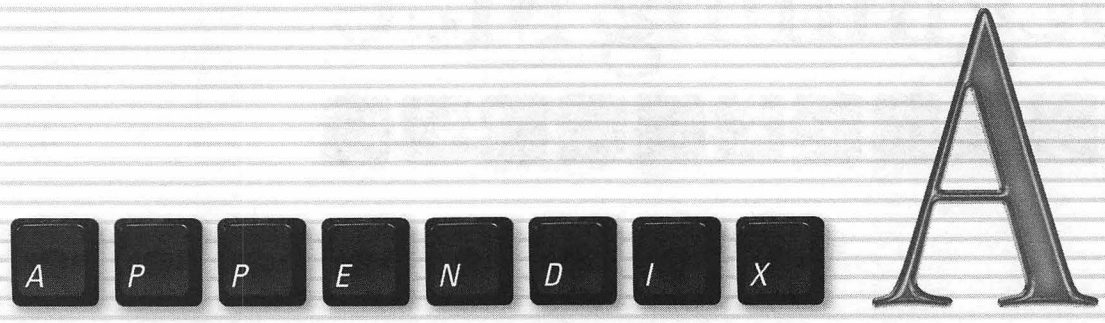
Advanced programming is something that can be achieved with just about any Macintosh development tools. Although the choice of tool is based more on preference than on anything else, you might want to talk to other developers to help you decide on the best avenue for you. The Apple Developer Connection is going to be your best first resource for finding information on Macintosh application development tools.

Beginning Mac®

PROGRAMMING




# Appendixes




# REALbasic Resources

We admit, albeit proudly, that REALbasic is an underdog IDE compared to CodeWarrior and other IDEs that use C++ and other popular programming languages and tools for Macintosh development. We know most developers out there prefer C++ or have been trained that way. For REALbasic, like the Macintosh itself, being an underdog has its advantages. You should know that underdogs like poodles are listed among dogs that bite the most.

There is a strong following of REALbasic developers who make their knowledge and tools available over the Internet. You'll find everything from tutorials to freeware to sample code, and lots of people willing to help you.

## REAL Software: The Official Home of REALbasic

**http://www.realbasic.com**

REAL Software, if you've been sleeping or studying this book through osmosis, is the creator of REALbasic. At REAL Software's Web site, shown in Figure A.1, you'll find the latest Standard and Professional versions, helpful tools, and tutorials. You can buy a copy online, if you like. There's contact information for technical support, and links to other

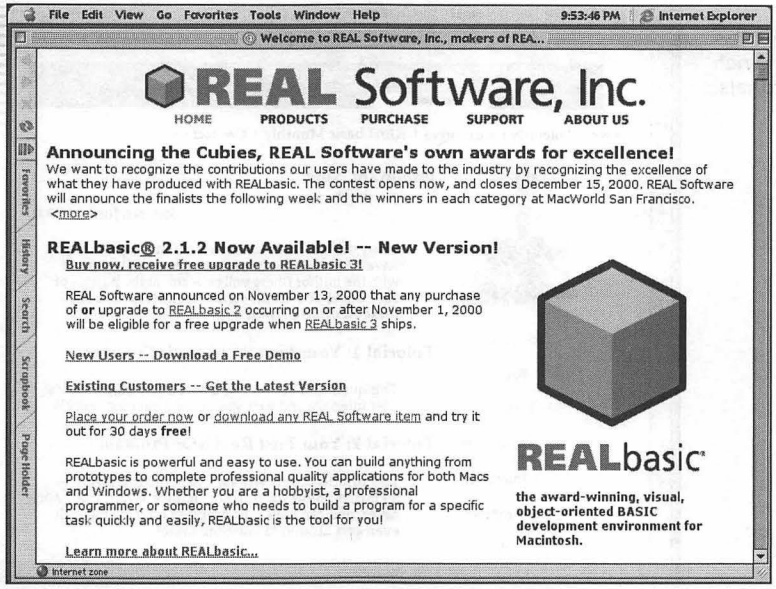



**Figure A.1**
The official home of REALbasic, from REAL Software

REALbasic-oriented sites. Geoff Perlman and his crew have sweated long and hard to bring you one of the best—if not the easiest—programming environments available for both classic Mac OS and Mac OS X development.

The REALbasic CD (of which we've provided a copy in the *REALbasic* folder on this book's CD) includes many applications created with REALbasic, including the electronic versions of the tutorial, developer's guide, and language reference. You can download the latest versions of these guides here.

REAL Software loves to expose developers to new alpha and beta versions of REALbasic so they can refine the features for a new general release. Remember that alpha and beta versions of REALbasic may have bugs and shouldn't be used for full-scale development. Use them at your own risk.

## REALgurus

**http://www.realgurus.com**

If there's any RB support site you should visit often, it's REALgurus, shown in Figure A.2. It offers a message board, a massive collection of tutorials, sample code, and much more.

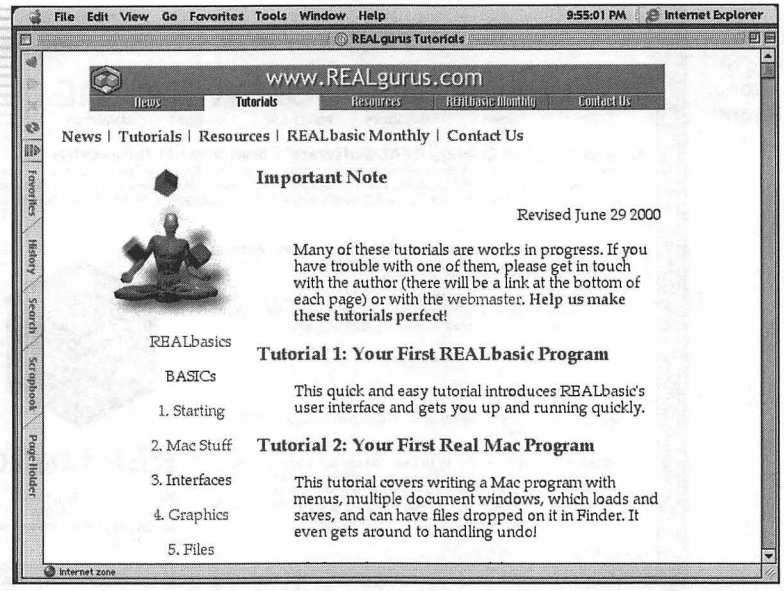



**Figure A.2**
REALgurus is rich with tutorials.

# REALGoodies

**http://www.geocities.com/SiliconValley/Station/7130/home.html**

The REALGoodies site provides a few projects that you can peruse to further your understanding of REALbasic programming. Kevin Mullins, the Webmaster, also has a part of a game written in REALbasic that uses many techniques you could use in your own game.

# REALbasic Monthly

**http://www.nd.edu/~jvanderk/rbm/**

The Webmaster and author of *REALbasic Monthly* decided to stop new publications not long ago, but left the site, shown in Figure A.3, available for developers to find some inventive resources on programming. Perhaps someone out there (maybe even you) can get RBM back on its feet.


**Figure A.3**
RBM is no longer published regularly, but its information is still available and informative.

# Einhugur Software

**http://www.einhugur.com/**

REALbasic's power can be extended through plug-ins, and Einhugur Software offers members and visitors samples of complex REALbasic plug-ins and classes for many project ideas. If you plan on developing Windows applications in addition to Macintosh, this site, shown in Figure A.4, offers items that can make your project shine.

# REALnews

**http://www.swssoftware.com/realnews/**

As a Macintosh technician, I regularly visit a handful of Macintosh news sites on the Web. Two sites, Macintouch (http://www.macintouch.com) and MacFixIt (http://www.macfixit.com) offer everything I need to know in daily happenings in the Apple world. REALnews, shown in Figure A.5, reminds me of these sites—it's one of the best programming news sites I've found,

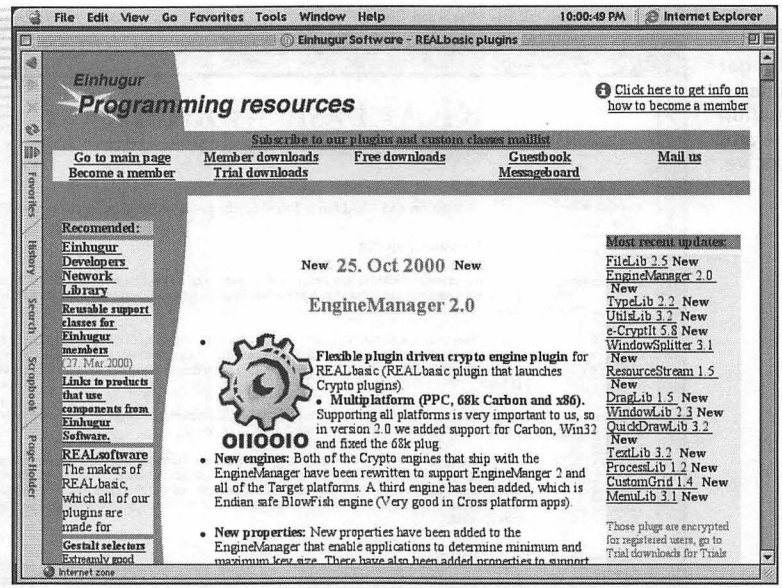



**Figure A.4**
Einhugur Software sells some software, but is also a comprehensive REALbasic resource site.

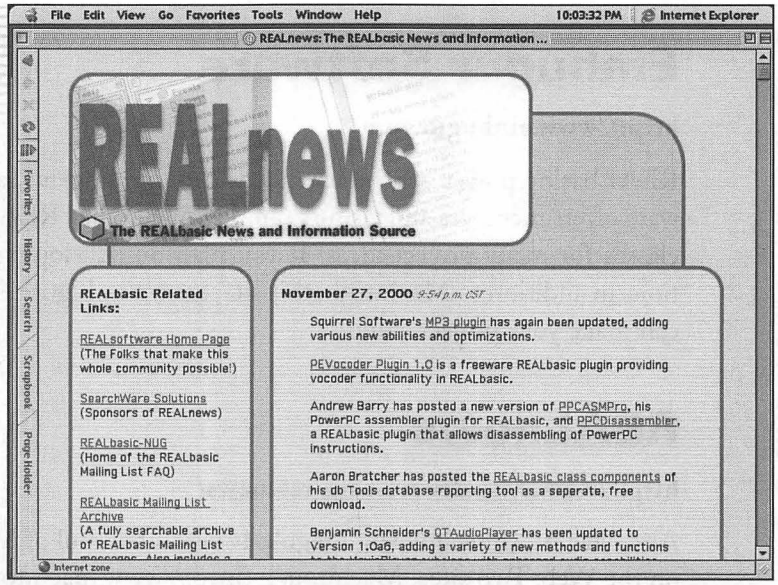



**Figure A.5**
Next to REAL Software's site, REALnews is the place to go for the latest news in the REALbasic world.

and an essential resource to keep up on REALbasic news. You can also send announcements to this site when you finish your big projects, telling people where they can find them on your Web site.

# Zegsoft

**http://zegsoft.tripod.com/realbasic.html**

Thinking of making the next great desktop-publishing or word-processing program? Perhaps you should visit Zegsoft, whose shareware ruler class and printing plug-in might be useful in your development. Best of all, Zegsoft claims that its plug-ins work in Windows applications, too. The Zegsoft site is shown in Figure A.6.

# The REALbasic Mailing List

**http://www.realsoftware.com/support.html**

REAL Software provides several mailing lists to which you can subscribe and contribute your tips and get help on using REALbasic. All five mailing lists are available as they are posted or in a single-message digest version. I've been pleasantly peppered with the digests each day, and each contains at least one useful tidbit. Also available from this page is a link to archives of each mailing list. A Sherlock plug-in is available for users of Mac OS 8.6 and later to search the archives quickly.

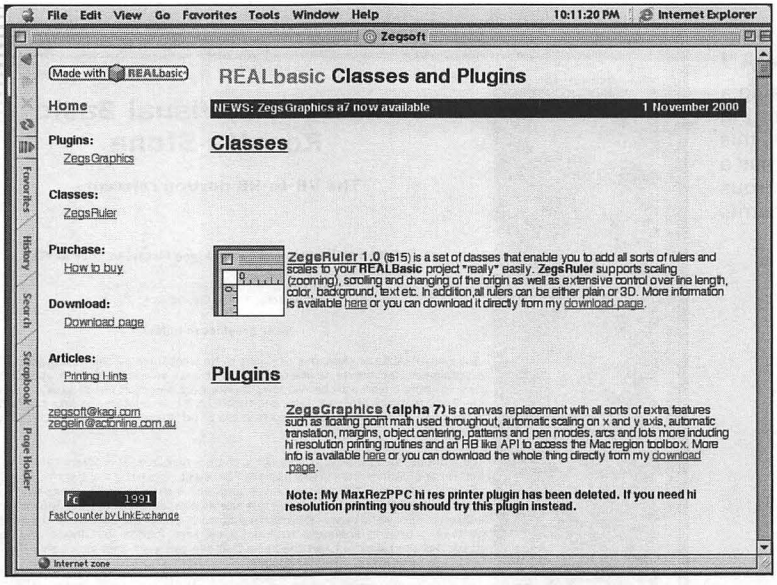



**Figure A.6**
Find some great shareware tools at Zegsoft.

TIP

Please visit the REALbasic support site for complete information on the focus of each list and how to subscribe via e-mail.

# The REALbasic/Visual Basic Rosetta Stone

**http://kode-fu.com/rosetta/**

Each time I type this URL, I think of that scene in the *The Matrix* in which Keanu Reeves's character downloads martial-arts training into his brain and announces, with awe, "I know kung-fu." My mind conjures up an image of a developer in martial-arts training robes at a computer.

If you happen to use Visual Basic or know of others who use that Windows-only IDE, you may find yourself with questions on the differences and compatibility between it and REALbasic. This Web site, shown in Figure A.7, provides information on many function and control uses available in Visual Basic, and how to adapt them to work in REALbasic. If you happen to know of a Visual Basic developer who's seriously considering making a Mac app, toss her a trial version of REALbasic and point her to this site. You'll be glad you did.

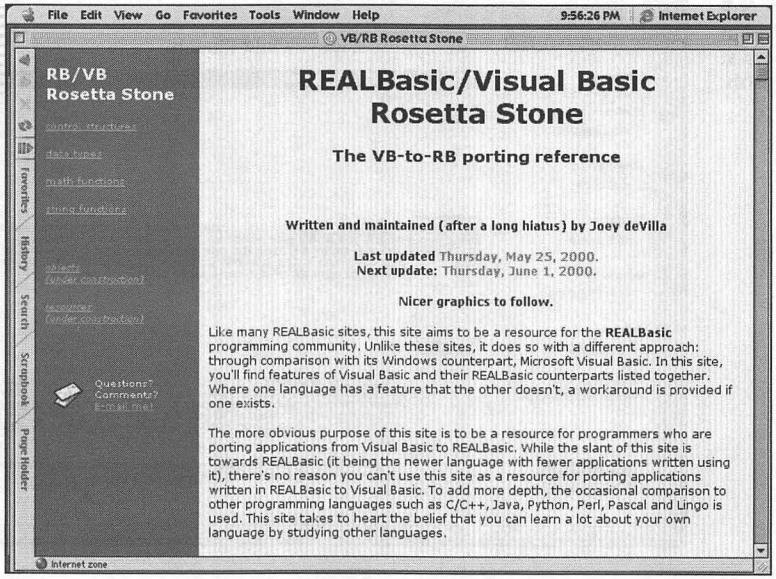
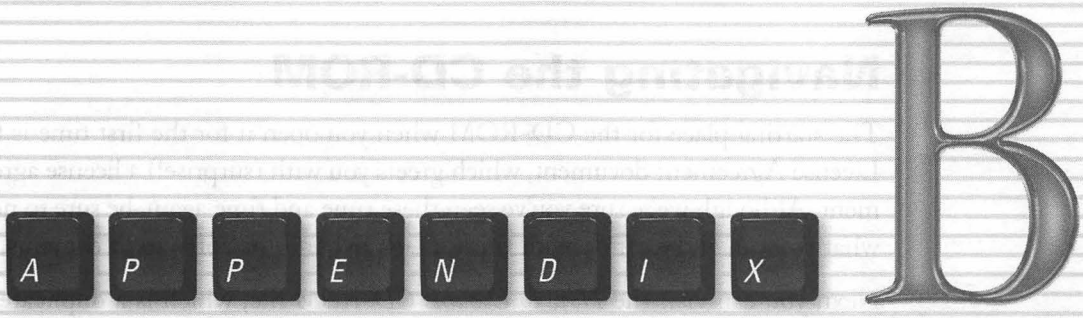



**Figure A.7**
Besides being a good porting reference, this site also has a humorous domain name.




# How to Use the CD-ROM

The *Beginning Mac Programming* CD-ROM contains a trial version of REALbasic and other tools you need to get started with Macintosh application development.

## System Requirements

To use our CD-ROM, your computer should meet the following minimum requirements:

- It must have a CD-ROM drive.
- It must run Mac OS 7.6.1 or later.
- It must have a hard disk with 6.5MB of free space.

Although REALbasic itself has pretty relaxed requirements, do note that, in order to benefit and create decent Macintosh applications within your lifetime (given that speed isn't the first thing you think about with a Macintosh Quadra), you really should use a Power Mac—*any* Power Mac. Note also that non-Power Macintosh versions of REALbasic cannot create Power Macintosh versions of REALbasic apps, although you can create non-Power Mac apps on a Power Mac.

# Navigating the CD-ROM

The starting place for the CD-ROM when you open it for the first time is the License Agreement document, which greets you with (surprise!) a license agreement. Although we're sure you've seen these time and time again, be sure to note what you can and can't do with this CD-ROM and the software it contains.

Next, you can visit the *How to Use this CD-ROM* file, which is a duplicate of this appendix, but may contain last-minute information that came too late to be added to the book-bound version.

## REALbasic 3.2

REALbasic is an integrated programming environment—that is, an application that creates other applications. The folder *Open Me for REALbasic 3* contains the same software and accessories available from the REALbasic trial CD and REAL Software's Web site.

To install REALbasic and its support folders, documentation, and third-party applications on any Power Mac or 68K Macintosh computer running Mac OS 7.6.1 or later (but **not** Mac OS X), do the following:

1. Open the folder named *Open Me for REALbasic 3*.
2. Drag the *REALbasic 3.2* folder to your *Applications* folder on your hard disk.

To install only the REALbasic application that's designed for your computer, follow the steps to install the REALbasic application and accessories as described above, then do the following:

1. Open the folder named *Open Me for REALbasic 3*.
2. Open the folder named *Other Versions*.
3. If you use a 68K Macintosh, open the *68K* folder, and drag the REALbasic application from that folder into the *REALbasic 3* folder on your hard disk. When prompted whether you want to replace the existing copy of REALbasic, click OK.
4. If you use a Power Macintosh, open the *PowerPC* folder, and drag the REALbasic application from that folder into the *REALbasic 3* folder on your hard disk. When prompted whether you want to replace the existing copy of REALbasic, click OK.

If you're a Power Macintosh user who wants to develop Carbon applications, you should install the REALbasic Carbon application for Mac OS 9.

1. Open the folder named *Open Me for REALbasic 3*.
2. Open the folder named *Other Versions*.
3. Open the *Mac OS 9/Carbon* folder and drag the REALbasic application from that folder into the *REALbasic 3* folder on your hard disk. When prompted whether you want to replace the existing copy of REALbasic, click OK.

Mac OS X users can use a Carbonized version stored in a Disk Copy for Mac OS X image. To install REALbasic in Mac OS X:

1. Open the folder named *Open Me for REALbasic 3*.
2. Open the folder named *Other Versions*.
3. Open the *Mac OS X Disk Image* folder.
4. Double-click the REALbasic3.2.dmg image file. (If the file fails to open, launch Disk Copy (located in Applications/Utilities on your Mac OS X hard drive) and drag the image to the Disk Copy window. A virtual disk of REALbasic will appear on your desktop.
5. In your Applications folder, create a new folder named "REALbasic".
6. Open the REALbasic disk image and drag all its contents into the REALbasic folder in your Applications folder.

In addition, the Open Me for REALbasic 3 folder contains links to REAL Software's tutorials and lots of sample code and third-party applications to help further inspire you. To read the documentation, you need the Adobe Acrobat Reader application. An HTML page containing a link for this application is included in the Open Me for REALbasic 3 folder.

## It's a Trial Version Until You Pay for It

The copy of REALbasic you install will work as a trial version of the Standard version for 30 days until you register it. In trial mode, applications you create will work for only a few minutes, and an annoying message will appear when you launch any application you created from the trial version. Certain other database features and Windows development options are limited, as well.

To get rid of the messages and the time-out of the application, just drop by REAL Software and kindly plunk down some change to purchase a license

code. This code will activate the copy you install as a Standard or Professional version, depending on how much you paid. Go ahead. You'll be glad you did.

### Sample Projects from the Book

Some chapters of the book describe REALbasic programming processes and projects you will create as you learn how to develop applications and use REALbasic tools. Included in the CD-ROM are those very same projects in the *My Paint-Sample Project* folder. Simply open the projects or access the code as you need.

## How Did We Make the Picture in the CD-ROM Window?

A few of you may notice that we created a picture of the book's cover within the window of the CD-ROM. Great trick, but how did we do it? Answer: a freeware application called Iconizer Pro. This application breaks any graphic into icons that form a picture in a Finder window as a mosaic.

After you make your first application, you can use Iconizer Pro to create a great folder arrangement where your application and support materials are stored. And yes, folder pictures made in Iconizer Pro work in both Mac OS 9 and Mac OS X, provided that you show things in Icon view (now that's obvious, right?).

Iconizer Pro is available for download from many locations on the Web. Although it is free, be sure to register the application with the author so that pictures you create with it don't show "unregistered" on each icon's information (from the Get Info window).

# Index

## A

# B



# C

# D

## E



## F

# G



# H

# I



# J



# K



# L

# M

# N



# O

# P

## Q



## R

# S

# T

# U



# V

# W



# X



# Z

# License Agreement/Notice of Limited Warranty

By opening the sealed disk container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disk to the place where you purchased it for a refund.

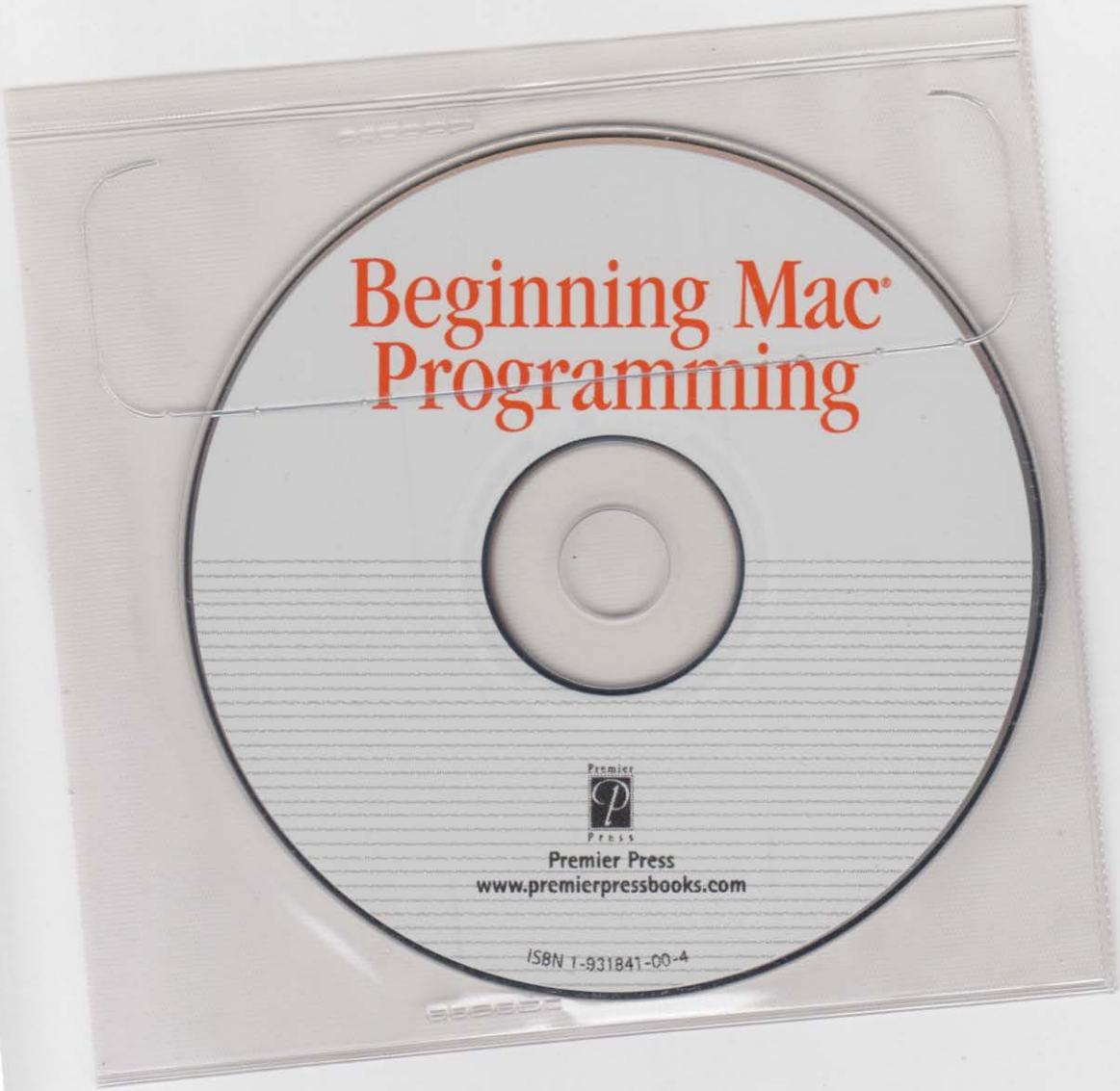
**License:**
The enclosed software is copyrighted by the copyright holder(s) indicated on the software disk. You are licensed to copy the software onto a single computer for use by a single concurrent user and to a backup disk. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disk only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

**Notice of Limited Warranty:**
The enclosed disk is warranted by Premier Press, Inc. to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disk combination. During the sixty-day term of the limited warranty, Premier Press, Inc. will provide a replacement disk upon the return of a defective disk.

**Limited Liability:**
THE SOLE REMEDY FOR BREACH OF THIS LIMITED WARRANTY SHALL CONSIST ENTIRELY OF REPLACEMENT OF THE DEFECTIVE DISK. IN NO EVENT SHALL PREMIER PRESS, INC. OR THE AUTHORS BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING LOSS OR CORRUPTION OF DATA, CHANGES IN THE FUNCTIONAL CHARACTERISTICS OF THE HARDWARE OR OPERATING SYSTEM, DELETERIOUS INTERACTION WITH OTHER SOFTWARE, OR ANY OTHER SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES THAT MAY ARISE, EVEN IF PREMIER PRESS, INC. AND/OR THE AUTHOR HAVE PREVIOUSLY BEEN NOTIFIED THAT THE POSSIBILITY OF SUCH DAMAGES EXISTS.

**Disclaimer of Warranties:**
PREMIER PRESS, INC. AND THE AUTHORS SPECIFICALLY DISCLAIM ANY AND ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY, SUITABILITY TO A PARTICULAR TASK OR PURPOSE, OR FREEDOM FROM ERRORS. SOME STATES DO NOT ALLOW FOR EXCLUSION OF IMPLIED WARRANTIES OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THESE LIMITATIONS MAY NOT APPLY TO YOU.

**Other:**
This Agreement is governed by the laws of the State of California without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Premier Press, Inc. regarding use of the software.


Beginning Mac®
Programming
Premier
Press
Premier Press
www.premierpressbooks.com
ISBN 1-931841-00-4

# Beginning Mac®

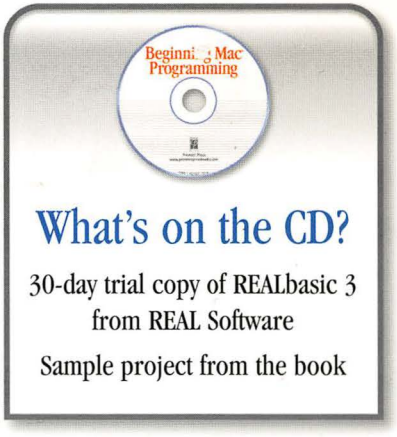
PROGRAMMING

# Build Your First Mac Program

Congrats! You've found your one-stop guide to programming for Mac® OS! Even if you have zero programming know-how, you'll be up to speed in no time as you progress from Macintosh programming basics to building your first application using REALbasic®. Along the way, you'll learn the ins and outs of Object Oriented programming, Classic, Carbon, and Cocoa!

## Use this book to make Mac applications!

- Get acquainted with REALbasic
- Learn the parts of a Mac program
- Understand variables and constants
- Develop and build your first Mac program
- Work with editing features and add final touches to your program
- Learn about Mac OS X programming
- Explore advanced programming for Mac OS




### What's on the CD?

30-day trial copy of REALbasic 3 from REAL Software

Sample project from the book

As a tech support specialist, **Kevin Spencer** has been explaining difficult topics to people for years. He is an avid Macintosh fan and knows the Mac inside and out. He is a frequent contributor to various Macintosh publications. Kevin lives in Indianapolis with his wife and two children.

Working as a software developer since 1985, **Jeff Thompson** has written applications in various languages such as BASIC, Z-80 Assembler, 6502 Assembler, DBL, 8088 Assembler, C, and C++. Jeff worked with and developed applications on various platforms from the good old days of the TRS-80, Apple II, DEC minicomputers, IBM PC's, and the Macintosh Plus all the way up to today's latest Pentium PC's and Macintosh PowerPC systems. He's currently employed by CTI Data Solutions as Senior Systems Analyst and Technical Lead on one of the highest rated Billing Analysis software applications in the country. Jeff currently resides in Indianapolis with his wife and two children.

Premier Press
www.premierpressbooks.com
User Level: Beginning/Intermediate
Category: Operating Systems
U.S. $39.99 Can. $59.95 U.K. £29.99

ISBN 1-931841-00-4