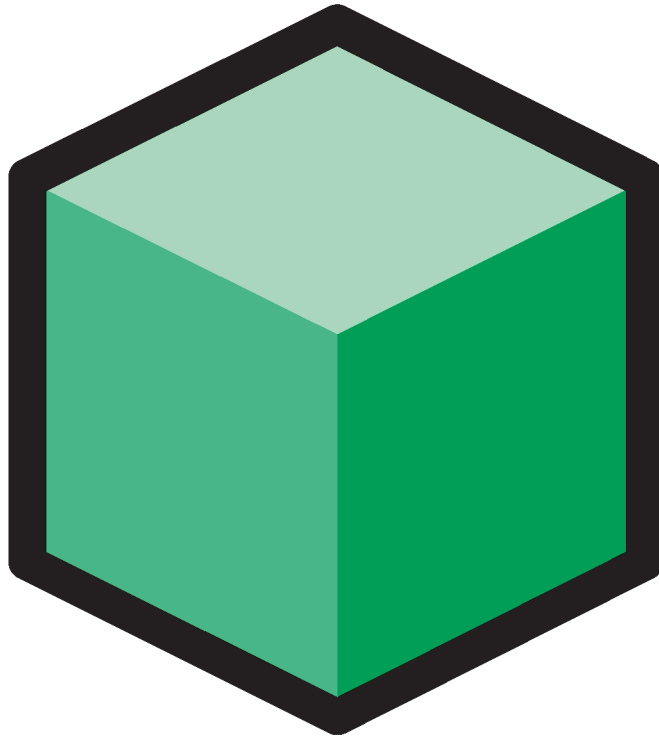


# REALbasic®



# TUTORIAL

---

# REALbasic Tutorial

Documentation by David Brandt and Dave McCollum;  
concept by Geoff Perlman.

© 1999-2000 by REAL Software, Inc. All rights reserved.  
Printed in U.S.A.

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

V 2.1, April 2000

---

# Contents

<b>CHAPTER 1</b>	<b>Introducing REALbasic</b>	<b>7</b>
	How to Use this Manual . . . . .	8
	Who Should Use this Manual . . . . .	8
	Presentation Conventions . . . . .	9
	Lesson Files . . . . .	11
	On Your Mark, Get Set, Go! . . . . .	11
<b>CHAPTER 2</b>	<b>Creating Windows</b>	<b>13</b>
	Starting Up REALbasic . . . . .	14
	REALbasic's Windows . . . . .	15
	Building a Document Window . . . . .	16
	Adding an EditField . . . . .	17
	Configuring TextField as a Text Editor. . . . .	21
	Review . . . . .	26
<b>CHAPTER 3</b>	<b>Creating Menu Items</b>	<b>27</b>
	Adding a Select All Menu Item . . . . .	28
	Adding the Menu Item . . . . .	28
	Enabling the Menu Item. . . . .	30
	Handling the Menu Item. . . . .	31
	Review . . . . .	33
<b>CHAPTER 4</b>	<b>Working with Documents</b>	<b>35</b>
	Getting Started . . . . .	36
	Working with Text Documents . . . . .	36
	Creating the New Menu Item . . . . .	36
	Enabling the New Menu Item . . . . .	37
	Handling the New Menu Item . . . . .	38
	File Types. . . . .	40

Saving Documents . . . . .	40
Adding the Save Menu Item . . . . .	41
Adding Properties to TextWindow . . . . .	41
Enabling the Menu Item . . . . .	43
Adding a SaveFile Method . . . . .	44
Using The Online Reference . . . . .	46
Managing the TextHasChanged Property . . . . .	48
Handling the Menu Item. . . . .	50
Adding a Save As Menu Item. . . . .	51
Adding a 'Save Changes' Dialog Box . . . . .	52
Creating the Dialog Box . . . . .	52
Displaying the Save Changes Dialog Box . . . . .	58
Adding an Open Menu Item . . . . .	61
Creating the Open Menu Item . . . . .	61
Handling the Menu Item. . . . .	62
Review . . . . .	64

## **CHAPTER 5**

<b>Working with Text</b>	<b>65</b>
Getting Started . . . . .	66
Configuring TextField for Styled Text . . . . .	66
Implementing the Style Menu . . . . .	66
Creating the Style Menu and its Menu Items . . . . .	67
Enabling the Style Menu Items . . . . .	68
Handling the Style Menu Items. . . . .	69
Managing Changes to the Text Selection . . . . .	72
Creating the Size Menu . . . . .	73
Creating the Size Menu and its Menu Items . . . . .	73
Enabling the Size Menu Items . . . . .	75
Adding the Menu Handler . . . . .	76
Managing Changes to the Text Selection . . . . .	78
Testing the Style and Size Menus . . . . .	79
Printing Styled Text . . . . .	79
Creating the Page Setup and Print Menu Items. . . . .	79
Enabling the Page Setup and Print Menu Items. . . . .	80

	Handling the Page Setup Menu Item . . . . .	80
	Handling the Print Menu Item . . . . .	81
	Testing Styled Text Printing . . . . .	83
	Review . . . . .	83
<b>CHAPTER 6</b>	<b>Creating Dynamic Menus</b>	<b>85</b>
	Getting Started . . . . .	86
	Implementing the Font Menu. . . . .	86
	Building the Font Menu . . . . .	87
	Enabling the Font Menu. . . . .	89
	Handling the Font Menu. . . . .	90
	Review . . . . .	93
<b>CHAPTER 7</b>	<b>Communicating Between Windows</b>	<b>95</b>
	Getting Started . . . . .	96
	Implementing the Find Dialog Box . . . . .	96
	Creating the Menu Item . . . . .	96
	Enabling the Find Menu Item . . . . .	97
	Creating the Find Dialog Box . . . . .	98
	Creating the Dialog Box . . . . .	99
	Review . . . . .	104
<b>CHAPTER 8</b>	<b>Wrapping Things Up</b>	<b>105</b>
	Getting Started . . . . .	106
	Using the Debugger . . . . .	106
	Automatic Debugging Features . . . . .	106
	Using the Debugger to Find Logical Errors	108
	Building a Stand-alone Application . . . . .	113
	Review . . . . .	115
	<b>Index</b>	<b>117</b>



# Introducing REALbasic

---

Welcome to REALbasic!

REALbasic is an integrated development environment based on a modern version of the BASIC programming language. The REALbasic application comprises a rich set of *graphical user interface* objects (commonly referred to as GUI), an object-oriented language, an object browser, and a debugger.

REALbasic provides you with all the tools you need to build virtually any application you can imagine.

If you are new to programming, you will find that REALbasic makes it fun and easy to create full-featured MacOS and Windows applications. If you are an intermediate or advanced programmer, you will appreciate REALbasic's rich set of built-in tools.

## How to Use this Manual

The *REALbasic Tutorial* comprises a series of practical lessons for learning REALbasic. The lessons are structured so that each one can be completed in about 30 minutes. Since the material in each chapter builds on the previous one, you should plan on working sequentially through this tutorial.

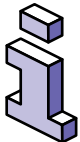
During the course of this tutorial, you will use REALbasic to build a complete application. You will build a text editor application that is similar to SimpleText, the text editor included with Macintosh computers.

You will quickly learn to appreciate REALbasic's power and ease of use. For the entire application, you will only need to create about 200 lines of programming code (SimpleText is built from over 20,000 lines of C/C++ programming code).

## Who Should Use this Manual

The tutorial is written for someone who is new to programming. You do not need any knowledge of programming in order to complete this tutorial.

If you have some programming experience, you may want to quickly review this tutorial so that you'll become familiar with REALbasic's *integrated development environment* (IDE) and language features.



If you are new to the MacOS, you should study the documentation that came with your computer. The documentation will help you learn how to use the mouse, menus, disks, and other aspects of the MacOS.

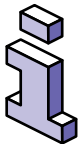


## Presentation Conventions

*Italic* type is used to emphasize the first time a new term is used, and to highlight import concepts. In addition, titles of books, such as *REALbasic Developer's Guide*, are italicized. *Courier* type is used for programming code. **Bold** is used to indicate text that you will type while using REALbasic.

When you are instructed to choose an item from one of REALbasic's menus, you will see something like "choose File ► New (⌘-N)". This is equivalent to "choose New from the File menu (⌘-N)".

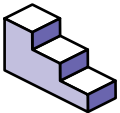
The items within the parentheses are *keyboard shortcuts* and consist of a sequence of keys that should be pressed in the order they are listed. The shortcut "⌘-O" means to hold down the Command key, press the "O" key, and then release the Command key.



When you see a paragraph with a large "i" to its left, you will know that the information provided will enhance your understanding of REALbasic.



When you see a paragraph with a big exclamation point to its left, you should pay careful attention to the paragraph contents. This style of paragraph is used to give you warning messages, or essential information.



A paragraph with an icon to its left like this lets you know that a series of instructional steps follows:

1. This is a sample step.
2. This is a second sample step in this set of instructions.

3. Hoping not to be left out, the third step is included with the other two steps.

Some steps ask you to enter lines of code into the REALbasic Code Editor. They appear in Courier (a monospaced font), like this:

```
If TextHasChanged then
  SaveChanges.ShowModal //display dialog & wait for input
  Select Case SaveChanges.ButtonPressed
    case "Don't Save"
    case "Cancel"
      Return True //cancel the quit
    case "Save" //call SaveFile to save the document
      TextWindow(Window(1)).SaveFile Window(1).Title, False
  End Select
  SaveChanges.Close //close the dialog
end if
```

When you enter code, please observe these guidelines:

- Type each printed line on a separate line in the Code Editor. Don't try to fit two or more printed lines into the same line or split a long line into two or more lines.
- Don't add extra spaces where no spaces are indicated in the printed code.

Whenever you run your application, REALbasic first checks your code for syntax errors as described in the section, "Automatic Debugging Features" on page 106. Syntax checking will direct your attention to the line of code that is causing problems. Check the line against the printed line. Also, if you have trouble getting your code to work, you can always open the lesson file for that chapter (described in the next section) and paste the corresponding code into your project.

## Lesson Files

REALbasic files for each completed chapter are included on the CD in the folder “Tutorial Files” inside of your “REALbasic Folder.” You can compare your work at different stages of this tutorial with that given in the provided files. You can also start a new chapter using the completed file from the previous chapter.

Since completed REALbasic project files are provided for each chapter, you can skip over a chapter if you get stuck. Later, you can easily return a particular chapter to revisit the material.

## On Your Mark, Get Set, Go!

You are now ready to begin learning REALbasic!



# Creating Windows

---

In this chapter you will be introduced to REALbasic and its Design Environment. You will learn how to:

- Start Up REALbasic
- Identify REALbasic's windows
- Build a document window that will hold the text of your text editor
- Run your application

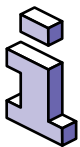
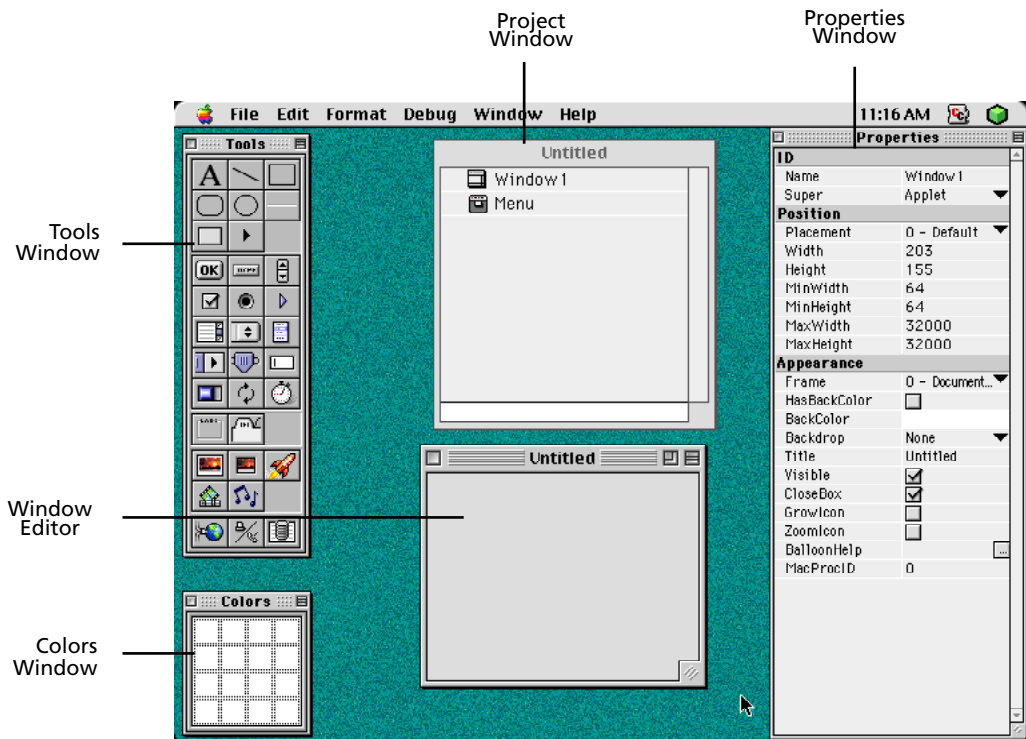
## Starting Up REALbasic



Locate the REALbasic application icon on your computer desktop (it's in the folder in which you installed REALbasic), and double-click it to start up REALbasic.

After REALbasic has started up, your screen should look like Figure 1:

**FIGURE 1. The REALbasic Design Environment.**



The screen shown in Figure 1 is from a computer running MacOS8. If you are running Mac OS 7.x, your screen may look slightly different.

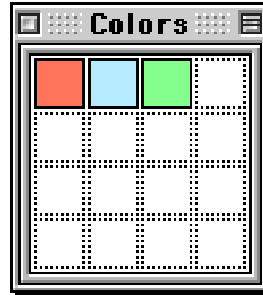
## REALbasic's Windows

As you can see in Figure 1 on page 14, there are five windows that open when you start up REALbasic:

- The *Project Window* contains a list of all of the parts that make up your REALbasic application.
- The *Window Editor* is the generic term for a window in the Design Environment. For a specific window, you will find its name listed in the *Name* property in the Properties Window.
- The *Tools Window* contains icons representing interface objects that you can drag and drop onto the Window Editor. Interface objects are referred to as *controls* in REALbasic.
- The *Properties Window* contains the list of the names of properties and their values for the *currently selected* object in your application. When you select a different object, the Properties Window changes to show the properties of that object. If no object is selected, the Properties Window is empty.
- The *Colors Window* is used to store colors that you have defined for use in your REALbasic application. It consists of a palette of up to 16 colors. You can use the Colors Window to assign a color to a property that accepts a color. To assign a color to a palette element, click it to display the Apple Color Picker. To assign a color to an object property, drag a color from the Colors Window to a property value that accepts a color (such as the BackColor property of a Window object).

The Colors Window with three colors is shown in Figure 2 on page 16.

FIGURE 2. The Colors Window with three colors assigned.



In addition, you can open the following window:

- The *Online Reference Window* contains the REALbasic Language Reference (Choose Window ► Reference to display the online reference). Use it as a convenient alternative to the printed or electronic (PDF) version of the *Language Reference*.

You will learn more about the features of REALbasic as you progress through the Tutorial.

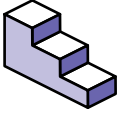
## Building a Document Window

Now that the introductions between you and REALbasic are over, you can start building your own application!

When you start REALbasic, it opens an untitled window in a Window Editor with the name *Window1*. The name of the window is listed in the Project Window and its properties are shown in the Properties Window. This is as shown in Figure 1 on page 14.

Since this will be the window that contains the text editor, we will first give it a more meaningful name.





To rename Window1, do this:

1. Click on Window1's name in the Project Window.

The Properties Window now shows Window1's current properties.

2. Change Window1's Name property to **TextWindow** and press the Return key.

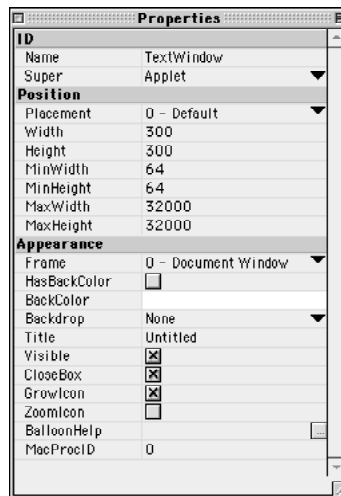
When you press Return, Window1's name in the Project Window changes to TextWindow.

Next, we need to tell REALbasic to add a standard Grow Box to TextWindow so the user can resize the window.

3. In TextWindow's Properties Window, check the GrowIcon property.

The Properties Window should now look like Figure 3.

**FIGURE 3. TextWindow's Properties Window.**

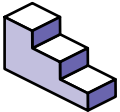
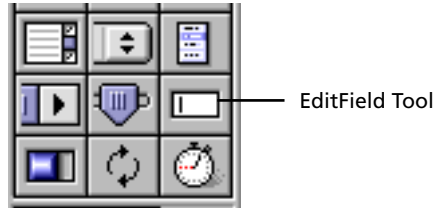


## Adding an EditField

In order to make TextWindow capable of handling text, we'll use an interface object called an *EditField* control. This is the interface object that accepts text input from the end-user. The EditField tool in the Tools window is shown in Figure 4 on page 18.

FIGURE 4. The EditField Tool in the Tools Window.

---



To add an EditField to TextWindow, do this:

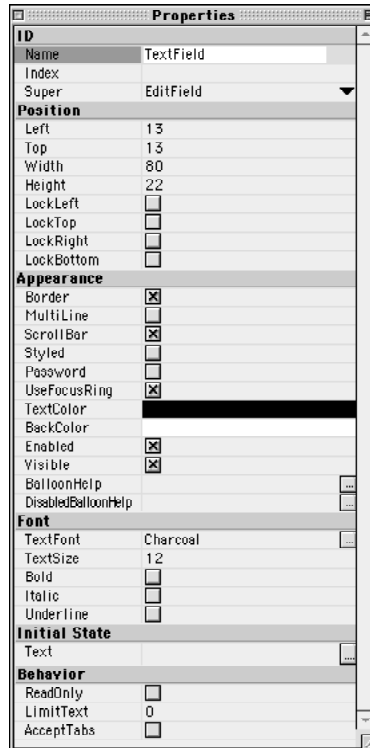
1. If TextWindow isn't already open in the Window Editor, double-click its name in the Project Window to open it.
2. Locate the EditField Tool in the Tools Window and drag it onto TextWindow.

Since the EditField is the currently selected object, the Properties Window now shows its properties. You have just created an *instance* of the EditField class; the instance inherits all the properties of its class.

3. Use the Properties Window to change the Name property of the EditField from EditField1 to **TextField**.

After renaming the EditField, its Properties Window should look like Figure 5 on page 19.

FIGURE 5. The Properties Window after renaming EditField1.

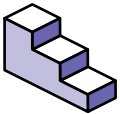


Although you have just started building your application, you may want to run it now, just to see what happens.

To run your application, do this:

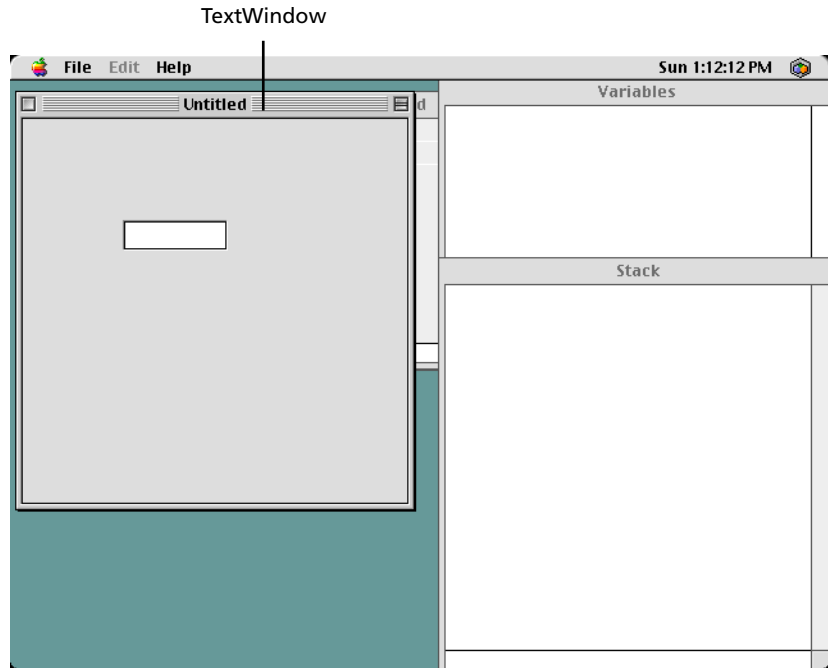
1. Choose Debug ► Run (⌘-R).

TextWindow appears and should look similar to that shown in Figure 6 on page 20.

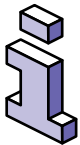


**FIGURE 6. First run of your application.**

---



2. Type something in the TextField to try it out.
3. After you are done exploring, choose File ► Quit (⌘-Q) to return to the Design Environment.



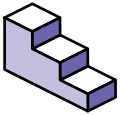
When you choose Run to launch your application, REALbasic automatically switches you to the *Runtime Environment*. The Runtime Environment is where you do “test runs” and debug your application. After you quit your application, you return to the Design Environment.

## Configuring TextField as a Text Editor

The TextField that you just created is, obviously, not an adequate text editor. In a text editor, the user can type as much text as he wishes. Also, the text editing area must be the same size as its window. Right now, the TextField you placed inside of TextWindow can handle only a small amount of text, all of which is on one line. To make a usable text editor, the TextField must have a scrollbar and accept many lines of text. In this section, you configure TextField so that it functions as a text editor and resize it so that its size matches its window.

In the first series of steps, you will use the Properties Window to fix the left and top sides of TextField in the top-left corner of TextWindow.

To resize TextField so that it will handle multiple lines of text, do this:



1. Click TextField to select it, if it isn't already selected.
2. Refer to the Properties Window and locate the Top and Left properties. Click to the right of the value corresponding to the Left property and type -1. Press the Return key to set the property value.

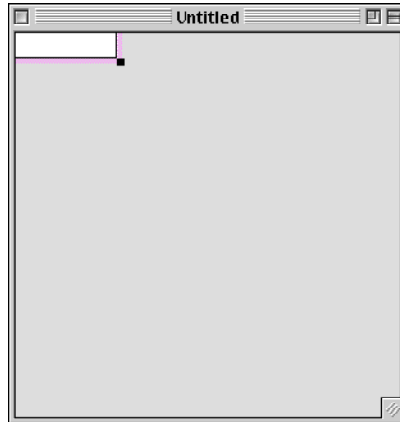
Notice how TextField moves to the left side of TextWindow. The value of -1 places the left edge of the TextField just outside the border of TextWindow.

3. Repeat step 2 for the Top property. Change its value to -1.

The TextField is now aligned with the top of TextWindow and should look like that shown in Figure 7 on page 22.

**FIGURE 7. Document Window with moved TextField.**

---

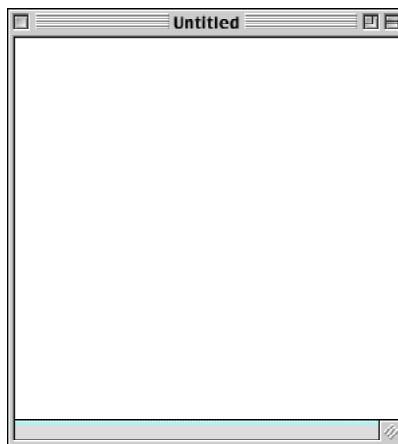


4. Now, drag the exposed (lower right) resizing handle of the TextField until it is close to the resizing handle of TextWindow.
5. To align the right side of the TextField with the right side of TextWindow, drag the resizing handle of TextWindow.

The window should now look as shown in Figure 8:

**FIGURE 8. The TextWindow after resizing.**

---

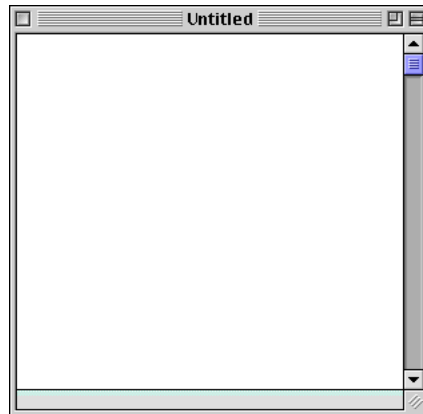


Next, you must tell REALbasic that you want TextField to accept as many lines of text as the user enters, display a scrollbar, and wrap text whenever a line of text reaches the right side of the TextField. This is done simply by assigning the MultiLine property to the TextField.

6. Select the MultiLine property of the TextField using the checkbox in the Properties Window.

The TextField now has a scrollbar. The Document Window should look similar to that shown in Figure 9.

**FIGURE 9. Document Window after adding the MultiLine property.**



To run the application again, do this:

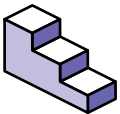
1. Choose Debug ► Run (⌘-R).

The text editing window appears.

2. Enter several lines of text.

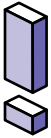
As you type you will notice that your lines wrap as they reach the end of the line. After you type enough lines to fill the window, the scrollbar will become active. You can use the scrollbar to get back to the top.

3. When you're done typing, choose File ► Quit (⌘-Q) to return to the Design Environment.



4. You should save your application project now. Choose File ► Save (⌘-S). Save your project file with the name **MySimpleText-ch2**.

The title of the Project Window is now “MySimpleText-ch2”.



In fact, it's always a good idea to save your project before running it. Doing so will help you avoid losing important work in the event that something unexpected happens while you are testing your application.

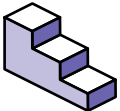
Lastly, you must configure TextField so that it remains the same size as its window when the user resizes the window using the window's Grow box. Unless you do this, the TextField and the Window will be the same size only if the user never resizes the window. This is not realistic.

REALbasic provides a very simple way to accomplish this. The Lock properties are used to fix the distance between the edge of the window and the edge of the control. The distance between edges is maintained during resizing if the corresponding Lock property is selected.

To lock the size of the TextField to its window, do this:

1. In the Design environment, resize TextWindow to make it larger. The TextField remains in place.
2. Adjust TextWindow so that it is aligned once again with the TextField.

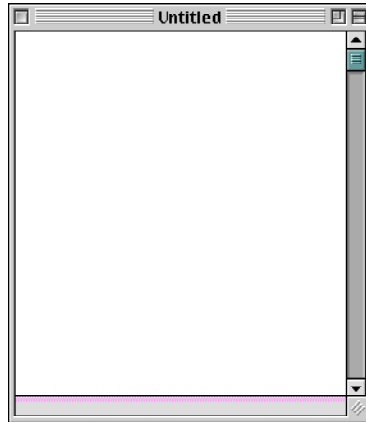
TextWindow should look like that shown in Figure 10 on page 25.





**FIGURE 10. TextField after final adjustments.**

---



3. Select the TextField. Locate the LockLeft, LockTop, LockRight, and LockBottom properties in the Properties Window and select them using their checkboxes.
4. Now, resize TextWindow.  
The TextField resizes as TextWindow resizes.
5. Run your application to test the resizing feature.
6. Choose File ► Quit (⌘-Q) to return to the Design environment and save your project once again.

At this point, you have created a very useful REALbasic object, TextWindow. TextWindow includes a TextField—an object derived from the EditField class that is configured for text editing. TextField inherits all the properties and methods of the EditField class. It is configured to accept multiple lines of text, has a vertical scrollbar, and is locked to its parent window. When you create another instance of TextWindow, you get all the properties of TextField “for free.” You’re going to do this later on in Chapter 4 when you create a New item in the File menu.

## Review

In this chapter you learned how to start up REALbasic, identify the windows of the Design Environment, add a multiline EditField to a document window, lock it to its window, and run your application.

---

***To Learn More About:***

REALbasic Design Environment

REALbasic commands and language

---

***Go to:***

*REALbasic Developer's Guide*

*REALbasic Language Reference*

# Creating Menu Items

---

In this chapter you will work with menus in REALbasic. You will learn how to:

- Add a menu item to your application
- Activate a menu item

You can continue working from the application you began in Chapter 2 or open the application “MySimpleText-ch2” in the Tutorial Files folder on your CD-ROM.

## Adding a Select All Menu Item

In this exercise, you will add a Select All menu item to the Edit menu. There are three required steps for implementing a menu item:

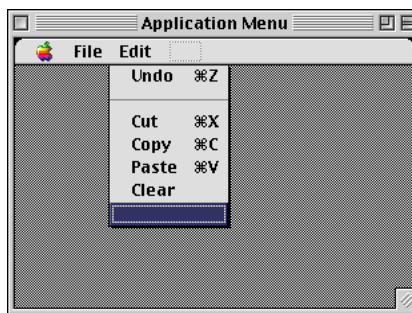
- Adding the menu item itself using the Menu Editor,
- Enabling the menu item. By default, menu items are disabled unless you add code to enable them.
- Adding a *menu handler* that tells REALbasic what to do when the user selects the (enabled) menu item. The menu handler can call other methods.

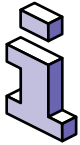
### Adding the Menu Item

To add a Select All item to the Edit menu, do this:

1. Bring the Project Window to the front by choosing Window ► Project (⌘-0) (Command-Zero).
2. Double-click the Menu object icon to open the Menu Editor.
3. Select the Edit menu in the Menu object window.
4. Select the blank menu item at the end of the list, as shown in Figure 11.

**FIGURE 11. The blank menu item selected.**





In the Menu Editor, there is always a blank menu item on each menu. You use it to create new menu items; it is not really a part of the menu and does not appear when you run your application.

If you add a menu item by mistake, you can remove it by selecting it and pressing Delete.

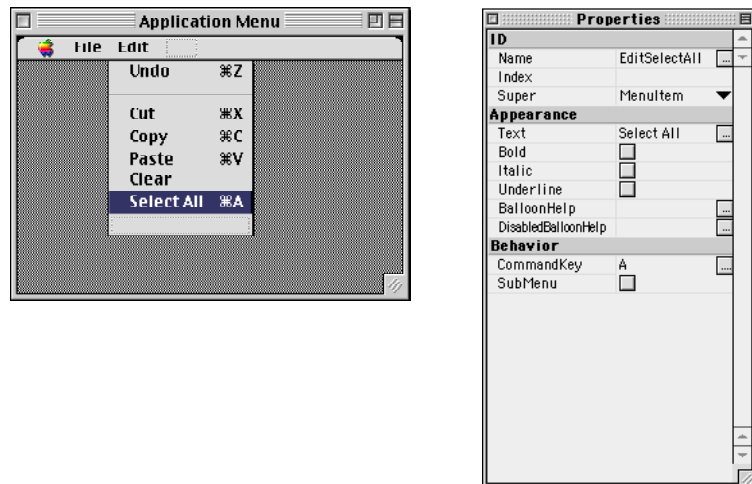
5. In the Properties Window, enter **Select All** in the Text property area and press Return.

The Name property is automatically filled in as EditSelectAll.

6. Assign the letter **A** to the Command Key property and press Return.
7. Your Menu Editor and Properties window should look like those shown in Figure 12.

**FIGURE 12. Menu Editor and Properties Window after adding the Select All menu item.**

---



8. Close the Menu Editor.

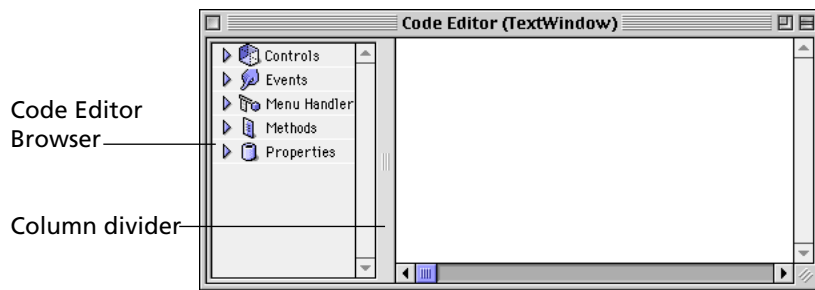
## Enabling the Menu Item

Since the Select All menu item applies to the TextField in TextWindow, we will add the code to enable the menu item to TextWindow. You will simply set the Enabled property of the menu item to True. This enables the Select All menu item whenever the user clicks on the Edit menu to display its items.

To enable the Select All menu item, do this:

1. Select TextWindow in the Project Window and press Option-Tab to open its Code Editor.

FIGURE 13. TextWindow's Code Editor.



Using the Browser pane of the Code Editor, you can expand items to associate code with various events or objects. You can also expand or reduce the size of the Browser area by dragging the Column Divider to the left or right.

2. In the Browser, click on the disclosure triangle to the left of the Events icon and select the EnableMenuItems item.

This is the event handler that runs whenever a user is just about to display the menu items in a menu

3. Type the following code in the Code Editor:

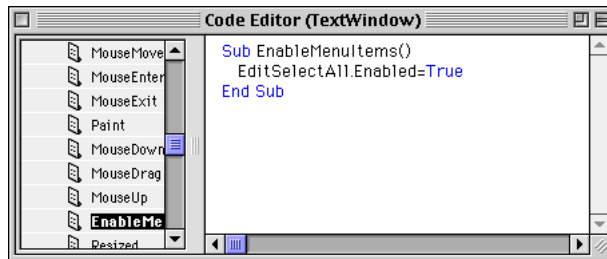
```
EditSelectAll.Enabled=True
```

When you assign a value to a property of an object, you use the syntax: *objectname.propertyname = value*. "EditSelectAll" is an object

in the MenuItem class which has the Enabled property. The Enabled property is a boolean—it can only be assigned the values of True or False.

Your TextWindow Code Editor should look like that shown in Figure 14.

**FIGURE 14. Select All Enabled in TextWindow.**



4. Save your project as MySimpleText-ch3.

## Handling the Menu Item

The last thing we need to do to make the Select All menu item functional is to add a menu handler and enter some code to perform the text selection. The menu handler runs automatically when the user actually chooses the Select All menu item.

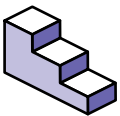
To add a menu handler for the Select All menu item, do this:

1. With the Code Editor for TextWindow as the frontmost window, choose Edit ► New Menu Handler...

A New Menu Handler dialog box appears.

2. Choose EditSelectAll from the pop-up menu and click OK.

A new method called EditSelectAll appears in the TextWindow Code Editor in the Menu Handlers category, and the Code Editor displays the function declaration.



3. Type the following:

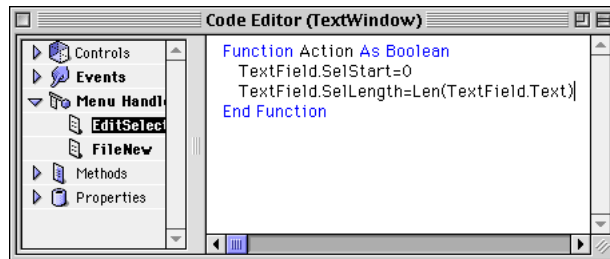
```
TextField.SelStart=0  
TextField.SelLength=Len(TextField.Text)
```

This code uses two properties of an EditField object, `SelStart` and `SelLength`, to determine which text to select. The `SelStart` property sets the position of the first highlighted character and `SelLength` sets the length of the highlighted text, beginning at `SelStart`. The `Len` function is a global function that returns the length of the string passed to it. In this case, it is passed all the text in `TextField`.

This code sets the start of the selection at the beginning of the text and sets the length of the selection to the length of the text in the `TextField`.

The TextWindow Code Editor looks like that shown in Figure 15.

**FIGURE 15. EditSelectAll menu handler with code entered.**



4. Save your project. Be sure to name the project "MySimpleText-ch3" if you haven't already.
5. Run your application.

If you have any trouble compiling your application, check to see that you have renamed the `EditField` to `TextField`. If the control hasn't been renamed, REALbasic won't recognize references to `TextField`'s properties.

6. Type some text into the text editor and try out the Select All menu item.

You should be able to select text using either the keyboard equivalent or the menu item, as shown in Figure 16 on page 33.



**FIGURE 16.** Text selected using the Select All command.

---



7. When you're done, choose **File ► Quit (⌘-Q)** to quit your application and return to the Design Environment.

## Review

In this chapter you learned how to add menu items to your application, to enable them, and to handle their events.

---

**To Learn More About:**

REALbasic Menus

REALbasic commands and language

**Go to:**

*REALbasic Developer's Guide: Chapters 3, 7.*

*REALbasic Language Reference*



# Working with Documents

---

In this chapter you will work with documents in REALbasic. You will learn how to:

- Create menu items for creating, opening, and saving documents
- Add code to your application to implement the menu items
- Add a 'Save changes' dialog box that is displayed when the user closes a document window or tries to quit the application with unsaved changes

## Getting Started

Locate the REALbasic project file that you saved at the end of last chapter ("MySimpleText-ch3"). Launch REALbasic and open the project file. If you need to, you may use the file "MySimpleText-ch3" that is located in the "Ch 3 Files" folder.

## Working with Text Documents

A text editor must be able to create, open, and save text documents. You will first add the ability to create new text documents. As you learned in the previous chapter, implementing a menu item involves three basic steps:

- Adding the menu item to a menu
- Enabling the menu item
- Adding a menu handler

### Creating the New Menu Item

To create the New menu item, do this:

1. Double-click the Menu item in the Project window and select the blank menu item in the File menu.
2. In the Properties Window, enter **New** in the Text area and **N** in the CommandKey property area.  
REALbasic automatically assigns the Name FileNew.
3. In the Menu Editor, drag the New menu item to the top of the menu.

Your Menu Editor should look similar to that shown in Figure 17. When you select the New menu item, the Properties window for that Item should look as shown in Figure 17.

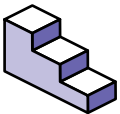
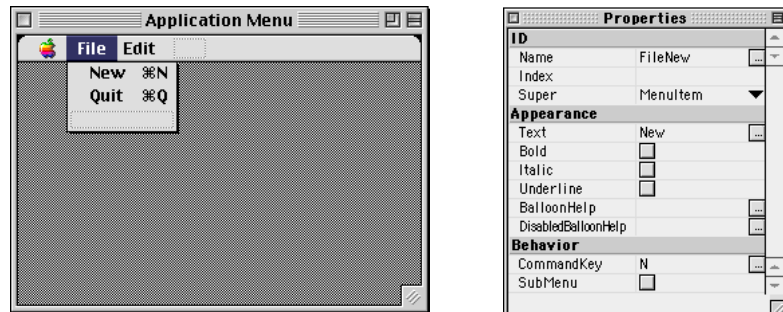


FIGURE 17. Updated File menu.



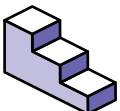
## Enabling the New Menu Item

The New menu item should be available for use whenever the Text Editor application is running — not just when a TextWindow is open. If you enabled the New menu item in the EnableMenuItems event handler for TextWindow (as you did for the Show All menu item), the New menu item would be available only when a document window is open. But suppose a user closes a document and then wants to create a new one. He can't do it! This is no good.

To enable the New menu item at all times, you need to make this a characteristic of the application as a whole, not just a window. You can do this by making it a property of a built-in class in REALbasic called the Application class.

To enable the New menu item, do this:

1. With the Project Window in the front, choose **File ► New Class**.  
REALbasic adds a class to the project called Class1.
2. Using Class1's Properties Window, change its name to **App**.
3. Using the Properties Window, change its Super property to Application using the Super pop-up menu.



The App class now belongs to the application as a whole, not any particular window. Its Code Editor window opens automatically.

4. In the Code Editor for the App class, expand the Events item.
5. Highlight the EnableMenuItems event handler and enter the following code:

```
FileNew.Enabled=True
```

Since FileNew is an object derived from the MenuItem class, it inherits all its properties and methods. This line of code sets the Enabled property of the menu item to True.

## Handling the New Menu Item

After you add and enable a menu item, you need to add a menu handler that tells REALbasic what to do when a user chooses the item. Without the menu handler, the menu item would do nothing.

The menu handler for the New menu item creates a new object of type TextWindow. That is, it is an *instance* of the class of objects that has all the properties of TextWindow that you set in Chapter 2. TextWindow is a window that already has a TextField object in it that is properly configured to work as a text editor.

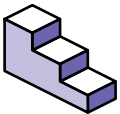
To handle the New menu item, do this:

1. With the Code Editor for App in front, choose Edit ► New Menu Handler... to create a menu handler for the New menu item.

If the Code Editor for the Application class is not in front, highlight it in the Project Window and press Option-Tab.

2. Select the FileNew menu handler and click OK.
3. Enter the following code into the Code Editor for the menu handler:

```
Dim w as TextWindow  
w=New TextWindow
```

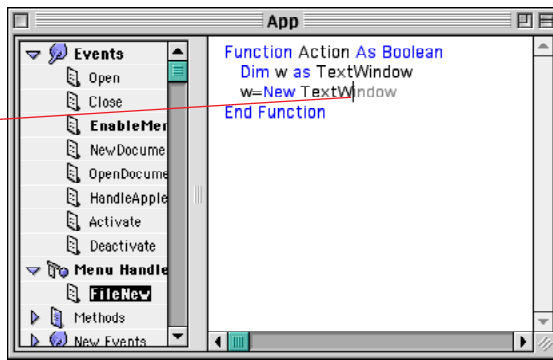


The Dim statement creates a new object of type TextWindow but does not create the instance. That is done in the next line with the New operator. It actually *instantiates* the class and returns the new instance of that class in w. The new instance, w, is a clone of TextWindow and is displayed immediately because the Visible property of TextWindow was set to True in Chapter 2.

You may have noticed that the Code Editor sometimes tries to guess what you are typing. If it recognizes a word, it tries to complete the word using translucent text, as shown in Figure 18.

FIGURE 18. Clairvoyance in REALbasic.

The Code Editor recognizes 'TextWindow' as a data type



If REALbasic's guess is correct, press the Tab key to enter the word without typing the remaining characters. If the guess is incorrect, just keep typing.

4. Save your project and then switch to the Runtime environment to test the New menu item.

As you can see, the New menu item creates a clone of the original default text window whose properties you specified earlier in the tutorial. Try it out even if no text window is open.

When you are finished, choose File ► Quit to return to the Design environment.

## **File Types**

In order for your application to recognize specific types of files, you can define the valid file types for the application. By default, it recognizes files of type 'text'. This is fine for this application, but you may want another application you write to recognize other common file types. For example, if you are writing a graphics application, you will need to tell REALbasic that it needs to open files of type PICT, TIFF, and so forth. Also, if your application saves documents in its own format, REALbasic must know that file type as well.

You use the File Types dialog box to specify valid file types. You display the File Types dialog box by choosing Edit ► File Types.... However, since REALbasic recognizes files of type text by default, you do not need to make any changes.

## **Saving Documents**

Since saving documents occurs only when a document is already open, we will let the text editor window manage these tasks. This means that you will add a Save menu item to the TextWindow window. In this section, you implement the Save menu item. Activating the Save menu item involves these operations:

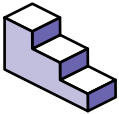
- Enabling the item. The Save menu item should be enabled only when the contents of the current window have been changed—not all the time, as is the case for the New and Select All menu items.
- Creating a menu handler. The menu handler tells the application what to do when the user chooses the Save menu item. The menu handler that you will add calls a generic save-file method that actually accomplishes the save.



- Adding a save-file method. The save-file method is called by the menu handler. It uses a *FolderItem* object to manage saving the contents of the window to a text file on disk.

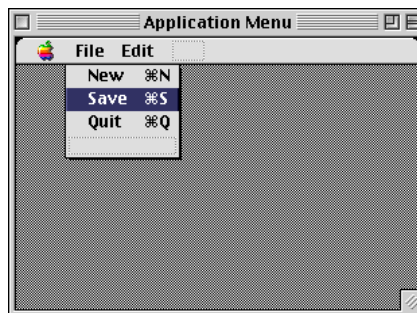
## Adding the Save Menu Item

To add the Save menu item, do this:



1. Open the Menu Editor in the Project Window.
2. Select the blank menu item in the File menu and use the Properties window to assign it the Text **Save** and CommandKey **S**.  
REALbasic automatically assigns the name `FileSave`.
3. Drag the Save menu item between the New and Quit items.  
The Menu Editor should now look like that shown in Figure 19.

FIGURE 19. Updated Menu Editor.

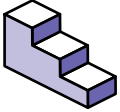


4. Close the Menu Editor and save your project.

## Adding Properties to TextWindow

When we open a file in our application, we will need to keep track of the filename so that we can save changes. In order to do this, we will define a new property for `TextWindow`. It makes sense to add the property to `TextWindow`, since each copy of `TextWindow` that is opened in the application is associated with a

specific file. We will also add a property to keep track of changes to the text in TextWindow.



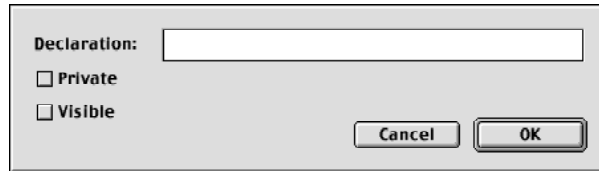
To add the properties to TextWindow, do this:

1. Select TextWindow in the Project Window.
2. Press Option-Tab to open TextWindow's Code Editor.
3. Choose Edit ► New Property...

The Property Declaration dialog box appears.

**FIGURE 20. The Property Declaration dialog box.**

---



4. Enter **Document as FolderItem** and then click OK (a *FolderItem* is the name of the REALbasic object that refers to files and folders).

5. Choose Edit ► New Property...

The Property Declaration dialog appears.

6. Enter **TextHasChanged as Boolean** and then click OK (Boolean is a data type which can take two values: *True* or *False*). In the section "Managing the TextHasChanged Property" on page 48 you will use this property to keep track of changes to the contents of the TextField.

7. Click the disclosure triangle next to the Properties category label in the Code Editor for TextWindow.

The Document and TextHasChanged properties are now listed. The Properties category label itself is now in boldface, indicating that properties have been added.

## Enabling the Menu Item

Since we want the Save menu item to be enabled only if there are unsaved changes to the document, the code will use the `TextHasChanged` property that you just added. The `TextHasChanged` property will function as a flag to let REALbasic know when the user has changed the contents of the `TextField` in `TextWindow`.

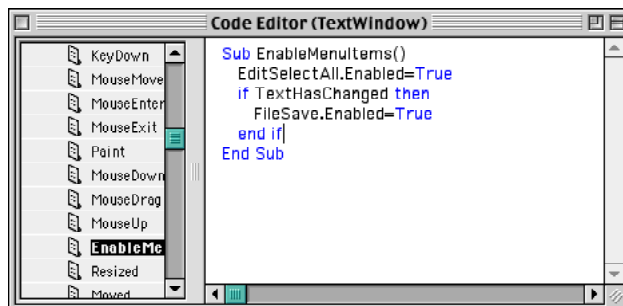
To enable the menu item, do this:

1. If the Code Editor for the `TextWindow` window is not already open, select `TextWindow` from the Project Window and open `TextWindow`'s Code Editor (Option-Tab).
2. Click the disclosure triangle next to the Events category label to reveal the events (or double-click the Events label).
3. Select `EnableMenuItems` and add the following code to the end of the method:

```
If TextHasChanged Then
    FileSave.Enabled=True
End if
```

The Code Editor should look like that shown in Figure 21.

**FIGURE 21. Updated EnableMenuItems Method.**



4. Save your project.

## Adding a SaveFile Method

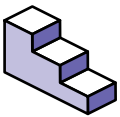
Next, you need to specify the actions to be taken when the Save menu item is chosen by the user. This is done in the SaveFile method. This method will be called by the menu handler for the Save menu item as well as the menu handler for the Save As menu item that you will add in the section “Adding a Save As Menu Item” on page 51. This method will manage two cases:

- The user chooses Save to save changes to an existing document.
- The user chooses Save or Save As to save an unsaved document or to save an existing document under a new name.

In the latter case, the application must first present a save-file dialog box that lets the user enter a filename. In the former case, the application saves the document using the existing filename.

In this method, the Boolean parameter DisplaySaveDialog is used to force the method to present a save-file dialog box. In this way, the same method can be called to manage either type of save.

To add the SaveFile method, do this:



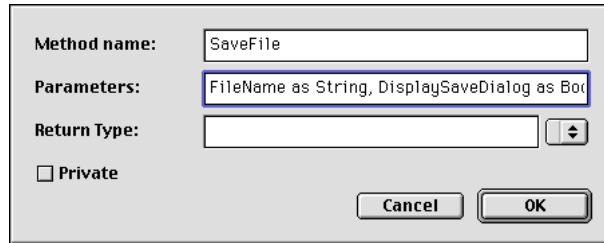
1. With the Code Editor for the TextWindow window in the front, choose Edit ► New Method.

The New Method dialog box appears.

2. Enter **SaveFile** as the method name and **FileName as String, DisplaySaveDialog As Boolean** in the Parameter area.

The dialog box should now look like Figure 22.

FIGURE 22. The New Method dialog box.



3. Click OK to close the dialog box.

The Code Editor for the SaveFile method appears. Note that the method name and parameters have been added. If you need to change the name or parameters, you can double-click the SaveFile method name in the Browser panel of the Code Editor.

In the next step, you will enter the code that will handle the two cases we described. Enter the following code for the SaveFile method into the Code Editor.

```
Dim f as folderitem
If Document = nil or DisplaySaveDialog = True then
    f=GetSaveFolderItem("text",FileName)
    If f <> nil then //if the user clicked Save
        Title=f.Name
        Document=f
    End if
End if
If Document <> nil then
    Document.SaveStyledEditField TextField
    TextHasChanged=False
End if
```

Remember to enter each printed line on a separate line in the Code Editor and do not split a long line into two lines.

If the Document property is undefined (i.e., its value is "Nil"), the document has not been saved, so the Save File dialog box must be presented. The GetSaveFolderItem function does this.

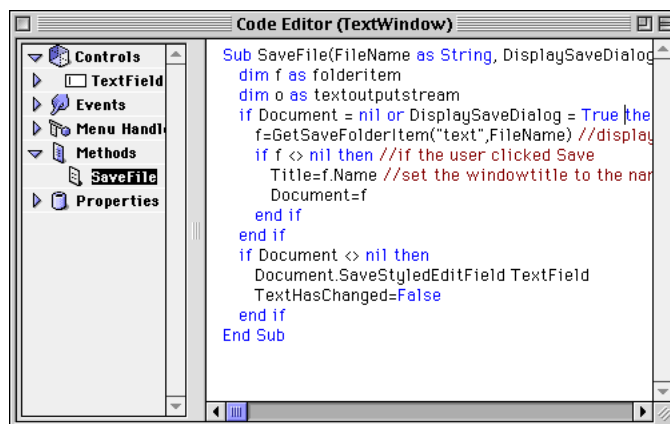
The line "Title=f.Name" sets the Title property of TextWindow to the name of the Name property of the FolderItem (i.e., the document). "Title" is a property of the Window class that TextWindow has

inherited. The next line, "Document=f" sets the Document property of TextWindow to the opened document.

If the document exists (i.e., the FolderItem is not Nil), the Save As dialog box does not have to be presented; the user wants to resave an existing document under its current name. In this case, you use the SaveStyledEditField method of a FolderItem to save the contents of the TextField. "TextField" is the value of the parameter that is passed to the SaveStyledEditField method of the EditField class. We also reset the TextHasChanged Boolean property to False because the document has not changed since its last save.

The Code Editor should now look like this:

FIGURE 23. Code entered for SaveFile method.



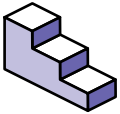
We are not quite ready to call this method because we haven't added the line of code that sets the TextHasChanged property to True when the text of TextField changes. This will be done in the section "Managing the TextHasChanged Property" on page 48.

## Using The Online Reference

The SaveFile method uses two built-in methods to do the hard work: It calls the global method GetSaveFolderItem to present

the save-file dialog box and the `SaveStyledEditField` method of the `FolderItem` class to save the contents of the `EditField` that is part of `TextWindow`. If you wish, you can look up these methods in the REALbasic *Language Reference* or, more conveniently, in the online version of the reference.

To look up `GetSaveFolderItem`, do this:



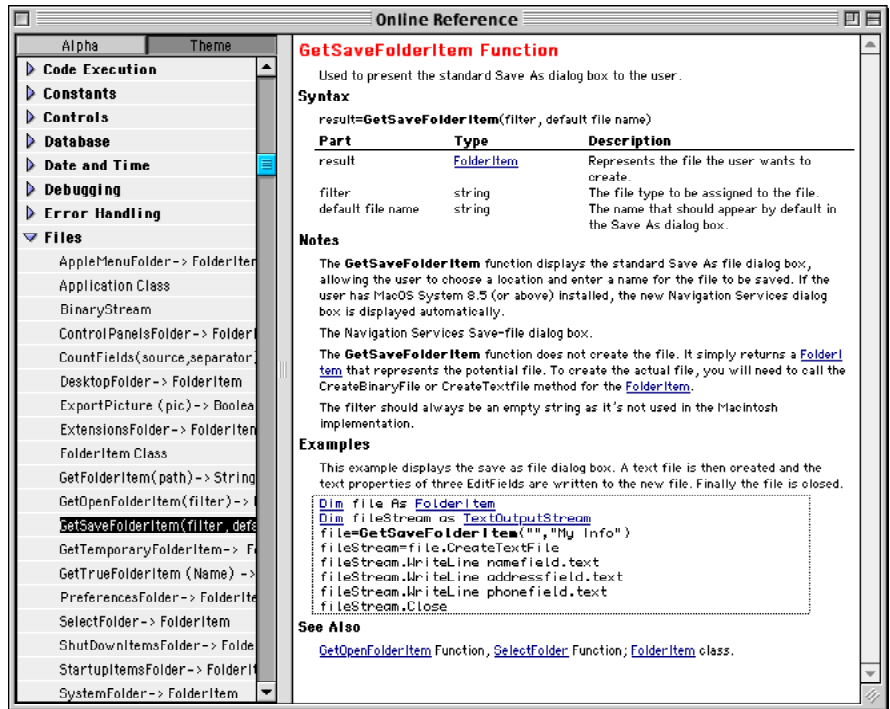
1. Choose **Window ► Reference**.

The online help dialog box appears. The browser on the left lists all the main entries in the *Language Reference*, sorted by theme (category) or alphabetically. The default sort order is by theme, but you can list the items alphabetically by clicking the Alpha tab at the top.

2. Expand the **Files** theme and highlight `GetSaveFolderItem`.

The window should look like Figure 24 on page 48.

FIGURE 24. The GetSaveFolderItem online documentation.



The main panel in Figure 24 presents the documentation for `GetSaveFolderItem`; hypertext links to related entries are in blue and are underlined. If you wish, you can click on a reference to `FolderItem` and then scroll down to read about the `SaveStyledEditField` method in the Methods table.

Also, code examples that are shown in dotted rectangles in the online reference can be dragged into your Code Editor Window.

3. When you are finished, click the close box to put away the online reference.

## Managing the TextHasChanged Property

The `TextHasChanged` property that is used in the `SaveFile` method must be assigned a value of `True` whenever there is a change to

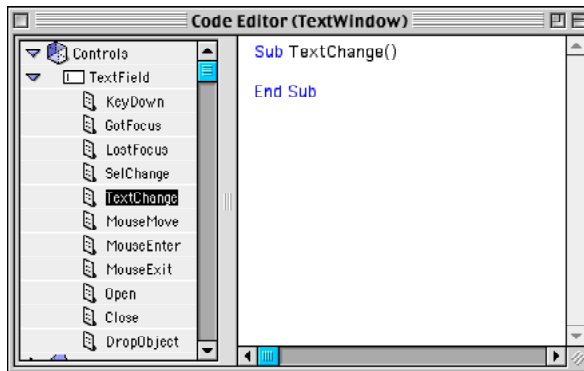


the text in the EditField. This is done in the TextChange event handler of TextField.

An *event handler* is a method that runs automatically when a particular event occurs. Each REALbasic interface object comes equipped with a set of empty event handlers. These are events that REALbasic is capable of detecting automatically. By adding code to an empty event handler, you specify what your application will do when a user interacts with the object in a certain way. This is the basic concept of *event-driven programming*.

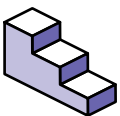
To see which event handlers are available for an EditField, bring the Code Editor for TextWindow to the front and expand the Controls item in the Browser. Then expand the TextField object. You will see a list of an EditField's event handlers. It will look like Figure 25 on page 49.

**FIGURE 25. TextField's Event Handlers.**



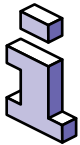
To manage the TextHasChanged flag, do this:

1. Highlight the TextChange event.
2. Add the following line of code to the blank event handler on the right of the divider:



```
TextChanged=True
```

Since you placed this line of code in the TextField's `TextChanged` event handler, it will run whenever there is a change to the text in the TextField. REALbasic has the job of figuring out when the text has changed.

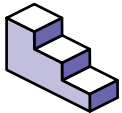


In the online reference, the event handlers that are available for each control are described in the Events table for that control.

## Handling the Menu Item

The menu handler for the Save menu item calls the `SaveFile` method that you just installed and passes the value of `False` to `DisplaySaveDialog` to prevent the method from displaying the save-file dialog box (unless it is an unsaved document).

To handle the menu item, do this:



1. Expand the Menu Handlers category in the Code Editor window to reveal the menu handlers.
2. Choose **Edit ► New Menu Handler**, select `FileSave` from the pop-up menu, and click OK.

A new menu handler named `FileSave` is added to the Code Editor Browser.

3. Enter the following code:

```
SaveFile Self.title, False
```

This menu handler actually calls another method, `SaveFile`, that does the work. The terms that follow this call —`Self.title`, and `False` —are the values of the parameters that are passed to the `SaveFile` method.

Remember that `SaveFile` takes two parameters. The first, a string, is the default name of the file to be saved and the second is a Boolean that tells `SaveFile` whether it needs to display a “save changes” dialog box<sup>1</sup>. The term “Self” in “Self.title” is a global function. It is a reference to the parent window in which the method is running—in this case, `TextWindow`. The term “Title” is the Title property of the

---

1. ...which we haven't created yet!

Window class—the string that appears in the window’s Title bar. You could leave “Self” off and the method would run anyway because it is assumed.

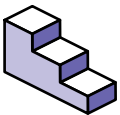
When you run the application and save the document the first time, the default text will be “Untitled,” since that is the default window title.

4. Save your project as MySimpleText-ch4.
5. Choose Debug ► Run (⌘-R) to test your application.  
Notice that the Save menu item is disabled initially.
6. Type some text. Use the Save menu item to save the document.  
Notice that the Save menu item becomes disabled until you modify the text in the text editor.
7. Choose File ► Quit (⌘-Q) to quit your application and return to the Design Environment.

## Adding a Save As Menu Item

A Save As menu item performs the same function as a Save menu item, except that it always presents a save-file dialog box that allows the user to save the existing document under a new name. It is implemented in the same fashion as the Save menu item.

To add the Save As menu item, do this:



1. Double-click the Menu item in the Project Window and add a new menu item to the File menu with the Text **Save As...** and Name **FileSaveAs** (without the three dots).
2. Drag it between the Save and Quit menu items.
3. Open the EnableMenuItems Event in TextWindow’s Code Editor and add the following line of code to the existing method:

```
FileSaveAs.Enabled=True
```

4. From TextWindow’s Code Editor, choose Edit ► New Menu Handler and choose FileSaveAs.

5. Enter the following code:

```
SaveFile Self.Title, True
```

This menu handler also manages the save using the `SaveFile` method but forces the save-file dialog box to be presented.

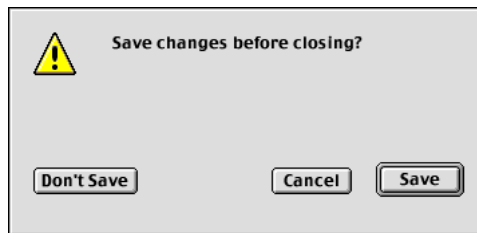
6. Test the Save and Save As commands by choosing Debug ► Run. Enter some text and save it using the Save command. Then try the Save As command. Notice that the Save As command uses the existing window title as the default document name.
7. When you are finished, choose File ► Quit to return to the Design environment.

## Adding a 'Save Changes' Dialog Box

In 'well-behaved' applications, the application gives the user a chance to save changes to all open documents whenever he closes a window with unsaved changes or he chooses the Quit command. This application is no different. In this exercise, you will create, install, and activate the Save Changes dialog box shown in Figure 26.

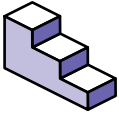
**FIGURE 26. The Save Changes dialog box.**

---



## Creating the Dialog Box

You create the dialog box by adding a new window to the project and adding the icon, button, and text controls to the window. The controls are added to a window by dragging them from the Tools Window (as you added the `EditField` in Chapter 2).



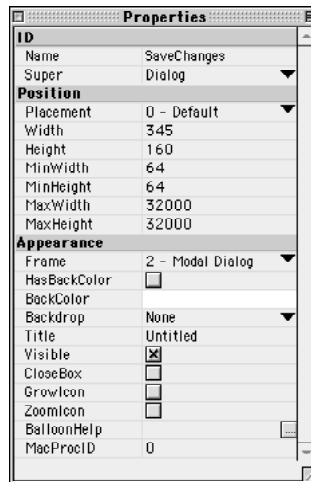
To create the dialog box, do this:

1. With the Project Window as the frontmost window, choose File ► New Window.  
REALbasic adds a window to the project and names it Dialog1.
2. Use Dialog1's Properties Window to change its name to **SaveChanges**.
3. Change the window's Width to **345** and Height to **160**.
4. Change its Frame Property to **Modal Dialog**.
5. Deselect the CloseBox, GrowIcon, and ZoomIcon check boxes.


This window will be used as a fixed-size dialog box.

The window's Properties Window should now look like this:

**FIGURE 27. Properties of the SaveChanges window.**



The following steps add the controls to the empty dialog box.

1. Use the Tools Window to drag a Canvas control  to the top-left area of the SaveChanges window. This control will display the caution icon shown in Figure 26 on page 52.

The Canvas control is a blank drawing canvas. It comes equipped with a set of drawing tools that you use programmatically to customize its appearance.

2. Click on the Canvas control and, using the Properties Window, assign it the properties shown in Table 1.

**TABLE 1. Properties of the Canvas Control**

Property	Value
Left	13
Top	13
Width	46
Height	46

3. Open the Code Editor for SaveChanges by selecting SaveChanges in the Project Window and pressing Option-Tab.
4. Expand the Controls item and then expand the Canvas1 item.

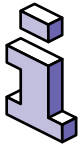
The list of event handlers for a Canvas control appears. To create the caution icon, you will add code to the Paint event handler. This event handler runs whenever REALbasic detects that the Canvas control needs to be redrawn.
5. Highlight the Paint item.

Notice that the parameter line for the Paint event handler contains one parameter — “g as Graphics.” The Graphics class in REALbasic contains the methods that allow you to completely customize the appearance of the Canvas control. They are your drawing tools.

6. Enter the following line of code:

```
g.DrawCautionIcon 0,0
```

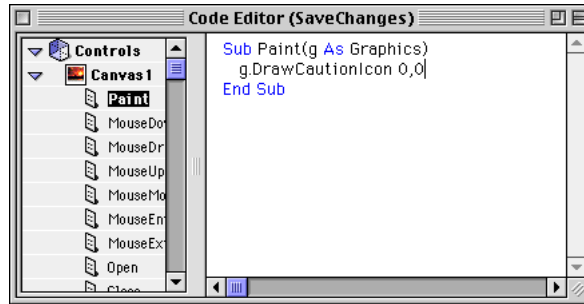
The syntax “g.DrawCautionIcon” indicates that DrawCautionIcon is a method within the Graphics class. It simply calls a built-in method in the Graphics class that draws the icon.



If you wish, check out the drawing tools available in the Graphics class using the online reference. You will see that the DrawCautionIcon method takes two parameters— the x and y coordinates where the top-left corner of the caution icon is to be positioned.

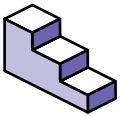
The Code Editor should look like Figure 28 on page 55.

FIGURE 28. Code for the Caution Icon.



Next, you need to add the text that appears to the right of the Caution icon. This is done by placing a StaticText control in the dialog box.

To add static text to the dialog box, do this:




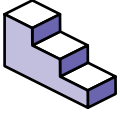
1. Click on the StaticText tool in the Tools Window  and drag it to the right of the Canvas control in the SaveChanges Window. Notice how alignment lines appear and it snaps in alignment with the top of the Canvas control.
2. With the StaticText control selected, assign it the properties shown below.


TABLE 2. Properties of the StaticText Control

Property	Value
Left	72
Top	13
Width	255
Height	20
Text	Save changes before closing?
MultiLine	Checked (Yes)

The next series of instructions adds the buttons that are placed below the Canvas and StaticText controls.



To add buttons to the dialog box, do this:

1. Using the Pushbutton tool  in the Tools Window, drag three pushbuttons into the approximate positions shown in Figure 26 on page 52.

Drag the Don't Save pushbutton first and let REALbasic align it with the left edge of the Canvas control. Then drag the Cancel and Save pushbuttons into place, using the horizontal alignment line to align them with the bottom of the Don't Save button.

Next, you will assign properties to each button by successively selecting each button and changing its properties using the Properties Window. As with the other controls, you use the Properties Window to set the exact position of the object.

2. Select each pushbutton and make the property assignments shown in Table 3.

**TABLE 3. Properties of the Don't Save, Cancel, and Save Pushbuttons.**

Property	Pushbutton		
	Don't Save	Cancel	Save
Name	Don'tSave	Cancel	Save
Left	13	191	272
Top	114	114	114
Width	78	60	60
Height	20	20	20
Caption	Don't Save	Cancel	Save
Default	Not checked	Not checked	Checked
Cancel	Checked	Checked	Not checked
Enabled	Checked	Checked	Checked
Visible	Checked	Checked	Checked

As with all controls, the Name property is the internal name of the object that you use to refer to the object. The Caption property is the label that appears in the Pushbutton.



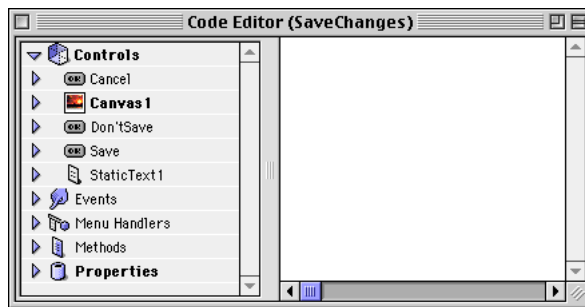
In the next series of steps, you assign a value to a property that identifies the button that is pressed. First you will declare the variable as a property of the SaveChanges window.

1. With the Code Editor for the SaveChanges window in front, choose **Edit ► New Property**.
2. Enter **ButtonPressed as String** in the dialog box and click OK.
3. Expand the Controls item in the Code Editor for the Save Changes window.

The three Pushbutton controls are listed by name, along with the Canvas control, as shown in Figure 29.

**FIGURE 29. Controls in the Save Changes dialog box.**

---



4. In succession, expand each pushbutton control and click the Action item.
5. Add the code for Action shown in Table 4.

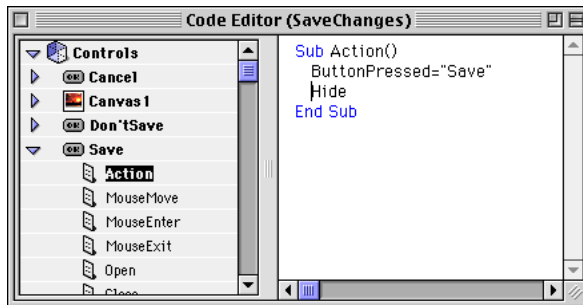
**TABLE 4. Code for Pushbutton Control Actions.**

Control	Code
Cancel	ButtonPressed="Cancel" Hide
Don't Save	ButtonPressed="Don't Save" Hide
Save	ButtonPressed="Save" Hide

The code assigns a string to the `ButtonPressed` property so that we can later determine which button was pressed. The `Hide` statement is a method of the `Window` class and, not surprisingly, makes the window disappear. (You could have equivalently written “`Self.Hide`”, as the parent object — the window — is assumed.)

For example, the Code Editor for the `Save` pushbutton looks as shown in Figure 30.

**FIGURE 30. Code for the Save Pushbutton.**



## Displaying the Save Changes Dialog Box

The final step is to add code that displays the `SaveChanges` dialog box when the user chooses the `Quit` menu item and there are unsaved changes to the contents of any open window. This is done in the `CancelClose` event handler in `TextWindow`.

The `Quit` menu item is different from the menu items that you have added in several ways. First, note that it is an instance of the `QuitMenuItem` class, rather than the `MenuItem` class. You can verify this by highlighting the `Quit` menu item in the `Menu Editor` and checking its properties.

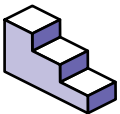
Second, it is enabled by default. You have been using the Quit menu item to return to the Design environment even though you have not enabled it.

Finally, the Quit menu item also has its own menu handler—it calls the built-in Quit method. Not surprisingly, this method tries to quit the application. If any windows are open, it calls each window's CancelClose event handler. This event handler gives you a chance to cancel the quit or perform actions prior to the quit.

If the CancelClose event handler returns False (the default action) then the window's Close event handler will be executed. It means, "Don't cancel the close." If the CancelClose event handler returns True, REALbasic stops sending CancelClose or Close events and the application will not quit.

The CancelClose method that you will write displays the Save Changes dialog box if the TextHasChanged property is True. It then determines which button in the dialog box the user has clicked. Only if the user clicks the Save button is the SaveFile method called.

To add the CancelClose code, do this:



1. In the Code Editor for TextWindow, expand the Events item and highlight the CancelClose event.
2. Enter the following code:

```
If TextHasChanged then
  SaveChanges.ShowModal //display dialog & wait for input
  Select Case SaveChanges.ButtonPressed
    case "Don't Save"
    case "Cancel"
      Return True //cancel the quit
    case "Save" //call SaveFile to save the document
      TextWindow(Window(1)).SaveFile Window(1).Title, False
  End Select
  SaveChanges.Close //close the dialog
```

```
end if
```

The code tests whether the text has changed by testing the value of the `TextChanged` property, and, if it has, it displays the Save Changes dialog box. The `SaveChanges.ShowModal` statement runs the `ShowModal` method of the `Window` class (the Save Changes dialog box is an instance of the `Window` class and, therefore, inherits all its properties and methods). This method stops code execution at this statement until the user clicks one of the three buttons in the dialog.

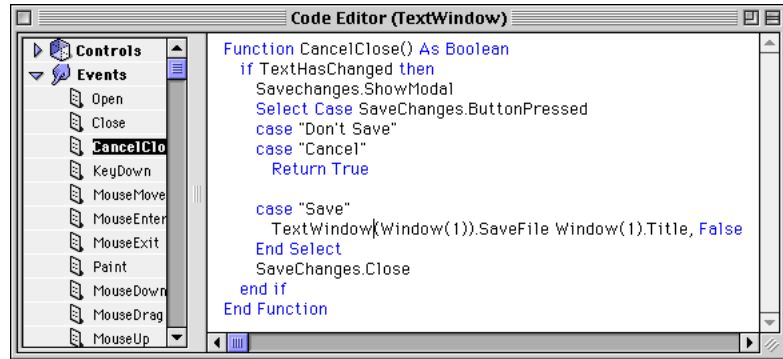
The Select Case structure determines which button the user clicks. If the Don't Save button is clicked, the quit continues without saving the document because `CancelClose` returns `False` and no method for saving the document is called. If the user clicks Cancel, the `CancelClose` event handler returns `True`, cancelling the close/quit. If the user clicks Save, the `SaveFile` method runs. Its parameters are the Title of the window (the `Title` property of the `Window` class contains the title of the window instance) and `False`—telling `SaveFile` not to display the Save Changes dialog box (since we're already in a Save Changes dialog box!). The syntax:

```
TextWindow(Window(1)).SaveFile
```

is a reference to the second window (Window zero is the dialog box). “`Window(1)`” is the `Window` function, which returns a reference to the specified window. Since at least two windows are open at this point, `REALbasic` needs to know which instance of `TextWindow` you are working with at the moment.

### 3. Save your project.

The Code Editor for `CancelClose` should now look like Figure 31 (minus the explanatory comments).

**FIGURE 31. The CancelClose method.**

4. Test the SaveChanges dialog box by switching to the Runtime environment, creating a new document, saving it to disk, modifying its contents, and then quitting to the Design environment.

The Save Changes dialog box should appear when you choose Quit.

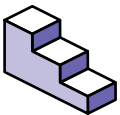
## Adding an Open Menu Item

Now that you have implemented the Save and Save As commands, the user needs to be able to open any of the documents that he has saved. The following exercise implements an Open menu item. You will:

- Add the Open menu item,
- Enable the menu item,
- Write a menu handler for the item.

### Creating the Open Menu Item

To create the Open menu item, do this:



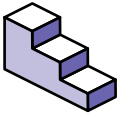
1. Double-click the Menu item in the Project window and select the blank menu item in the File menu.

If the Menu Editor does not open, check to see if the Debug ► Kill menu item is active. If it is, choose it.

2. In the Properties Window, enter **Open...** in the Text property area, **O** in the CommandKey property area, and **FileOpen** in the Name area (without the "...").
3. Drag the Open menu item between the New and Save menu items.

Like the New menu item, the Open menu item should be available even if there are no open documents. Therefore we are going to use the Application object to manage the menu item.

To enable the Open menu item, do this:

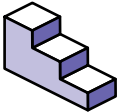


1. Highlight the App object in the Project Window and press Option-Tab to open the Code Editor for App.
2. Expand the Events item and select the EnableMenuItems item.
3. Add the following code to end of the EnableMenuItems method:

```
FileOpen.Enabled=True
```

## Handling the Menu Item

To handle the Open File menu item, do this:



1. Choose Edit ► New Menu Handler... to create a menu handler for Open.
2. Select the FileOpen menu handler in the New Menu Handler dialog and enter the following code into the FileOpen menu handler in the Code Editor:

```
Dim f as FolderItem
Dim w as TextWindow
f=GetOpenFolderItem("text") //displays open-file dialog
If f <> nil then //the user clicked Open
    w=New TextWindow //create new instance of TextWindow
    f.OpenStyledEditField w.TextField
```

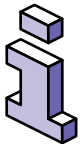
```
w.Document=f //assign f to document property of TextWindow  
w.title=f.Name //assign name of f to title property of w  
End if
```

The first two lines of this method create a new `FolderItem` object—a reference to a document—and a new instance of the `TextWindow` class to display the document. At this point, `f` contains no value, it is just a container that is capable of referring to a document. Similarly, the object “`w`” doesn’t actually refer to a new instance of `TextWindow` until the `New` function creates it.

The method then calls the global method `GetOpenFolderItem` which displays the open-file dialog box and returns a reference to the document that the user selects. The parameter (“`text`”) instructs the `GetOpenFolderItem` method to display only text documents.

If the user successfully opens a text document, the `GetOpenFolderItem` returns a reference to the document in the `FolderItem` object, `f`. The code first tests whether `f` is still `nil`—it would be `nil` if the user cancelled out of the open-file dialog—before creating a new window for the document and calling the `OpenStyledEditField` method of the `FolderItem` class. This method places the text that is now in `f` into the `TextField` belonging to the new instance of `TextWindow`.

The menu handler then sets the `Document` property of `TextWindow` to the `FolderItem` (the document the user selected) and sets the title of the window to the name of the document.



If you are confused about how the various calls to built-in methods work, consult the online reference entries for `FolderItem`, `GetOpenFolderItem`, and the `Window` class.

3. Save your project.
4. Choose **Debug ► Run** and try out the **Open** and **Save** commands.

5. When you are finished, quit the application to return to the Design environment.

## Review

In this chapter you learned how to add the capability to open, create, close, and save documents in your application.

---

***To Learn More About:***

REALbasic Files

REALbasic commands and language

---

***Go to:***

*REALbasic Developer's Guide: Chapters 6, 7, 8.*

*REALbasic Language Reference*



# Working with Text

---

In this chapter you will work with the styled text features in REALbasic. You will implement Style and Size menus. These menus allow the user to apply a different style and font size to selected text. Your code will also place check marks next to the currently selected style and font size so that the user knows the current settings.

## Getting Started

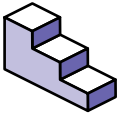
Locate the REALbasic project file that you saved at the end of last chapter ("MySimpleText-ch4"). Launch REALbasic and open the project file. If you need to, you may use the file "MySimpleText-ch4" that is located in the "Ch 4 Files" folder.

## Configuring TextField for Styled Text

Before implementing the Style and Size menus, you need to tell REALbasic to allow the TextField to accept multiple font styles and sizes. You do this by setting the Styled property of the TextField.

To configure the TextField for styled text, do this:

1. Double-click the TextWindow item in the Project Window.  
The window opens in a Window Editor. Its properties are now displayed in the Properties Window.
2. Click on the TextField in the window and use the Properties Window to set the Styled property.
3. Close the Window Editor.



## Implementing the Style Menu

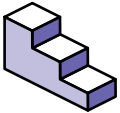
To change the style of document text, you need to add a Style menu to using the Menu Editor. Your Style menu will contain menu items for Plain, Bold, Italic, and Underline. You could also add the Outline, Shadowed, Condensed, and Extended styles using exactly the same process that is explained in this chapter. The latter styles are supported only on applications built for the

Macintosh platform. In the tutorial, you will implement only the basic four styles to avoid redundant steps.

You will first use the Menu Editor to add the new menu and menu items. You will then enable the menu items, install code that places a check mark next to the currently selected style and font size, and write the menu handlers for the Style menu items.

### Creating the Style Menu and its Menu Items

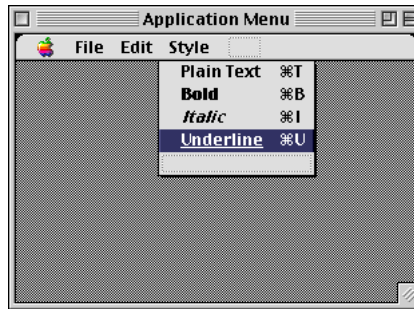
To create the Style menu, do this:



1. Double-click the Menu item in the Project Window.
2. Select the blank item in the menu bar.
3. Use the Property Window to set its Text property to **Style**.  
The Style menu now appears in the menu bar.
4. Select the blank menu item in the Style menu and use the Properties Window to set its Text to **Plain Text** and CommandKey to **T**.
5. Add a new menu item with Text **Bold** and CommandKey **B**. Select the Bold property so the menu item appears in bold.
6. Add a new menu item with Text **Italic**, and CommandKey **I**. Select the Italic property so the menu item appears in italic.
7. Add a new menu item with Text **Underline**, and CommandKey **U**. Select the Underline property so the menu item is underlined.

Your Style menu should look like that shown in Figure 32.

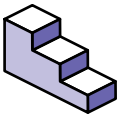
FIGURE 32. Newly created Style menu.



8. Close the Menu Editor and save your project as “MySimpleText-ch5”.

## Enabling the Style Menu Items

Next, you need to enable the new menu items. You enable these menu items by adding code to the EnableMenuItems event handler in the TextWindow Code Editor.



To enable the menu items, do this:

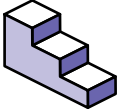
1. Select TextWindow in the Project Window and open its Code Editor by typing Option-Tab.
2. Expand the Events item in the Browser and click the EnableMenuItems event handler.
3. Add the following code to end of the method:

```
// Enable Style menu items
StylePlainText.Enabled=True
StyleBold.Enabled=True
StyleItalic.Enabled=True
StyleUnderline.Enabled=True
```

## Handling the Style Menu Items

As you would expect, you need to add a menu handler for each new menu item. The menu handlers that you will add will use a

method, `SetPlainStyleMenu`, which is used several times to manage the Plain Text menu item. It is convenient to add this method first.



To create `SetPlainStyleMenu`, do this:

1. Open the Code Editor for `TextWindow`.
2. Choose **Edit ► New Method...**  
The New Method dialog appears.
3. Enter **`SetPlainStyleMenu`** as the method name. It has no parameters.
4. Click **OK** to close the dialog.

The Code Editor displays the method name in the Browser.

5. Enter the following code in the method:

```
Dim start, length, i as Integer
if TextField.selbold or TextField.selitalic or
   TextField.selunderline then
   StylePlainText.Checked=False
else
   StylePlainText.Checked=True //assume selected text is plain
   Start=TextField.SelStart
   Length=TextField.SelLength
   For i=Start to Start+Length
     TextField.SelStart=i
     TextField.SelLength=1
     if TextField.selbold or TextField.selitalic or
        TextField.selunderline then
       StylePlainText.Checked=False
     Exit
   End if
 Next
 //now restore the selected text
 TextField.SelStart=Start
 TextField.SelLength=Length

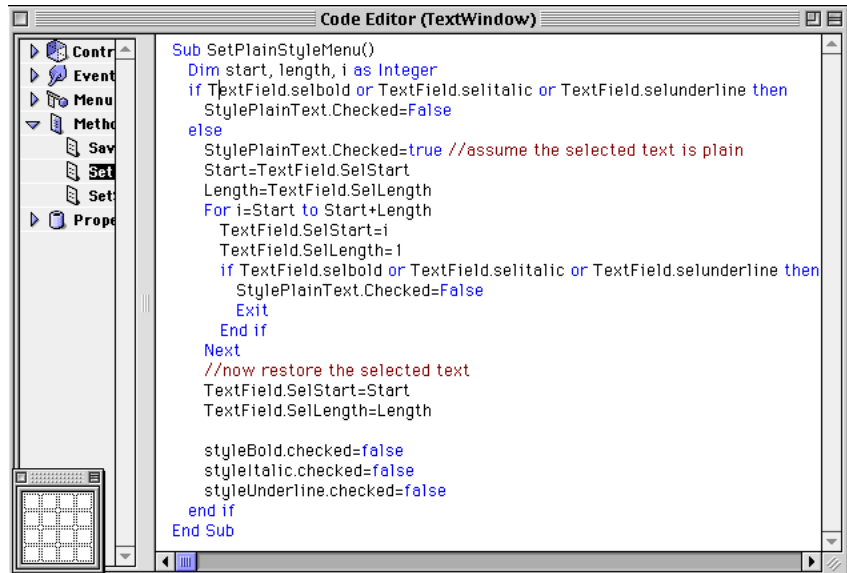
 styleBold.checked=False
 styleItalic.checked=False
 styleUnderline.checked=False
end if
```

The first If statement uses the `SelBold`, `SelItalic`, and `SelUnderline` properties of an `EditField`. In this statement, they test whether the selected text is bold, italic, or underline. If so, it unchecks the Plain-

Text item. If the selected text is not bold, italic, or underlined, then it checks the Plain menu item.

The Code Editor should now look like Figure 33 on page 70. Notice that the long If...then conditions must be entered on one line in the Code Editor.

FIGURE 33. The SetPlainStyle method.

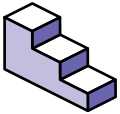


## 6. Save your project.

To handle the Style menu events, do this:

1. Use the Edit ► New Menu Handler command to create new menu handlers for each of the four menu items in the Style menu.
2. Add the following code to the StylePlainText menu handler:

```
//remove all styles from the selected text
TextField.selbold=false
TextField.selitalic=false
TextField.selunderline=false
SetPlainStyleMenu
```



The first three lines turn off bold, italic, and underline from the selection. The last line calls the `SetPlainStyleMenu`, which will check the Plain menu since the bold, italic, and underline properties are all False.

**3. Add the following code to the StyleBold menu handler:**

```
TextField.ToggleSelectionBold
StyleBold.Checked=TextField.SelBold
SetPlainStyleMenu
```

The first line toggles (reverses) the Bold attribute for the selected text (`ToggleSelectionBold` is a method of an `EditField` and `TextField` was derived from the `EditField` class). The second line checks the Bold menu item if the toggling applies bold or unchecks it if the toggling removes bold. The `SetPlainStyleMenu` method checks or unchecks the Plain menu item, depending on whether the toggling adds or removes the Bold attribute.

The next steps use the same logic to manage the italic and underline text attributes.

**4. Add the following code to the StyleItalic menu handler:**

```
TextField.ToggleSelectionItalic
StyleItalic.Checked=TextField.SelItalic
SetPlainStyleMenu
```

**5. Add the following code to the StyleUnderline menu handler:**

```
TextField.ToggleSelectionUnderline
StyleUnderline.Checked=TextField.SelUnderline
SetPlainStyleMenu
```

**6. Save your project.**

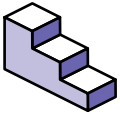
## **Managing Changes to the Text Selection**

The final step is to add code that places a check mark next to the current style when the text selection changes. Without this addition, the Style menu would not be properly updated when

the user moves the insertion point to text in a different style without actually changing the text style.

To do this, you will add code to the `SelChange` event handler of the `TextField`. This event runs when the text selection has changed.

To manage the Style menu check mark, do this:



1. In the Code Editor for `TextWindow`, expand the `Controls` item and then expand the `TextField` item.
2. Click on `SelChange` and enter the following code:

```
//uncheck all style items
StyleBold.Checked=False
StyleItalic.Checked=False
StyleUnderline.Checked=False

//now check the appropriate items
StyleBold.Checked=Me.SelBold
StyleItalic.Checked=Me.SelItalic
StyleUnderline.Checked=Me.SelUnderline
SetPlainStyleMenu
```

The use of “`Me`” in this code is a reference to the event handler’s control, `TextField`. In other words, `Me.SelBold` indicates whether the currently selected text in `TextField` is bold. If the selected text is Bold, the corresponding menu item is checked; otherwise, it is not. As before, the `SetPlainStyleMenu` method manages the Plain menu item.

## Creating the Size Menu

In this section you will create a Size menu and several menu items that specify a particular font size. You will use an *array* to manage the menu items in the Size menu. An array is simply a multi-valued object. It has only one name but several *elements*. In this



case, the array elements correspond to the Size menu items. The elements are managed by the *Index* property of the array elements. Arrays in REALbasic are generally zero-based, meaning that the first element of the array has an index value of zero. For example, the notation "aControl(2)" actually refers to the third element of the array aControl.

### Creating the Size Menu and its Menu Items

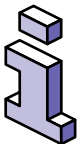
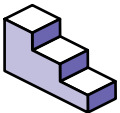
In this exercise, you will create a Size menu and menu items corresponding to the font sizes of 9, 10, 12, 14, 18, 24, 36, 48, 72, and 128. Each menu item will have the same Name ("Size") but have different Text and Index properties. As you know, the Text property controls the text of the menu item. Because these menu items will be managed by an array, both the Name and the Index properties will be used to refer internally to a particular menu item.

To create the Size menu, do this:

1. Double-click the Menu item in the Project Window.
2. Select the blank item on the menu bar.
3. Use the Property Window to set its Text property to **Size**.
4. The Size menu now appears in the menu bar.
5. Select the blank menu item in the Size menu and use the Properties Window to set its Name to **Size**, Index to **0** (zero), and Text to **9**.

This menu item is the first element of the Size array, so the value of the index property is zero. In the next step, you will create nine more array elements. REALbasic now "knows" that you want to create an array, so it will increment the value of the Index property for you when you set the Name of each menu item to Size

If you forget to specify the value of Index for the first menu item, REALbasic will ask you whether you want to create an array when you give the second menu item the same Name as the first item. Otherwise, it will increment the value of Index automatically when you enter "Size" as the Name.



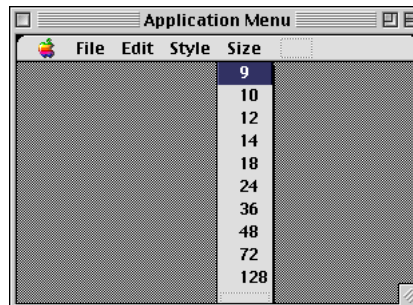
6. Continue adding menu items with the Name **Size** for all menu items and Text properties of **10, 12, 14, 18, 24, 36, 48, 72, and 128**. In other words, the Size menu items should be specified as follows:

**TABLE 5. Properties of the Size menu items.**

Menu Item	Name	Text	Index
9	Size	9	0
10	Size	10	1
12	Size	12	2
14	Size	14	3
18	Size	18	4
24	Size	24	5
36	Size	36	6
48	Size	48	7
72	Size	72	8
128	Size	128	9

Your Size menu should look like that shown in Figure 34.

**FIGURE 34. Newly created Size menu.**



7. Close the Menu Editor and save your project as "MySimpleText-ch5".

## Enabling the Size Menu Items

Next, you need to enable the new menu items. You enable these menu items by adding code to the EnableMenuItems event handler in the TextWindow Code Editor. Since the menu items are an array, you can use a loop with an index counter to enable the items.

To enable the menu items, do this:

1. Select TextWindow in the Project Window and open its Code Editor by typing Option-Tab.
2. Expand the Events item in the Browser and select EnableMenuItems.
3. Add the following line as the first line of the method:

```
Dim i as Integer
```

The variable *i* will be the index counter for the loop. You must place declarations (which define the data types of variables) prior to any 'regular' lines of code.

4. Add the following lines to the end of the method:

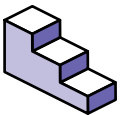
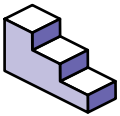
```
//enable font size menu items  
for i=0 to 9  
    Size(i).Enabled=True  
next
```

## Adding the Menu Handler

The menu handler for the Size menu items uses a method, SetSizeMenu. It places a check mark next to the selected Size menu item by setting the value of the Checked property of the menu item on-the-fly. You can add this before writing the menu handler itself.

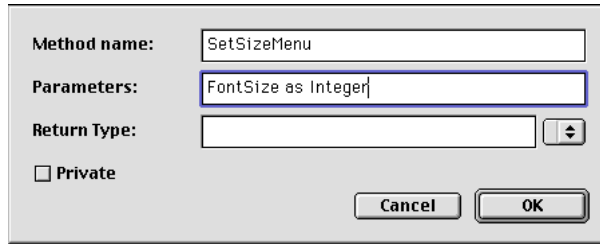
To add the SetSizeMenu method, do this:

1. With the Code Editor for TextWindow in the front, choose Edit ► New Method.



2. In the New Method dialog box, enter the Method name as **SetSizeMenu** and parameter **FontSize as Integer**.
3. The New Method dialog box should look like Figure 35.

**FIGURE 35. The SetSizeMenu declaration.**



4. Enter the following code into the Code Editor for SetSizeMenu:

```
Dim i as Integer
For i=0 to 9
  if Size(i).Text=Str(FontSize) Then
    Size(i).Checked=True
  else
    Size(i).Checked=False
  end if
Next
```

The If statement determines whether the Text property of the  $i^{\text{th}}$  Size menu item is equal to the FontSize passed to the method. If so, it checks that menu item.

Now, you can add the menu handler itself. It uses a Select Case structure to determine which menu item was selected and change the font size.

5. Choose Edit ► New Menu Handler and choose Size from the pop-up menu.

Notice that you need only one menu handler for all the Size menu items. REALbasic understands that there is an array controlling this group of menu items and adds the index of the array as a parameter in the menu handler.

**6. Enter the following menu handler code.**

```
Dim i as Integer
For i=0 to 9
    Size(i).Checked=False
Next
Select Case Index
Case 0
    TextField.SelTextSize=9
Case 1
    TextField.SelTextSize=10
Case 2
    TextField.SelTextSize=12
Case 3
    TextField.SelTextSize=14
Case 4
    TextField.SelTextSize=18
Case 5
    TextField.SelTextSize=24
Case 6
    TextField.SelTextSize=36
Case 7
    TextField.SelTextSize=48
Case 8
    TextField.SelTextSize=72
Case 9
    TextField.SelTextSize=128
End Select
SetSizeMenu Val(Size(Index).Text)
```

The menu handler uses the `SelTextSize` property of the `EditField` class to set the font size of the selected text. The last line of the menu handler passes the selected font size (as an integer) to `SetSizeMenu`. This method places a check mark next to the selected menu item.

## **Managing Changes to the Text Selection**

The final step is to add code that places a check mark next to the currently selected font size when the text selection changes. Without this addition, the `Size` menu would not be properly

updated when the user moves the insertion point to text in a different font size. That is, if the user selected text and changed its font size and then moved the insertion point to text out of the selection, the Size menu would not update.

Since this is a property of the TextField, the code is added to the SelChange property of TextField.

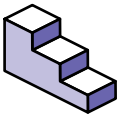
To manage checking and unchecking Size menu items, do this:

1. In the Code Editor for TextWindow, expand the Controls item and then expand the TextField item.
2. Highlight the SelChange item and add the following line of code:

```
SetSizeMenu Me.SelTextSize
```

This line of code runs SetSizeMenu when the text selection changes. It passes the font size of the current text selection to SetSizeMenu.

3. Save your project.

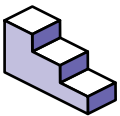


## Testing the Style and Size Menus

Now that all the code is in place, you are ready to see how it works.

To use the styled text editor, do this:

1. Choose Debug ► Run and enter some text in the text editor.
2. Select some text and try changing the font size and style.  
Notice that the check marks in both menus show the font size and style.
3. When you are finished testing, choose File ► Quit to return to the Design environment.

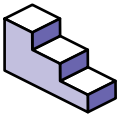


## Printing Styled Text

Now that you can enter text and change the font, font size, and style of selected text, you will want to be able to print out documents that retain your styled text attributes.

In this section, you will add Page Setup and Print items to the File menu to accomplish this task.

### Creating the Page Setup and Print Menu Items



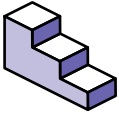
To create the menu items, do this:

1. Double-click the Menu item in the Project window and select the blank menu item in the File menu.
2. In the Properties Window, enter **Page Setup...** in the Text area and press Enter.  
REALbasic automatically assigns the Name FilePageSetup.
3. Next, select the blank menu item in the File menu and enter **Print...** in the Text area and press Enter.
4. Assign **P** to the CommandKey property.
5. Position the two new menu items between the Save As and Quit menus.
6. Select the empty menu item at the bottom of the Edit menu and enter a dash "-" as its Text property.  
This creates a separator between groups of menu items.
7. Drag the divider between the Save As and Page Setup menu items.
8. Create another divider and drag it between the Print and Quit menu items.
9. Close the Menu Editor.

## Enabling the Page Setup and Print Menu Items

You want the user to be able to access these menu items whenever a document window is open, so you should enable them in TextWindow's Code Editor.

To enable the menu items, do this:



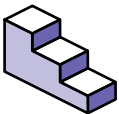
1. In the Code Editor for the TextWindow window, expand the Events item.
2. Highlight the EnableMenuItems event and add the following line to the existing code:

```
FilePageSetup.Enabled=true  
FilePrint.Enabled=true
```

## Handling the Page Setup Menu Item

To manage the user's selections in the Page Setup dialog box, you need to create a PrinterSetup object. This object has a property, SetupString, that contains many of these selections. You will first add this property to TextWindow's Code Editor.

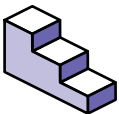
To add the property, do this:



1. Choose Edit ► New Property and enter **PageSetup as String** in the Property definition dialog box.
2. Click OK to close the window.

Next, you need to write the menu handler for the Page Setup menu item.

To add the Page Setup menu handler, do this:



1. Choose Edit ► New Menu Handler and choose FilePageSetup from the pop-up menu.
2. Enter the following code in the Page Setup menu handler:



```
Dim ps as PrinterSetup
ps=New PrinterSetup
If PageSetup <> "" then
    ps.setupstring=PageSetup
end if
If ps.PageSetupDialog then
    PageSetup=ps.setupstring
end if
```

The PrinterSetup property, ps.SetupString, contains the page setup selections that the user makes in the Page Setup dialog box. The second If statement displays the Page Setup dialog box. If the user clicks OK, ps.PageSetupDialog returns True and the SetupString is assigned to the PageSetup property.

The menu handler uses the PageSetup property to store the user's selections.

## Handling the Print Menu Item

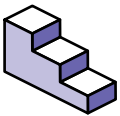
You use an object of type StyledTextPrinter to print styled text. It uses its DrawBlock property to "draw" the styled text on the page.

To add the Print menu handler, do this:

1. Choose Edit ► New Menu Handler and choose FilePrint from the pop-up menu.
2. Enter the following code in the Print menu handler:

```
Dim stp as StyledTextPrinter
Dim g as Graphics
Dim ps as PrinterSetup
Dim pageWidth as Integer
Dim pageHeight as Integer
ps=new PrinterSetup

If PageSetup <> "" then //PageSetup contains properties
    ps.setupString=PageSetup
    pageWidth=ps.Width
```



```
    pageHeight=ps.Height
// open Print dialog with Page Setup properties
    g=openPrinterDialog(ps)
else
    g=openPrinterDialog() //open dg w/o Page Setup properties
    pageWidth=72*7.5 //default width and height
    pageHeight=72*9
end if
If g <> nil then //user didn't cancel Print dialog
    stp=TextField.StyledTextPrinter(g,pageWidth)
    stp.drawBlock 0,0,pageHeight
End if
```

The menu handler uses the `StyledTextPrinter` method of the `EditField` class to create a `StyledTextPrinter` object ("stp"). If the user used the Page Setup dialog box to set properties, the `PageSetup` property is not null and its properties are used for printing. Otherwise, default page setup properties are used.

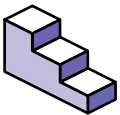
If the user accepts the Print dialog, the `DrawBlock` method is used to draw the contents of `TextField` to the printer.

## Testing Styled Text Printing

Now that styled text printing has been installed in your application, you are ready to see how it works.

To print styled text, do this:

1. Choose **Debug ► Run** and enter some text in the text editor.
2. Select some text and change the font size and style.
3. Use the **Page Setup** and **Print** menus to test styled text printing.
4. When you are finished testing, choose **File ► Quit** to return to the Design environment.



## Review

In this chapter you learned how to add the capability to work with styled text in your application.

---

***To Learn More About:***

REALbasic Text Handling

REALbasic commands and language

---

***Go to:***

*REALbasic Developer's Guide: Chapters 7, 8.*

*REALbasic Language Reference*



# **Creating Dynamic Menus**

---

In this chapter you will learn how to create a menu whose items will be created on-the-fly when the user starts his copy of the application. You will add a Font menu to the application and add code that will load the names of the fonts installed on the user's computer at the time he starts the application.

Unlike the Style and Size menus, you cannot specify the Font menu items in advance. Different users will see different Font menus.

## Getting Started

Locate the REALbasic project file that you saved at the end of last chapter ("MySimpleText-ch5"). Launch REALbasic and open the project file. If you need to, you can use the file "MySimpleText-ch5" that is located in the "Ch 5 Files" folder.

## Implementing the Font Menu

Implementing the Font menu involves the same basic steps for menu creation that you learned in earlier chapters. The key difference here is that you will add a method that loads the names of existing fonts into an array. This method runs when the application starts up.

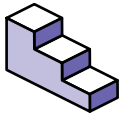
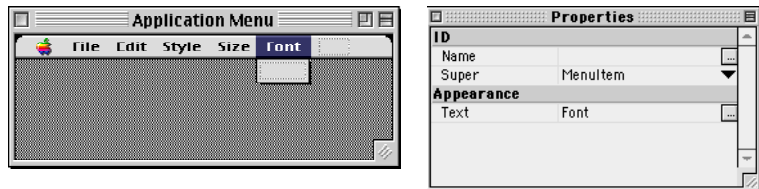
First, you add the Font menu to the menubar.

To create the Font menu, do this:

1. Open the Menu Editor from the Project Window.
2. Select the blank item on the menu bar.
3. Change its Text property to **Font**.

The Font menu should look like that shown in Figure 36.

**FIGURE 36. Font menu and Its Properties Window.**



4. In the Menu Editor, drag the highlighted Font menu so that it lies between the Edit and Style menus.

As you drag, a vertical bar in the menubar indicates the position of the Font menu during the drag.

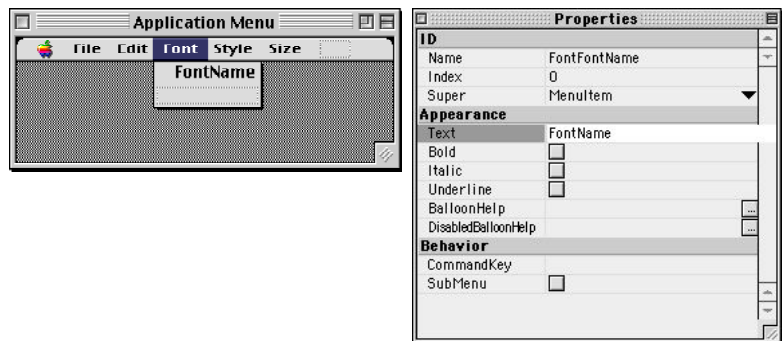
5. Select the empty menu item in the Font menu and enter **FontName** as the Text property.

The Name property is automatically assigned "FontFontName".

6. Enter **0** (zero) in the Index property of the FontFontName menu item.

The Menu Editor and the Properties window should look like that shown in Figure 37.

**FIGURE 37. Updated Font menu and Its Properties Window.**



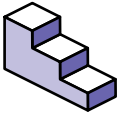
7. Close the Menu Editor and save your project as MySimpleText-ch6.

## Building the Font Menu

The menu items for the Font menu are built when the user launches the application. We, therefore, want to add this code to the Application class, not to the text editor window. The code will run when the application's Open event occurs, that is, when the application itself opens. (There is no need to rebuild the Font menu for each window instance, for example.)

You should not be surprised to see that we will load the font names into an array.

You will write a method for the Application class's Open event handler that builds the menu items from the fonts installed in the user's operating system.



To build the Font menu items, do this:

1. With the Project Window in the front, highlight the App class and press Option-Tab to bring the Application class's Code Editor to the front.
2. Click the disclosure triangle next to the Events item to display all of the event handlers for the App object.
3. Select the Open event handler.
4. Add the following code to the method:

```
Dim m as MenuItem
Dim i as Integer

//build the font menu
FontFontName(0).text=Font(0)
For i=1 to FontCount-1
    m=New FontFontName
    m.Text=Font(i)
Next
```

This method uses the global Font function which returns the name of the  $i^{\text{th}}$  font on the user's computer. The first font is handled separately, since we already created a menu item for it. Menu items for the remaining fonts are created dynamically using the New function. FontFontName is an instance of the MenuItem class.

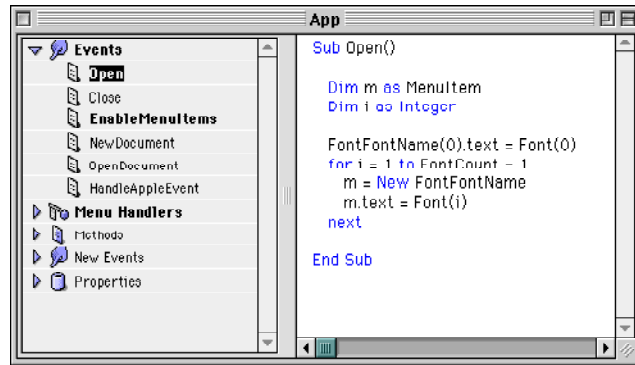
The code builds a new menu item (m) for each font and assigns its name to the Text property of the menu item.

The Code Editor for App should look like that shown in Figure 38.



**FIGURE 38. App Object with code added to its Open event handler.**

---



5. Close the App Code Editor and save your project.

## Enabling the Font Menu

Since Font menu items should be enabled only when a document is open, we will enable them in TextWindow.

To enable the Font menu, do this:

1. Open TextWindow's Code Editor from the Project Window.
2. Select the EnableMenuItems event handler from the Browser.
3. Add the following code to the end of the method:

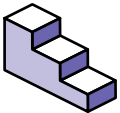
```

// Enable all fonts in the Font menu
For i = 0 to FontCount - 1
    FontFontName(i).Enabled = True
Next

```

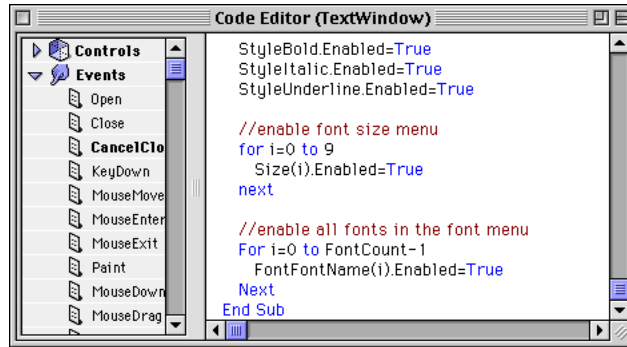
The event handler uses the FontCount function to determine how many fonts are on the user's computer.

The Code Editor should look like that shown in Figure 39 on page 90.



**FIGURE 39. EnableMenuItems with font enabling added.**

---



4. Save your project.

## Handling the Font Menu

The Font Menu menu handler needs to set the currently selected text to the font that the user chooses from the Font menu. This will be done with the SelTextFont property of the TextField. It also needs to add a check mark next to the name of that font.

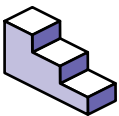
To handle Font menu events, do this:

1. Open the Code Editor for TextWindow create a new menu handler for the FontFontName menu item (choose Edit ► New Menu Handler...).

The New Menu Handler dialog box should look like that shown in Figure 40.

**FIGURE 40. New Menu Handler dialog box.**

---



**2. Click OK.**

The Code Editor displays the FontFontName menu handler. As was the case for the Size menu, REALbasic understands that all the Font menu items are controlled by an array. The menu handler includes an Index parameter so that you can manage all the menu items using one menu handler.

**3. Add the following code to the FontFontName menu handler:**

```
TextField.SelTextFont=Font ( Index)
```

This line of code sets the font of the selected text to the font selected in the Font menu. Next, you need to add code that places a check mark next to the name of this font in the Font menu. This code loops through the font names in the Font menu and finds the one whose name matches the name of the font of the selected text.

**4. Add the following declaration as the first line of the FontFontName menu handler:**

```
Dim i as Integer
```

**5. Now, add the following code after the last line of the menu handler:**

```
For i=0 to FontCount-1
  if FontFontName(i).text = Font(index) then
    FontFontName(i).Checked = True
  else
    FontFontName(i).checked = False
  End if
Next
```

This code loops through the names of all the installed fonts and checks the selected font.

Finally, you need to add similar code to the SelChange event of the TextField to set the check mark whenever the text selection changes. This occurs, for example, when the user selects text, makes a selection from the Font menu and then moves the

insertion point. The Font menu must update to indicate the Font of the new selection.

1. Open the SelChangeEvent of TextField and enter the following declarations as the first two lines of the method:

```
Dim i as Integer
Dim theFont as String
```

2. Next, add the following lines after the last line of the method:

```
//check the correct font when the selection changes

TheFont = Me.SelTextFont
If TheFont <> "" then
  For i=0 to FontCount-1
    If FontFontName(i).text = TheFont then
      FontFontName(i).Checked = True
    Else
      FontFontName(i).Checked = False
    End if
  Next
End if
```

The variable TheFont stores the name of the selected font. The For...Next loop compares this name to the names of every font in the Font menu and checks the menu item whose name matches TheFont.

3. Save your project.
4. Choose Debug ► Run (⌘-R) to test the Font menu.  
Notice that you can now change the font, font size, and style of selected text.
5. After you're done testing the application, choose File ► Quit (⌘-Q) to return to the Design Environment.

## Review

In this chapter you learned how to dynamically create menu items in your application.

---

**To Learn More About:**

REALbasic Font Handling

REALbasic Menu Handling

REALbasic commands and language

---

**Go to:**

*REALbasic Developer's Guide: Chapters 3, 4, 7.*

*REALbasic Developer's Guide: Chapters 3, 5, 7.*

*REALbasic Language Reference*



# Communicating Between Windows

---

In this chapter you will work with object communication features in REALbasic. You will learn how to:

- Add a Find dialog box to your application
- Add code to your application to allow communication between the Find dialog box and the text editor

The Find function that you will build is a simple function that searches from the text insertion point to the end of the text. It does not contain an option to go back and start from the beginning of the document to the text insertion point, although you could easily add this capability later. This Find function is not case-sensitive.

## Getting Started

Locate the REALbasic project file that you saved at the end of last chapter ("MySimpleText-ch6"). Launch REALbasic and open the project file. If you need to, you may use the file "MySimpleText-ch6" that is located in the "Ch 6 Files" folder.

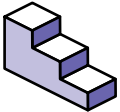
## Implementing the Find Dialog Box

By now you are familiar with the process of adding a menu item, enabling it, and adding a menu handler. The new feature in this chapter is that the dialog box that is displayed by the menu item must communicate with another window in the application.

You will start by adding a menu item for the Find function to the Edit menu.

### Creating the Menu Item

To create the menu item, do this:



1. Double-click the Menu object in the Project Window.
2. Select the Edit menu on the menu bar.
3. Select the empty menu item at the bottom of the Edit menu and enter **Find...** as its Text property.

The Name property automatically is filled in as "EditFind" in the Properties Window.

4. Type an F for the CommandKey property.
5. Select the empty menu item at the bottom of the Edit menu and enter a dash "-" as its Text property.

This creates a divider between groups of menu items.

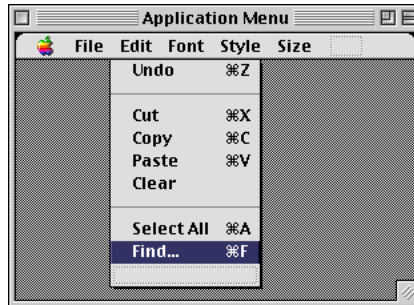
6. Drag the divider between the Clear and Select All menu items.



The Edit menu and the Find menu item should look like that shown in Figure 41.

**FIGURE 41. The Find menu item.**

---



7. Close the Menu Editor and save your project as "MySimpleText-ch7".

## Enabling the Find Menu Item

The Find menu should only be enabled when a document window is open, so you enable it in TextWindow's Code Editor.

To enable the menu item, do this:

1. Open the Code Editor for TextWindow from the Project Window.
2. Select the EnableMenuItems event handler from the Browser.
3. Add the following code to the end of the method:

```
EditFind.Enabled = True
```

The Code Editor should look like that shown in Figure 42 on page 98.

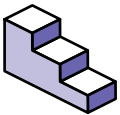
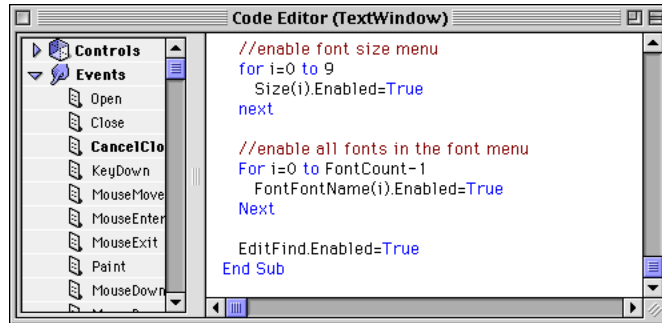


FIGURE 42. Updated Code Editor for EnableMenuItems.



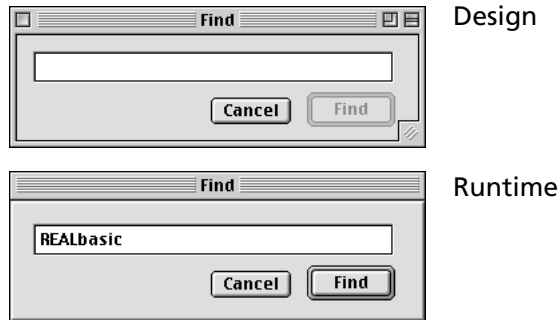
4. Close the Code Editor and save your project.

## Creating the Find Dialog Box

The next task is to create the Find dialog box itself. You will use the same process that you used in Chapter 4 to create the dialog box described in the section “Adding a ‘Save Changes’ Dialog Box” on page 52. You create a new window, add controls to the window, and assign properties to the controls. When you are finished, the Find dialog box will look like Figure 43 on page 99:

**FIGURE 43.** Find dialog box as seen in the Design and Runtime environments.

---



## Creating the Dialog Box

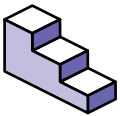
You begin by adding a new window to the project.

To create the dialog box, do this:

1. With the Project Window as the frontmost window, choose **File ► New Window**.  
REALbasic adds a window to the project and names it **Dialog1**.
2. Use Dialog1's Properties Window to change its name to **FindWindow**.
3. Change the window's Width to **300** and Height to **79**.
4. Deselect the Growlcon and Zoomlcon properties.  
These properties are deselected because FindWindow will be a fixed-sized dialog box.

The following steps add the controls to the empty window.

1. If it is not already open, double-click FindWindow in the Project Window to display it in a Window Editor.
2. Use the Tools Window to drag an EditField control to the top-left area. This control will serve as the entry area for the text to be searched for.



3. Click on the EditField control and, using the Properties Window, assign it the properties shown in Table 6.

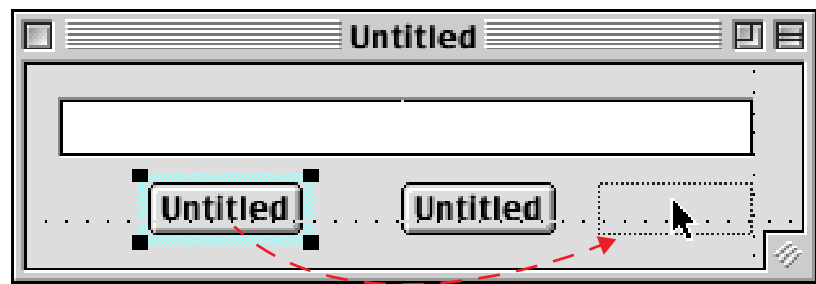
**TABLE 6. Properties of the EditField Control**

Property	Value
Name	FindText
Left	13
Top	13
Width	268
Height	22

4. Next, drag a PushButton from the Tools Window to the area occupied by the Cancel button in Figure 43 on page 99.
5. Select the Pushbutton control in FindWindow and choose Edit ► Duplicate (⌘-D) to create the Find pushbutton.
6. Drag the Find pushbutton into place, letting REALbasic align it to the baseline of the Cancel button using the alignment line.

You can use the horizontal and vertical alignment lines to align the Find pushbutton to both the Cancel button and the edge of the EditField, as shown in Figure 44.

**FIGURE 44. Aligning the Find button.**



7. Select each pushbutton and make the property assignments shown in Table 7.

**TABLE 7. Properties of the Cancel and Find Pushbuttons**

Property	Pushbutton	
	Cancel	Find
Name	CancelButton	FindButton
Left	145	220
Top	46	46
Width	60	60
Height	20	20
Caption	Cancel	Find
Default	Not checked	Checked
Cancel	Checked	Not checked
Enabled	Checked	Not Checked
Visible	Checked	Checked

In the next series of steps, you specify the actions of each control.

1. Click on FindWindow in the Project Window and press Option-Tab to open its Code Editor.
2. Expand the Controls item.  
You will see the names of the three objects that you just placed in FindWindow.
3. Expand FindText and then click the TextChange event handler. It runs whenever a user enters text in the Find dialog box. Enter the following code.

```
If Len(Me.Text)>0 then //if the user entered text
    FindButton.Enabled=True
Else
    FindButton.Enabled=False
End if
```

The If statement determines whether the FindText field contains some text after the change (The Me function is a reference to the control that owns the event handler—in this case FindText). If so, it enables the Find button.

4. Expand `CancelButton` and then click `Action`. Then enter the following code:

```
Self.Close
```

This line of code closes the window. The `Self` function is a reference to the button's parent window—not the button.

5. Expand `FindButton` and then click `Action`. Then enter the following code:

```
TextWindow(Window(1)).Find FindText.Text  
Self.Close
```

This method uses a method called `Find` which does the real work. It takes as its argument the text the user has entered into the dialog box. `FindText` was derived from the `EditField` class and `Text` is a property of `EditFields`.

The `Window` function is used to specify the `TextWindow` in which to search. The expression "`Window(1)`" refers to the second window—`Window (0)` is the `Find` dialog itself—so `Window(1)` is the frontmost document window.

The next step is to add the `Find` method to `TextWindow`.

1. Select the `TextWindow` item in the `Project Window` and press `Option-Tab` to open its `Code Editor`.
2. Choose `Edit ► New Method` to create the `Find` method.
3. Enter **Find** as the method name and **Value as String** as the parameter. Click `OK` to display the `Code Editor` for the `Find` method.
4. Enter the following into the `Find Code Editor`.

```
Dim FoundAt as Integer  
FoundAt=InStr(TextField.SelStart,TextField.Text,Value)  
If FoundAt>0 then //select the target text  
    TextField.SelStart=FoundAt-1  
    TextField.SelLength=Len(Value)  
Else  
    Beep  
    MsgBox "The text "+chr(210)+Value+chr(211)+" could not be  
        found."
```

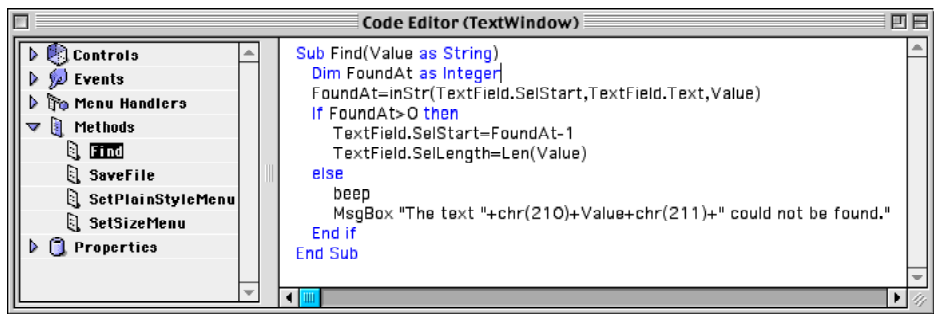
End if

This method locates the string to be searched for (the parameter *Value*) using the `InStr` function. `InStr` takes three parameters, the position at which to begin the search, the text to search, and the text to search for. It then sets the `SelStart` property of `TextField` to the position of the first highlighted character and `SelLength`, the length of the highlighted text, is set to the length of the string to be searched for.

Note that the `MsgBox` text should be entered on one line. It is split into two lines here because of space limitations. The Code Editor should look like Figure 45.

**FIGURE 45. The Find method in the Code Editor.**

---



The last step is to add the menu handler for the Find menu item. The menu handler simply displays the dialog box.

1. With `TextWindow`'s Code Editor as the frontmost window, choose **Edit ► New Menu Handler**.
2. Choose `EditFind` from the Menu Handler pop-up menu and enter the following code into the menu handler method.

```
FindWindow.Show
```

"Show" is a method of the `Window` class. This line of code simply displays the dialog box.

3. Save your project.

You are now ready to test the new feature. Choose Debug ► Run, enter some text, and test the Find menu item.

## Review

In this chapter you learned how to create objects that communicate with each other in your application.

---

***To Learn More About:***

REALbasic Object Communication

REALbasic commands and language

---

***Go to:***

*REALbasic Developer's Guide: Chapters 3, 5, 9.*

*REALbasic Language Reference*



# Wrapping Things Up

---

In this chapter you will work with the REALbasic Debugger and build a stand-alone application from your project. You will learn how to:

- Use the Debugger to find logical errors in your code,
- Turn your project into stand-alone MacOS and Windows applications.

## Getting Started

Locate the REALbasic project file that you saved at the end of last chapter ("MySimpleText-ch7"). Launch REALbasic and open the project file. If you need to, you may use the file "MySimpleText-ch7" that is located in the "Ch 7 Files" folder.

## Using the Debugger

The REALbasic Debugger is the part of REALbasic that helps you fix parts of your application that aren't working properly. As with the rest of REALbasic, the Debugger is easy to use. In fact, you probably have already used the Debugger without knowing it.

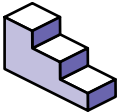
### Automatic Debugging Features

A portion of the REALbasic Debugger is active whenever you enter code in your application. The syntax coloring and code indentation in the Code Editor is one way that REALbasic proactively helps you to debug your code. Another is automatic syntax checking. Whenever you choose Debug ► Run, REALbasic checks the syntax of all your code and stops when it finds a syntax error.

To demonstrate REALbasic's syntax checking, do this:

1. Open the Code Editor for TextWindow.
2. Select the EnableMenuItems event handler to display its code.
3. At the top of the method, change the line

```
dim i as Integer  
  
to  
  
dim i as Integers
```



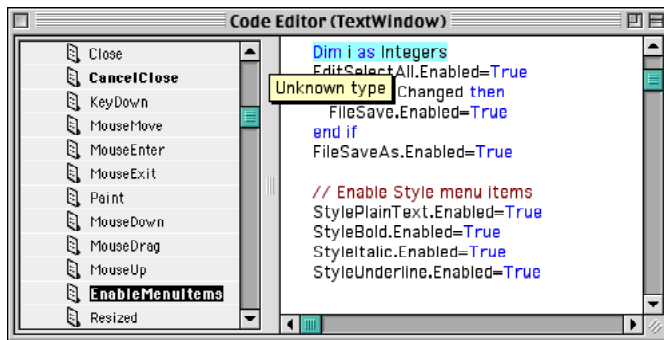
If you are paying attention, you will notice that the color of the data type changed from blue to black when you added the 's'. This should clue you the fact that 'Integers' is not a data type that REALbasic recognizes. But if you were not paying attention, REALbasic will point out the syntax error when you try to run the modified application.

4. Now, choose Debug ► Run (⌘-R).

An "Unknown type" error message appears and the offending line of code is highlighted. Your Code Editor should look like that shown in Figure 46.

**FIGURE 46. Syntax error message in the Code Editor.**

---



5. To fix the error, simply delete the extraneous 's' from the line of code.

Notice that the color of the data type changes to blue.

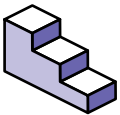
## Using the Debugger to Find Logical Errors

Errors that occur while your program is running are usually logical errors. To debug these errors, you will need to indicate to the REALbasic Debugger where it should check your code.

First, you need to set *breakpoints* in the source code in the region where you think the program is failing. Breakpoints are locations in your code where the application will pause and enter the Debugger while it is running. Once you are in the Debugger, you can examine the current values of variables, properties, and other parameters. You can check for unexpected, improper, or undefined values and take appropriate corrective action. You can also verify that your methods are actually being called when you expect them to be called.

Breakpoints don't alter your code and do not pause a stand-alone application built with REALbasic. The following exercise shows you how you can pause the application, check on the current values of variables, and continue executing a method line-by-line.

To see how the REALbasic Debugger works, do this:

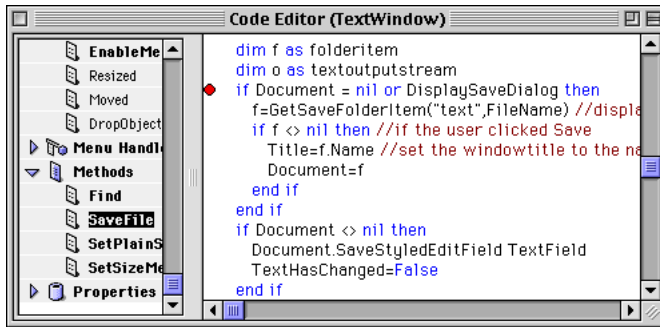


1. Open the Code Editor for TextWindow.
2. In the Browser, expand the Methods item and select the SaveFile method.  
The SaveFile method is displayed.
3. Click to the left of the line containing the first "If" keyword to place the insertion point there.
4. Choose Debug ► Set Breakpoint to place a breakpoint at that line of code.

A red diamond icon appears in the margin of the Code Editor, signaling a breakpoint. This breakpoint will cause REALbasic to pause when you try to save a document in the Runtime Environment. When you try to save a new document, the Debugger will appear instead of the save-file dialog box.

Your Code Editor should look like that shown in Figure 47.

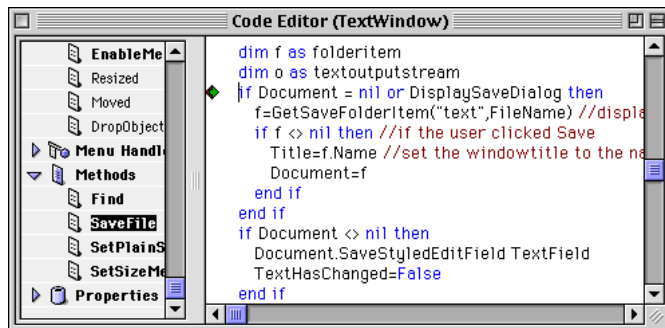
FIGURE 47. A breakpoint set in the Code Editor.



5. Save your project as "MySimpleText-ch8".
6. Choose Debug ► Run (⌘-R) to start your application in the Runtime Environment.
7. Type some text into the text editor and choose File ► Save (⌘-S).

Your application stops at the Breakpoint and displays the Code Editor as shown in Figure 48. A green arrow icon is located in the margin to the left of the line containing the breakpoint.

FIGURE 48. Debugger stopped at the breakpoint.



8. Locate the windows titled *Variables* and *Stack*.

By default, these windows are located on the right side of your screen.

When the execution of an application is paused at a breakpoint, the Variables and Stack windows are automatically opened. You can then inspect the current values for various properties and objects.

The Variables window contains a list of all the variables local to the method containing the breakpoint, along with their current values. Any objects that are defined in the method have a button in place of a value field. If you click the View... button, a window called the *Object Viewer* opens, containing the list of current property values for the object.

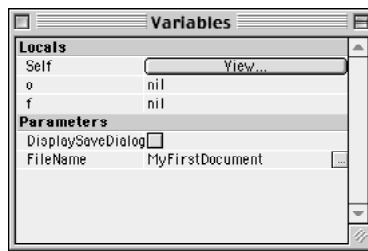
The Stack Window contains the name of the current method, along with any methods that invoked the current method. You can check the Stack Window to verify that methods are actually called when you expect them to be called.

**9. Select the Variables window to make it active.**

The Variables window should look like that shown in Figure 49.

**FIGURE 49. The Variables Window.**

---



In the Variables window you see that both variables (f and o) are undefined. This is as it should be since the document that you are

trying to save has not yet been saved. The variable `f` will be defined when you actually save the document.

When you are in the Debugger, you can execute code line by line and monitor the contents of the Variables Window. You do this using the Step Into or Step Over menu items.

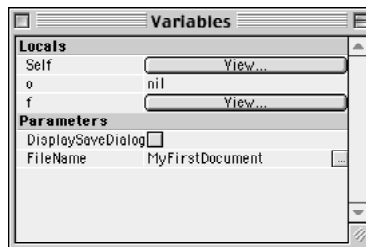
10. Bring the Code Editor to the front and choose Debug ► Step Into (⌘-I) until the save-file dialog box appears.

Each time you select this menu item, the current line of code is executed and the green arrow shown in Figure 48 on page 109 moves down one line.

11. Save the document under a filename and then examine the Variables window.

Notice that the `f` variable now has a View button because it has just been defined, as is shown in Figure 50.

**FIGURE 50. The Variables Window with a View button for “f”.**



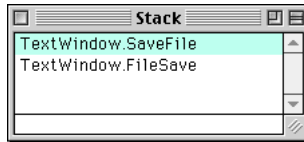
12. Click the variable `f`'s View button to see the current value of `f` in the Object Viewer.

The Object Viewer will show the absolute pathname to the document and the filename that you just gave it.

13. Click OK to close the Object Viewer.
14. Locate the Stack window.

The Stack window lists the current method and should look like that shown in Figure 51.

**FIGURE 51. The Stack Window.**



This is as it should be, i.e., the SaveFile method was called when the FileSave method handler was executed.

15. To see your debugging options choose the Debug menu.

The Debug menu should look like that shown in Figure 52.

**FIGURE 52. The Debug Menu.**



The Step Over and Step Into commands both execute the current line of code. The difference is that, if the line of code calls another method, Step Over will execute the line without stepping through each line of the *other* method's code.

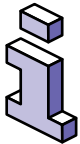
16. To resume execution of your application, choose Debug ► Run.
17. Choose File ► Quit (⌘-Q) to exit the Runtime environment and return to the Design environment.

Please refer to the *Developer's Guide* for a complete description of REALbasic's debugger.



## Building a Stand-alone Application

If you have tested your project and everything works as expected, then you will want to turn your REALbasic project into a stand-alone application. As a stand-alone application, your program will work like any other MacOS or Windows application.



In fact, once you build a stand-alone version of your REALbasic application, you do not need to have REALbasic to run the application.

To create a stand-alone application from your REALbasic project, do this:

1. Choose File ► Build Application....

A dialog box similar to that shown in Figure 53 on page 114 appears.

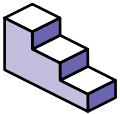
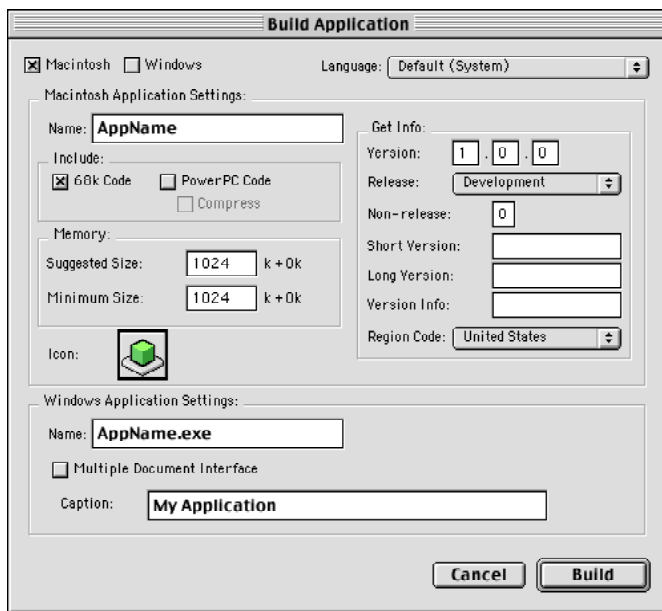


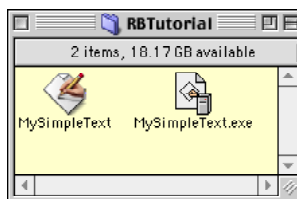
FIGURE 53. The Build Application Dialog.



2. Change the name to **"MySimpleText"** in the Macintosh Application Settings area.
3. If you have a Windows computer handy, click the Windows checkbox above the Macintosh Application Settings area and enter **MySimpleText.exe** in the Windows Application Settings area.
4. Click Build.

A new application file icon appears in the same folder as your REALbasic project file, ready for you to use. If you also created a Windows version, its icon appears as well.

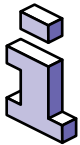
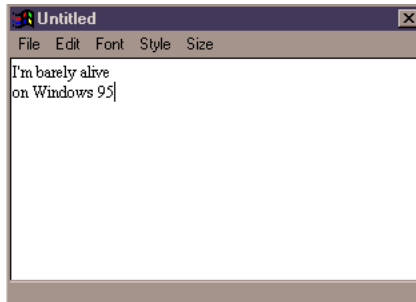
FIGURE 54. Macintosh and Windows versions of MySimpleText.



You can now quit REALbasic and double-click the MySimpleText icon from the Finder to edit text to your heart's content. If you have a Windows computer, drag MySimpleText.exe over and try it out.

**FIGURE 55. MySimpleText running on Windows.**

---



To learn about the other options in the Build Application dialog, consult the REALbasic *Developer's Guide*.

## Review

In this chapter you learned how to use the REALbasic Debugger and to build a stand-alone application from your REALbasic project.

---

**To Learn More About:**

Building stand-alone Applications

REALbasic Debugger

REALbasic commands and language

---

**Go to:**

*REALbasic Developer's Guide: Chapter 13.*

*REALbasic Developer's Guide: Chapter 10.*

*REALbasic Language Reference*



# Index

## A

- App object
  - building Font menu with 88
  - event handlers for 88
- Application 87
- application 7
  - building standalone 113
  - debugging 106
  - debugging your 108
  - fixing 106
  - naming 114
  - running 19
  - starting 19
- Application class 87
- array 72, 86
  - elements of 72
  - zero-based 73
- arrays
  - zero-based 73

## B

- boolean 42
- breakpoints 108
- bugs 105
- building a standalone application 113

## C

- CancelClose event handler 59–61
- Canvas control 53
  - Paint event handler 54
- caution icon 54
- class
  - creating a new 37, 38
- Code Editor 32, 43, 54
  - Browser pane 30
  - command key for displaying 30
  - dragging example code into 48
- Code Editor window 30
- Colors Window 15
- compiling an application 113
- controls 15
  - TextField 17

scrollbar [23](#)  
locking [24](#)

## **D**

data types  
  boolean [42](#)  
Debug menu [112](#)  
Debugger [105](#)  
debugging  
  error messages [107](#)  
  manual [108](#)  
  menu options [112](#)  
  Object Viewer [110](#)  
  setting breakpoints [108](#)  
  Stack window [110](#)  
  Variables window [110, 113](#)  
dialog box  
  creating a [53](#)  
document window [23](#)  
dynamically created menu items [88](#)

## **E**

E enableMenuItems event handler [30](#)  
EditField  
  event handlers for [49](#)  
  lock properties [24](#)  
  MultiLine property [23](#)  
  Properties Window [18](#)  
EditField control [17](#)  
EnableMenuItems event handler [75](#)  
error messages [107](#)  
event handler [37, 49, 50, 54](#)  
event handlers [50](#)  
event-driven programming [49](#)

## **F**

file types  
  recognizing [40](#)  
files  
  lesson [11](#)  
  tutorial [11](#)  
Find menu items  
  adding  
    Find in Edit menu [96](#)  
fixing programming code [105](#)  
FolderItem [42](#)  
FolderItem class [47, 63](#)

Font menu [86](#)  
fonts [86](#)

## **G**

GetOpenFolderItem function [62](#)  
GetOpenFolderItem method [63](#)  
graphical user interface [7](#)  
Graphics class [54](#)  
GUI [7](#)

## **I**

IDE [8](#)  
indenting lines of code [106](#)  
Index property [73](#), [87](#)  
InStr function [103](#)  
integrated development environment [8](#)  
interface objects [15](#)

## **L**

language  
    programming [8](#)  
local variables [110](#), [113](#)  
locking properties [24](#)

## **M**

Menu Editor [36](#), [41](#)  
    opening [28](#)  
menu handler [28](#), [38](#), [50](#), [75](#), [90](#), [103](#)  
    adding a [31](#)  
menu item  
    command key property [29](#)  
    deleting a [29](#)  
menu item divider [79](#)  
menu item dividers [96](#)  
Menu items  
    adding  
        Open menu item [61](#)  
menu items  
    adding [28](#)  
        "Select All" to the Edit menu [28](#)  
        Close, Save, and Save As... in the File menu [41](#)  
        Font [86](#)  
        New in the File menu [36](#)  
        Style [67](#), [73](#)  
    dynamically created [88](#)  
    enabling

- "Select All" in the Edit menu [30](#)
- Close, Save, and Save As... in the File menu [43](#)
- Find & Replace [97](#)
- Font [89](#)
- Open in the File menu [62](#)
- Size and Style [68](#), [75](#)
- handling
  - "Select All" in the Edit menu [31](#)
  - Close, Save, and Save As... in the File menu [50](#)
  - Font [90](#)
  - Open in the File menu [62](#)
  - Style [70](#)
- Index property [73](#)
- managed by an array [72](#)
- menus
  - See also* menu items
- method
  - adding a [44–46](#), [102](#)
- methods
  - Stack window [110](#)
- MySimpleText [24](#), [114](#)

## **N**

- New function [63](#), [88](#)
- New Menu Handler dialog box [31](#), [90](#)

## **O**

- Object Viewer [110](#)
- object-oriented [7](#)
- online reference [46–48](#), [54](#)
- Online Reference Window [16](#)
- Open [87](#)
- Open event [87](#)
- Open event handler [88](#)
- opening the Menu Editor [28](#)

## **P**

- Paint event handler [54](#)
- program. *See* application
- programming language
  - BASIC [7](#)
  - object-oriented [7](#)
- project
  - REALbasic [24](#)
  - saving [24](#)
- project file [24](#)



Project Window 15  
properties 110, 113  
Properties Window 15  
Property Declaration dialog box 42  
PushButton control  
    properties of 56  
Pushbutton tool 56

## **R**

REALbasic  
    Debug menu 112  
    Debugger 105  
    design environment 14  
        Code Editor Window 30  
        Colors Window 15  
        Online Reference Window 16  
        Project Window 14  
        Properties Window 14  
        Tools Window 14  
        Window Editor 14  
    MySimpleText application 114  
    project 113  
    project file 24  
    runtime environment 20  
running an application 19  
runtime environment 20

## **S**

scrollbar 23  
SelChange event handler 72, 91  
SelChange property 78  
Self function 50–51, 52  
SelLength property 103  
SelStart property 103  
SelTextFont property 90  
SetSizeMenu method 75  
SimpleText 8  
stack 110  
Stack window 110  
standalone application 113  
StaticText control 55  
StaticText tool 55  
styled text  
    printing 79–82  
Super property 37  
syntax coloring 106

**T**

TextChanged event handler [101](#)

Tools Window [15](#), [52](#)

tutorial files [11](#)

**V**

Variables window [110](#), [113](#)

**W**

window

    adding a [99](#)

    creating a [53](#)

Window Editor [15](#)

Window function [60](#), [102](#)

windows

    adding properties to [41–42](#)

    creating [13](#), [17](#)

    dialog

        Build Application... [113](#)

    document [23](#)

    properties of [17](#)