



User's Guide

REALbasic 2010 Release 1 User's Guide

Documentation by David Brandt.

© 1999-2009 by REAL Software, Inc. All rights reserved.

Printed in U.S.A.

Mailing Address	REAL Software, Inc. PO Box 162181 Austin, TX 78716
Web Site	http://www.realsoftware.com
ftp Site	ftp://ftp.realsoftware.com
Support	REALbasic Feedback at the REAL Software web site.
Bugs/ Feature Requests	Submit via REALbasic Feedback at the REAL Software web site.
Database Plug-ins	The REALbasic CD; the most recent versions are at www.realsoftware.com .
Sales	sales@realsoftware.com
Phone	512-328-REAL (7325)
Fax	512-328-7372

Version 2010 Release 1, February, 2010

Contents

CHAPTER 1 Introduction	19
Contents	19
Welcome to REAL Studio	20
Installing REAL Studio	21
Windows Requirements	21
Linux Requirements	21
Macintosh Requirements	21
Where to Begin	22
Documentation Conventions	22
Using the On-Line Help	23
Searching the Online Reference	25
Context-Sensitive Help	25
Context-Sensitive Error Messages	26
Using Tips	26
Electronic Documentation	27
Our Support Web Page	28
End User Web Sites	28
REAL Studio Developer	28
REAL Studio third-party Books	28
Our Internet Mailing Lists	28
Obtaining Updates	28
Technical Support from REAL Software	29
Contacting REAL Software	29
Reporting Bugs and Making Feature Requests	30
 CHAPTER 2 Getting Started with REAL Studio	 31
Contents	31
Concepts	32
Applications are Driven by Events	32

Developing Software with REAL Studio	32
The Development Environment	33
The REAL Studio IDE Window	35
The Window Layout Editor	36
The Project Editor	40
The Code Editor	46
The Menu Editor	49
Configuring the IDE for Multiple Windows	51
Working with the Tabs bar	53
The Contextual Menu	53
Hiding the Tabs bar	53
Dragging a Tab	54
The Main Toolbar	55
Customizing the Main Toolbar	58
The Bookmarks Bar	59
REAL Studio IDE Menus	61
The File Menu	61
The Edit Menu	63
The Project Menu	65
The View Menu	68
The History Menu	70
The Bookmarks Menu	71
The Window Menu	72
The Help Menu	73
Working with Projects	73
Creating a New Project	74
Configuring the Project Editor Toolbar	77
Adding Items to Your Project	79
Removing Items from Your Project	81
The Project Editor Contextual Menu	81
Saving Your Project	84
Creating Project Templates	87

CHAPTER 3 Building a User Interface 89

Contents	90
Working with Windows.	90
Window Types.	90
Creating Windows.	102
Removing Windows	103
Setting the Default Window	104
Encrypting Windows	107

Message Dialog Boxes	108
The MsgBox function	109
The MessageDialog Class	109
Interacting with the User Through Controls	113
Favorites Controls	114
Adding, Changing, and Removing Controls	116
Understanding Control Layers	131
Understanding The Focus	132
Full Keyboard Access	137
Duplicating Controls	139
The Object Hierarchy	139
Button Controls for Performing Actions	140
Controls for Displaying and Entering Text	143
HTMLViewer	146
Controls for Displaying and Entering Numeric Values	147
Controls for Presenting a List of Choices	149
Controls for Visually Grouping Other Controls	155
Controls for Displaying Graphics and Pictures	160
Controls for Playing Movies, Music, and Animation	163
Miscellaneous Controls	164
Controls for Handling Communications	165
Toolbar Control	168
The Timer	173
Controls for Working With Databases	173
The Spotlight Query Control	174
ActiveX Controls	174
The Container Control	177
Opening an Old Project	178
Changing The Tab Order	179
Using the Edit Tab Order Mode	180
Auto-Adjustment of the Tab Order	183
Aligning Controls with Other Controls	183
Spacing Controls Evenly	184
The Control Hierarchy	184
Control Hierarchy Features	187
Adding Menus and Menu Items	189
The Default Menubar	190
Adding Menubars	192
Adding Menus	193
Adding a Help Menu	196
Adding Menu Items	196
Adding a Submenu	200

Adding a Menu Item to the Mac OS X Apple and Application Menus .	202
Moving Menus and Menu Items	204
Converting a Menu Item to a Menu	205
Removing Menu Items	205
Adding A Menu Item Separator	205
Creating Menultems on the Fly.	205
Importing and Exporting Menus	207
User Interface Guidelines	208

CHAPTER 4 BASIC Programming Concepts 211

Contents	211
BASIC versus REAL Studio.	212
Storing Values in Properties and Variables	213
What are Properties?	213
Variables.	213
Data Types	213
Changing a Value From One Data Type to Another.	219
Assigning Values to Properties	219
Getting Values From Properties	222
Getting and Setting Values in Variables	223
Declaring Objects	227
Using Arrays	229
Mathematical Operators	235
Operator Precedence	236
Constants	236
Reserved Words	240
Executing Instructions with Methods	241
Passing Values to Methods	241
Returning Values from Methods	243
Passing Parameters by Value and by Reference	244
Using the Meta-Constant	245
Documenting Your Code	247
Comparison Operators	249
Logical Comparisons	250
Bitwise Comparisons	251
Executing Instructions Repeatedly with Loops	252
While...Wend	253
Do...Loop	253
For...Next	254
The For...Each statement	257

Adding Loops to your Code	258
Making Decisions with Branching	260
If...Then...End If	260
If...Then...Else...End If	261
If...Then...Elseif...End If	262
If...Then...Else	262
#If...#Endif	262
Select...Case	264

CHAPTER 5 Programming with Events and Objects 271

Contents	271
Understanding Event-Driven Programming	272
Using The Code Editor	273
Opening the Code Editor	273
Configuring the Code Editor	275
The Browser	278
Understanding Methods in the Code Editor	282
Opening a Window from its Code Editor	296
The Code Editor's Contextual Menu	297
Searching your Project	297
Finding using the Contextual Menu	302
Copying and Pasting Code	304
Printing Your Code	304
Importing and Exporting Your Classes, Menus, Modules, and Windows	305
External Project Items	305
Importing	305
Exporting	307
Encrypting Your Source Code	307
Responding To User Actions with Event Handlers	309
Object-Oriented Programming	310
Windows Events	311
Opening Windows	313
Adding Properties to Windows	314
The Scope of a Property	315
Declaring an Array as a Property	316
Computed Properties	321
Shared Methods and Properties	322
Adding Constants to Windows	324
The Scope of Window Constants	324
Localizing an Application using Constants	325
Converting a Literal to a Constant	327

Adding Methods to Windows	329
Passing Parameters to Methods	329
Returning Values from Methods	330
The Scope of Methods	331
Dynamic Method Creation	334
An Example Method	335
Passing a Parameter by Value or Reference	336
Setting Default Values for a Parameter	338
Setter Methods	340
Constructors and Destructors	341
Attributes	341
Accessing Items of Other Windows	342
Controls	345
Events	345
Creating New Instances of Controls On The Fly	346
Sharing Code Among An Array of Controls	348
Drag and Drop	350
Dragging Text From TextFields	350
Dragging a Row From a ListBox	351
Dragging from an ImageWell	351
Dragging from a Canvas Control	352
Dropping.	352
Dropping Items On TextAreas	353
Dropping Items on ListBoxes	354
Dropping Items on ImageWells and Canvas controls	355
RawData and PrivateRawData Properties	356
Menus and Menu Items	357
Adding Code To a Menu Handler	359
Enabling Menu Items	360
Handling Menu Items From Individual Controls	361
Handling Menu Items When a Window Is Open.	361
Handling Menu Items When No Windows Are Open	361
Creating New Menu Items On The Fly	362
Displaying a Contextual Menu	364
Classes	365

CHAPTER 6 Adding Global Functionality with Modules . . 367

Contents	368
Understanding Modules	368
Adding A New Module	368
Scope of a Module's Items	370

Adding Methods to Modules	370
Adding Properties to Modules	373
Adding Constants to Modules	375
Adding a Constant to a Module	376
Color constants	378
Using Constants to Localize your Application	378
Adding Classes to Modules	384
Converting a Project Class to a Module Class	385
Adding Class Interfaces to Modules	387
Adding Event Definitions to Modules	392
Adding Delegates to Modules	393
Structures	394
Creating a Structure	394
Using Structures	396
Structure Alignment	397
Adding an Enumeration to a Module	397
Nesting a Module in a Module	399
Class Extension Methods	401
Importing and Exporting Modules	402
Exporting	402
Importing	402
Encrypting Modules	403

CHAPTER 7 Working With Text and Graphics 405

Contents	405
Working With Fonts	406
The System and SmallSystem Fonts	406
What Fonts Are Available?	406
Working with the Selected Text	407
Creating a Password Field	408
Formatting and Filtering Text Entry	408
The Format Property	408
The Mask Property	409
Handling Styled Text	409
Determining the Font, Size, and Style of Text	410
Setting the Font, Size, and Style of Text	411
Working with StyledText Objects	412
Working with Text Encodings	416
Text Encodings: From ASCII to Unicode	416
Changing Your Code To Handle Text Encodings	417

Formatting Numbers, Dates, and Times	420
Numbers	420
Dates	421
Times	423
Searching using Regular Expressions	424
Adding Pictures and Drawing Graphics	427
Understanding the Coordinates System	427
Displaying Pictures In a Window	428
Creating Pictures	431
Drawing Standard Dialog Icons	435
Drawing Pixels	436
Drawing Lines	437
Drawing Ovals	437
Drawing Rectangles	437
Drawing Polygons	438
Drawing into a Region in the Graphics Object	439
Creating Custom Controls with the Canvas Control	440
Working with Vector Graphics	442
Drawing and Displaying a Vector Object	443
Opening and Saving Vector Graphics	445
Working With Color	445
Determining the RGB Values For a Color	446
The Pixel Property of Graphics Objects	448
Printing Text and Graphics	448
Working with the Page Setup Dialog Box	448
Printing With The Print Dialog Box	449
Printing Without The Print Dialog Box	450
Printing Styled Text	450
Transferring Text and Graphics with the Clipboard	451
Testing The Clipboard For Specific Data Types	451
Getting Data From The Clipboard	452
Putting Data On The Clipboard	452

CHAPTER 8 Creating Reports 455

The Report Layout Editor	456
Report Editor Controls	456
Report Editor Toolbar	457
Report Editor Areas	457
Adding a Report to a Project	458
Adding a Grouping Section	460
Report Editor Examples	464

Using a Database as a Data Source	464
Using a Text File as a Data Source	470
CHAPTER 9 Working With Files	479
Contents	479
Understanding File Types	480
Using The File Types Editor	480
Creating Custom File Types for Your Application	489
Understanding FolderItems	491
How Are Shortcuts and Aliases Handled?	491
Getting a File at a Specific Location	492
Accessing Specific System Folders	494
Verifying that you have accessed the Item.	495
Creating a New FolderItem	496
Getting Information About a FolderItem	496
Deleting a FolderItem	496
Getting and Setting Ownership	497
Getting and Setting Permissions	497
Getting The Path To Your Application's Folder	500
Getting Specific Items In the Application's Folder	500
Getting The Selected File From An Open File Dialog Box	501
Getting The Selected Folder From An Open Folder Dialog Box.	504
Using the Save As Dialog Box.	507
Working With Text Files	510
Reading From a Text File	510
Writing to a Text File	512
Limitations of Text Files.	514
Working With Styled Text Files.	514
Loading Styled Text Into a TextArea	514
Writing Styled Text From a TextArea to a File	514
Working With Picture Files	516
Saving Pictures.	516
Opening Pictures	518
Working With Sound Files	519
Working With Movie Files	520
Working With Binary Files	521
BinaryStreams	522
Reading From a Binary File	522
Writing to a Binary File	523
Working With Macintosh Resources	524

Opening a File's Resource Fork	524
Adding a Resource Fork to a File	525
Adding a Resource Fork to a Project	525
Supported Resource Types	526
Reading Resources	526
Writing To Resources	528
More Information on the ResourceFork	528
Files Opened From the Desktop	529
Files Opened by Double-Clicking	529
Files Dropped On Your Application's Icon	529
Creating New Files	529

CHAPTER 10 Creating Reusable Objects with Classes 531

Contents	531
The Benefits of Classes	532
Reusable Code	532
Smaller Projects and Applications	532
Easier Code Maintenance	532
Easier Debugging	532
More Control	532
Understanding Instances	533
Understanding Subclasses	533
What is a Subclass?	533
Examples of Subclasses	533
Referring to a Class's Properties and Methods From Within the Class	538
Creating Classes	539
Creating a Subclass from an Existing Class	540
Creating a Superclass from an Existing Class	541
Saving Classes	542
External Project Items	542
Modifying Classes	543
Scope of a Class's Methods, Properties, and Constants	543
Adding Properties	544
Customizing the Properties List	545
Adding Computed Properties	549
Adding Shared Properties	552
Adding Constants	553
Adding Methods	555
Adding Shared Methods	557
Adding Event Definitions	558
Adding Structures	561

Structure Alignment	564
Adding Enumerations	564
Adding Delegates	566
Virtual Methods	568
Extending Classes	568
Writing a Class Extension Method	568
Calling a Class Extension Method	569
Constructors and Destructors	569
Constructors	570
Old Syntax	571
Destructors	571
Overloading	572
Overloading Custom Classes	572
Assigning a Value to a Method	573
Using Arrays of Classes	575
Casting	576
Managing Menus within Classes	577
Using Classes in Your Projects	577
The Class	577
The Instance	577
The Reference	578
Subclasses Based on Controls	578
Classes Based on Classes Other Than Controls	578
Accessing the Properties and Methods of a Class	579
When are Instances of Classes Removed From Memory?	580
The Application Class	580
Special Event Handlers	581
Scope of the App Class's Properties	582
Scope of the App Class's Methods	582
Creating Custom Interface Controls with Classes	583
Drawing Your Custom Control	584
Class Interfaces	585
Implementing Methods	590
Modifying and Deleting Interfaces	591
A Class Interface Example Project	591
Creating a new Class Interface from an Existing Class	593
Interface Inheritance	594
Introspection	597
Importing Classes From Other Projects	598
Importing External Project Items	599
Exporting Classes For Use In Other Projects	599
Encrypting Your Source Code	599

Deleting Classes From a Project	602
---	-----

CHAPTER 11 Creating Databases with REAL Studio 603

Contents	603
REAL Studio's Database Architecture	604
Structured Query Language	604
REAL Studio's Database Tools	605
Selecting a REAL Data Source.	605
Creating and Modifying Databases from the Project Editor	607
Adding Indexes	609
Viewing Data	610
Storage Types and Column Type Affinities	615
The DatabaseQuery Control	617
The DataControl Control	618
Creating a Database Front End Programmatically.	619
Accessing a Data Source	619
Creating a Database in Code	620
Opening a Data Source	621
Editing Records	623
Listing Records.	626
Adding Records	627

CHAPTER 12 Debugging Your Code 631

Contents	631
What is Debugging?	632
Logical Bugs	632
Syntactical Bugs	632
Analyzing the Project	632
Filtering Types of Issues	636
The Debugger	636
Breaking into the Debugger	637
The Debugger Screen	638
Controlling Execution	647
Following the Execution of Methods	648
Step	648
Step In	648
Step Out	649
Tracking Method Execution with the Stack	649
Watching Your Values	649

Local Values	649
Parameters	650
Global Values	650
Object IDs	650
Starting and Stopping Your Project	651
Runtime Exception Errors	652
Handling Runtime Errors	653
Profiling your Project	655
Remote Debugging	657
About Firewalls	663

CHAPTER 13 Communicating With The Outside World 665

Contents	665
Communicating With Serial Devices	666
Getting Set Up	666
Opening the Serial Port	666
Reading Data	666
Writing Data	667
Changing a Serial Control's Configuration on the Fly	668
Closing the Port	668
Communicating With Modems	668
Communicating with USB and FireWire Devices	668
TCP/IP Communications with the TCPSocket Control	669
Getting Set Up	669
Making a Connection to Another Computer	670
Listening For a Connection From Another Computer	670
Reading Data	670
Writing Data	671
Handling Errors	672
Orphaning a Socket	673
Maximum Number of Sockets	673
Closing the Connection	674
Sending and Receiving Email via TCP/IP	674
HTTP Communications	675
Handling Multiple Connections with the ServerSocket Control	675
Reference Counting	676
Handling Secure TCP Connections with the SSLSocket Control	676
UDP Connections with the UDPsocket Control	677
Datagrams	678
UDPsocket Modes	678
Making Networking Easy	679

The AutoDiscovery Class	680
Understanding Protocols	681

CHAPTER 14 Extending the Capabilities of REAL Studio . . . 683

Contents	683
Making API calls to the Operating System	684
Calling AppleScripts	685
Preparing an AppleScript to Work in REAL Studio	685
Adding an AppleScript to a Project	685
Passing Values To an AppleScript	686
Returning Values From an AppleScript	686
Calling an AppleScript	686
Removing an AppleScript	686
Communicating with AppleEvents	687
Sending AppleEvents	687
Receiving AppleEvents	687
Sophisticated AppleEvents	688
Using and Writing REAL Studio Plug-ins	689
Loading Plug-ins	689
Using Plug-ins	689
Including Plug-ins in Your Stand-Alone Applications	689
Writing Your Own Plug-ins	689
Using PowerPC Shared Libraries	690
Microsoft Office Automation	690
ActiveX Components	691

CHAPTER 15 Building Stand-Alone Applications 693

Contents	694
Choosing a Target Platform	694
Building Your Application	695
Building for Windows	696
Incremental Compilation	697
Analyzing the Project	699
Customizing the Standalone Application's Properties	700
Appearance Settings	701
Version Information	702
Windows Settings	705
Linux Settings	706
Mac Settings	706

Debugger	706
Advanced	707
Preparing your Application for Compilation	712
Compiling for Windows.	713
Mac OS X Considerations	716
Linux Considerations	718
Assigning Custom Document Icons.	719
Region Codes	720

CHAPTER 16 Converting Visual Basic Projects to REAL Studio
723

Contents	723
VB Migration Assistant	724
What doesn't it do?	724
Supported Versions of VB.	724
Third-party Controls.	724
Converting a VB Project.	724
Encoding Issues on Windows	727
Encoding Issues on Macintosh and Linux	727
Non-English File Names	727
Auto-opening your Project	728
Database Options	728

Index	729
------------------------	------------

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

Contents

- Welcome to REAL Studio
- Installing REAL Studio
- Documentation conventions
- Using the Online Reference
- Other helpful resources
- Contacting REAL Software

Welcome to REAL Studio

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

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

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

Most forms of BASIC are interpreted. This means that they include a translator that has to constantly translate BASIC code into the code that the computer can actually understand. REALbasic has no interpreter. REALbasic compiles your code when you run your application.

REALbasic's form of the BASIC language is also "object-oriented." This means that it uses a modern architecture that most popular programming languages (like C++ and Java) are using today. Object-oriented programming languages make it easier to write and debug because the code is written as individual objects that are similar to objects in the real world. In fact, in many ways REALbasic is more object-oriented than languages like C++ and certainly easier to learn and program.

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

Installing REAL Studio

The REAL Studio application has the following hardware and operating system requirements:

Windows Requirements

To run REAL Studio on Windows, you must have the following:

- A PC with at least a 1.0 GHz processor and at least 1 GB of RAM. (2.0 GHz processor and 2 GB of RAM recommended),
- The Windows 2000, Windows XP, or Windows Vista or Windows 7 operating system.

Compiled Windows applications run on Windows 2000, XP, and Vista/Window 7.

Run the Windows installer to install REAL Studio for Windows.

Linux Requirements

To run the REAL Studio IDE on Linux, you must have the following:

- A PC with at least a 1.0 GHz processor and at least 1 GB RAM; 2 GB of RAM and 2 GHz processor recommended,
- Any x86-based Linux distribution that includes GTK+ 2.8 (or higher), glibc-2.4 or higher, the CUPS (Common Unix Printing System), and libstdc++ .so.6. Ubuntu 6.10 or above, SUSE Linux Enterprise Desktop 10, and Red Hat Enterprise Linux 5 are officially supported.

The Linux version is available as a tar archive (.tgz file), Red Hat rpm file, and a deb file for Ubuntu.

A .tgz file was processed by tar and compressed by gzip. Untar the file and locate the REAL Studio application in the REAL Studio folder.

Compiled applications require an x86-based Linux distribution with GTK+ 2.8 or above, glibc-2.4, the CUPS, and libstdc++ .so.6.

Macintosh Requirements

To run REAL Studio on Macintosh you must have the following:

- A Macintosh with a 1.0 GHz G4 processor and 1 GB of RAM or any Intel-based Macintosh and Mac OS X 10.3 or later. REAL Studio for Macintosh is a Universal Binary; it runs “natively” on both PowerPC and Intel-based Macintoshes. The recommended configuration is any Intel Macintosh with 2 GB of RAM and Mac OS X 10.4 or above.

Compiled Macintosh applications run on any Intel or PowerPC Mac with Mac OS X 10.2 or later.

To install the Macintosh version of REAL Studio, drag the REAL Studio application from the disk image to your hard disk. It is recommended that you store it in your Applications folder.

Where to Begin

If you are new to programming, you should begin by going through the *QuickStart* and then the *Tutorial*. This will give you a good overview of REAL Studio and introduce you to the programming language. Next, read the *User's Guide*. This guide will provide you with detailed information on the language and the various components that make up REAL Studio. When you need details about a specific control or command in the language, consult the *Language Reference*.

Documentation Conventions

This documentation uses the following typographical conventions:

Initial References

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

Menu References

When you are told to select a menu item, the menu name is listed first, followed by an arrow, then the item name and command key shortcut. For example File ► Exit means “choose Exit from the File menu”.

Keyboard Equivalents

Most menu items have keyboard equivalents. On Windows and Linux, the Ctrl key is the primary modifier, and sometimes the Alt and Shift keys are used. On Macintosh, the Command key is the primary modifier; sometimes the Option and Shift keys are also used. When keyboard equivalents are given, Windows and Linux equivalents are given first, followed by the Macintosh keyboard equivalent. For example “Ctrl+Q or ⌘-Q” means “Ctrl+Q on Windows and Linux or Command-Q on Macintosh.”

Screen Illustrations

REAL Studio is truly a cross-platform application. The application itself runs under Windows 2000 and above, Linux with GTK+ 2.8 installed, and Mac OS X. REAL Studio can build applications that run on Windows 2000, Windows XP, Vista/Windows 7, Mac OS X, and Linux. The screen snapshots of the REAL Studio development environment are a mixture of Windows XP, Windows Vista, Ubuntu Linux, and Mac OS X Leopard. Where necessary, windows and controls that appear in built applications are shown for the Windows, Macintosh, and Linux platforms. Some interface features of REAL Studio are specific to a platform, so only that platform is shown.

Code Examples Code examples appear in gray boxes:

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

Icons

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

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



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



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



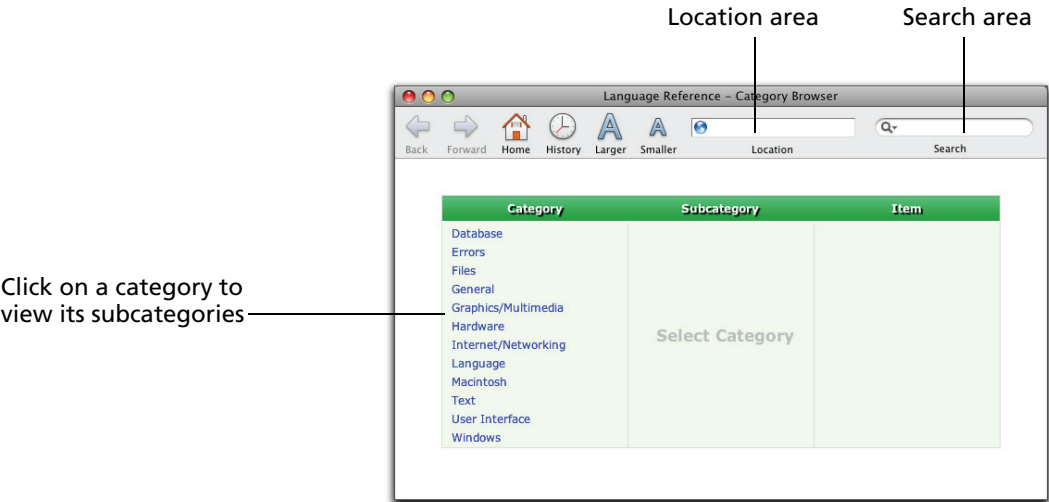
This icon indicates that the text to the right pertains to Mac OS X only.



Using the On-Line Help

The REAL Studio *Language Reference* is built-in to REAL Studio. To access this language reference, choose Help ► Language Reference (F1 on Windows and Linux or ⌘-? on Macintosh) or press the Help key.

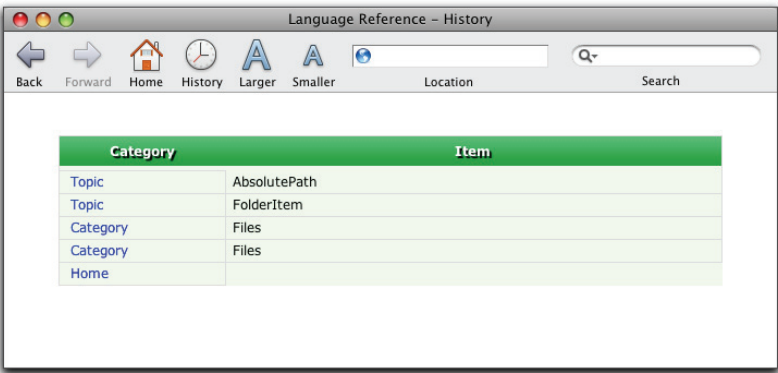
Figure 1. The On-Line Reference



Click a category name to view the subcategories and then click a subcategory to view its language items. Click on item to navigate to it.

Use the Arrows in the header area to move backward and forward through the items you have been browsing. The keyboard equivalents are Ctrl+[and Ctrl+] on Windows and Linux and Command-Left Arrow and Command-Right Arrow on Macintosh. The Home button takes you back to the page with the list of categories and subcategories. The History button displays the list of pages you have navigated to. A sample History screen is shown in Figure 2. The Item column shows the names of the screens you have navigated to, with the most recent screen at the top of the list. The Category column is a hypertexted list of the categories of the items.

Figure 2. The History screen.



The Larger and Smaller buttons increase or decrease the font size that is used in the Online Reference.

When you are programming, context-sensitive help is also available. Select the item in your Code Editor for which you want help and right+click (Command-click on Macintosh). A contextual menu appears. Choose the Help for *ItemName* menu command. REAL Studio will then open the online reference to the desired item.

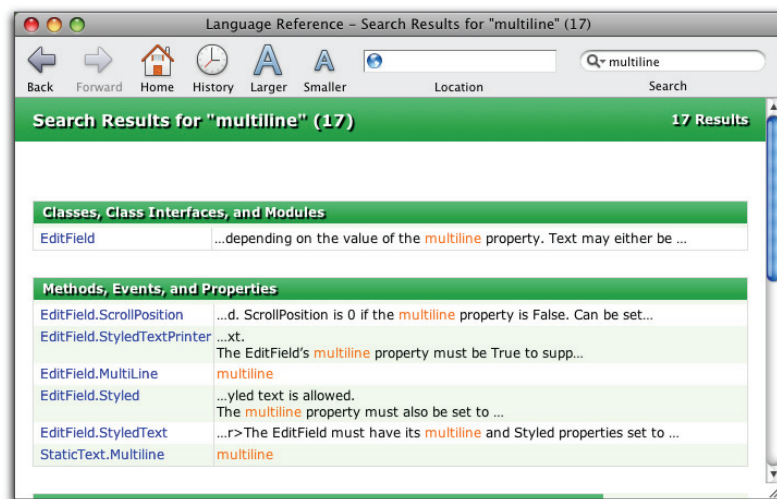
Searching the Online Reference

Use the Location area in the header of the window to search the Online Reference. To search for a REAL Studio language object (i.e., an item listed in the Browser), simply begin typing the name of the object.

As you type, REAL Studio tries to guess the name of the object. If it has one guess, the guess appears in gray text. To accept the guess, press the Tab key. If it has several guesses, three dots appear. Press Tab and a pop-up menu of choices appears. Use the up and down arrow keys to highlight the desired item and press Tab or Return to select it. Press Enter or Return to accept the entry and jump to the item.

Use the Search area to search for any term, such as the name of a property, method or event. Press Enter to do the search. If it finds matching items, it will return a list of found items. For example, the following shows the results for a search on the term “multiline”.

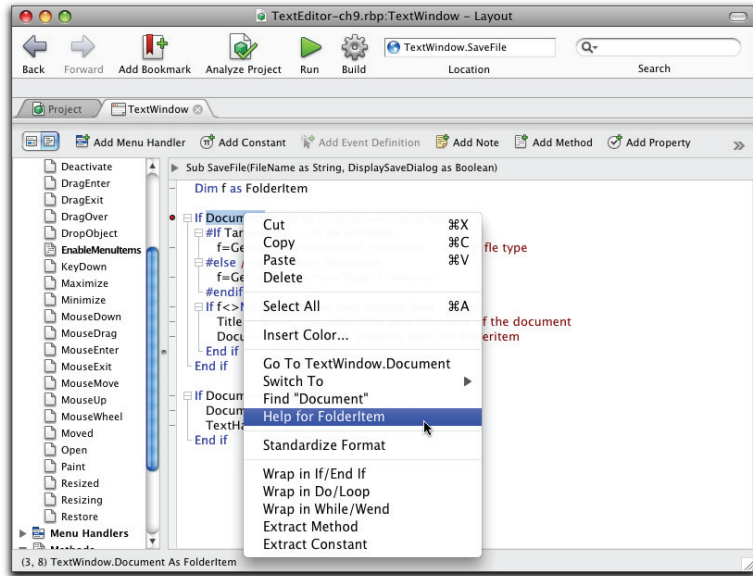
Figure 3. Search results in the Online Reference.



Context-Sensitive Help

The contextual menu in the Code Editor has a menu item for help. Highlight a command in your Code Editor for which you want help and then choose “Help for...” from the contextual menu. In the following example, the user has right+clicked (Control-click on Macintosh) on the term “FolderItem” in the Code Editor. The contextual menu appears and offers to look up this term in the Online Reference.

Figure 4. Looking up a term from the Code Editor.



Context-Sensitive Error Messages

When you attempt to compile REAL Studio code, the application first analyzes your project for syntactical errors and other issues that may cause the application to behave unexpectedly. If it finds a problem, it opens a new tab in the IDE called “Issues” and describes the problem it found. For more information about REAL Studio’s error checking, see the section “Analyzing the Project” on page 632.

Using the HyperText Links in the Online Help

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

Using the Code Examples

The Online Reference contains many code examples that you can use in your projects. You can copy and paste the examples into your Code Editor.

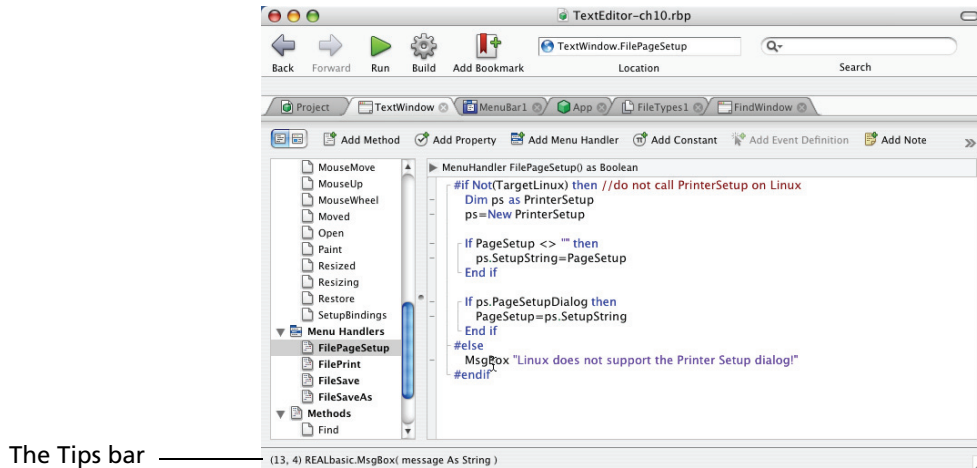


Some code examples omit irrelevant code for the sake of clarity. Omitted code is indicated by dots. Insert your own code in its place. Also, some examples include a Sub or Function statement, which is added automatically by the Code Editor. If you copy and paste such an example into the Code Editor, you need to remove the duplicate statements.

Using Tips

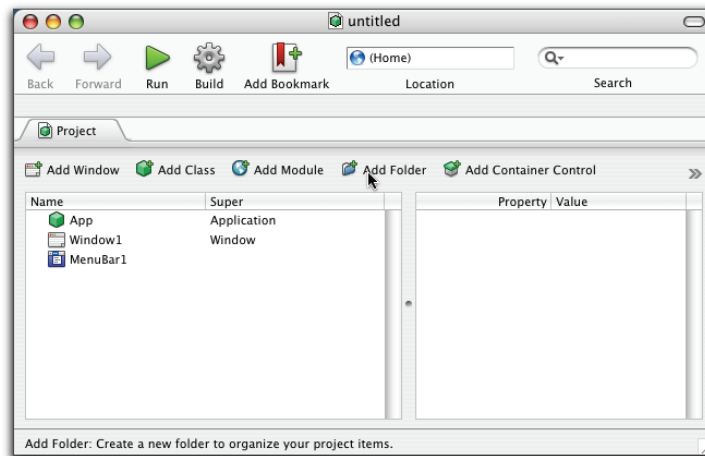
As you work, the REAL Studio IDE “watches” your activities and displays hints in the Tips bar. This is the bar at the bottom of the IDE window.

Figure 5. The Tips bar.



For example, as you write code, tips remind you of the correct syntax for the commands you are entering. This is shown in Figure 5. The insertion point is in the MsgBox command and the Tips bar shows its syntax. Also, the text in the Tips bar changes as you move the mouse around the IDE window. Simply place the mouse over an object to get information about it. For example, in Figure 6 the mouse pointer is over the Add Folder item in the toolbar and the Tips bar shows information on this item.

Figure 6. Help on the Add Folder item.



Electronic Documentation

All of the REAL Studio documentation is available at our web site (<http://www.real-software.com>). These documents are available in PDF (Adobe Acrobat) form.

You can purchase printed copies of the documentation from REAL Software for an extra charge.

Our Support Web Page

Our support page is located at <http://www.realsoftware.com/support>. This is the place to check for information on REAL Studio. You'll find information on customer service, technical support, and links to mailing lists and news groups.

End User Web Sites

There are dozens of web sites created by other users dedicated to REAL Studio. Check our web site for links to these sites.

REAL Studio Developer

REAL Studio Developer (<http://www.rbdeveloper.com>) is a magazine devoted to REAL Studio programming tips and techniques. REAL Studio Developer publishes articles by both experienced REAL Studio users and the authors of REAL Studio.

REAL Studio third-party Books

The REALsoftware web site contains information on third-party books, web sites, magazines and other resources that offer support. Check for the most current information at <http://www.realsoftware.com>.

Our Internet Mailing Lists

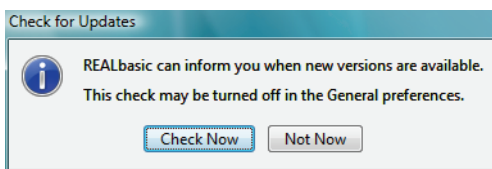
We sponsor several Internet Mailing lists that give you the opportunity to ask questions, and share information with other REAL Studio users via email. For more information on the available Internet Mailing Lists, see our support page at www.realsoftware.com/support.

Obtaining Updates

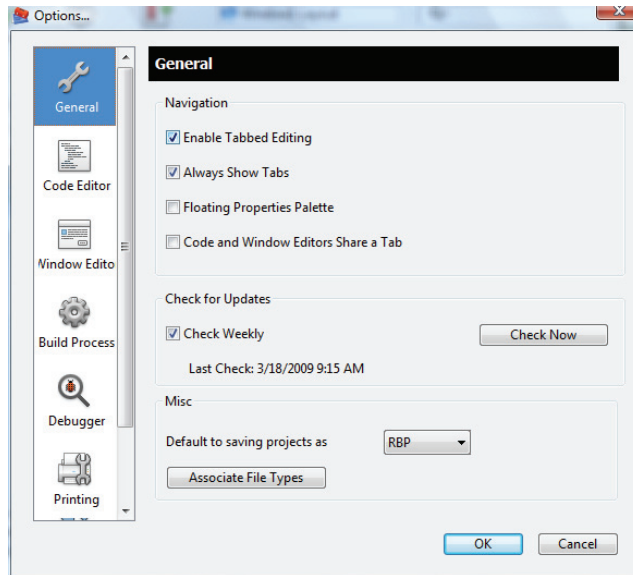
REAL Software provides new releases of REAL Studio no more than three months apart. A new release is available to all users who have a current license.

REAL Studio contains an option to check for a new release automatically. By default, this option is selected. If you wish, you can deselect this preference or check for a new release whenever you wish. When you first launch REAL Studio, it asks whether you would like to check for updates immediately.

Figure 7. The Check for Updates message box.



This option can also be turned on in the General preferences screen in the options dialog box. Choose Edit ► Options (REAL Studio ► Preferences on Macintosh), select General Preferences, and then click Check Weekly. You also have the option in General Preferences of checking for updates immediately

Figure 8. Checking for REAL Studio updates.

Deselect the Check Weekly checkbox if you don't want automatic updates.

The new release can also be downloaded from REAL Software's home page (<http://www.realsoftware.com>).

Technical Support from REAL Software

REAL Software provides free email support for all users who have a current license. Older versions of REAL Studio do not qualify for free email support. See the technical support page on our web site for information about our technical support policy.

We also have Developer Program that is available at an extra charge. It entitles you to priority technical support. See the support page at our web site (<http://www.realsoftware.com/support>) for more information or call us at 512-328-7325.

If you purchased REAL Studio from an international distributor, you need to contact your distributor for technical support. You will find a list of all of our international distributors on our web site at <http://www.realsoftware.com>.

Contacting REAL Software

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

Phone	512 328-REAL (512 328-7325) from 9AM to 6PM Central Time, Monday through Friday
Fax	512 328-7372
email	Submit via REAL Studio Feedback (Help ► REAL Studio Feedback)

Mail	REAL Software, Inc. PO Box 162181 Austin, TX 78716
------	--

Reporting Bugs and Making Feature Requests

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

You can access Feedback directly from within REAL Studio. Just choose Help ► REAL Studio Feedback and your default web browser will open the Feedback page.

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

Getting Started with REAL Studio

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

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

Contents

- Concepts
- The Integrated Development Environment (IDE)
- Working with projects

Concepts

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

Applications are Driven by Events

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

Now that desktop computers use a graphical user interface, users have a far more intuitive way to interact with applications. However, one thing hasn't changed: applications are still driven by events. The difference is that back in the old days there were very few events the application had to worry about responding to. The old-fashioned application was always in a modal state: It only had to respond to the limited number of choices it presented to the user. With a graphical user interface, many more choices and ways of interacting with the computer are available. The user might choose a menu item, click on a button, or type in a field. Also, the applications themselves may cause events to occur that were not directly caused by the user. For example, when a window opens, an event occurs (the window opened). When a window is moved or resized, an event occurs.

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

Developing Software with REAL Studio

If you have written computer programs using traditional programming languages, you already know that the process of development is three steps: write some code, compile the code (turning the code into something the computer can really understand), and test your application. When you find a problem in your application, you start the process over again. Developing software applications with REAL Studio isn't much different than that. The big difference is how often you go through this process. Compilers for traditional languages can take several minutes or more to

compile an application before you can begin testing. Consequently, you spend a lot of time writing code before compiling to avoid waiting for the compiler. REAL Studio's compiler is so fast that you will find you can make a small change to your code and immediately run it to make sure the change you made works as expected. You can also ask REAL Studio to check your code for errors before you even try to compile it.

Like traditional programming language compilers, REAL Studio's compiler will stop if it finds a syntactical error in your code and inform you what the error is so you can fix it. But unlike traditional compilers that require you to track down the line of code where the error occurred, REAL Studio's compiler takes you right to the point in your source code where the error occurred.

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

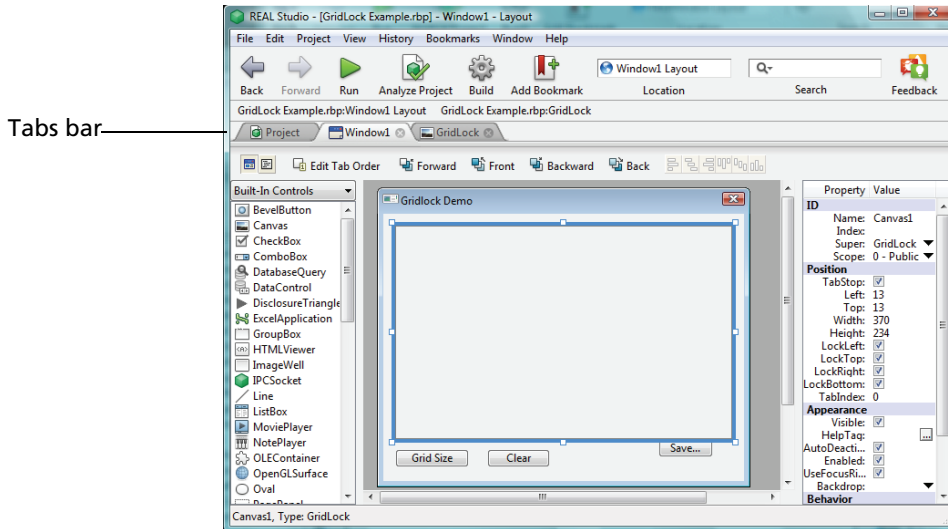
The Development Environment

REAL Studio is an *Integrated Development Environment* (IDE). This means that it contains everything you need to build an application. An interface builder, code editor, compiler, and debugger are all integrated into one package. In traditional programming languages, these items would each be a separate application.

The REAL Studio user interface is extremely configurable. By default, all components of your application are organized into a single window, the *IDE window*. With a single IDE window, you can browse among project items by clicking on tabs at the top portion of the window. You can also open more than one project at the same time; each will be shown in its own window.

Figure 9 on page 34 shows an example of the user interface configured for a single IDE window.

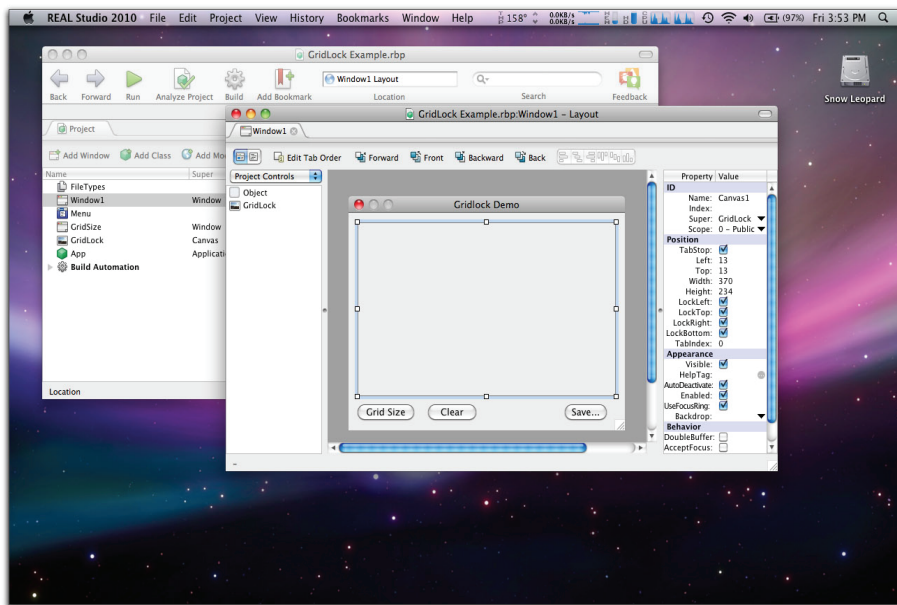
Figure 9. A project organized into one IDE window.



In Figure 9, editors for the components are open and they are identified in the Tabs bar. The tab for the “Window1” item is in front and its editor fills the main section of the screen. You move among the components by clicking their tabs.

The user interface can also be configured to open some or all components of your application in separate windows. If you prefer to move among editors without displaying and hiding each one sequentially, then you will like this configuration. For example, the following illustration shows the editor panel in Figure 9 in its own window.

Figure 10. A Window Editor open in a separate window.



Notice that its tab has been removed from the main IDE window in the back. You can open additional editors in their own windows or configure REAL Studio to open all editors in separate windows.

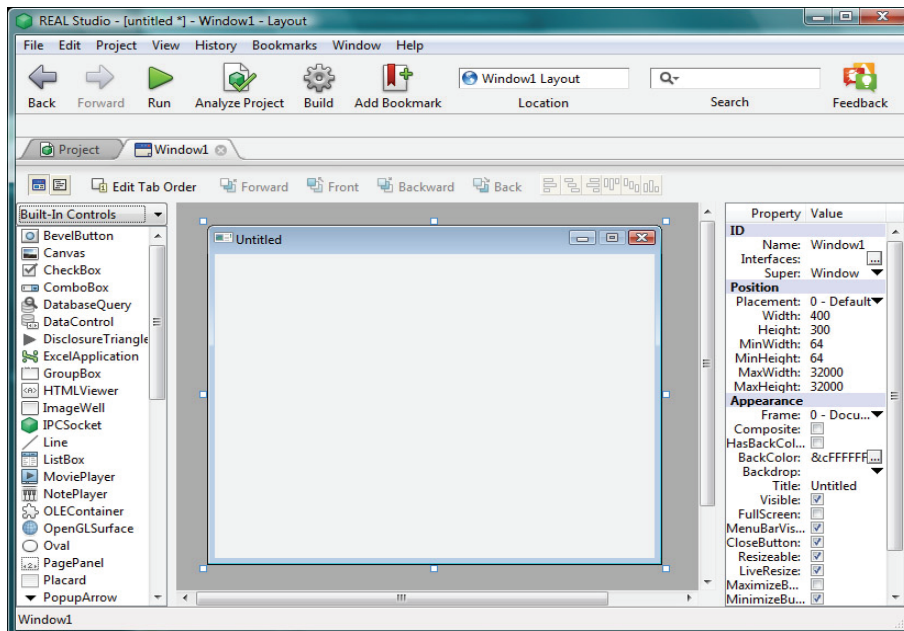
You will learn more about these and other interface options later in this chapter.

The REAL Studio IDE Window

The main IDE window organizes all the components of your application into a series of screens. The design should be familiar to anyone who is used to working with an internet browser that has the ability to use multiple tabs. This includes Internet Explorer 7 on Windows, Firefox on all platforms, and Safari on Windows and Macintosh.

When you start REAL Studio, the main IDE window appears. Two tabs are open, the Project Editor and the Window Editor for the default window. The Window Editor is in front.

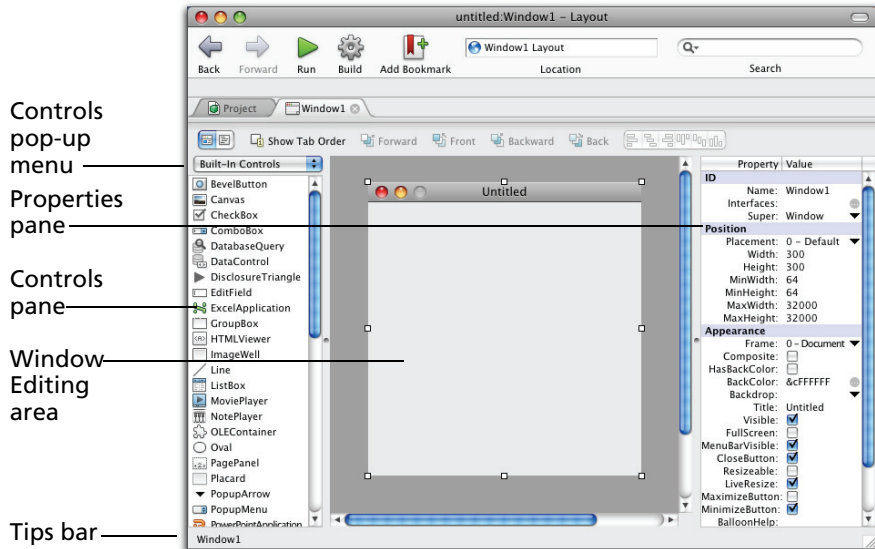
Figure 11. The REAL Studio Window Editor.



The Window Layout Editor

You use the Window Layout Editor to design each window in your project. To access the Window Layout Editor for a window, double-click the window's name in the Project Editor. By default, REAL Studio adds a new tab to the Tab bar and displays the editor for that window in that screen. The Window Layout Editor for the default window appears as shown in Figure 12.

Figure 12. An example window displayed in its Window Editor.



Notice that the Tabs bar now has two tabs. You can return to the Project Editor by clicking on its tab. Also, the Location field has changed to show the name of the editor. For a Window Editor, the name of the screen consists of the name of the window followed by the word “Layout.” You can always navigate to this screen by entering its name in the Location area, followed by “Layout”. To navigate to a control on the layout, use the format:

WindowName.ControlName Layout

This Window Editor can be closed by clicking its tab’s close box, but the Project Editor cannot be closed.

The window being edited is in the center of the screen and its properties are shown in the Properties pane on the right. These are the same properties as you saw from the Project Editor when you clicked on Window1 item in the Project Editor.

The dividers on either side of the editing area can be dragged to the left or right to resize the editing area within the IDE window. When the mouse pointer is over a resizing hot point in a divider (indicated by a dot in Figure 12), it changes to a resizing pointer with arrows pointing to the left and right. Drag to resize the editing area. Resizing is shown in Figure 16 on page 41.

The window editing area has horizontal and vertical scroll bars that enable you to view different areas of the window without resizing the IDE window or resizing the editing area. You can also reposition the window in the editing area by dragging its title bar.

You can configure REAL Studio to open editors in their own windows. If you have selected this option, double-clicking the “Window1” item in the Project editor



opens the editor shown in Figure 12 in its own window. For more information on this option, see the section “Configuring the IDE for Multiple Windows” on page 51.

The Controls Pane

The list on the left side of the Window Editor screen shows the names of REAL Studio’s built-in controls. *Controls* are interface elements such as buttons, checkboxes, text entry fields, lists, tab panels, pop-up menus, and movie players.

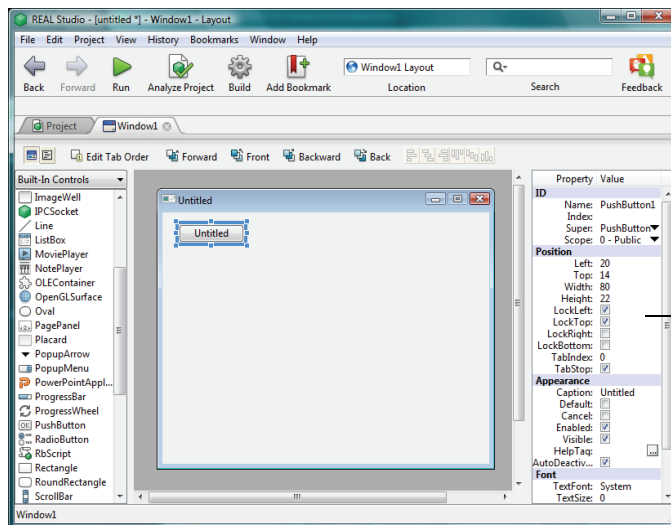
You use the Controls pane to add controls to the window that you are designing. There are several ways to add a control to a window:

- Double-click on the control,
- Drag a control from the Controls pane to the window editing area,
- Select the control in the Controls pane and then press the Enter key (Return on Macintosh),
- Select the control in the Controls pane and then drag an area in the Window Editing area in the location, size, and shape that you want.

When you add a control to a window, an instance of that type of control appears in the Window editing area and the Properties pane changes to show the properties for that control.

For example, in Figure 13, a PushButton control has been added to the window. Its resizing handles indicate that it is selected.

Figure 13. A PushButton control added to Window1.



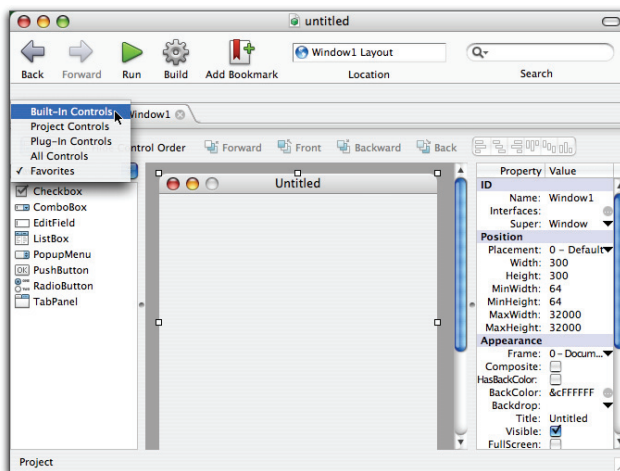
The Properties pane now shows the properties of the PushButton

Since the PushButton control is selected in the Window Editor, the Properties pane has changed to show the properties of the PushButton. To redisplay the window’s properties, deselect the PushButton by clicking on the surface of the window.

The Controls Pop-up Menu

The pop-up menu above the list of controls enables you to populate the Controls list with one of three types of controls, all the controls, or a subgroup of the controls:

Figure 14. The Controls pop-up menu.



The Controls pop-up menu offers the following choices:

- **Built-in Controls:** The controls that are built into REAL Studio. This is the default choice. The built-in controls are shown in Figure 13. For more information on REAL Studio’s built-in controls, see the section “Interacting with the User Through Controls” on page 113.
- **Project Controls:** Custom controls that are based on built-in controls. Project controls are also listed as items in the Project Editor. For information on creating custom controls, see the section “Understanding Subclasses” on page 533 and the procedures for creating subclasses based on controls in Chapter 10 on page 531.
- **Plug-in Controls:** Controls that you add to REAL Studio by installing plug-ins. Third-parties can market custom controls in the form of plug-ins that are installed by placing the plug-in in the Plugins folder in the REAL Studio folder. This list is empty if you have no third-party plug-in controls installed.
- **All Controls:** The built-in, project, and plug-in controls in one alphabetized list.
- **Favorites:** Controls from any of the three types of controls that you have marked as Favorites. REAL Studio ships with a selection of the most frequently used controls as Favorites. Those controls are shown in Figure 14. For more information, see the section “Favorites Controls” on page 114.

The current selection has a checkmark to its left.

The Project Editor

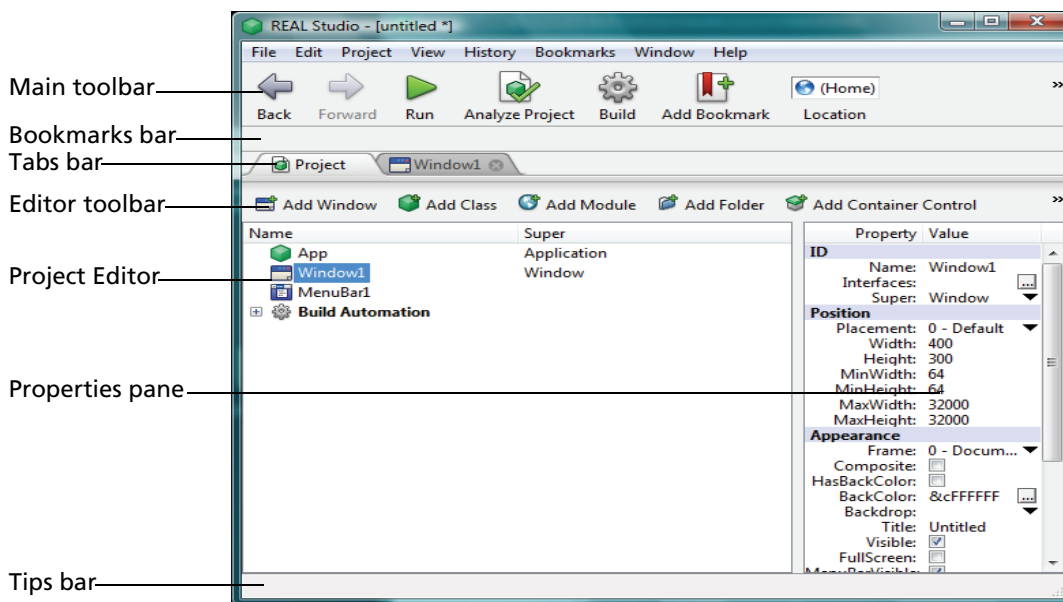
A *project* is the collection of items that make up the application you are developing. The Project Editor organizes all the major components of the application. You add these items to the application from the Project and you can access each item from this screen.

For example, each of the windows that makes up your application will be listed in the Project Editor. Some of the other items that might be listed in the Project Editor are classes, modules, menubars, pictures, sounds, databases, and movies. You will learn about these items in later chapters.

You use the Project Editor toolbar or the Project ► Add submenu to add items to the project. You will learn more about projects in the next chapter.

The Project Editor displays a list of these elements to give you easy access to them. You click on a project item to go to an editor for that item or, if there is no editor, a viewer.

Figure 15. The REAL Studio Project Editor window.



The main IDE window resembles an internet browser window. The Main Toolbar contains controls for navigating among parts of your project. The Back and Forward buttons allow you to redisplay previously viewed screens, the Location field lets you go to an item by name, and the Search area lets you search for specific objects throughout your project.

With the Add Bookmark button, you can add frequently used items to the Bookmarks menu or the Bookmarks bar. You can navigate to bookmarked items

simply by choosing its name from the Bookmarks menu or clicking on it in the Bookmarks bar.

The History menu tracks the series of screens that you have worked on. You can redisplay a screen by choosing its name from the History menu.

You can open an item listed in the Project Editor by double-clicking it. Its editor (or viewer) appears in the IDE window and a tab for that editor is added to the Tabs bar. The Tabs bar contains a tab for each editor that is open. You can view any open editor by clicking on its tab.

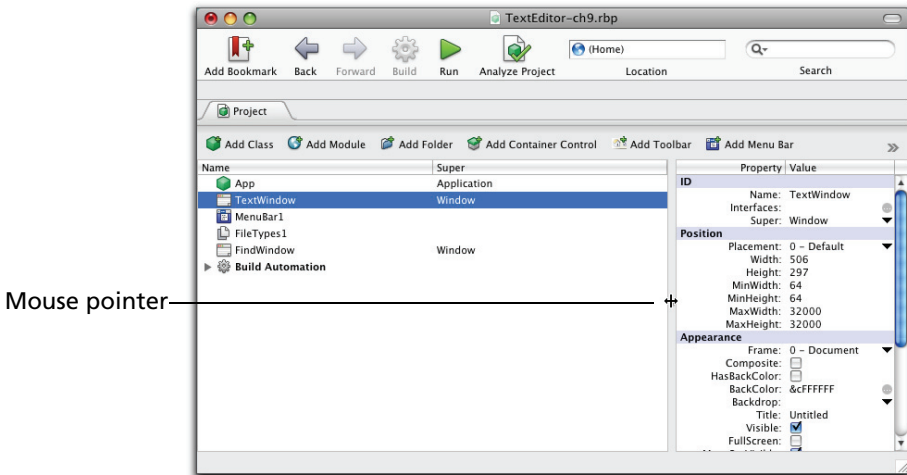
Just below the Tabs bar is a toolbar that belongs to the editor that is displayed. You use it to add or manipulate items in the editor you are viewing. The Editor toolbar changes depending on which editor you are on.

Editor Panes

Each Editor screen is divided into two or more *panes*. In the case of the Project Editor screen, it is divided into the *Project Editor* pane and the *Properties* pane. Panes are separated by dividers that can be moved to the left or right by holding down the mouse button on the divider and dragging to the left or right.

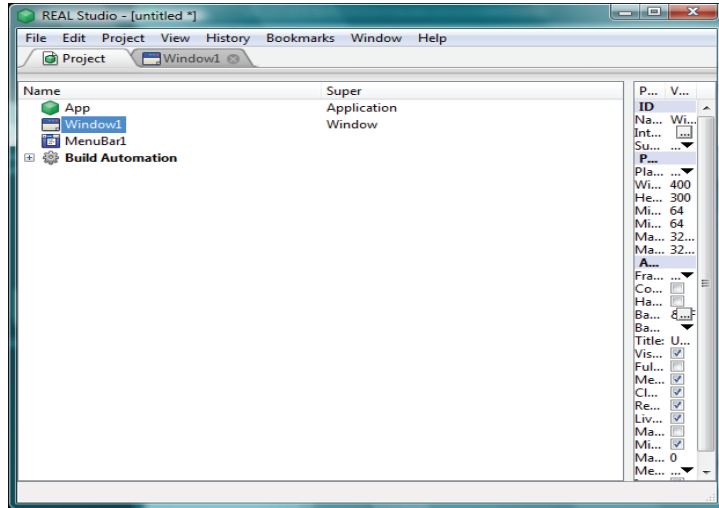
To resize the panes, move the pointer to the divider until the mouse pointer changes to an Arrow pointer that points to the left and right. Then hold down the mouse button and drag in either direction.

Figure 16. The Properties pane being resized.



You can increase the usable area in an editor by selecting the View ► Editor Only menu command. This menu command hides the Main toolbar, the Bookmarks bar, and the Editor toolbar, leaving only the Tab bar above the editor. It also minimizes the space taken up by secondary panes in the editor area without hiding them completely.

Figure 17. The Project Editor after choosing Editor Only.



When the Editor Only menu command is selected, REAL Studio places a checkmark to its left. Selecting the Editor Only command again changes the IDE to its previous configuration.

Another way of increasing the area of the IDE window that is devoted to the editor is to move the Properties pane out of the window entirely. For information on this option, see the section “Displaying the Properties pane in its own window” on page 44.

The following sections give an overview of each type of Editor screen.

The Properties Pane

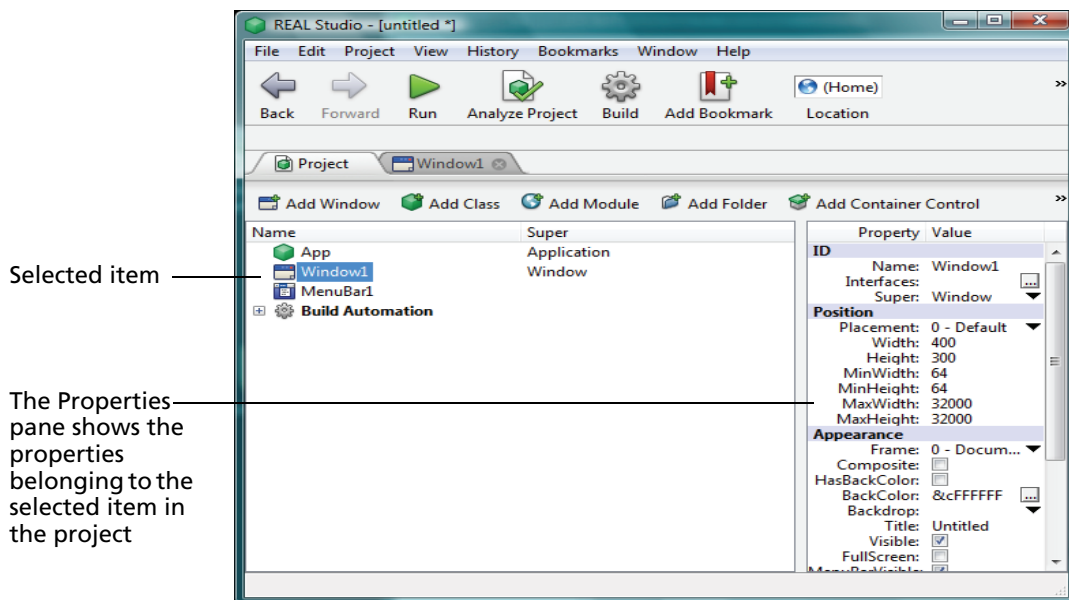
When you select an item in the Project Editor by clicking on it, the Properties pane (the pane to the right of the divider) shows properties that belong to the selected item. For example, in Figure 18 on page 43 the `Window1` item is selected in the Project Editor, so its properties are shown in the Properties pane.

Properties are values that are owned by an item. They characterize the item. Some examples of items that have properties are a window, a menu item, and a control in a window. For example, a window has a `Title` property that holds the text that is shown in the window’s Title bar. A window also has `Width` and `Height` properties that store the window’s size. A window’s `Left` and `Top` properties describe the position of its top-left corner.

The Properties pane displays all of the properties that *can be modified in the IDE* for the currently selected item. This is an important point because some objects have properties that can be modified only by your programming code. An item may also have properties that cannot be modified or can be modified only from the IDE. The appearance of the Properties pane depends on which object is selected.

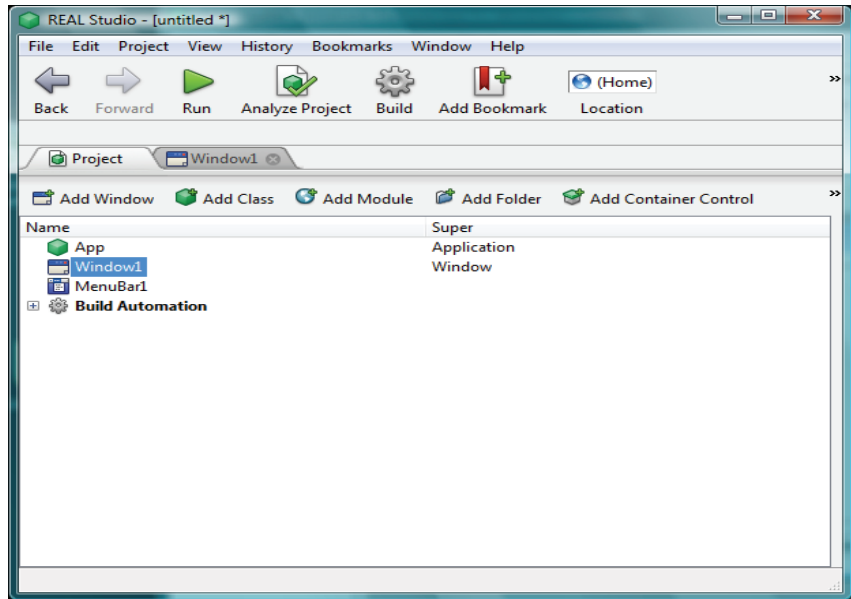
For example in Figure 18, Window1 is selected (highlighted) in the Project Editor and the Properties pane shows various properties of this window, such as its name, frame type, height, and width. For more information about the Properties pane, see the section “Using a Window’s Properties Pane” on page 100 and “Changing a Control’s Properties with the Properties Pane” on page 126.

Figure 18. The Project Editor with Window1 selected.



If you want to maximize the area of the IDE window devoted to the editor, you can choose **Window ► Hide Properties**. This menu command removes the Properties pane from the display, allocating all the horizontal space to the editor.

Figure 19. The Project Editor after choosing Hide Properties.

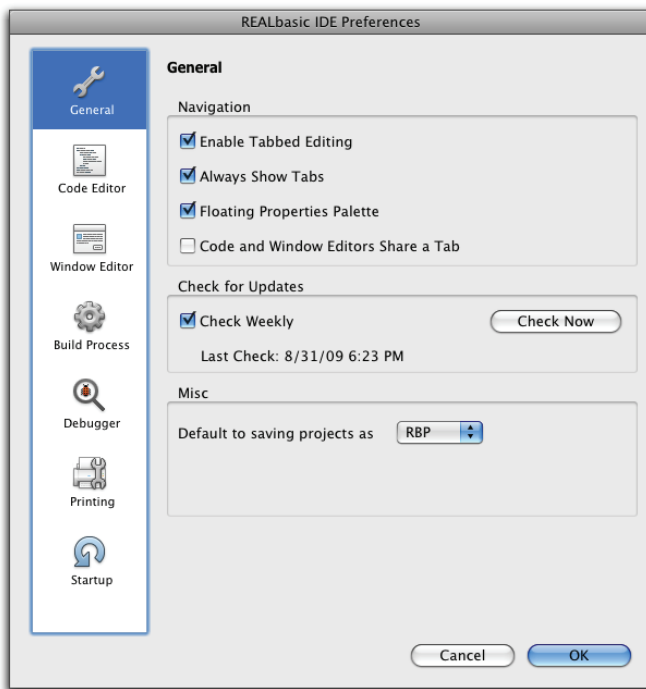


When you choose Hide Properties, the menu command changes to Show Properties. Choose Show Properties to revert to the previous display. The Show Properties command remembers the previous position of the divider.

Displaying the Properties pane in its own window

By default, the Properties pane is shown on the right side of each editor's panel. If you wish, you can display the properties in a separate floating window.

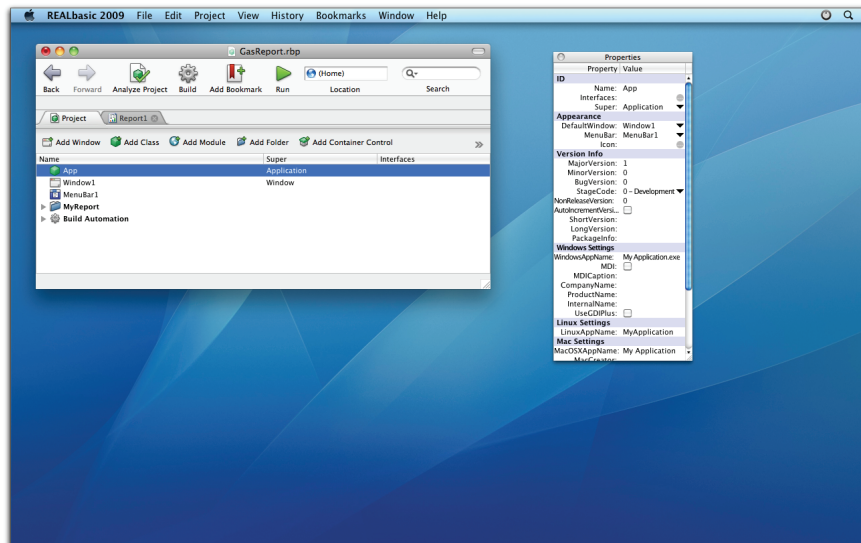
To do so, choose Edit ► Options (REAL Studio ► Preferences on Macintosh) to display the Options dialog box (Preferences dialog on Macintosh). Select the General topic and select the Floating Properties Palette option.

Figure 20. General Preferences with the Floating Properties Palette option selected.

With this option selected, the Properties pane for all the editors will appear in a separate floating window.

Figure 21 shows the Project Editor of a project with the Floating Properties Palette option selected. The Properties palette shows the properties of the Window1 item, since it is selected in the Project Editor. The Tabbed Editing option is also selected.

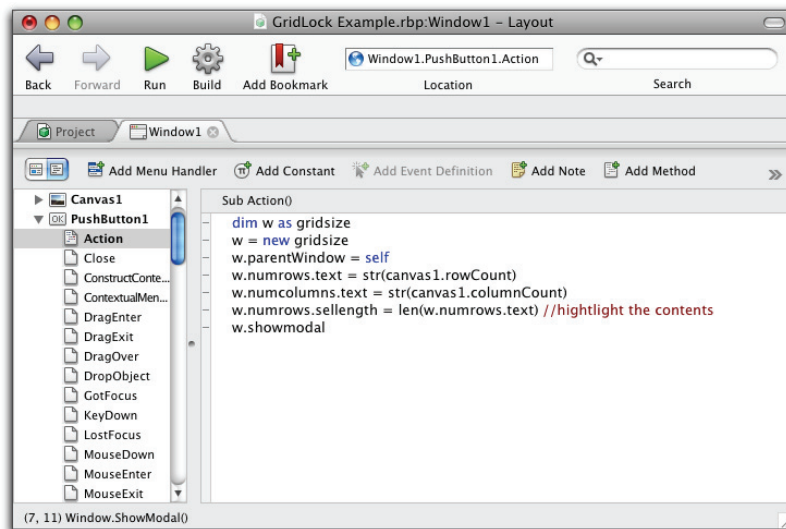
Figure 21. The Project Editor with a floating Properties palette.



The Code Editor

Use the Code Editor to add programming code to items in your project, such as controls, classes, and windows. The Code Editor has a browser area that makes it easy to locate the object to which you want to add code. The Code Editor is shown in Figure 24.

Figure 22. The Code Editor for a control in a window.



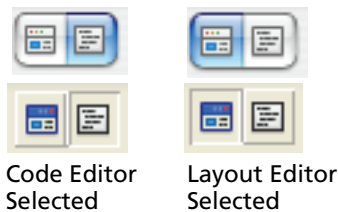
This Code Editor is for a window named Window1. It currently displays code for a PushButton control in that window.

Each Window Editor has two views: The Layout editor view (for example, Figure 13 on page 38) and the Code Editor view (for example, Figure 22). As you build your application, you move back and forth between these views. You design your interface with the Layout Editor view and then add code to windows and controls in the window by switching to the window's Code Editor.

You switch between the Code Editor and Layout Editor views using the View ► Show Code or View ► Show Layout menu commands or with the Edit Mode buttons in the editor's Toolbar. By default, these buttons are shown on the extreme left of the Window Editor toolbar and are illustrated in Figure 23 (The Window Editor toolbar can be customized; for more information, see the section "Customizing the Window Editor Toolbar" on page 105).

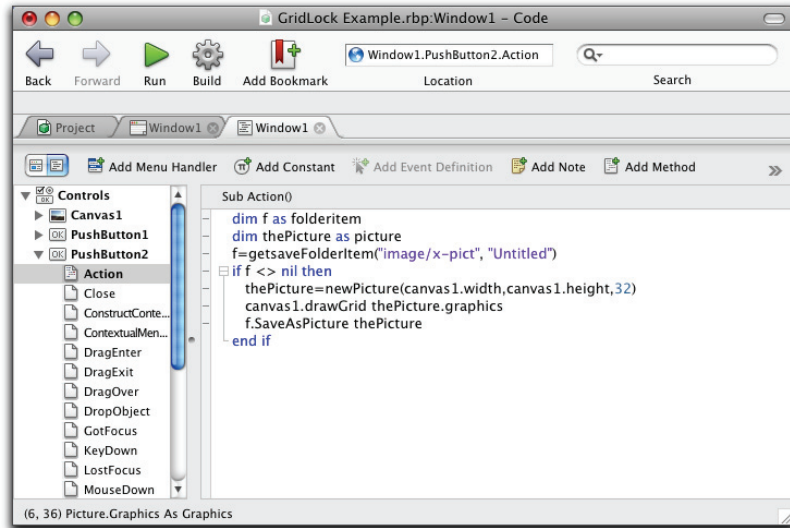
On Macintosh and Linux, the selected icon has the system highlight color; on Windows the selected icon is in its depressed state.

Figure 23. The Edit Mode buttons.



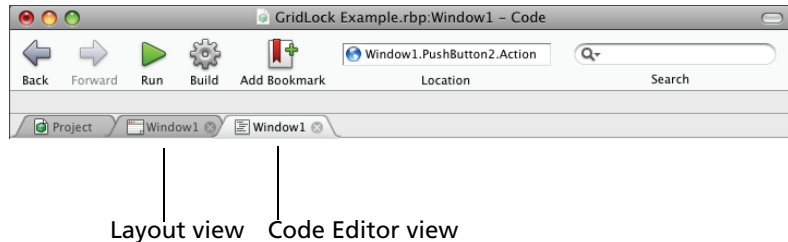
You can configure the REAL Studio IDE so that each window's Layout and Code Editors get their own tab in the Tabs bar. For example, in Figure 24 the two views for Window1 both have tabs in the Tabs bar. The Code Editor view is in front and the Layout Editor tab is in the middle of the Tabs bar. They are both identified by the name of the window and are distinguished by the small icon in the tab. The Layout Editor small icon has a mini Title bar, while the small icon for the Code Editor view depicts lines of code only. With this configuration of the Tabs bar, you can switch to either the Layout or Code views from the Tabs bar.

Figure 24. The Code Editor for a control in a window with its own tab.



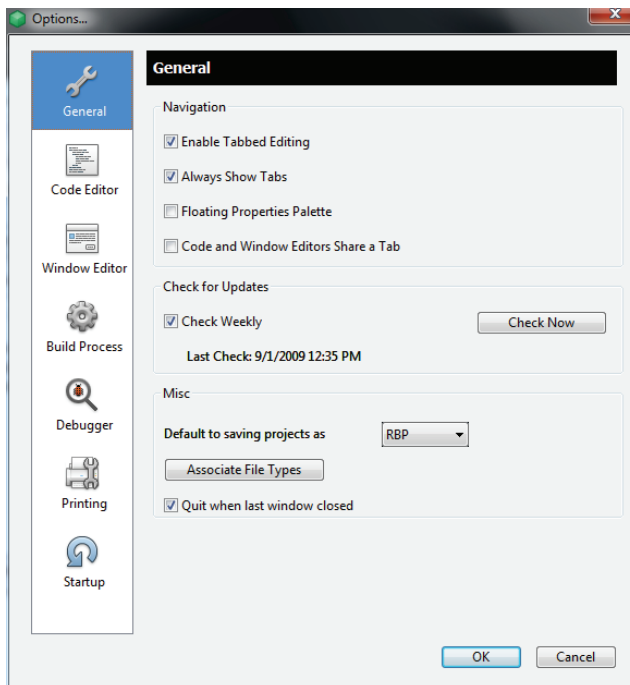
A Tabs bar with both Window Editor tabs is shown in Figure 25.

Figure 25. A Tabs bar with both the Layout view and Code Editor views for Window1.



At that point, both views belonging to the window are available from the Tab bar. If you are working with another editor, you can go to either view directly by clicking its tab.

You can select this configuration in the Options dialog box (Preferences on Macintosh). Choose Edit ► Options (REAL Studio ► Preferences on Macintosh), select the General topic, and deselect the Code and Window Editors Share a Tab preference.

Figure 26. Configuring the IDE for separate Window Editor tabs.

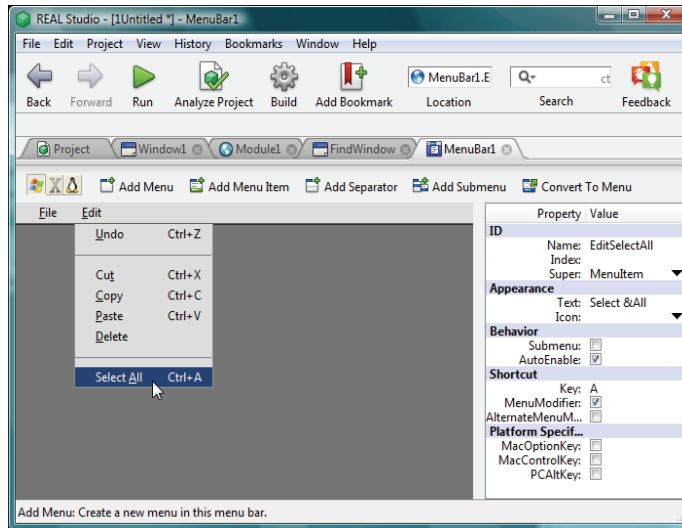
If this option is selected, each window gets one tab in the Tabs bar and the tab for the window gets the Layout Editor small icon for both the Layout and Code Editor views. This configuration of the IDE window was the only option available prior to REAL Studio 2006 Release 3.

If the Code Editor and Layout Editor get their own tabs, then the Edit Mode buttons perform the same functions as the Code and Layout Editor tabs.

For a detailed description of the Code Editor, see the section “Using The Code Editor” on page 273.

The Menu Editor

You use the Menu Editor to create the menus and menu items that will be displayed when your application runs. The Menu Editor is shown in Figure 27.

Figure 27. The Menu Editor.

When you click on a menu or menu item in the Menu Editor, the Properties pane changes to show the properties of the item. The Text property of the menu item is the text that is displayed by the menu item. You can also assign keyboard shortcuts to menu items and even create sub-menus (a menu item that is actually just another menu). REAL Studio adds File, and Edit menus for you by default. For Mac OS X, it also adds the Apple and Application menus.

The row of icons on the left side of the Menu Editor toolbar allows you to preview the look of the menu system under every possible operating system on which you can build the application. Click on one of the View Mode buttons to preview the menu system on that platform.

Figure 28. The Menu Editor's View Mode buttons (Linux selected).

Configuring the IDE for Multiple Windows

By default, the Tabs bar is populated with tabs for each editor as you open new editors. The tabs appear in the order that you open the editors. This mode is called *Tabbed Editing*. The entire project and all of its editors is organized into one window. All of the editors for the project are accessed via tabs in the Tabs bar. Also, the Debugger appears as a tabbed editor window. For more information about the Debugger, see Chapter 12, “Debugging Your Code” on page 631.

If you wish, you can configure the REAL Studio IDE so that each editor appears in its own window. In this case, clicking an item in the Project Editor opens its editor (or viewer) in its own window and each Tabs bar only has one tab. You do this by turning Tabbed Editing off.

To turn off Tabbed Editing, choose Edit ► Options (REAL Studio ► Preferences on Macintosh) to display the Options dialog box (Preferences on Macintosh). Select the General topic and deselect the Enable Tabbed Editing preference. This turns Tabbed Editing off globally; each IDE editor will now open in a separate editing window.

Figure 29. Disabling Tabbed Editing in the Options (aka Preferences) dialog box.

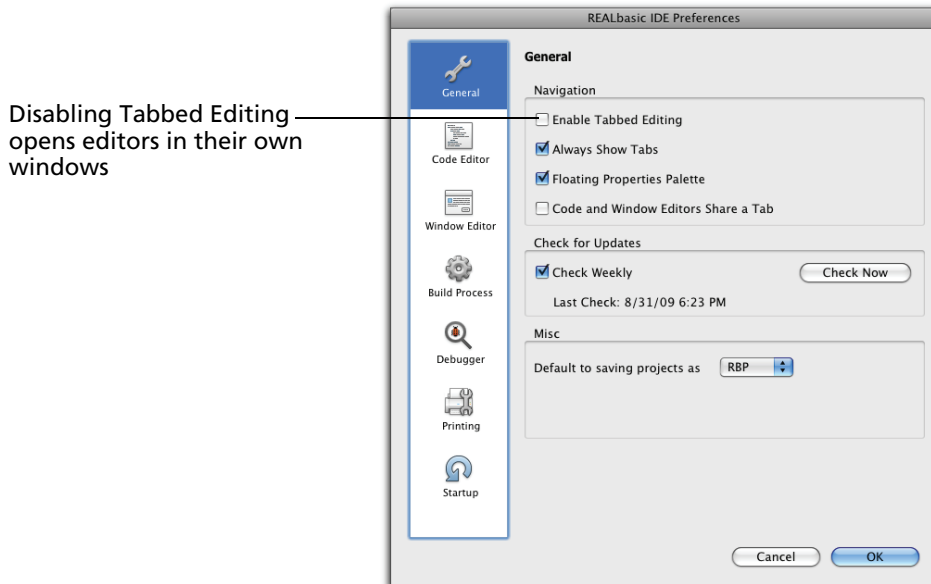
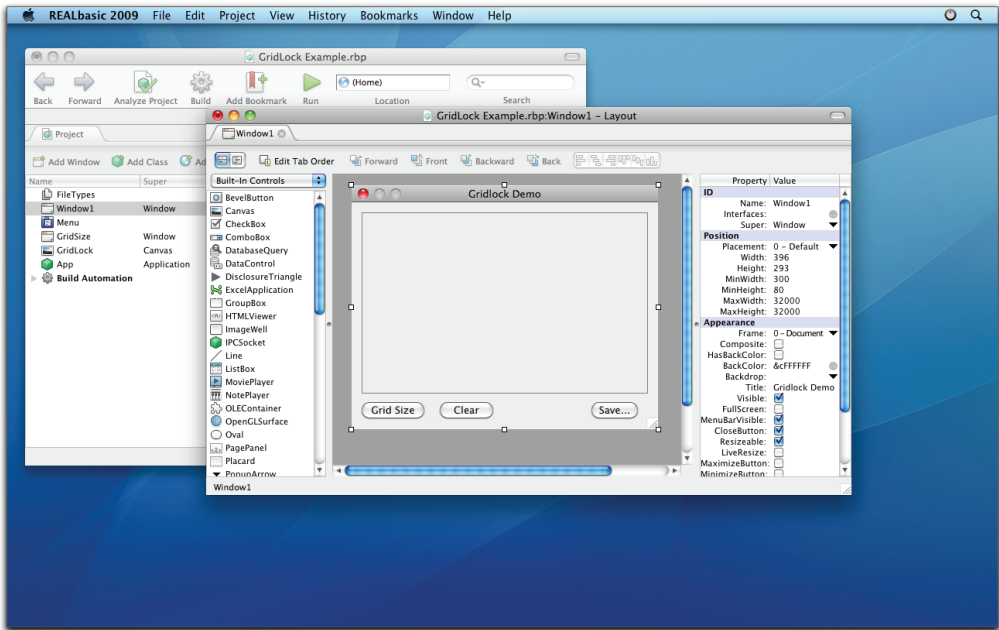


Figure 30 shows the default window in its Window Editor with Tabbed Editing off and the Project Editor is in its own window. The window that contains the Window Editor does not have a main toolbar or a Bookmarks bar.

Figure 30. The Window Editor with Tabbed Editing off.



When you turn Tabbed Editing off, then all Editors are displayed in separate windows. If you want to use the main IDE window for most editors and display only one or some editors in their own windows, then you can instead drag a tab out of the IDE window. For more information, see the section “Dragging a Tab” on page 54.

When Tabbed Editing is on, double-clicking a window in the Project Editor opens the window’s Layout Editor in its own window. Shift-double-click to open the window’s Code Editor in its own window.

You can optionally open an editor in the main IDE window. Hold down the Control key (Command key on Macintosh) and double-click the editor’s name in the Project Editor. Click Shift+Control (Shift-Command on Macintosh) to open the window’s Code Editor in the IDE window.

If an editor is already open in a tab, you can drag that tab out of the IDE window. It will then open in its own window. For more information, see the section “Dragging a Tab” on page 54.

Working with the Tabs bar

Except for the Project Editor, each tab has a close box that you can use to close that screen. Closing a close box only closes the screen; it does not delete the item.

The Contextual Menu

Each tab in the Tabs bar has a contextual menu that has the following menu commands:

- **Open in New Window:** The Editor for that tab opens in its own window and is removed from the current IDE window. You can also do this by dragging the tab out of the IDE window.
- **Close Tab:** The Editor is closed, leaving the other tabs in the Tabs bar. This command is not available for the Project Editor tab.
- **Close Other Tabs:** Closes all Editors except the Project Editor and the current editor. This leaves the IDE window with two tabs, the current tab and the Project Editor tab, or, if the current tab is the Project Editor, only the Project Editor. If the Project Editor tab is the only “other” tab, then this menu item is disabled.
- **Close All Tabs:** Closes all the Editors except the Project Editor.
- **Select Tab:** Displays a submenu with the names of all the tabs, enabling you to choose a tab to move to the front. The name of the current Editor has a checkmark to its left.
- **Show on Disk:** Available only for items that are stored on disk, such as the Project Editor and any external project items. Opens the folder containing the item and selects it.

If you right+click (Command-click on Macintosh) on the Tabs bar itself not a tab, the contextual menu has three items:

- **Close Other Tabs:** Closes all Editors except the frontmost Editor and the Project Editor.
- **Close all Tabs:** Closes all Editors except the Project Editor. If the only tab in the window is not the Project Editor, then it closes the window.
- **Select Tab:** Displays a submenu with the names of all the tabs, enabling you to choose a tab to move to the front. The name of the Editor that is currently in front has a checkmark to its left.

Hiding the Tabs bar

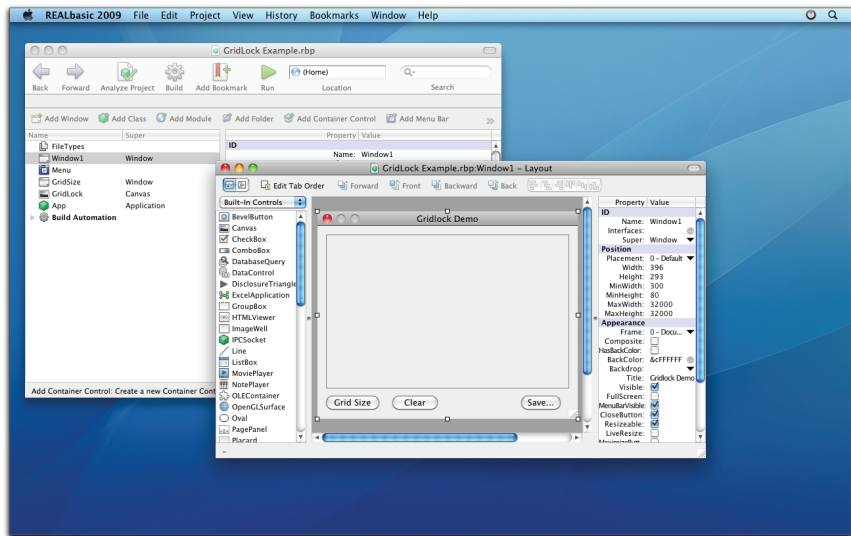
If you like, you can hide the Tabs bar when only one editor is open. When this option is on, the initial view of the IDE would be shown without a Tabs bar. However, when you double-click an item in the Project Editor, a tab for the item is added to the IDE window, assuming Tabbed Editing is on. If Tabbed Editing is off, the editor opens in a new window.

If you use this option in conjunction with the multiple windows option, each editor window will be displayed without a Tabs bar.

To hide the Tabs bar for IDE windows with one editor, choose **Edit ► Options** (REAL Studio ► Preferences on Macintosh) to display the Options dialog box (Preferences on Macintosh). Select the General topic and deselect the Always Show Tabs option. This option shows the tabs bar when Tabbed Editing is on and more than one Editor is open.

In Figure 31, the Tabbed Editing and Always Show Tabs options are both off. As in Figure 32, window1 is shown in its own Layout Editor window, but the Tabs bar is not shown in either the Project or Layout Editor windows.

Figure 31. A project with Always Show Tabs turned off.



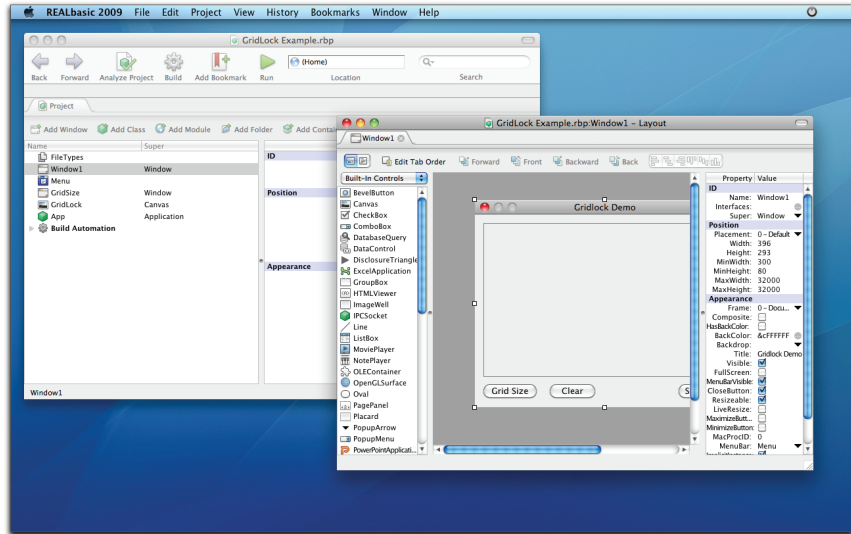
Dragging a Tab

When you have several tabs in the Tab bar, you can rearrange the tabs by dragging a tab to the left or right. As you drag over an existing tab, its text changes to the text of the tab being dragged. Drop the tab to put it in its new position.

When there is more than one tab in the Tabs bar, you can drag a tab out of the IDE window to the desktop. The editor whose tab you drag will be redisplayed in its own window.

For example, in Figure 32, the tab for the default window's Window Layout Editor has been dragged out of the IDE window. It is now displayed in its own window. Note that the project is in Tabbed Editing mode; the main IDE window has tabs for the Menu Editor and another window's editor.

Figure 32. A project after dragging the default window's Window Layout Editor out of the IDE window.



In the case of the Window Layout Editor, you can open the Code Editor for an item in a new window by holding down Ctrl (Command on Macintosh) and double-clicking the item. For example, if you want to display the Code Editor for one of the PushButtons shown in Figure 32, you would Ctrl+Double-click on the PushButton. If you double-click, you will switch to the Code Editor within the same IDE window.

The Main Toolbar

The REAL Studio Main Toolbar has items for navigating among editors in your project, for testing your project, and for building a standalone application. Figure 33 shows the default Main Toolbar.

Figure 33. The default IDE Main Toolbar.

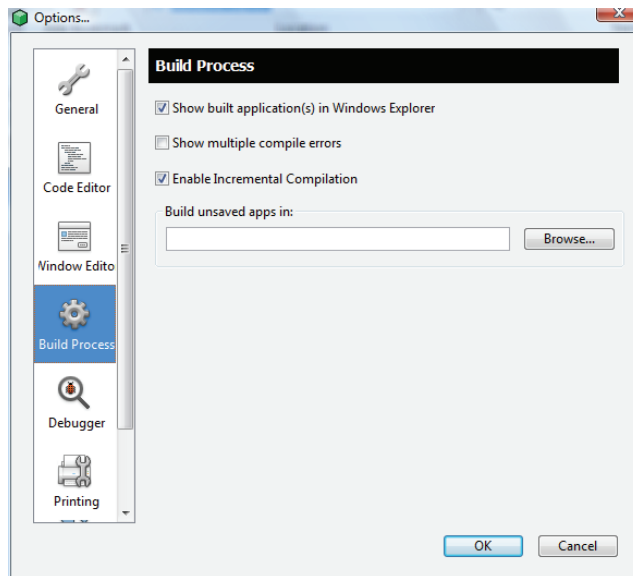


- **Back and Forward buttons:** The Back and Forward buttons work exactly like the Back and Forward buttons in a Web browser; they move backwards and forwards among IDE screens in the order that you viewed them. This list of screens is also displayed as items in the History menu. You can use the History menu to jump to any previously visited page.
- **Run button:** Click the Run button to compile the project and test it. When you click Run, REAL Studio attempts to compile your application. If the project compiles successfully, the application launches in its own window and REAL Studio adds a new screen in the IDE window for real-time debugging that's called the

“Run” screen. Click Stop in the Run screen’s Editor toolbar to halt execution of the application.

By default, REAL Studio saves the compiled application in the Temp folder for your computer’s operating system and launches it automatically. If you wish, you can choose another location in the Build Process panel of the Options dialog box. Choose Edit ► Options (REAL Studio ► Preferences on Macintosh), click the Build Process icon, and then modify the path in the “Build unsaved apps in” field.

Figure 34. The Build Process Options screen.



You can switch to another editor in the IDE and continue working on your application without quitting the debugging application, but your changes won’t be reflected until you stop the debugging application and rerun it.

If REAL Studio finds any syntax errors, the attempt to compile and launch the application will stop and REAL Studio will display the errors in an Issues screen or in the Code Editor in which it found the error. You need to fix all the syntax errors to allow REAL Studio to successfully compile and run the application. For information about checking for syntax errors, see the section “Syntactical Bugs” on page 632.

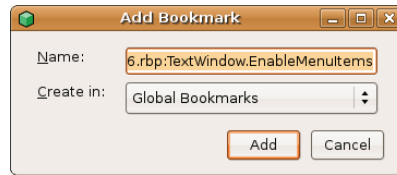
For information on testing and debugging the application within the IDE, see Chapter 12, “Debugging Your Code” on page 631.

- **Build Button:** Click the Build button to build a standalone (double-clickable) application based on the current Build settings. The difference between “Run” and “Build” is that the result of a “Build” is a standalone double-clickable application that doesn’t need the REAL Studio application at all. A “Run” is a test-run of your program that is intended for debugging inside the REAL Studio IDE. See Chapter

15, “Building Stand-Alone Applications” on page 693 for information on Build Settings and building standalone applications. The attempt to build will fail if there are any syntax errors.

- **Add Bookmark button:** Click the Add Bookmark button to add the current item to either the list of Bookmarks in the Bookmarks menu or to the Bookmarks bar. An item can be an entire editor, a method, a property, a control, a file type set, and so forth. This list is displayed in the Bookmarks menu and the Bookmarks bar is just above the Tabs bar and below the Main Toolbar. When you choose this menu item, the dialog box shown in Figure 35 appears:

Figure 35. The Bookmarks dialog box.



Choose Global Bookmarks from the Create In drop-down list to add the item's name to the Bookmarks menu. It's called “global” because the bookmark will appear in all of your projects. Choose Local Bookmarks bar to place it in the current project's Bookmarks bar. Each project has its own Bookmarks bar.

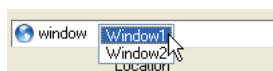
If you have created bookmark folders, they will also be offered as choices in the Create In drop-down list. This enables you to place a new bookmark directly in a folder.

You can navigate directly to a bookmarked item by choosing its name from the Bookmarks menu or clicking on its name in the Bookmarks bar.

- **Location area:** Use the Location area in the same way you use the URL area in a Web browser. Enter the name of an item and press Enter (Return on Macintosh) to go to it. You can enter “(Issues)” to display an open Issues tab, “(Search)” to display an open Search Results list, or “(Super)” to display a class's super class. Please refer to Chapter 10, “Creating Reusable Objects with Classes” on page 531 for information on classes and super classes.

The Location area uses the autocomplete feature of REAL Studio. As you type, REAL Studio attempts to complete the term that you are typing. Its guess is shown in gray. If it has more than one guess, it shows three dots and you can press Tab to see a contextual menu of the possibilities.

Figure 36. Using Autocomplete in the Location area.



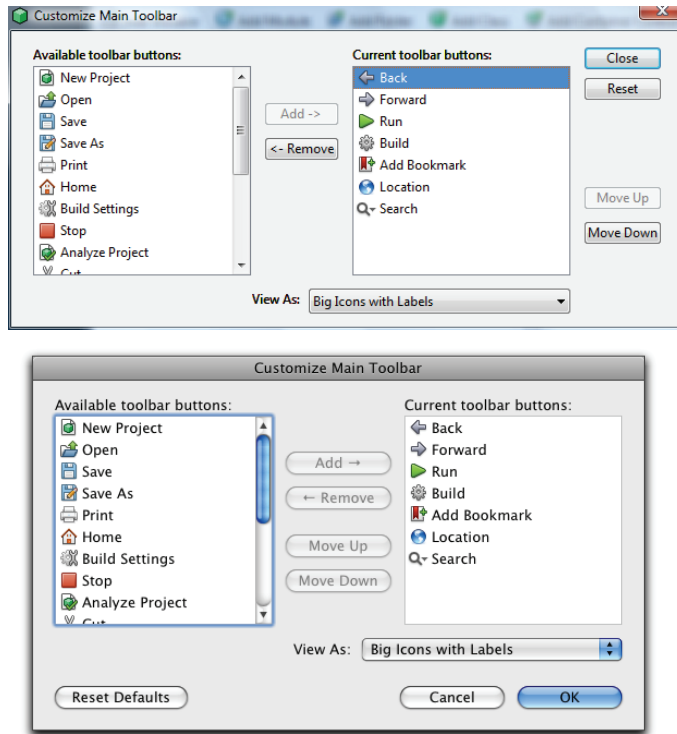
For more information about autocomplete, see the section “Autocomplete” on page 291.

- **Search area:** Use the Search area to find items in your code. A pop-up menu lets you choose among searching the whole project, the current item, or the current method (if applicable). On Mac OS X, it also offers to search your computer using Spotlight.

Customizing the Main Toolbar

If you like, you can add, remove, or reposition the items in the Main Toolbar. To do so, choose View ► Main Toolbar ► Customize. The Customize Main Toolbar dialog box appears.

Figure 37. The Customize Main Toolbar dialog box.



The Customize Main Toolbar dialog box uses a “mover” interface to configure the toolbar. Listed in the right panel are the current items in the toolbar. Listed on the left are optional items that can be added to the toolbar.

The following operations are available:

- To add an item, highlight it in the left panel and click the Add button (shown in Figure 37).
- To remove an item, highlight it in the right panel and click the Remove button. This moves the item to the list on the left.
- To reorder an item, highlight it in the right panel and click either Move Up or Move Down or click the item you want to move, drag it to the desired location, and

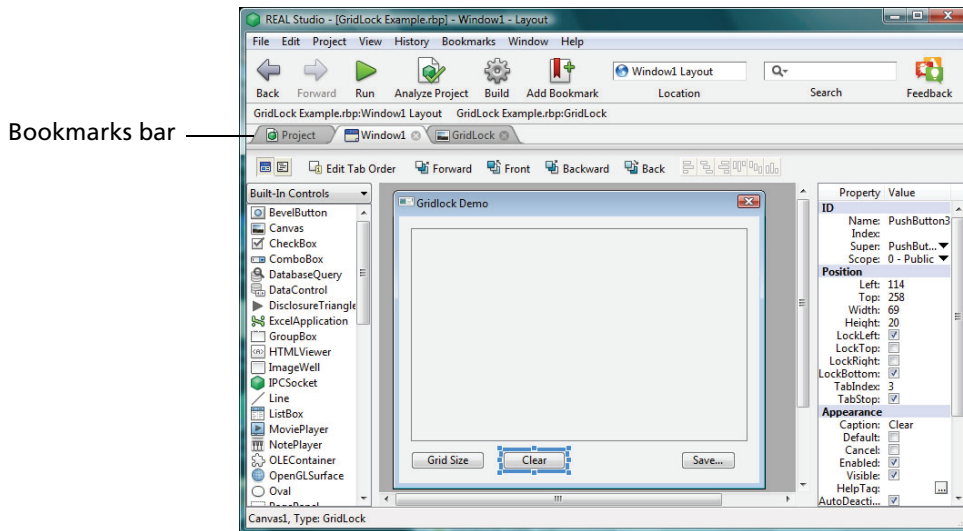
then drop it between two items. The order in which the items are listed is the left-to-right order in the toolbar.

- To change the appearance of the items in the toolbar, choose an item from the Display As drop-down list. Your choices are:
 - Big icons with labels.
 - Small icons with labels,
 - Big icons (no labels),
 - Small icons (no labels),
 - Labels only.
- To reset the toolbar to the default toolbar, click the Reset button (Windows and Linux) or Reset Defaults button (Macintosh).

The Bookmarks Bar

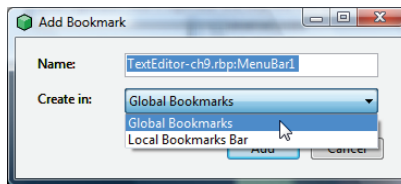
The Bookmarks bar, which is just below the Main toolbar, displays the locations that you have added to the local bookmarked locations. You can go to a bookmarked location simply by clicking on the item in this toolbar.

Figure 38. The Bookmarks bar.



To add an item to the Bookmarks bar, click the Add Bookmark button in the Main Toolbar or choose Bookmarks ► Add Bookmark. The Add Bookmark dialog box shown in Figure 39 appears.

Figure 39. The Add Bookmark dialog box.



By default, the Create In drop-down list offers two choices:

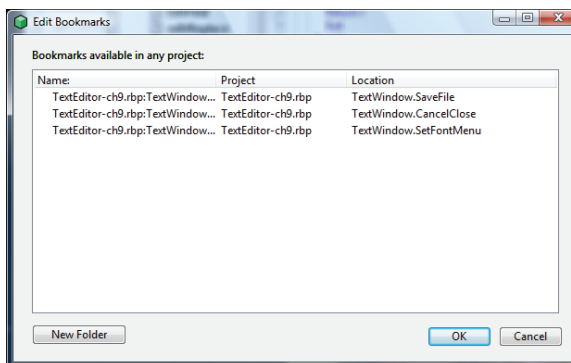
- **Global Bookmarks:** Global bookmarks appear as items in the Bookmarks menu. They are global in the sense that they appear in all of your projects.
- **Local Bookmarks bar:** Local bookmarks appear in the Bookmarks bar in the project's IDE window. When you add a bookmark to the Bookmarks bar, it is local to that project; it does not appear in the Bookmarks bar of your other projects.

You can also add methods and properties to the Bookmarks bar by dragging the name of the method or property from the Code Editor browser list to the Bookmarks bar. When the item to be added to the Bookmarks bar is over the Bookmarks bar, it shows a selection rectangle, indicating that it can accept the dragged item.

If you have created bookmark folders for the Bookmarks menu, they will also be offered as choices in the Create In drop-down list. This enables you to place a new bookmark directly into a folder. When you add a item to a folder, the Bookmarks menu shows the folder's name as a menu item and the items in the folder as submenu items.

You can also modify existing bookmarks. To do so, right+click (Control-click on Macintosh) on the Bookmarks bar to display the Bookmarks bar's contextual menu and choose Customize. The Edit Bookmarks dialog box appears:

Figure 40. The Edit Bookmarks dialog box.



With the Edit Bookmarks dialog box, you can modify the names and locations of your bookmarks. You can edit the name to make it easier to recognize.

Locations use the “dot” syntax to specify the project and Editor, and item names. That is, the syntax is *projectname.editorname.itemName*. If it is a control on a window,

the dot syntax is extended: *projectname.editorname.controlname*. For windows, the window name refers to the Code Editor view and “*window name* Layout” refers to the Window Editor for the window. When you create the bookmark, REAL Studio suggests the name that will work, so you can simply accept the default name.

To modify either the name or location, click twice in the text to get an insertion point and type the replacement text.

REAL Studio IDE Menus

The REAL Studio IDE has the following menus: File, Edit, Project, View, History, Bookmarks, Window, and Help. The Macintosh version adds the standard Apple and REAL Studio menus.

The File Menu The File Menu has menu items for creating, opening, and saving items.

- **New Project:** Creates a new project. It first gives you a choice of built-in or user-defined templates. See the section “Creating Project Templates” on page 87 for information on defining new project templates. When you make your choice, REAL Studio opens a new IDE window for the project. The Project Editor contains the items specified in the selected template. You can have more than one project open at the same time. For more information about Desktop and Console applications and custom templates, see the section “Creating a New Project” on page 74.
- **New Window:** Opens a new REAL Studio IDE window for the current project. The new window includes the Project Editor and the editor that was frontmost when you chose New Window. Use this menu command to view two or more editors in your project simultaneously. Changes in one window cause the others to be updated.
- **Open:** Displays an open-file dialog box that allows you to open an existing REAL Studio project or a REAL Studio script. You can open standard REAL Studio projects or projects saved in the XML or Version Control Project formats. For information on the XML and Version Control formats, see the sections “Saving as XML” on page 85 and “Saving as a Version Control Project” on page 85. The Save and Save As menu commands give you a choice of formats in which to save your project. When you select a project, it opens in its own IDE window, keeping the current IDE window open. This menu command enables you to have several projects open at once. A REAL Studio script is created with the IDE Script editor (see the File ► IDE Scripts menu command) and is used to automate the REAL Studio IDE.
- **Open Recent:** Open Recent has a submenu of recently opened REAL Studio projects. When you choose a project from the submenu, it opens in its own REAL Studio IDE window while keeping the current IDE window open. The Startup screen of the Options dialog (Preferences on Macintosh) contains a preference to open the most recently opened project automatically when REAL Studio starts up.

- **Close Tab:** Closes the tab panel that is currently in front. You can also close a Tab by clicking the Tab's close box. If you close the Project tab, REAL Studio will try to close the project itself. If you have unsaved changes, REAL Studio will give you a chance to save your changes.
- **Close Window:** Closes the current IDE window. If you have unsaved changes to the project, REAL Studio will give you a chance to save your changes.
- **Save:** Saves the current project or the project whose IDE window is active. A drop-down list enables you to save in the standard project format, in XML, or in Version Control Project format (VCP). For information on the XML and Version Control formats, see the sections “Saving as XML” on page 85 and “Saving as a Version Control Project” on page 85. The standard REAL Studio project format is the default and is the recommended format if you have no special need to work with XML and you are not working with a Version Control system. VCP format should be used only in conjunction with version control systems. If no changes have been made since the last save operation, this menu item is dimmed.
- **Save As:** Saves the current project or project whose IDE window is active under a new name. A drop-down list enables you to save in the standard project format, in XML, or in Version Control Project (VCP) format. The standard REAL Studio project format is the default and is the recommended format if you have no special need to work with the XML and you are not using a Version Control system. VCP format should be used only in conjunction with version control systems.
- **Revert to Saved:** Reverts the current project or the project whose IDE window is active (i.e., in the front) to its last saved state. You will lose any changes that you made since the project was last saved.
- **IDE Scripts:** Available only for the Studio version of REAL Studio. Used to script the IDE. The File ► IDE Scripts menu command has a submenu with the New IDE Script command and the names of any existing IDE scripts. Existing scripts must be in a folder named “Scripts” in the same folder as the IDE or the project in order to appear as submenu items. The New IDE Script menu command opens an IDE Script Editor window that enables you to write code that automates the REAL Studio IDE. You can either type in the code or have the IDE Script Editor record your actions. In the latter case, click the Record button, perform the actions that you want the script to automate, and then click it again to stop the recording process. It will generate the code and add it to the Script Editor. In either case, you use the RBScript language. See the entry for IDE Scripts in the *Language Reference* for the list of IDE Script commands.
- **Import/Import as External:** Import opens an open-file dialog box that enables you to import an item into the project. You can import any type of item that can appear in the Project Editor. This includes object such as windows, classes, modules, pictures, sounds, and movies. You can also import items by dragging them from the desktop to the Project Editor. To import an item as an external project item, hold

down the Alt key (Option key on Macintosh) and the Import menu item changes to Import as External. For more information about external project items, see the section, “External Project Items” on page 542.

- **Export Item...**: Exports the contents of the currently selected item. The Save-file dialog box gives you a choice of the REAL Studio format for the item, the XML format, or the VCP (Version Control Project) format.
- **Export Localizable Values...**: Exports all of the dynamic string constants in the application to a file. This file is readable by REAL Software’s free localization application, Lingua. Lingua is the utility that you use to localize REAL Studio applications. When you are finished localizing the string constants in Lingua, you import this file back into your application with either the File ► Import command or by dragging it into the Project Window. For information about Lingua, see the section “Using Lingua to Localize your Application” on page 383.
- **Page Setup**: Displays the Page Setup or Print Setup dialog box for your operating system.
- **Print**: Prints the project’s properties, which are shown in the Properties pane for the App class, and all the project’s code. Source code is printed in color. The Printing panel of the Options dialog box (Preferences on Macintosh) has an option for printing in color. For information on the project’s properties, see the section “Customizing the Standalone Application’s Properties” on page 700.
- **Print Item**: Prints the selected item according to the current Page Setup settings.
- **Exit or Quit**: Closes all open IDE windows and Quits the REAL Studio IDE application. On Macintosh, the Quit menu item is located under the REAL Studio menu.

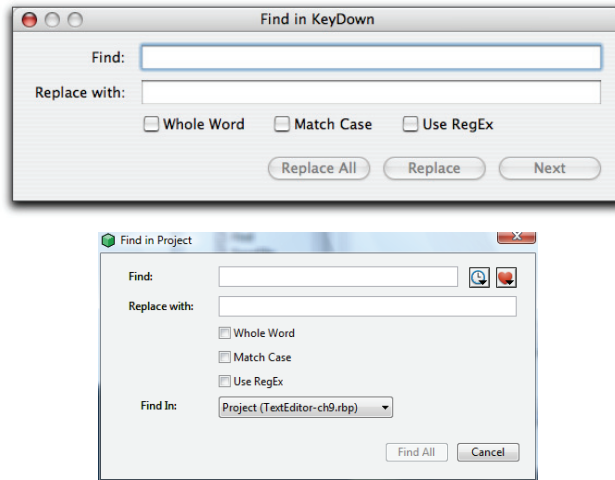
The Edit Menu The Edit menu has the standard editing commands and some REAL Studio-specific commands for working with the Code and Window editors.

- **Undo**: Undoes the last action. If this is impossible, this item changes to Can’t Undo.
- **Redo**: If you have just chosen Undo, the Redo menu item becomes active, offering to redo the action that was undone. In other cases, this menu item is dimmed.
- **Cut**: Cuts the selected text or item and places it on the Clipboard.
- **Copy**: Copies the selected text or item and places it on the Clipboard.
- **Paste**: Pastes the item on the Clipboard at the insertion point (text) or into the current object, such as a window in a Window editor.
- **Delete**: Deletes the selected text or item without putting it on the Clipboard.
- **Select All**: Selects all of the text or all the items in the current editor. It does not place the items on the Clipboard.

- **Deselect All:** Deselects all the currently selected items.
- **Comment:** Valid only for Code Editors. Changes the line in which the text insertion point is located into a nonexecutable comment line. If the current line is already a comment, this menu item changes to Uncomment. The Uncomment menu item changes the line back to a line of executable code. These two menu items are equivalent to the Comment and Uncomment buttons in the Code Editor toolbar.
- **Encrypt Item:** Displays a dialog box for encrypting the selected project item. Encryption is supported only in the REAL Studio Professional and Studio editions. Decryption is supported in all editions of REAL Studio. An encrypted item cannot be viewed or edited in its editor. If the selected project item is already encrypted, this menu item changes to Decrypt *Item*. Decrypt *Item* presents a dialog box enabling you to decrypt the item. The Encrypt and Decrypt Item menu items duplicate the functionality of the optional Encrypt and Decrypt buttons that can be installed in the Project Editor toolbar.
- **Property List Behavior:** Available only for classes. Displays the *ClassName* Property List Behavior dialog. It enables you to customize several aspects of the Properties list when an instance of the class has been added to a window. For information, see the section, “Customizing the Properties List” on page 545.
- **Duplicate:** Duplicates (copies and pastes) the selected text or item.
- **Arrange:** Displays a submenu for changing the control order of objects in a window, Bring Forward, Bring to Front, Send Backward, Send to Back. Available only for Window Layout editors. These choices affect the order in which the user can move from one control to another by pressing the Tab key. It also affects the display order for controls that are overlapped. You can display the control order with the View ► Control Order command. Available only for Window and ContainerControl Editors. These submenu items duplicate the functionality of the Window Editor toolbar.
- **Align:** Displays a two-part submenu for aligning objects in a window. The menu items above the separator align two or more objects by their left edges, right edges, top edges, and bottom edges. The items below the separator are for spacing three or more objects evenly, in either the horizontal and vertical directions. These items are available only for Window Layout and Container Control Editors. These submenu items duplicate the functionality of the Window Editor toolbar.
- **Auto Adjust TabIndexes:** Displays a submenu that enables you to change the tab order to either of two standard configurations: Top-down left-right or Left-right top-down. For more information, see the section “Auto-Adjustment of the Tab Order” on page 183.
- **Find:** Displays a submenu for displaying the Find and Find All dialog boxes for finding and replacing text in the current method or throughout the project. The Find submenu item displays a dialog that searches and replaces in the current

project item (e.g., the current method), while the Find All submenu item displays a dialog with an additional menu item that lets you specify the scope of the search.

Figure 41. The Find dialog box (top) and Find All dialog box.



For information on the Find dialog, see the section “The Find in Project and Find dialogs” on page 299. You can also find project items using the Search field in the Main toolbar.

- **Find ► Go to Search:** Moves the insertion point to the Search field in the Main toolbar. From there, you can enter a search string.
- **Options:** Displays the Options dialog box for setting application-wide preferences. On Mac OS X, this item appears in the REAL Studio menu as “Preferences”.

The Project Menu

The Project menu has items for adding items to the project, testing it within the IDE, debugging code with the debugger, and building a standalone application.

- **Add:** The Add menu item has a submenu for adding a window, a class, a class interface, a container control, a module, a folder, a menu bar, a file type set, a toolbar, an ActiveX component (Windows OS only), a report, a build automation step, and a database. The Project Editor’s toolbar duplicates much of the functionality of the Add menu. Below a separator, the Add menu lists items that are particular to the type of editor that is currently shown. These menu items duplicate the functionality of the editor’s toolbar.
- **Window:** When Window’s Code Editor is shown, the Add menu item has submenu items for adding methods, properties, computed properties, constants, menu handlers, and notes. These items duplicate the functionality of the Code Editor toolbar. When a Window Editor’s Layout Editor is shown, these menu items are not available; the commands that correspond to a Window Editor’s toolbar are in the Edit menu’s Align and Arrange submenu items.

- **Menu Bar:** When a Menu Editor is shown, the Add menu item has submenu items for adding a menu, menu item, submenu, and a separator. These submenu items duplicate the functionality of the Menu Editor toolbar.
- **Module:** When the Code Editor for a module is shown, the Add menu item has a submenu for adding methods, properties, event definitions, constants, and notes. These submenu items duplicate the functionality of the Code Editor toolbar for a module.
- **Class Interface:** When the Code Editor for a class interface is shown, the Add menu item has a submenu for adding methods and notes. These submenu items duplicate the functionality of the Code Editor toolbar for a class interface. Other items that are normally available for a Code Editor are not appropriate for a class interface. For more information about class interfaces, see the section “Class Interfaces” on page 585.
- **Container Control:** When the Code Editor for a Container Control editor is shown, the Add menu item has submenu items for adding methods, properties, event definitions, constants, menu handlers, and notes. These items duplicate the functionality of the Code Editor toolbar. When the Container Control’s Layout editor is shown, the Arrange and Align submenus of the Edit menu contain the items that duplicate the functionality of its editor toolbar. For information about container controls, see the section “The Container Control” on page 177.
- **Report:** Adds a blank Report Layout editor window to the project. Use the Report Editor to design printed reports. For more information, see the chapter “Creating Reports” on page 455.
- **Build Automation:** Adds build scripts to the project. *Building* a project produces your standalone application. Use a script to automate the build process, such as setting properties of the standalone application. For more information, see the section “Build Automation” on page 707.
- **Turn Breakpoint On:** The Turn Breakpoint On menu item sets a breakpoint in the Code Editor in the selected line or the line in which the text insertion point is located. This is equivalent to clicking in the left margin in the Code Editor to set the breakpoint. Lines on which you can set a breakpoint are indicated by a dash in the first column of the Code Editor. When a Breakpoint is set in a line, this menu item changes to Turn Breakpoint Off. For information on setting breakpoints, see the section “You Have Set A Breakpoint In Your Code” on page 637.
- **Clear All Breakpoints:** The Clear All Breakpoints item removes all breakpoints in the project. It is available only for Code Editors and only when at least one breakpoint is set.
- **Break on Exceptions:** If the Break on Exceptions menu item is selected (has a checkmark to its left), the REAL Studio compiler will break into the Debugger when it encounters a runtime exception error. For information on the Break on Exceptions option, see the section “Runtime Exception Errors” on page 652.

- **Profile Code:** If checked, it enables the REAL Studio Profiler. The Profiler monitors the built application while it is running. It measures the amount of time spent in each method and it also reports how many times the method is called. The Profiler is available only in the Studio version of REAL Studio. If you don't own the Studio version, this command is not available. For more information, see the section "Profiling your Project" on page 655.
- **Warnings:** Displays the Issue Type dialog box. This dialog is used in conjunction with the Analyze Project feature of REAL Studio. It enables you to choose the types of issues that you wish to be warned about. Deselect the items that you wish Analyze Project to ignore. For more information about Analyze Project, see the section "Analyzing the Project" on page 632.
- **Run Paused:** Used for debugging REAL Studio plug-ins at the same time as debugging REAL Studio code. Run Paused builds the application and starts the debugger but it will not launch the executable. This allows you to debug the REAL Studio project but have an external entity responsible for launching the executable. For more information, see the section "The Debugger" on page 636.
- **Run:** Attempts to compile the application and launch it. If REAL Studio is successful, the application is launched and a new Debugger screen labeled "Run" is added to the IDE window. If the attempt to compile is unsuccessful, REAL Studio will display the syntax errors that prevented compilation. This item is equivalent to the Run button in the Toolbar. Use Run to test and debug your application. For more information, see Chapter 12, "Debugging Your Code" on page 631.
- **Run Remotely:** Available only in the Professional and Studio versions of REAL Studio. The Run Remotely item has submenus that enable you to test the application on another computer. Most often you will use the Run Remotely feature to test the application on another platform. It has at least one submenu item, Setup, and submenu items for each remote computer that is set up for remote debugging. The Setup menu item displays a dialog box in which you can choose the remote machines on which you will debug. Each target machine must be visible on your network and have the Remote Debugger Stub utility configured and running. For more information, see the section "Remote Debugging" on page 657.
- **Pause:** This menu item pauses execution of the test application in the debugger. It is enabled only if you are testing the application in the debugger. For more information on this command, see the section "Controlling Execution" on page 647.
- **Stop Debugging:** This menu item stops execution of the test application in the debugger and returns to the IDE. This item is dimmed when you are not running a test application (i.e., by clicking the Run button). For more information on this command, see the section "Controlling Execution" on page 647.
- **Step:** Use the Step command with the Debugger to step through your code line by line. Step has submenu items for stepping over, into, and out of a method. This item is dimmed when you are not running a test application (i.e., by clicking the Run

button). This duplicates the functionality available in the Debugger toolbar. For information on the Step options, see the section “Controlling Execution” on page 647.

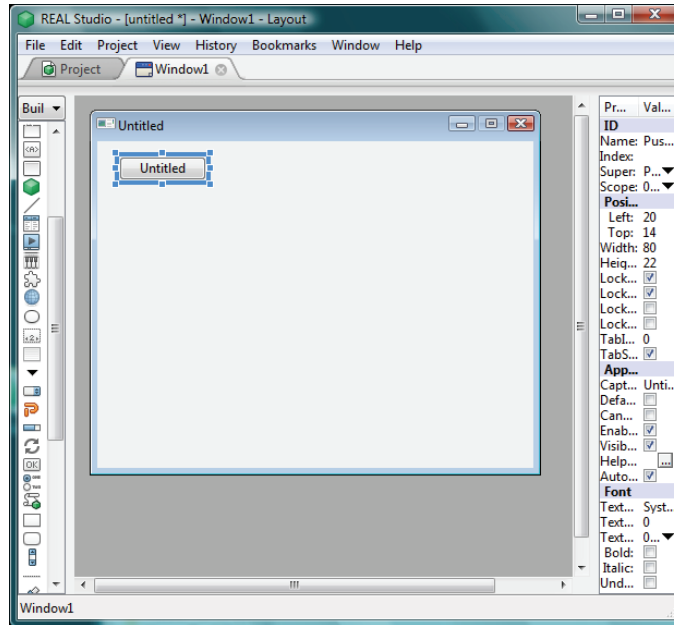
- **Analyze Project:** Analyzes the project for syntactical errors and problems that may cause the built application to behave unexpectedly. If it finds issues, REAL Studio opens an Issues tab that lists the errors. If it finds no issues, it reports that no problems were found in the Tips bar. It does not build the application. For more information, see the section “Analyzing the Project” on page 699.
- **Analyze Item:** Analyzes *Item* for syntactical errors and issues that may cause the application to behave unexpectedly. *Item* is the project item in the frontmost tab. If it finds issues, REAL Studio opens an Issues tab that lists the errors. If it finds no errors, it reports that no errors were found in the Tips bar. It does not build the application. For more information, see the section “Analyzing the Project” on page 699.
- **Build Settings:** Displays the Build Settings dialog box. Use this dialog box to choose the platform or platforms for which you will build standalone applications. You can build for all platforms that REAL Studio supports. For information on the Build Settings dialog box, see the section “Choosing a Target Platform” on page 694.
- **Build Application:** The Build Application item is equivalent to the Build button in the Main Toolbar. It builds your application according to the current Build Settings and App object’s properties. If you have selected more than one target platform in Build Settings, it builds for all the selected platforms simultaneously. The built application will be created in a “Builds” folder for that project. It will contain subfolders for each selected platform. For more information about the build process, see the section “Building Your Application” on page 695.

The View Menu

The View menu has IDE configuration options that enable you to show, hide, and customize the toolbars and maximize the area of the window used by the current editor:

- **Editor Only:** If the Editor Only menu item is selected (has a checkmark to its left), the area of the window devoted to the editor is maximized, hiding the toolbars and minimizing any panes to the left and right of the editor area. It applies to all editors until it is turned off. For all editors, it hides the Main and Editor toolbars and the Bookmarks bar. It also minimizes the Properties pane and Controls pane, if appropriate for that editor. Figure 42 shows a Window Editor in its maximized state. For a Code Editor, the Editor Only command reduces the Browser area to a little sliver, like the Controls pane in Figure 42. If Editor Only is selected, you can revert to the “normal” view by deselecting this menu item. If you have resized the panes prior to selecting Editor Only, REAL Studio remembers your settings when you deselect Editor Only.

Figure 42. A Window Editor in Editor Only mode.



- **Show Code/Show Layout:** Available only for Window and Container Control Editors. If the Layout editor for a window or container control is shown, the Show Code menu item switches to its Code Editor; if the Code Editor is shown, the Show Layout menu item switches to the Layout Editor view. These menu items are the equivalents of the Code Editor and Window Editor icons on the left side of the Window Editor and Code Editor toolbars. If the Code and Layout Editors for a window are shown in separate tabs, then this menu item brings the selected Editor to the front, leaving the current tab unchanged.
- **Show/Hide Empty Events:** Available only for Code Editors. By default, Show Empty Events is selected and all event handlers are listed in the browser area of the Code Editor. The Hide Empty Events command suppresses the event handlers that have no code in them. If Hide Empty Events is selected, this menu item changes to Show Empty Events. For more information on showing/hiding empty events, see the section “Showing and Hiding Empty Events” on page 280.
- **Tab Order:** Available only for Window and Container Control Layout Editor views. The Tab Order is the order in which controls are selected as the user presses the Tab key repeatedly. If the Tab Order menu item is selected (has a CheckBox to its left), the Tab Order for the controls in the window are shown as numbered badges. To hide this information, deselect the Tab Order menu item. There are several ways to change the Tab Order. You can use the Arrange submenu items in the Edit menu, change the value of the TabIndex property in the Properties pane for each control,

use the Edit ► Auto Adjust TabIndexes menu item, or change the value of the TabIndex property via code.

- **List Bindings:** Available only for Window and Container Control Layout Editor views and for layouts that contain bindings. It displays a list of object bindings in the layout in a separate window. REAL Studio bindings were deprecated in REAL Studio 2007 Release 5. Object bindings can no longer be created. This menu item can be used to identify bindings that were created with previous versions of REAL Studio.
- **Menu Layout:** Available only for a Menu Editor. It has submenu items for previewing the current menubar on any of the platforms on which you can build the application: Windows, Mac OS X, or Linux. These submenu items are equivalent to the Preview Mode icons on the left side of the Menu Editor toolbar.
- **Main Toolbar:** The Main Toolbar menu item has a submenu for hiding and customizing the Main Toolbar. The Customize submenu item displays a “mover” dialog box that you can use to add or subtract items from the Main Toolbar.
- **Bookmarks ToolBar:** The Bookmarks ToolBar submenu has items for hiding and customizing the Bookmarks ToolBar. This is the row of local bookmarks that you have added to the current project via the Add Bookmark button or the Bookmarks ► Add Bookmark menu item. The Customize submenu item displays a dialog box in which you can edit the text and location of the bookmarked items.
- **Editor Toolbar:** The Editor Toolbar is just below the row of tabs and has buttons for adding items to the object being edited. Each editor has its own toolbar, so this menu item pertains to the current editor’s toolbar. The Editor Toolbar menu item has a submenu for hiding and customizing the current Editor Toolbar.

The History Menu

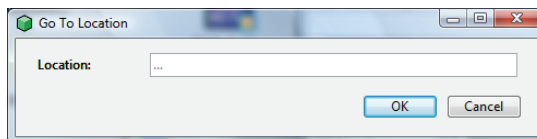
The History menu keeps track of the screens that you use as you work on your project. As you work, it automatically adds the names of the screens to the menu. You can go directly to any screen that you have previously visited by choosing its name from the History menu.

The History menu also includes menu commands for the navigation-related items in the Toolbar:

- **Backward:** Moves to the previously-viewed screen. This is equivalent to the Back button in the Toolbar.
- **Forward:** Moves to the next screen that you viewed (assuming that you have moved backwards in the list of viewed screens). This is equivalent to the Forward button in the Toolbar.
- **Home:** Moves to the Project Editor, the screen that you see when you double-click the REAL Studio IDE application or choose File ► New Project. This is equivalent to the optional Home button that can be added to the Main Toolbar.

- **Go to Location:** Allows you to enter the name of an Editor or an item. This is equivalent to entering the item name into the Location area in the Main Toolbar. If the Location area is shown when this menu item is selected, it moves the text insertion point into the Location area. Otherwise, it opens a dialog box or sheet window into which you can enter the location. Figure 43 shows the Go To Location dialog box that is displayed only when the Main toolbar is hidden.

Figure 43. The Go To Location dialog box.



When you are in a Code Editor window and want to go to the definition of an item, you can hold down Ctrl and double-click the item (hold down Command on Macintosh).

The Bookmarks Menu

The Bookmarks menu contains commands for bookmarking individual items in your projects. As you add bookmarks, the names of the bookmarked items are added to the Bookmarks menu. Go to a bookmarked item by choosing its name from the Bookmarks menu or clicking it in the Bookmarks bar.

You can bookmark items from several projects and they will be available in all open REAL Studio IDE windows. The syntax that REAL Studio uses is *projectname.editorname*, that is, the name of the project on disk, followed by a dot, followed by the name of the editor. If the item is a control in a window, the dot syntax is extended: *projectname.editorname.controlname*. If the project name is omitted, the current project is assumed. Use the Global Bookmarks option when you want the bookmark to be available in all of your projects. This makes it easy to navigate to an item in another project.

- **Show all Bookmarks:** Opens a dialog box that lists all the global bookmarks available. You can edit the names and/or locations of your global bookmarks from this dialog box. It works the same way as the Edit Bookmarks dialog box for your local bookmarks in the Bookmarks bar. For information on editing bookmarks, see the section “The Bookmarks Bar” on page 59.
- **Add Bookmark:** Adds the current item to either the Bookmarks menu or the current Bookmarks bar. If you have added a Bookmark Folder to either the menu or the bar, you can also add the current item to any folder. For information on adding a bookmark, see the section “The Bookmarks Bar” on page 59.
- **Add Bookmark Folder:** Displays a dialog box that enables you to name and add a folder to either the Global or Local Bookmarks menu (this assumes that both Global and Local bookmarks exist). When you choose Add Bookmark Folder, the following dialog box appears:

Figure 44. The Add Bookmark Folder dialog box.



After you have created a bookmark folder, the Add Bookmark dialog's Create In drop-down list offers to add the new bookmark to either the Global or Local Bookmarks folder.

- **Global Bookmarks:** Displays a submenu of the global bookmarks that you have added, if any. Select a global bookmark to go to that location.

The Window Menu

The Window menu has items for managing the REAL Studio IDE window and its contents. You can have more than one window open for the same project and you can have multiple projects open. It also has items for selecting among tabs in the current window.

- **Minimize:** Minimizes the IDE window to the Taskbar (Windows and Linux) or the Dock (Mac OS X). This is equivalent to clicking the Minimize button in the window's Title bar. When the IDE window is minimized, you can click on its name or icon to restore the window to its previous size and position.
- **Maximize/Zoom:** Maximizes the IDE window to fill the screen. It is called "Zoom" on Macintosh. This is equivalent to clicking the Maximize button in the IDE window's Title bar. When the window is maximized, this menu item changes to Restore. Choose Restore to return the IDE window to its previous size and position.
- **Show/Hide Properties:** Available only for the Project and Window Editors. Shows or hides the Properties pane in the window. When the Properties pane is hidden, the menu command changes to Show Properties.
- **Builds:** Opens the Build Progress dialog box. It lists all the builds since the session started or since you last cleared the list. For more information on the Build Progress dialog box, see the section "Building Your Application" on page 695.
- **Next Tab:** Next Tab is equivalent to clicking on the tab to the right of the current tab in the Tab bar.
- **Previous Tab:** Previous Tab is equivalent to clicking on the tab to the left of the current tab in the Tab bar.
- **Bring All to Front:** Available only on Macintosh. If more than one REAL Studio window is open, it brings all of them to the front with respect to other application and Finder windows.

Below these menu items, the Windows menu displays a menu item for each open window. Each item includes the name of the project and the name of the currently displayed editor.

The Help Menu

The Help menu provides easy access to the built-in REAL Studio documentation as well as online information from the REAL Studio web site. The Help menu has the following menu items:

- **Getting Started:** Opens the REAL Studio QuickStart using Adobe Acrobat Reader. If the QuickStart file is not installed, this menu command displays a dialog box enabling you to download it from the REAL Software web site.
- **Tutorial:** Opens the REAL Studio Tutorial in a new window using Adobe Acrobat Reader. If the Tutorial file is not installed, this menu command displays a dialog box enabling you to download it from the REAL Software web site.
- **User's Guide:** Opens the REAL Studio User's Guide in a new window using Adobe Acrobat Reader. If the User's Guide file is not installed, this menu command displays a dialog box enabling you to download it from the REAL Software web site.
- **Language Reference:** Opens the online Language Reference. Use it to read descriptions of REAL Studio objects, check syntax, and use code examples. For more information about the online Language Reference, see the section "Using the On-Line Help" on page 23.
- **REAL Studio on the Web:** Opens your default web browser application to REAL Software's home page on the web.
- **REAL Studio Feedback:** Opens the REAL Studio Feedback page, where you can enter bug reports and feature requests and search the Feedback database by keyword.
- **About REAL Studio:** Opens a window that provides information on the version of REAL Studio that is running and identifies the licensed user. On Mac OS X, the About REAL Studio menu item is in the REAL Studio menu.
- **Enter License Key:** Opens a dialog box that allows you to enter your license code into a your copy of REAL Studio. On Mac OS X, the Enter License Key menu item is in the REAL Studio menu.

Working with Projects

All of the windows, menus, modules, pictures, sounds, movies, plug-ins, databases, and programming code that make up an application are stored in a Project document. Projects give you a convenient way to organize the objects that make up your application. You can have several projects open at once and you can have several windows open per project.

Projects can contain any of the following items:

- Windows
- Menu bars

- Classes
- Class interfaces
- Modules
- File Type sets
- Container Controls
- Pictures
- Sounds
- Movies
- Databases
- AppleScripts (Mac OS only)
- Resource files (Mac OS only)
- Cursor resources
- Folders
- Documents of other types, which are treated as text strings

Each item is denoted by an icon that denotes its type and, in some cases, its subtype. For example, different window types have their own icon.

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

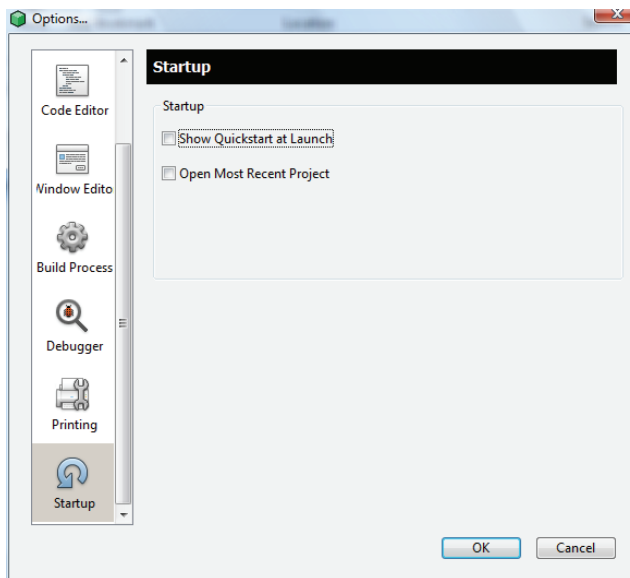
Double-clicking on an item in the Project Editor will either display the item in its editor or a viewer for the item, if REAL Studio has no editor for that type of item.



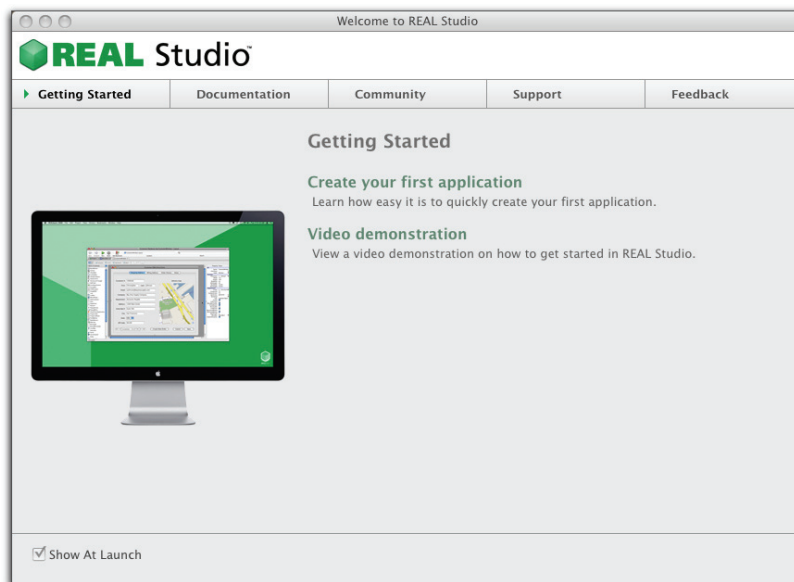
NOTE: There is one exception to this rule. You can encrypt an item in the Project Editor to prevent others from accessing the item and any code associated with it. An encrypted item functions normally in the built application, but it cannot be viewed or edited within the REAL Studio IDE. Encrypted items have a small key in the lower-right corner of its icon. When you double-click an encrypted item, REAL Studio displays an alert that tells you that the item is encrypted. For more information, see “Encrypting Your Source Code” on page 307.

Creating a New Project

When you open REAL Studio by double-clicking on the REAL Studio application icon, either a new project is created for you automatically or the most recent project is opened. Optionally, REAL Studio will display a Quickstart screen that leads to an introduction to REAL Studio. These options are controlled by the Startup options screen in REAL Studio options. To set your options, choose **Edit ► Options** (or **REAL Studio ► Preferences** on Macintosh). Scroll the left panel down and click the Startup icon. The following screen appears.

Figure 45. The REAL Studio Startup options screen.

If you select the Show Quickstart at Launch option, then the following screen is presented in front of the REAL Studio IDE.

Figure 46. The Quickstart options screen.

By default, the Quickstart screen is shown the first time you launch REAL Studio. After you have viewed the demo and finished the Quickstart, you can deselect it in the Startup options screen.

If you turn off the Quickstart screen, REAL Studio will either open a new untitled project or your most recently opened project. It will do the latter if you select the Open Most Recent Project in the Startup screen in the Options dialog (see Figure 15 on page 40). By default, the new project is a Desktop Application project but you can change the type of project. For more information, see the section “Creating Project Templates” on page 87.

If you immediately resize the IDE window after creating a new project, REAL Studio will remember this size and use it as the default IDE window size for subsequent new projects.

If you have a project open and wish to begin a new one, simply choose File ► New Project. REAL Studio presents the New Project dialog box. It enables you to choose the type of project template that you want to base your new project on. By default, it contains four items. The first two items are built into REAL Studio and have no user-modifiable templates. They are:

- **Desktop Application:** This is the default template for any standard REAL Studio application that uses a graphical user interface (GUI). A REAL Studio Desktop Application project has a window and a menubar. The Desktop Application template includes these two items. Except where noted, the *User's Guide* deals only with Desktop Applications. When it refers to a REAL Studio project, it means Desktop Application project.
- **Console Application:** This is the default template for a REAL Studio application that runs only in the Terminal window (Mac OS X and Linux), the command line (Windows), or completely in the background. A Console Application cannot have a menubar or windows. These items are omitted from the Console Application template. Use this option only when you want to create an application that has no graphical user interface of its own. For example, a web or mail server application can be written so that it has no graphical user interface. For more information on console projects, see the entries in the *Language Reference* for the `ConsoleApplication` and `ServiceApplication` classes.

By default, a new Desktop Application project contains three items, `Window1`, the main window, `MenuBar1`, the default menu bar and menus, and a special class called the `App` class, which you can use to manage application-wide properties and functions of the application. For information about this class, see the section “The Application Class” on page 580.

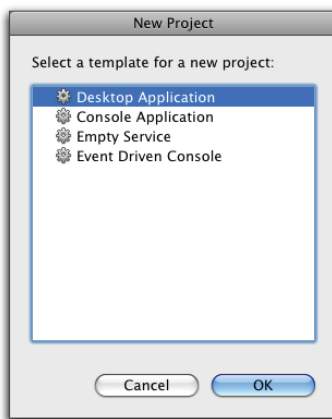
The default project for a Console Application lacks the items that are needed to build and maintain an interface. It has only one item, the `App` class. It is based on the `ConsoleApplication` class instead of the `Application` class. Moreover, the Project ► Add submenu does not allow you to add interface items to a console application project such as windows, menubars, and `ContainerControls`. For more information on these classes, see the entries for these classes in the *Language Reference* and Chapter 10, “Creating Reusable Objects with Classes” on page 531.

REAL Studio ships with two additional templates that are based on items in the Project Templates folder. Since you can open and resave these templates, you can modify them for your own use. They are:

- **Empty Service:** The template for an Empty Service is designed for an application with no user interface, such as a web server. Its App class item is based on the Service Application class rather than the Console Application class. The code for the Service should be placed in the App class's Run event handler.
- **Event Driven Console:** The template for an Event Driven console is for an event-driven application with no user interface. Its App class is based on the ConsoleApplication class and has no Project items for building and maintaining a user interface. The event driven code runs in the Run event of the App class. The code is in a main DoEvents loop.

When you create a new project from a template, the Project Editor will contain all the additional objects that were saved with the project when the template project was created. For more information on adding templates, see “Creating Project Templates” on page 87.

Figure 47. The New Project dialog box.

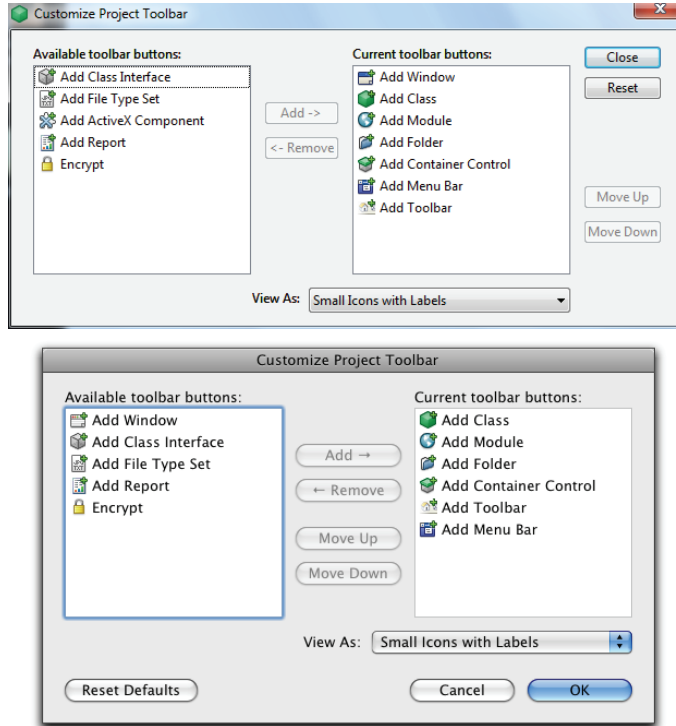


Configuring the Project Editor Toolbar

The Project Editor Toolbar (just below the row of Tabs) has buttons for adding items to the project. By default, it has buttons for adding classes, class interfaces, modules, folders, menubars, and windows. If you like, you can modify the Project Editor toolbar with the View ► Editor Toolbar ► Customize submenu. Note that the Project pane must be selected to customize the Project Editor toolbar rather than another editor's toolbar.

The following dialog appears:

Figure 48. The Customize Project Editor Toolbar dialog box.



The Customize Project Toolbar dialog box uses a “mover” interface to configure the toolbar. Listed in the right panel are the current items in the toolbar. Listed on the left are optional items that can be added to the toolbar.

The following operations are available:

- To add an item, highlight it in the left panel and click the Add button (shown in Figure 37).
- To remove an item, highlight it in the right panel and click the Remove button. This moves the item to the list on the left.
- To reorder an item, highlight it in the right panel and click either Move Up or Move Down or select the item to be moved, drag it to the desired location, and drop it between two items. The order in which the items are listed is the left-to-right order in the toolbar.
- To change the appearance of the items in the toolbar, choose an item from the Display As drop-down menu. Your choices are:
 - Big icons with labels.
 - Small icons with labels,

- Big icons (no labels),
 - Small icons (no labels),
 - Labels only.
- To reset the toolbar to the default toolbar, click the Reset button (Windows and Linux) or the Reset Defaults button (Macintosh).

Adding Items to Your Project

The method you use to add items to a project depends on the type of item you wish to add. You add REAL Studio project items such as windows, classes, class interfaces, menubars, and modules by clicking on a button in the Project Editor toolbar (just below the row of tabs) or with the Project ► Add submenu.

If you have a picture, sound, movie, or REAL database you wish to use in your project, you can add with the File ► Import menu item. It displays an import-file dialog box that enables you to navigate to the item to be imported. Choose the item and click the Import button.

You can also import items by dragging a file from the desktop and dropping it into the Project Editor. If REAL Studio stores a shortcut (alias on Macintosh) to an external item, it is displayed in italics within the Project Editor. You will learn in later chapters how to add each type of item that can appear in the Project Editor.

Dragging to the Tabs Bar

If the Project Editor is not displayed, you can still import an item into the Project Editor by dragging. Do this by dragging the item to the Tabs bar. Any item that can be dragged to the Project Editor can also be dragged to the Tabs bar.

When you complete the drag on Macintosh, the Tabs bar gets a marquee that indicates that it can accept the dragged item. On Windows and Linux, the pointer changes to show a marquee.

Organizing Project Items

If your application has many items in its Project Editor, you may want to organize them rather than leaving them in the order in which they were created. First, you can reorder an item by holding the mouse button down on the item and dragging it up or down. As you drag, you can drop it to a new position. The new position is indicated by a horizontal line between existing items as you move the dragged item over it. Release the mouse button to drop the item into its new position.

You can also group items by adding a folder to the Project Editor and storing some items in the folder. For example, you might want to create a folder to hold all the movies and another folder for all the sounds. First, create the folder by clicking the Add Folder button in the Project Editor toolbar or choose Project ► Add ► Folder. A folder has one property, its name. Use the Properties pane to give the new folder a meaningful name. Then drag items that you wish to group together into the folder. To drag into a closed folder, drag in a diagonal direction to the right to indicate that the item should be placed in the folder instead of above or below. If you want to add new items to a folder, open the folder and then click the Add button in the toolbar for the desired type of item.

Note that your use of these two features does not affect the functionality of the project. They are available as a convenience to you, making it easier to work with projects with a lot of items.

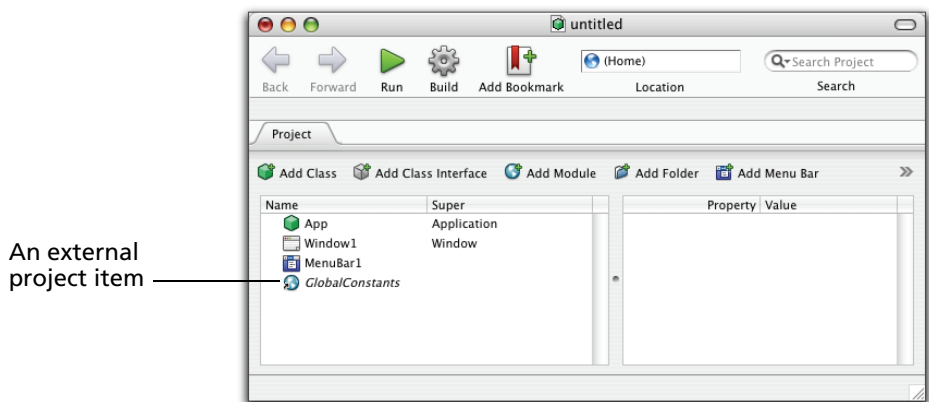
External Project Items

It is also possible to include windows, classes, or modules in a project that are actually stored in external files. This feature allows more than one project to use the externally stored item. When you modify an external project item in REAL Studio and then save your project, your changes are written out to the external file on disk. When you open any other project that refers to the same external file, that project will reflect your changes. However, REAL Studio does not allow you to open more than one copy of REAL Studio accessing the same external project item simultaneously.

To convert a project item to an external item, select it in the Project Editor and right+click (Control-click on Macintosh) to display the contextual menu for the item and choose Make External. REAL Studio will display a Save-file dialog box. Navigate to the directory in which you want to save the item, name the item, and click Save. When you have successfully saved the item, it will be shown in the Project Editor with a shortcut badge and its name will be in italics. This is shown in Figure 49.

To add an item to a project as an external item, hold down Alt key (Option key on Macintosh). The File ► Import menu command changes to Import as External. Choose that command and select the item to be imported. You'll know you've done it correctly because the icon in the Project Editor will have a small shortcut (alias on Macintosh) badge and the name will be in italics, just like shortcuts on the desktop. In Figure 49 an externally stored module has been added to the project.

Figure 49. A module as an external project item.



You can add an item to another project as either an external item or a regular item. To add it as an external item, hold down the Ctrl and Shift keys (⌘ and Option keys on Macintosh) when dragging to the Project Editor.

When you save a project that contains external items, REAL Studio will display a Save As dialog box for each external item, followed by the one for the project itself. This gives you an opportunity to save your modified external project items to a new location, leaving the original ones intact. External project items that you haven't modified are not presented in this way.

If an external project item is set to Read Only in Windows or Linux or locked in the Finder (on Macintosh), you can't modify that item within the REAL Studio IDE, though you can still view it. This provides a convenient way to protect external items (which may be shared by many projects) from accidental modification. It also provides a way to use REAL Studio with some version control systems, as long as these systems can use locking at the OS level to reflect the checked in/out state of each file. (Note that if an external file is changed on disk by something other than REAL Studio — such as a version control system — you'll need to re-open the project to reload that item.)

In addition to REAL Studio items that can be stored either internally or externally, several other types of items are stored as external items by default. These include movies, sounds, and pictures.

Removing Items from Your Project

You can remove items from a Project by clicking once on the item in the Project Editor to select it and then pressing the Delete key. You can also select the item and choose Edit ► Cut or Edit ► Delete. You can also use the Delete command in the Project Editor's contextual menu, described in the following section.

The Project Editor Contextual Menu

The Project Editor has its own contextual menu that makes it easy to perform actions directly from the Project Editor. Right+click on a project item (Control-click on Macintosh) or on the blank space in the Project Editor to display a contextual menu. The items on the menu vary depending on object type and may include some or all of the following items:

- **Add to Project:** Has a submenu that offers to add any type of item that can be added to the Project Editor. Items that can be added include windows, classes, class interfaces, container controls, modules, toolbars, folders, menu bars, file type sets, REAL SQL Databases, report, build automation script, and ActiveX controls (Windows only). This menu item duplicates the functionality of the Project ► Add menu item.
- **Add to *ModuleName*:** Available only when the selected item is a module. Has a submenu that offers to add any type of item that can be added to a module namespace. Items that can be added are classes, class interfaces, and other modules. If the contextual menu is displayed while a module's Code Editor is displayed, the submenu also offers to create a method, property, computed property, constant, note, structure, or enum. For information about adding items to modules, see Chapter 6, "Adding Global Functionality with Modules" on page 367.

- **Edit Source Code:** Opens the Code Editor for editing the item within REAL Studio. For example, if the item is a module, it opens the Code Editor for the module. If the item is a movie, it opens the movie using the default media player for the movie type, if available. You can also open files by simply double-clicking the item in the Project Editor. (If the object is encrypted, the Edit command is unavailable).
- **Edit Window:** Opens the window for editing in a Window Layout Editor. This item is available only if the item is a window or a container control.
- **Find Item:** Finds all instances of the selected item in the project. When you choose Find Item, REAL Studio opens Search Results panel that lists the results of the search. For more information on the Search Results panel, see the section “Searching your Project” on page 297.
- **Delete:** Removes the item from the project. You can undo this action with the Edit ► Undo Delete menu command.
- **Duplicate:** Duplicates the selected project item.
- **Play:** Appears only if the item is a sound file. Plays the sound.
- **Open File:** Opens the file as if double-clicked from the desktop. For example, if the item is a picture, it opens the item in the application indicated by its file extension or Type and Creator codes.
- **Show on Disk:** Opens the file’s folder and highlights the file. For external items only.
- **File Path:** Displays a hierarchical menu to the external item, allowing you to open any folder enclosing the file.
- **Relocate:** Opens an open-file dialog box, enabling you to reopen the item if it has been moved from its original location. For external items only.
- **Encrypt:** Displays the Encrypt... dialog box, allowing you to enter an encryption password and encrypt the object. Encryption is supported only in REAL Studio Professional and Studio editions. Decryption is supported in all editions. This feature is most typically used to hide code in windows, classes, or modules that you distribute or sell to other developers. For an example, see the section “Encrypting Modules” on page 403. Available only if the item is not encrypted.
- **Decrypt:** Displays the Decrypt... dialog box, allowing you to enter the encryption password to decrypt the object. An encrypted object in the Project Editor has a small key in the bottom-right corner of its icon. Available only if the item is encrypted.
- **Add to Class:** Appears for classes, allowing you to add structures, enums, and delegates to classes.

- **New Subclass:** Appears only when the selected item in the Project Editor is a class or subclass. Creates a new subclass, adds it to the Project Editor, and automatically sets its super class to the selected class. That is, it creates a subclass of the selected class. The new subclass is automatically named *CustomClassName*, where *ClassName* is the name of the super class. After creating the subclass, you can use the Properties pane to make any necessary modifications to the new subclass, including changing its Super class. See Chapter 10, “Creating Reusable Objects with Classes” on page 531 for information on classes and subclasses.
- **Extract Superclass** Opens the Extract Superclass dialog box. Available only for classes that do not have a superclass. This enables you to create a new superclass from the current class. You can give the new superclass a name and select some or all of the methods and properties of the current class to be methods and properties of the superclass instead. When you accept this dialog, the new superclass is added to the project and any methods or properties that you selected become methods of the superclass and are removed from the current class. For information on classes and superclasses, see “Creating a Superclass from an Existing Class” on page 541.
- **Implement Interface:** Opens the Implement Interface dialog box. This enables you to implement a class interface for the selected item. It is available only for windows and classes. It contains a drop-down list of all the built-in and custom interfaces in the project. Choose among the types of interfaces presented in the drop-down list. When you specify a class interface via this dialog box, REAL Studio will automatically add all the method declarations specified by the class interface to the item’s Method Editor and will open the Method Editor to the first such method. A window or class can have more than one class interface. Repeat this process to specify additional class interfaces for the item. For more information on interfaces, see the section “Class Interfaces” on page 585.
- **Property List Behavior:** Available only for classes. Displays the *ClassName* Property List dialog box. It enables you to customize several aspects of the Properties pane when an instance of the class has been added to a window. For information on the Properties List Behavior dialog, see the section, “Customizing the Properties List” on page 545.
- **Make External:** Saves the item to disk as an external project item. The difference between Make External and Export is that Make External also converts the item from a “regular” item to an external item for the present project. Like exported items, the copy on the desktop can be imported into other projects. For information on external project items, see the section “External Project Items” on page 80.
- **Make Internal:** Appears only when the selected item in the Project Editor is an external item. Make Internal converts the item to an internal project item.
- **Export:** Used to export the item to disk. When you choose Export, a Save-file dialog appears, enabling you to save the item to disk. You can import the item into another project by dragging it from the desktop to the Project Editor or importing

it as an external item (in which the item remains on disk and can be shared among projects) by holding down the Ctrl and Shift keys (⌘ and Option keys on Macintosh) while you drag the item from the desktop to the Project Editor.

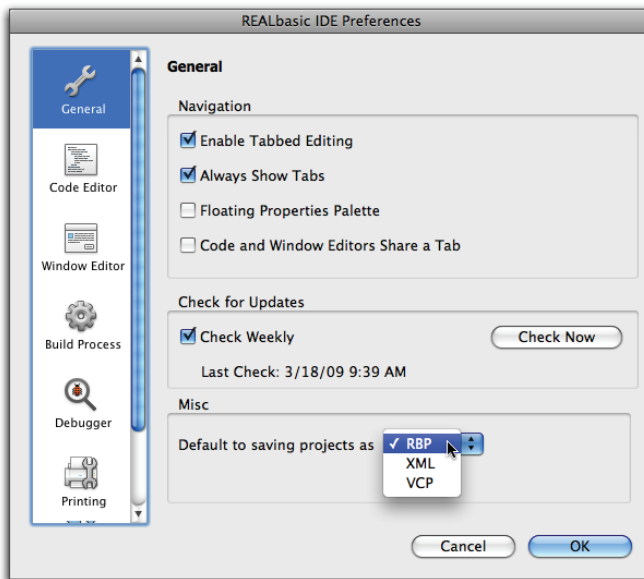
- **New Implementor:** Available only for class interfaces. Creates a new class that implements the selected class interface. The new class is named *CustomInterfaceName*. It has no Super class but implements the methods of the selected interface. For more information on interfaces, see the section “Class Interfaces” on page 585.
- **Find Implementors:** Available only for class interfaces. Searches the project and finds all classes that implement the current class interface. When it is finished, it creates a new Search Results tab that lists the results of the search. For more information on interfaces, see the section “Class Interfaces” on page 585.
- **Extract Interface** Opens the Extract Interface dialog box. Available only for windows and classes. This enables you to create a new class interface that uses some of the methods of the current item in the new class interface and makes the current class or window an implementor of the new class interface. For more information on interfaces, see the section “Class Interfaces” on page 585.
- **Attributes:** Available for project items and methods and properties in the Code Editor. Opens the Attributes Editor for the item. Attributes are compile-time properties. An attribute is added to an item via the Attributes Editor in the Project Editor. They can be accessed by the *AttributeInfo* class in the language. For more information on Attributes, see the section “Attributes” on page 237.

Saving Your Project

When you want to save the changes you have made to your project, choose File ► Save. If you aren’t sure whether you want to keep the changes you have made, you can choose not to save your project or choose Save As from the File menu and save the project under another name. This will keep your original project intact.

By default, REAL Studio saves the project in its own binary format. This is the preferred format for normal project development with REAL Studio. In addition, you can also save in either the XML or Version Control Project (VCP) formats. These options are offered in the “Save as Type” or Format (on Macintosh) pop-up menus in the Save As dialog box.

If you need to save in either alternate format, you can set the default preference in the General Options screen (Preferences on Macintosh). You can set the default to the binary format (RBP), XML, or VCP format.

Figure 50. Setting the default format in Preferences.

As noted above, you can save external project items independently of the project via the items' contextual menus.

Saving as XML Projects can be saved in XML format using the File ► Save or File ► Save As commands and then choosing XML from the Save as Type (Windows and Linux) or Format (Macintosh) pop-up menu in the Save-file dialog box. When you save in XML format, you are really doing an export, not a save; the project stays associated with its original file location and format.

Opening XML Projects can be imported from XML using the File ► Open command. Also, all text files will appear in the Open dialog, but only those that contain proper XML data can actually be read. If the open is successful, an IDE for the project appears, just as if it had been saved in the REAL Studio format.

Saving as a Version Control Project The Version Control format saves the project in a human-readable text format. This format is intended only for use with a Version Control System to handle version control and project sharing among multiple programmers. In the Version Control system, programmers can easily reconcile different versions of the same project by working with the text files.

Although the Version Control Project format produces editable text files, you should not attempt to write your code or continue development of the project in a text editor. Project development is supported only within the REAL Studio IDE.

When you save a project in the VCP format, you actually get several files that collectively represent the information in the project. Here are general descriptions of the types of files that you get:

- The Project file is store as an .rbvcp file that contains the properties of the application as a whole. The .rbvcp file can be opened from the REAL Studio IDE via the File ► Open command.
- Source code project items (interfaces, modules, classes, file type sets) are stored in .rbbas files.
- Each window and container control is stored in a .rbfrm file that contains the window's or container control's contents as well as the source code.
- The menus are stored in an .rbmnu file.
- External items (pictures, sounds, movies, and so forth) don't generate any extra files since they're already external.
- Binary items are saved in a single binary .rbres file. Items such as the application's icon, file type icons, encrypted items, and so forth are stored here. This file is not meant to be human readable.

In a text file, each item begins with a #tag statement that indicates the type of the item, such as “#Tag=Window”. This is followed by a text representation of the item. For example, properties are represented as *key-value* pairs (such as “Width=350”), methods and menu handlers are stored in a Version Control project, and so forth. The details of the format are not officially documented, but you will be able to recognize what each item is.

On the Macintosh a, VCP uses Unix line endings.

To use a version control system, simply save the project in VCP format. Then you can check the items in. For detailed information about the VCS “Subversion,” see <http://subversion.tigris.org>.

When you delete files or folders from a project in VCP format, they are left on disk. This is actually useful for Subversion and similar VCS users as it gives you a chance to manually delete the file from Subversion.

Renaming a project item produces a “.obsolete” file. For example, if you have Foo.rbbas and rename it to Bar.rbbas, you are left with Foo.rbbas.obsolete. To fix this in Svn you have to manually (outside of REAL Studio) do the following:

- Rename Bar.rbbas to a temporary name such as Bar.rbbas.hide
- Rename Foo.rbbas.obsolete to Foo.rbbas
- Use Svn to rename Foo.rbbas to Bar.rbbas (no need to commit, just need the rename operation pending)
- Manually delete Bar.rbbas

- Rename your Bar.rbbas.hide back to Bar.rbbas
- Commit



When you update the project, you should have the project closed. Otherwise, the IDE may save over what got just got checked out. It does not notice file changes when the project is open.

Opening a Version Control Format Project

The files that are generated by REAL Studio when you save in the Version Control Project format should be kept in the same directory. To open a project in Version Control Project format, choose File ► Open and choose Version Control Project from the Format drop-down list (Not needed on Macintosh). Navigate to the correct directory and then open the .rbvcp file that corresponds to the desired project.

Creating Project Templates

If you have several items you commonly use in every project, you can save them in a project file and make the project file a template for new projects. The template can include custom windows, classes, modules and other project items.

When you create a new project based on the template, REAL Studio creates a new untitled project that is an exact copy of the template. The template project itself remains unchanged. This lets you create a new project using existing project items without worrying about modifying the original items.

The most efficient way to use a template is to place it in a special directory in the same directory as REAL Studio called “Project Templates”. If you do so, your list of templates will be listed in the New Project dialog box whenever you choose File ► New Project.

If you want REAL Studio to use a particular custom template automatically, name it “Default New Project” and place this template in your Project Templates folder. This template will be used when REAL Studio is launched.

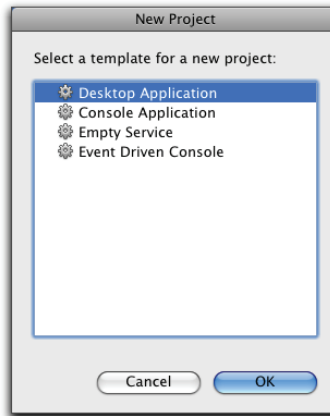


To use a template to start a new project, do this:

- 1 Copy the projects you wish to use as templates to the “Project Templates” folder in the folder that contains the REAL Studio application.**
- 2 Launch REAL Studio and choose File ► New Project.**

REAL Studio presents the New Project dialog box. It lists all the templates in the Project Templates directory as well as the templates shipped by REAL Studio. Figure 51 on page 88 shows the set of default projects. The first two are built into REAL Studio; the last two are shipped with REAL Studio as editable templates that are placed in the Project Templates folder.

Figure 51. The New Project dialog box.



3 Highlight the desired template and click OK.

REAL Studio will use that item as the basis of the new project. It will open as “Untitled” but will have all the classes, modules, windows, and so on from the template file. Changes to a new project based on a template will not affect the template.

Building a User Interface

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

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

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

Contents

- Working with windows
- Interacting with the user through controls
- Adding menus
- User interface guidelines

Working with Windows

Typically, most of an application's user interface will be in the application's windows. This, of course, is highly application-specific. REAL Studio makes it easy to create new windows of just about any type. You create your user interface by creating its windows and adding interface controls such as PushButtons and CheckBoxes.

By default, a REAL Studio Desktop Application project has one window that is displayed automatically when the application runs. Typically, you will begin designing your application's interface by adding controls to this window and enabling the controls by writing code.

To add additional windows to an application, you follow the following procedure:

- Add a new window to the project by clicking the Add Window button in the Project Editor toolbar or by choosing the Project ► Add ► Window command,
- Set the window's Frame type and other properties using its Properties pane,
- Add controls to the window,
- Add code as needed,
- Add code to display the window in the finished application (see the section "Opening Windows" on page 313).

The following sections review the types of windows supported by REAL Studio. In addition to these window types, you can design message dialog boxes without formally creating a window for the message. You do so using the `MessageDialog` class, which is described in the section "The MessageDialog Class" on page 109.

Window Types

REAL Studio supports twelve types of windows. The window type is set by its Frame property. Some types, however, are rarely used in modern applications and are retained only for historical reasons. A few specialized windows are supported on Mac OS X only. The types of windows are:

- Document
- Movable Modal dialog
- Modal dialog
- Floating

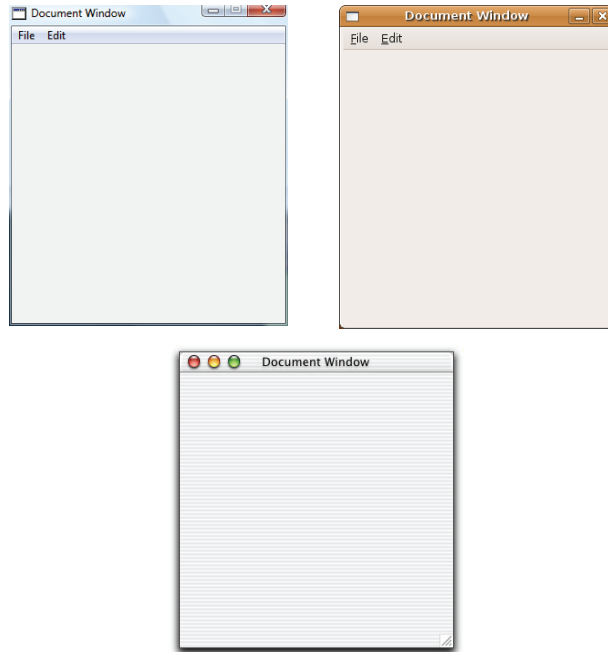
- Plain box
- Shadowed box
- Rounded (functionality was available only on Mac OS 9; it appears as a Document window on all currently supported platforms)
- Global Floating
- Sheet window (functionality available only on Mac OS X)
- Metal window (functionality available only on Mac OS X 10.2 and above)
- Drawer window (functionality available only on Mac OS X 10.2 and above)
- Modeless Dialog

The type you choose for a particular window depends mostly on how the window will be used.

Document

The Document window is the most common type of window. When you add a new window to a project, this is the default window type. It is also the window type for the default window, Window1. Document windows are most often used when the window should stay open until the user dismisses it by clicking its close box (if it has one) or clicking a button programmed to close the window. The user can click on other windows to bring them to the foreground, moving the document window behind the others. Figure 52 on page 92 shows an example of a small, blank document window. When you first launch REAL Studio, the default Desktop Application project includes one blank Document window.

Figure 52. Document windows.



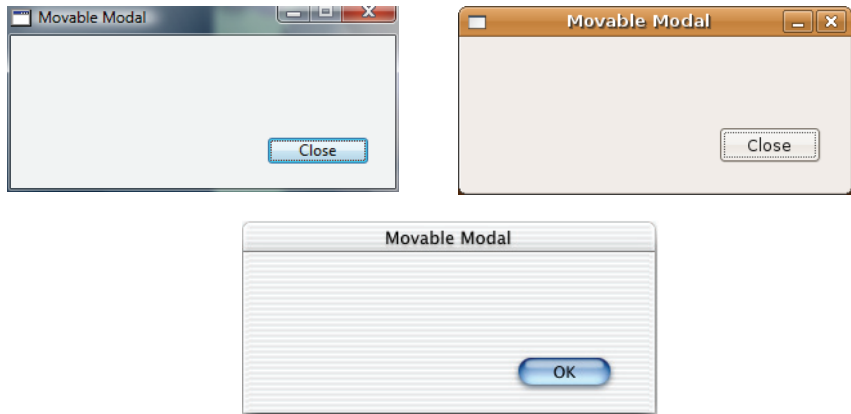
Document windows can have a close box, a maximize box, and a grow handle (making them user-resizable). Mac OS X Document windows have the standard set of red, yellow, and green buttons in the Title bar.

On Windows and Linux, the default menubar, `MenuBar1`, appears in the window by default. You can choose to display the window with no menubar by setting the `MenuBar` property of the window to `None`, or, if you have created additional menubars, choose a different menubar.

When you add a new window to your project, it defaults to the Document Window type. You can modify its type by setting the `Frame` property (see “Using a Window’s Properties Pane” on page 100).

Movable Modal

This type of window stays in front of the application’s other open windows until it is closed. Use a Movable Modal window when you need to briefly communicate with the user without the user’s having access to the rest of the application. Because the window is movable, the user will be able to drag the window to another location in case they need to see information in other windows in order to finish what they are doing in the Movable Modal window. Figure 53 shows examples of a blank Movable Modal window on each operating system.

Figure 53. Movable Modal windows.

On Windows, a Movable Modal window has minimize, maximize, and close buttons in the Title bar. In Windows MDI interfaces, the window opens in the center area of the screen rather than in the center area of the MDI window. Therefore, the Movable Modal window may open outside the MDI window.

On Linux, the window has minimize and close buttons in its Title bar.

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



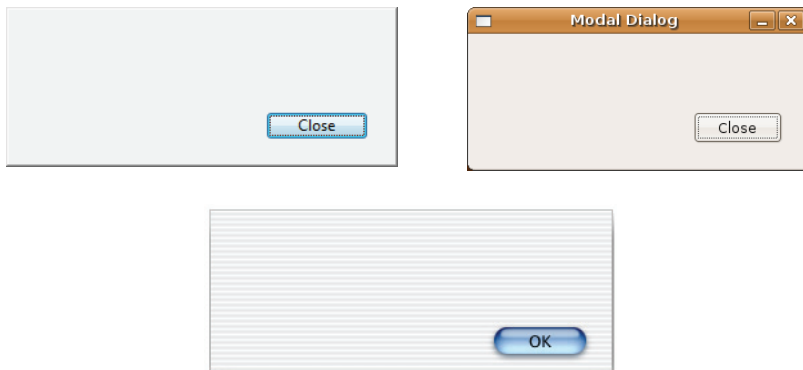
NOTE: There is one exception to the rule regarding Movable Modal windows being in front of all other windows. If a Movable Modal window or one of its controls executes code that opens a Floating window, the Floating window will be in front of the Movable Modal window. However, it is poor interface design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

Modal Dialog

These windows are very similar to Movable Modal windows. The only difference is that Modal Dialog windows have no Title bar, so they cannot be moved. On Windows, a Modal dialog box has no minimize, maximize, or close buttons. In Windows MDI applications, a Modal Dialog window opens in the center area of the screen rather than the center area of the MDI window. Therefore, a Modal Dialog box may open outside of the application's MDI window. On Linux, Modal Dialogs are modal but have a Title bar and close and minimize buttons.

The Page Setup dialog box is an example of a Modal Dialog window.

Figure 54. A Modal Dialog window.



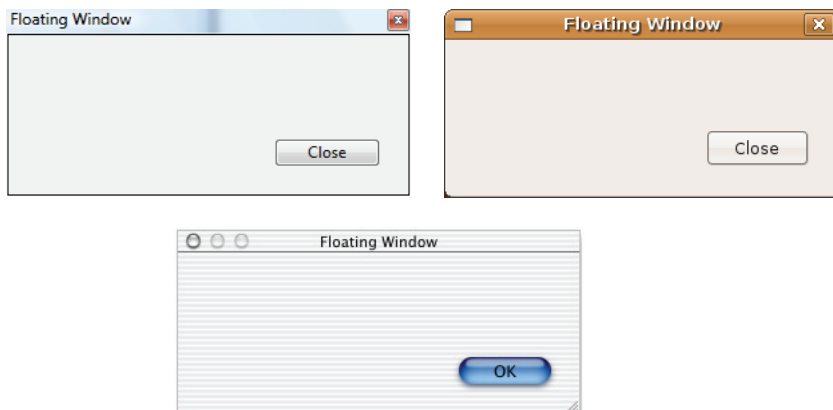
NOTE: Because Modal Dialog windows and Movable Modal windows are both modal, the same exception applies regarding floating windows opening in front of Modal windows. See the note for Movable Modal windows on page 93.

Floating

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

A Global Floating Window is a Floating window that can float in front of a particular application's window or all applications' windows. They are described in the section "Global Floating" on page 97.

Figure 55. Floating windows.



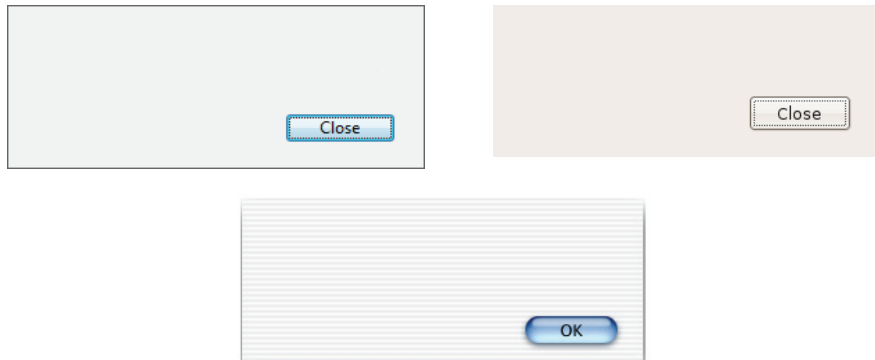
Like Document windows, Floating windows can have a close box and can be user-resizable. On Linux, Floating windows have minimize and maximize widgets.

In Windows MDI applications, a Floating window can float outside the application's own window. By default, a Windows MDI Floating window opens in the top-left area of the screen, regardless of the location of the MDI window.

Plain Box

These windows function as Modal Dialog windows. The only real difference is their appearance, as you can see in Figure 56 on page 95. Plain Box windows are commonly used for About Box windows and for applications that need to hide the desktop.

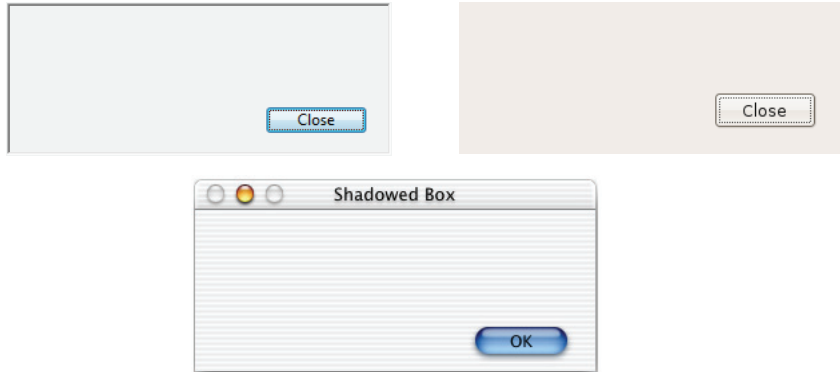
Figure 56. Plain Box windows.



On Windows MDI applications, a Plain Box window opens in the center area of the screen rather than the center area of the MDI window. Therefore, a Plain Box window may open outside the MDI window.

Shadowed Box Like Plain Box windows, Shadowed Box windows function as Modal Dialog windows. The only difference is their appearance, as you can see in Figure 57. Shadowed Box windows are commonly used for About Box windows.

Figure 57. Shadowed Box windows.



On Windows MDI applications, a Shadowed Box window opens in the center area of the screen rather than the center area of the MDI window. Therefore, a Shadowed Box window may open outside the MDI window.

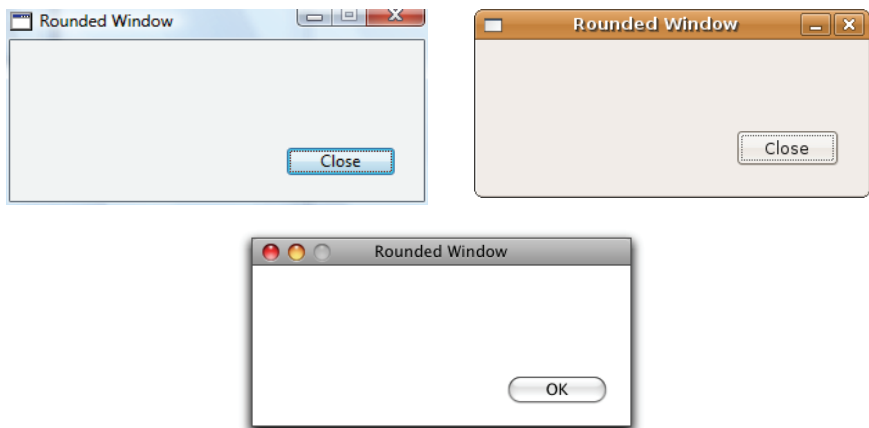
On Mac OS X, a Shadowed Box window works like a Modal Dialog box with a minimize button.

Rounded

Rounded windows act like Document windows, as the true “rounded” window is obsolete. It existed as a distinct window type on Mac OS 9 and earlier. Currently, the only differences are appearance (as you can see in Figure 58) and the fact that, on Macintosh, Rounded windows cannot have a zoom box or be resizable. On Windows, a “Rounded” window has the standard minimize, maximize, and close buttons in the Title bar and the corners are not actually rounded.

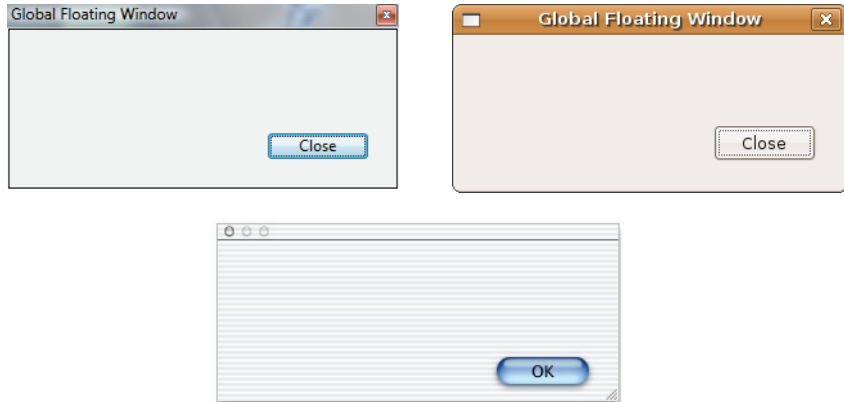
Rounded windows are not used any more and there is really no reason to choose this option instead of Document windows. Rounded windows appear as Document windows in REAL Studio Mac OS X applications.

Figure 58. Rounded windows.



Global Floating A Global Floating window looks like a Floating window, except that it is able to float in front of other applications' windows, even when you bring another application window to the front. This doesn't work for a Floating window. A 'regular' Floating window floats only in front of its own application's windows.

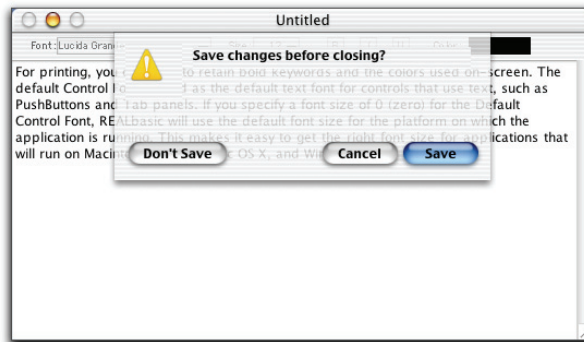
Figure 59. Global Floating windows.



On Windows MDI applications, a Global Floating window can float outside of the MDI window. By default, it opens in the top-left area of the screen.

Sheet Window A “Sheet window” is the official name for drop-down dialog boxes that were introduced with Mac OS X. Mac OS X uses them in place of modal dialog boxes. Figure 60 illustrates an appropriate usage of a Sheet window.

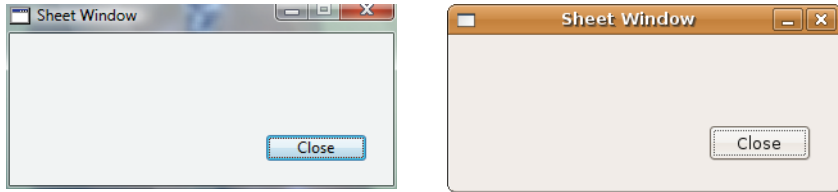
Figure 60. A Sheet window in action.



A Sheet window behaves like a Modal dialog window, except that the animation makes it appear to drop down from the parent window's Title bar. It can't be moved from that position and it puts the user interface in a modal state. The user must respond to the choices presented in the Sheet window.

Sheet windows behave as sheets only under Mac OS X. On Windows and Linux, Sheet windows behave like Movable Modal dialog windows.

Figure 61. Sheet windows on Vista and Linux.



Metal Window A Metal window uses the metallic background that Apple introduced with Mac OS X and certain Mac OS X software products such as QuickTime, iTunes, and Safari. A Metal window has a Grow handle and the standard Title bar with close, minimize, and maximize buttons of a Document window. The Metal window type requires Mac OS X version 10.2 or above.

In Mac OS X 10.5 Leopard, Apple replaced the metallic look with a gray. This is reflected in REAL Studio's Metal window type when it is used under Mac OS X Leopard.

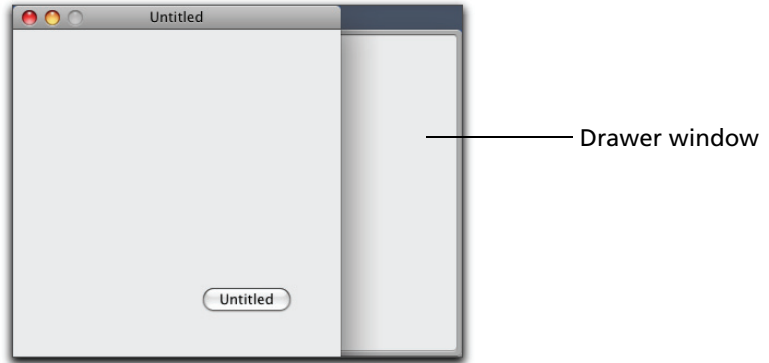
Figure 62. A blank Metal window under Mac OS X 10.2 and 10.5.



On Windows and Linux, a Metal window looks like a regular Document window.

Drawer Window

A Drawer window was introduced in Mac OS X. A Drawer slides out of a side or the top or bottom of a main window to provide supplemental information. For example, the Mac OS X Mail application uses a Drawer window to display the user's list of mailboxes. On Windows and Linux, a window of this type appears as a separate floating window.

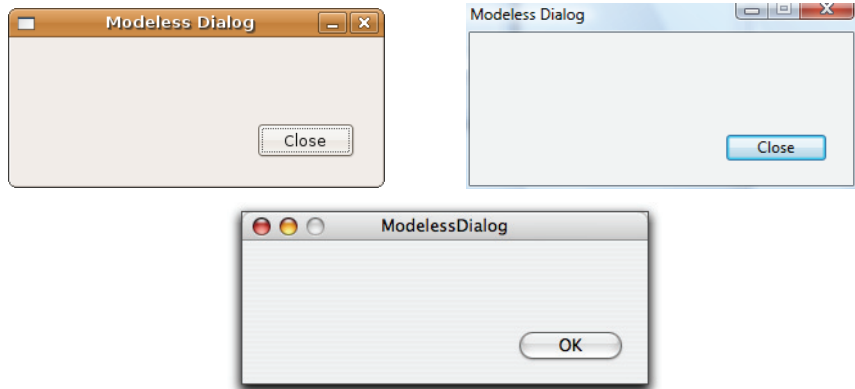
Figure 63. A blank Drawer window (Mac OS X).

Display a Drawer window using the `Show`, `ShowWithin`, or `ShowWithinModal` methods of the `Window` class. When the Drawer window is displayed, it is in a modeless state even if it is called by `ShowWithinModal`. You can also control the border from which the window slides (either side, top, bottom, or system default, in which the system figures out where there's room for the Drawer window).

Modeless Dialog

The Modeless Dialog window is similar to the Modal Dialog, except that it is paired with a parent window (usually a Document window). Unlike a Modal Dialog, it allows you to access the parent window while it is displayed. If you hide the parent window, the Modeless dialog hides as well. If you show the parent window, the dialog reappears.

The Modeless Dialog is supported on Windows and Linux. On Macintosh, it behaves as a Document window.

Figure 64. Modeless Dialog boxes.

Custom Window Types

The `Window` class has a property, `MacProcID`, that allows you to create custom window types. This property gives you more options than described in this section.

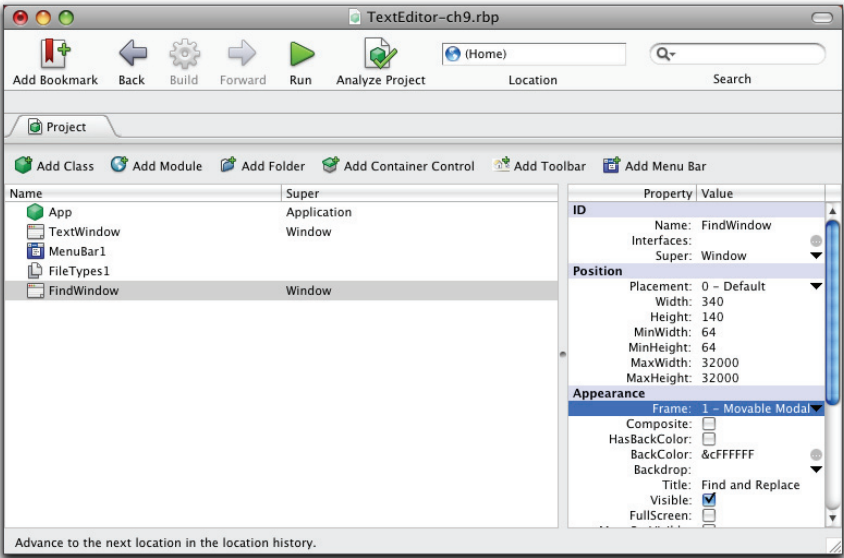
However, these custom types are supported only on Macintosh. All of the window types described in this section are cross-platform, with the limitations noted in each section.

For more information, see the discussion of the MacProcID property of the Window class in the *Language Reference*.

Using a Window's Properties Pane

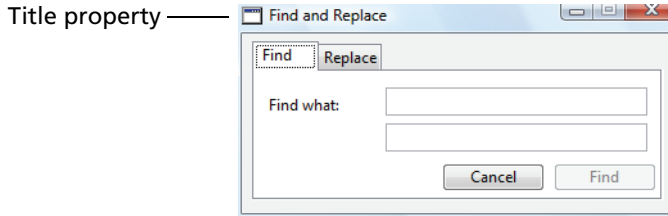
When you click on a window's name in the Project Editor, the Properties pane changes to show the window's properties that can be set *in the development environment* — as opposed to properties that can be set via code. A selected item in the project and its properties is shown in Figure 65.

Figure 65. A Movable Modal dialog window in the project and its properties.

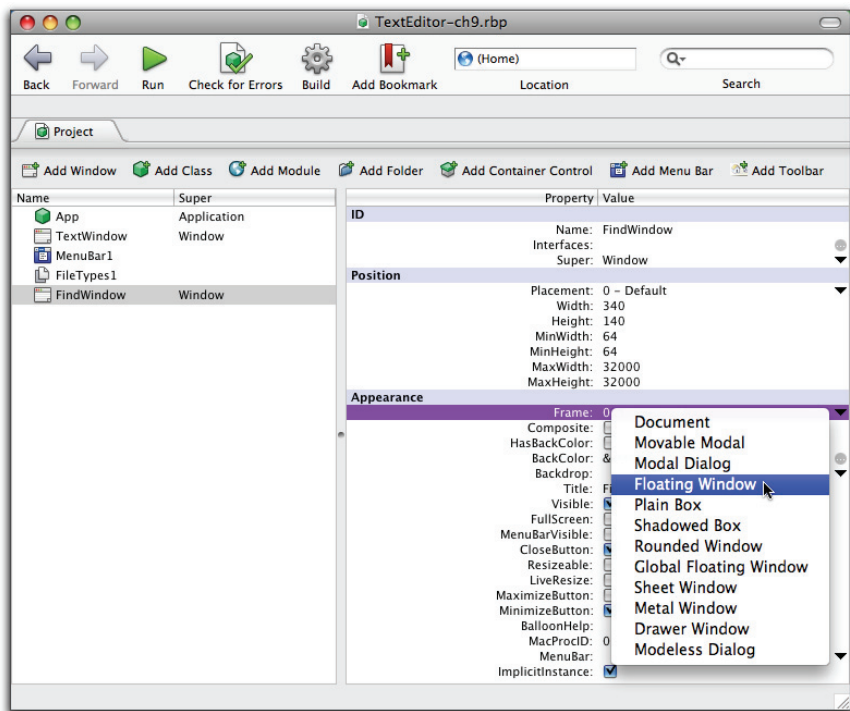


You change a window's properties by entering values into the enterable areas in the Properties pane, by making menu selections, and by selecting or deselecting CheckBoxes.

When you enter a value into an entry area, press the Return key to commit the entry. For example, this window's Title property was changed from the default text, "Untitled", to "Find and Replace". The text of the Title property appears in the window's title bar when the window appears. Figure 66 shows the window in the finished application.

Figure 66. The value of the Title property in the finished application.

In the Properties pane, drop-down menus are denoted by the downward pointing arrow on the right side of line. For example, this window's Frame property was changed from Document to Movable Modal using the drop-down menu.

Figure 67. Changing the Frame property.



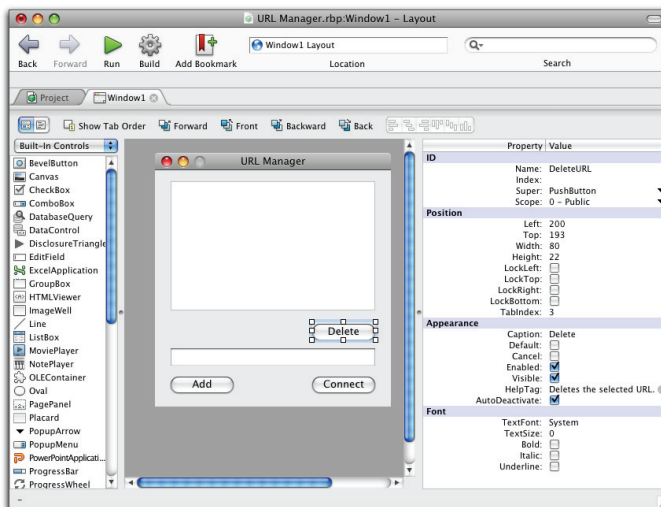
The icon with the ellipsis  (Windows and Linux) or  (Mac OS X) indicates that you can enter text by clicking the icon to display a text entry dialog. For example, the HelpTag property of the selected PushButton accepts text entry.

Figure 68. The HelpTag property with help text.




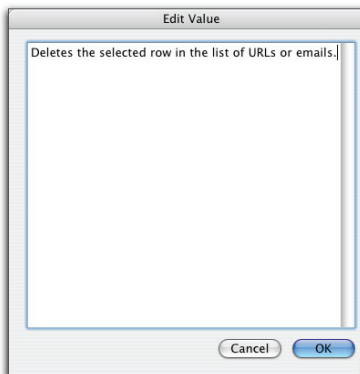
You can enter the text directly into the entry area or click the  icon to display a much larger text entry area. An example is shown in Figure 69.

Figure 69. Entering Help text in the Edit Value area.



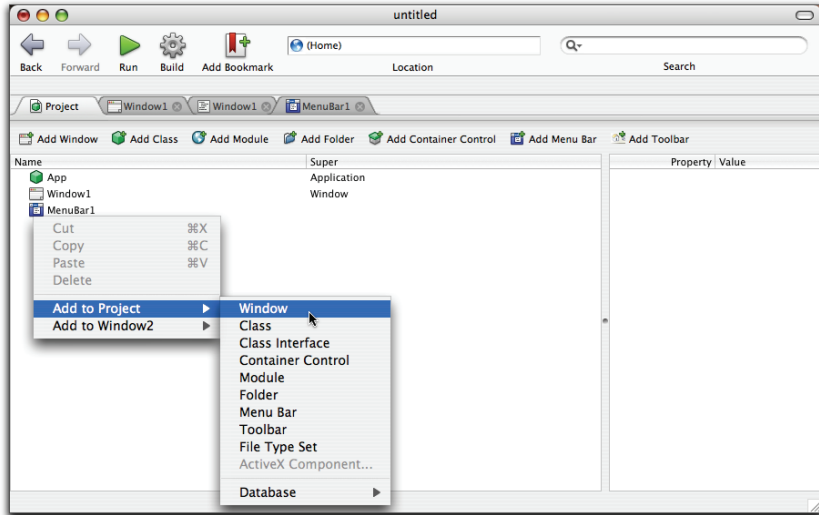
Creating Windows

When you create a new Desktop Application project, REAL Studio adds a window named “Window1” to your project automatically. This is the default project window. Unless you specify otherwise, it will appear when you launch your compiled application. For information on changing the default window, see the section “Setting the Default Window” on page 104.

There are several ways to add an additional window to your project. You can click the Add Window button in the Project Editor toolbar, use a menu command, or a contextual menu.

To add a window using the menu command, choose **Project ► Add ► Window**. To use the contextual menu, right+click (on Windows and Linux) or Control-click (on Macintosh) and choose **Add to Project ► Window**.

Figure 70. Adding a window to the project using the Project Editor's contextual menu.

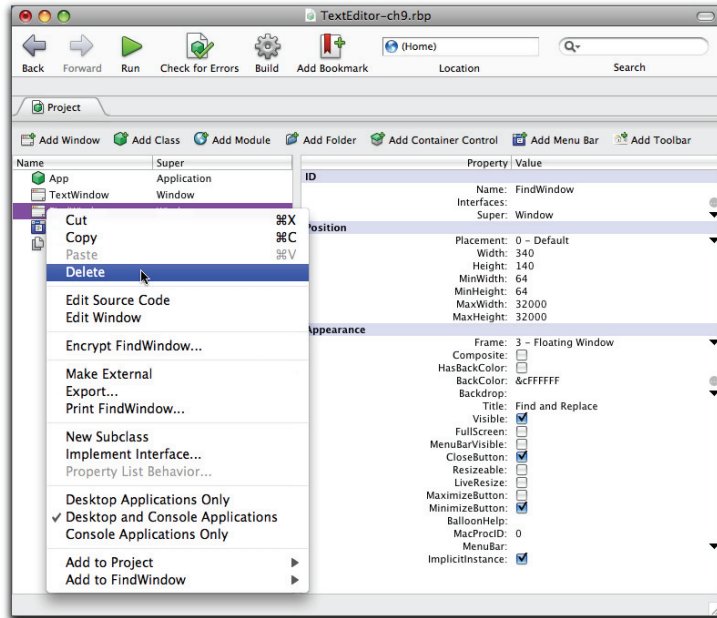


The windows in your project act as templates for windows in your application. When your application opens one of these windows, it's really opening a copy of the window. This means that your application can have several copies of the same window open at the same time. It's important to understand this when creating your user interface because there is no need to go to the extra trouble of duplicating a window in the IDE if your application needs to open two of them at the same time. For example, a text processing application uses one window template as the document window. The **File ► New** command in the finished application allows the end-user to create as many document windows as needed from the same template.

Removing Windows

To remove a window from your project, simply click on it once in the Project Editor to select it and press the Delete key or choose **Edit ► Delete**. You can also Right+click the window (Control-click on Macintosh) to display the contextual menu and remove it using the Delete contextual menu item.

Figure 71. Deleting a window via its contextual menu.



You can undo many actions in REAL Studio. For example, if you delete a window by mistake, choose Edit ► Undo (Ctrl+Z or ⌘-Z on Macintosh).

Setting the Default Window

By default, the window that is included in a Desktop Application project automatically opens when your application is launched. It is called the default window. If you have created more than one window, you can make a different window the default window. Or, you can specify that no windows will open automatically when the application starts.

You set the default window using the DefaultWindow property of the App class object. The App class is added to the project automatically when you create a new Desktop Application project.

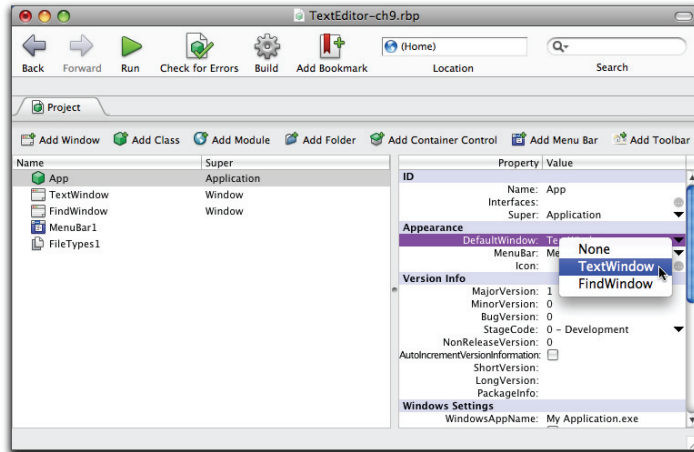
To set the default window, do this:

- 1 Click on the App class in the Project Editor.**

The Properties pane changes to show the properties of the App class. The DefaultWindow property is in the Appearance group. Its pop-up menu contains all the windows in the project as well as the choice of “None.”

- 2 Choose the desired window from the DefaultWindow pop-up menu or, if you want no window to open, choose None.**



Figure 72. Setting the default window for the project.

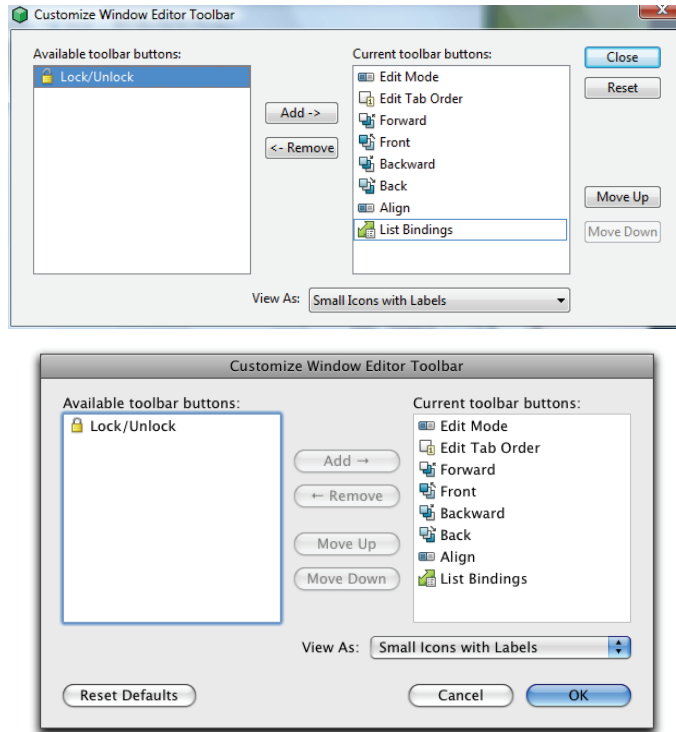
When your application launches, the default window (or no window) will open automatically.

Customizing the Window Editor Toolbar

The Window Editor toolbar has buttons for manipulating the controls in the window. By default, it has buttons for changing the Tab Order (the order in which controls are selected when the user presses Tab) and aligning controls. To modify the Window Editor toolbar, choose View ► Editor Toolbar ► Customize submenu. (Note that a Window Editor must be selected to customize the Window Editor toolbar rather than another editor's toolbar.)

The Customize Window Editor Toolbar dialog box appears:

Figure 73. The Customize Window Editor Toolbar dialog box.



The Customize Window Editor toolbar dialog box uses a “mover” interface to configure the toolbar. Listed in the right panel are the current items in the toolbar. The left panel contains any available items.

The following operations are available:

- To add an item, highlight it in the left panel and click the Add button.
- To remove an item, highlight it in the right panel and click the Remove button. This moves the item to the list on the left.
- To reorder an item, highlight it in the right panel and click either Move Up or Move Down or select an item, drag it to the desired location, and drop it between two items. The order in which the items are listed is the left-to-right order in the toolbar.
- To change the appearance of the items in the toolbar, choose an item from the Display As drop-down menu. Your choices are:
 - Big icons with labels.
 - Small icons with labels,
 - Big icons (no labels),
 - Small icons (no labels),

- Labels only.
- To reset the toolbar to the default toolbar, click the Reset button (Windows and Linux) or the Reset Defaults button (Macintosh).

Encrypting Windows



You can encrypt (protect) or decrypt (unprotect) a window from the Project Editor. A protected window cannot be opened in a Window Editor and no one can access any code associated with the window or any of the controls in the window. It's a good idea to encrypt any REAL Studio items that you want to sell to others. They will be able to use the item in their project but cannot modify your code.

Encryption is supported only in the REAL Studio Professional and Studio editions. Decryption is supported in all editions.

To encrypt a window, you supply a password which can be used to decrypt it later.

To encrypt a window, do this:

- 1 **Right-click on the window in the Project Editor (Control-click on Macintosh) and choose Encrypt from the contextual menu (see Figure 71 on page 104) or choose Edit ► Encrypt.**

The Encrypt *Window* dialog box appears, as shown in Figure 74.

Figure 74. The Encrypt *Window* Dialog box.



- 2 **Enter and confirm a password for decryption.**

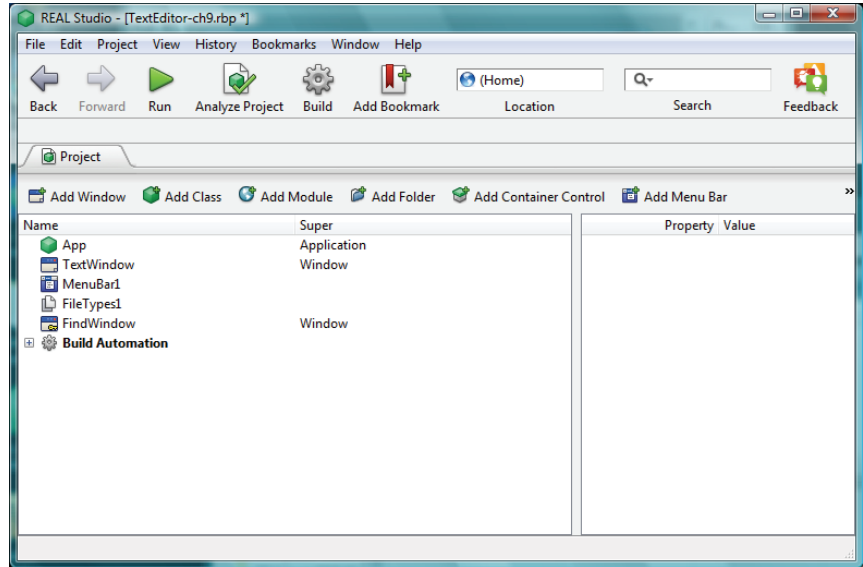
The Encrypt button will not be enabled if the “Confirm” password does not match the initial entry.

Important: Don’t forget the password.

- 3 **If you want the window to be accessible only to REAL Studio 2006 Release 3 and above, then click the “Use REAL Studio 2006r3 Encryption” checkbox.**
- 4 **When you are finished, click Encrypt.**

An encrypted window appears in the Project Editor with a small key in the lower right corner of the window icon. In Figure 75, the “FindDialog” window has been encrypted.

Figure 75. A project with an encrypted window.



When a user tries to open an encrypted window in a Window Editor, REAL Studio displays the *Decrypt Window* dialog box, shown in Figure 76.

To edit the window and/or its code, you must decrypt it using its encryption password.

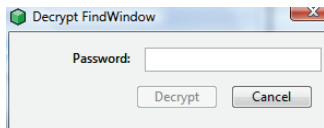


To decrypt an encrypted window, do this:

- 1 Right-click on the window's name in the Project Editor (Control-click on Macintosh) and choose Decrypt from the contextual menu or click on the window's name and choose Edit ► Decrypt.**

The *Decrypt Window* dialog box appears.

Figure 76. The Decrypt Window dialog box.



- 2 Enter the decryption password and click Decrypt.**

In a few moments, the key will disappear from the window's icon, indicating that it has been successfully decrypted. If you entered an incorrect password, a dialog box will inform you of that fact.

If you don't know the password, there is no way to decrypt the window.

Message Dialog Boxes

REAL Studio offers an especially quick way to create standard message dialog boxes. There are two built in language commands that create message dialog boxes auto-

matically. They bypass the process of creating a window, adding it to the project, and adding controls to it via the Controls pane and the Window Editor. These commands are the `MsgBox` function and the `MessageDialog` class.

These commands create only a limited range of window types. The windows are message boxes that have an icon, some text, and one or more buttons. These commands cannot be used to create dialog boxes or any other type of window that displays information in a different form. The only user input that is supported is the selection of a button to click. Also, you do not have the ability to arrange the buttons, text, and icon freely. If you need any of these features, you should create a window and add it to the project.

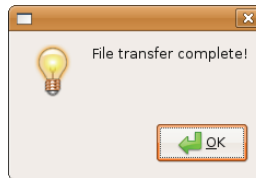
The MsgBox function

Of the two commands, the `MsgBox` function is the simpler one. Use it when you want to present a brief message to the user in the form of a modal message box. Simply pass the text that you want to display in the message box to the `MsgBox` function. For example, the following message is shown when a file upload has completed successfully:

MsgBox "File transfer complete!"

This line of code displays a Message box with the message and one button that the user can click to dismiss the window. This message box looks as shown in Figure 77.

Figure 77. The basic MsgBox message box.



If the string that you pass to `MsgBox` contains a null character or unprintable characters, you should first filter them out prior to using the `MsgBox` function. The null character will terminate the string, no matter where it appears.

The `MsgBox` function has two optional parameters that provide some customization. You can pass an integer that indicates that the function should display additional buttons, indicate which button is the default button, and set the icon to be displayed.

If you use the optional parameters, the `MsgBox` function returns an integer that tells you which button the user clicked. If you display more than the one button, you should examine the value returned by the `MsgBox` function.

The MessageDialog Class

Use the `MessageDialog` class when you need to design more complex message dialog boxes than shown in Figure 77. With the `MessageDialog` class, you can present up to three buttons and control their text and functionality. You can also present subordinate explanatory text below the main message.

However, since `MessageDialog` is a class, you cannot accomplish all of this with one line of code. You need to declare a variable as type `MessageDialog`, instantiate it, set its properties, and handle the result returned, which tells you which button the user pressed.









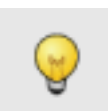



Before using the `MessageDialog` class, its best to read about REAL Studio classes and how they work. Please refer to Chapter 10 for this information. After that, read the section on the `MessageDialog` class in the *Language Reference*. This section is intended to give you an idea of the features that this class offers.

A `MessageDialog` can have up to three buttons, which are based on the `ActionButton`, `CancelButton`, and `AlternateActionButton` classes. They have the following properties:

Property	Description
Caption	The text displayed in the button.
Visible	Set to True to show the button.
Default	Set to True to highlight the button as the default button in the <code>MessageDialog</code> . By default, the <code>ActionButton</code> 's Default property is True.
Cancel	Set to True to indicate that the button will respond to the Escape key. On Macintosh, it will also respond to the Command- sequence. Only one button in a dialog can have the its Cancel property set to True. Some operating systems do not permit one button to be both the Default button and the Cancel button.

By default, only the `ActionButton` is shown, but you can show the others simply by setting their Visible properties to True.

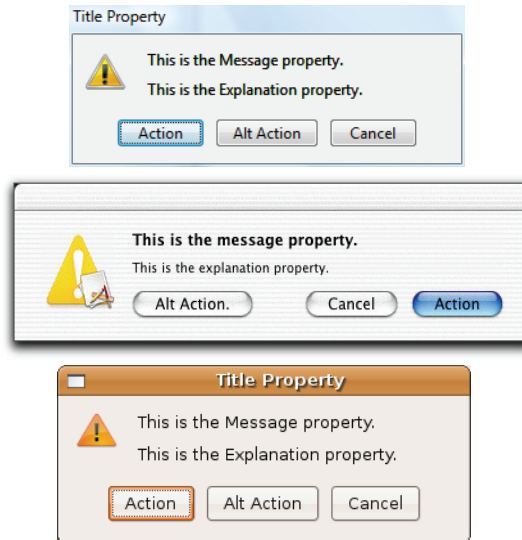
In addition, you can set the text of the message, the subordinate explanation, the type of icon shown in the dialog (no icon, Note, Warning, Stop, or Question), and the title. Not all of the icons are presented in Mac OS X. The following table shows how the icons appear on each platform and value of the Icon property.

Platform	Value of the Icon Property			
	0 (Note)	1 (Caution)	2 (Stop)	3 (Question)
Windows				
Mac OS X				
Linux				

You present the customized alert by calling the ShowModal method of the MessageDialog class.

The positions of the three buttons and the Message and Explanation properties are shown in Figure 78.

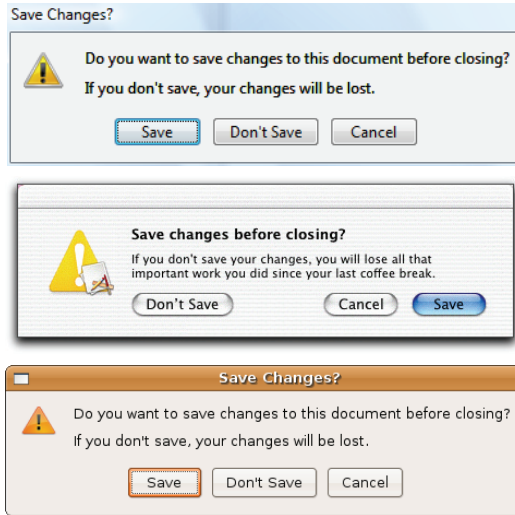
Figure 78. The positions of the buttons and text.



After the user clicks a button, the MessageDialog returns a MessageDialogButton object, which is either an ActionButton, CancelButton, or AlternateActionButton. By determining the type of object that was returned, you learn which button the user pressed. You can also examine the returned object's properties, if necessary.

Figure 79 shows a sample MessageDialog box. See the entries in the Language Reference for the MessageDialog class and the MessageDialogButton classes for more information and examples.

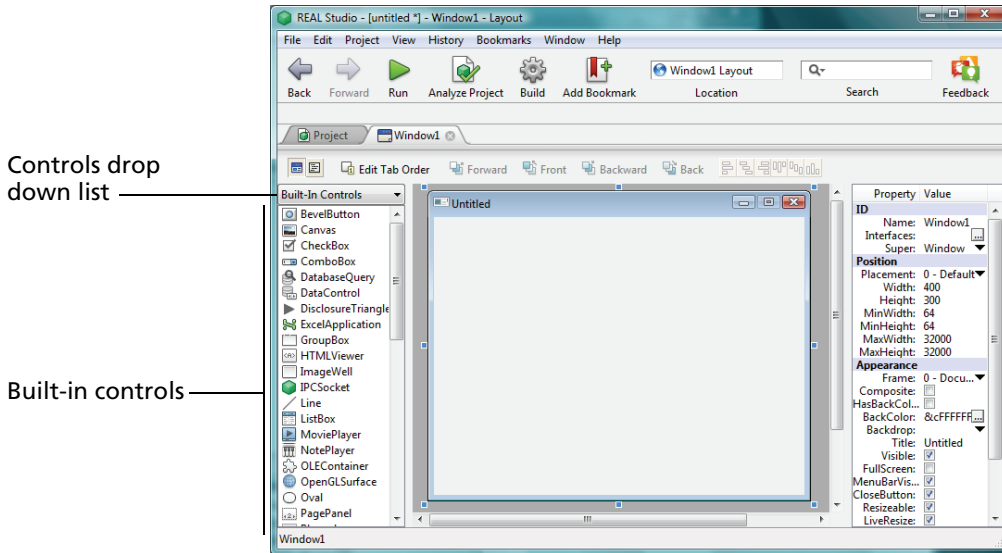
Figure 79. The customized MessageDialog alert.



Interacting with the User Through Controls

Users provide information to your application through user interface controls. REAL Studio provides a tremendous amount of flexibility in this area. Not only are there many built-in controls, but you can even create your own controls (you will learn more about this later). REAL Studio's built-in controls are added to windows using the Controls pane, shown in Figure 80.

Figure 80. The Controls pane showing the Built-in Controls list.



The Controls drop-down list lets you select among three types of controls for display in the Controls pane, plus a customizable list of controls. The three types of controls are Built-In controls, Project controls, and Plug-in controls. You use the Controls drop-down list to choose the type of controls that is displayed:

- **Built-In Controls:** The controls that are built into REAL Studio. This is the default choice. The built-in controls are shown in Figure 80.
- **Project Controls:** Custom controls that are based on built-in controls. You can create customized versions of any of the built-in controls. If you do so, they appear in this list automatically. Project controls are also listed in the Project Editor as classes. By default, this list has one item, Object. It is a “placeholder” item that you can convert to a control or object of any type. You do this by dragging an instance to the window and changing its Super property to the desired type of object. For information on creating custom controls, see the section “Understanding Subclasses” on page 533 and the section “Creating Custom Interface Controls with Classes” on page 583.
- **Plug-in Controls:** Controls that you added to REAL Studio as by installing plug-ins. Third-parties can create custom controls in the form of plug-ins that are

installed by placing the plug-in in the Plugins folder in the REAL Studio IDE folder. This list is empty if you have no control plug-ins installed.

- **All Controls:** The Built-in, Project, and Plug-in controls in an alphabetized list.
- **Favorites Controls:** Controls that you have added to your list of favorite controls. See the following section for information on creating your Favorites list.

A few of the items in the Built-in Controls pane add objects to a window that are not visible to the end user. Technically, they are not really controls. They are in the list for your convenience. You can use them to add capabilities to the application that become available when the window is open. For example, you can use the TCPSocket control to support network communications via the TCP/IP protocol. Other controls support Microsoft Office Automation. They enable you to automate PowerPoint, Excel, and Word via REAL Studio, but don't add a visible interface item to your application's windows. When you add one of these controls to a window, you then add code to the control.

Favorites Controls

To make the Controls pane more manageable, you can create a list of frequently used controls. Your Favorites list can include controls from all three built-in lists, the Built-In controls, Project controls, and Plug-In controls. A shorter list that is made up of only the controls you use frequently will be easier to work with. REAL Studio ships with a short list of frequently used controls in the Favorites list.

If you use Project and Plug-In controls as well as the built-in controls, you can combine all types of controls in your Favorites list of controls. For example, in Chapter 10 you will learn how to create a custom TextArea control that is based on the built-in TextArea control but prohibits the user from copying the text in the control.

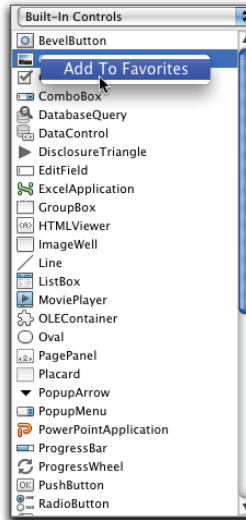
To add a control to your Favorites list, do this:



- 1 If it is not already displayed, use the Controls drop-down list to switch to the type of Controls list that contains the control you want to make a Favorite.**
- 2 Right+click (Control-click on Macintosh) on the desired control.**

A contextual menu appears.

Figure 81. The Add to Favorites contextual menu item.



3 Choose Add to Favorites from the contextual menu.

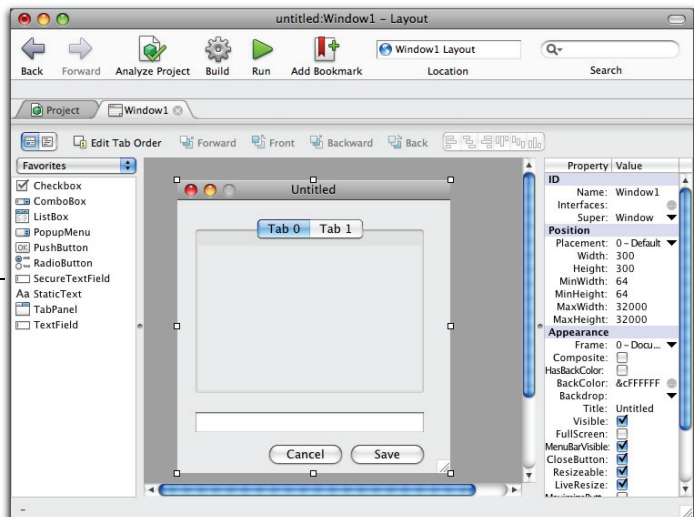
When you switch to your Favorites list, the selected control will have been added to the end of the list. (If the control is already in the Favorites list, the contextual menu item does not appear.)

4 Repeat this process to add additional controls from the current list or switch to another type of list and add additional controls.

For example, Figure 82 shows both Built-in and Project controls in a Favorites list.

Figure 82. A Favorites list with both Built-in and Project controls.

A Project
control
based on
TextField





To remove a control from your Favorites list, do this:

- **Right-click on the control (Control-click on Macintosh) and choose Remove from Favorites from the contextual menu.**

The control will still be available from the list that “owns” it—built-in, project, or plug-in.

Adding, Changing, and Removing Controls

REAL Studio makes adding, changing, and removing controls easy.

Adding Controls to a Window

There are several ways to add a new control to a window:

- Drag a control from the Controls pane to the Window Editor,
- Double-click a control in the Controls pane,
- Select a control in the Controls pane and draw a region in the Window Editor,



To add a control by dragging, do this:

- 1 **If the Window Editor for the window is not visible, click its tab or double-click its name in the Project Editor.**
- 2 **Display the desired type of Controls pane with the Controls drop-down list and then drag the control from the Controls pane to the desired location in the window and drop it onto the window.**

REAL Studio adds an instance of the control at the location of the drop.



To add a control by double-clicking, do this:

- 1 **If the Window Editor for the window is not visible, click its tab or double-click its name in the Project Editor.**
- 2 **Display the desired type of Controls pane with the Controls drop-down list and double-click the control.**

REAL Studio places an instance of the control in the window.



To add a control by selecting, do this:

- 1 **Click once on the control in the Controls pan to select it.**
- 2 **Press Enter (Return on Macintosh).**

REAL Studio adds an instance of the control at the default location in the window.



To add a control by drawing, do this:

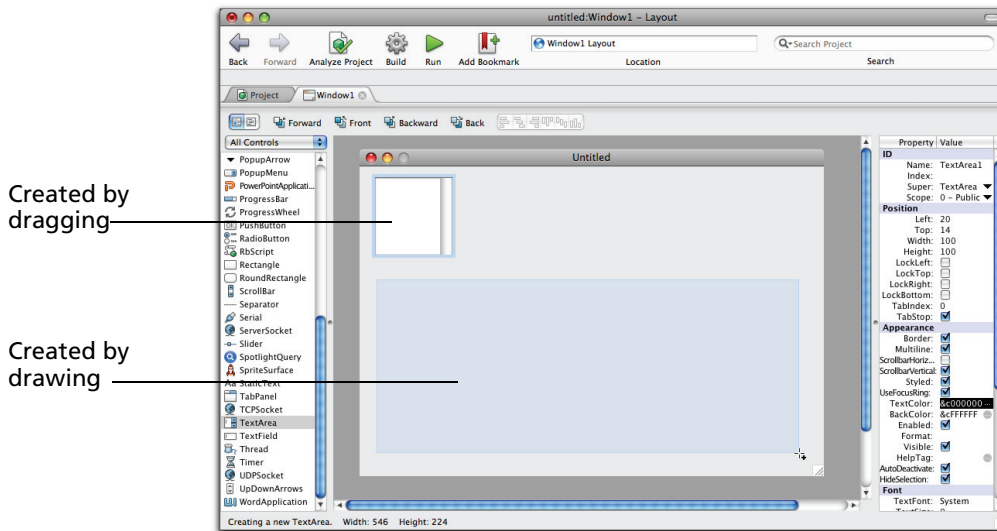
- 1 **If the Window Editor for the window is not visible, click its tab or double-click its name in the Project Editor.**

- 2 **Display the desired type of Controls pane with the Controls drop-down list.**
- 3 **Click once on the control to select it and then use the pointer to draw a region in the window.**

REAL Studio places an instance of the control at the specified location and in the specified size.

The last way of adding a control is especially convenient when the default size of the control is not appropriate. Suppose you want to add a `TextArea` control to a window. If you use either of the first two methods of adding the control, REAL Studio will use the default size for the `TextArea`. You will need to resize the control in another step. However, if you draw the region, you can specify both the location and size of the control in one step. Here is a comparison:

Figure 83. TextAreas created by dragging and by drawing an area.



To use the contextual menu to add a control, do this:



- 1 **Right+click the mouse button in the window (Control-click on Macintosh).**
A contextual menu appears. The Add item is at the top level of a hierarchical menu of objects that can be added to the project. It displays the full REAL Studio object hierarchy in the form of a hierarchical menu.
- 2 **Use the Add menu item to choose the control you wish to add from the list of built-in controls.**

The Add submenu item enables you to add any object to a window. The submenu command exposes the list of REAL Studio classes and the REAL Studio object hierarchy, which is indicated by multiple levels of submenus. The visible controls are in the Add ► Control ► RectControl submenu.

When you choose an object that is not a control, the object is represented in the window by a generic REAL Studio icon. When you click on the generic icon, it is selected in the window's Code Editor, just as if it were a visible control. Of course, it will not be visible in the built application.

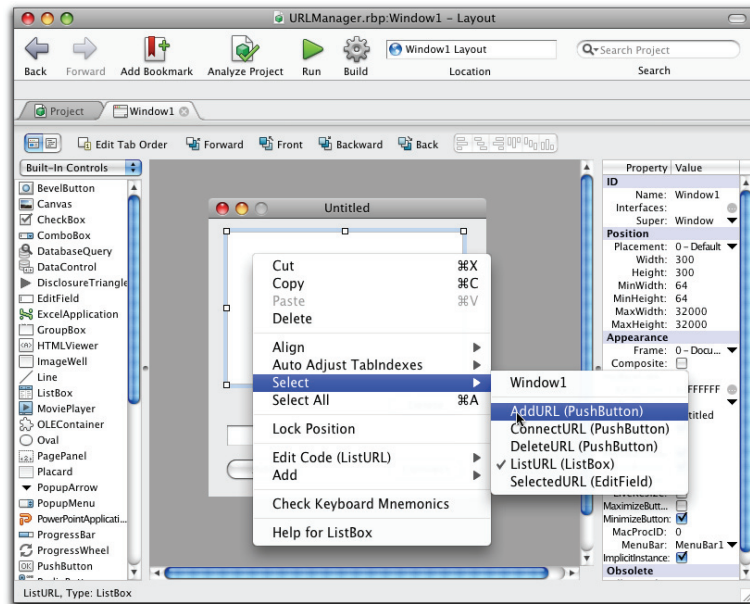
You can modify its properties in the Properties pane and add code to the object using the Code Editor, providing it has built-in events.

Selecting Controls

Controls can be selected in one of three ways: using the mouse button, the Tab key, or the contextual menu. The easiest way to select an individual control is to click on it. If you click on a control, it will be selected.

The contextual menu contains two items for selecting controls, Select and Select All. The Select All command selects all controls in the window and Select has a submenu that lists all of the window's controls. This enables you to select any control in the window even if it is not currently visible (see “Selecting Invisible Controls” on page 119 for more information). Choose a control from the submenu to select it.

Figure 84. Selecting a control with the window's contextual menu.



You can also move through the controls in a window by pressing the Tab key. Each time you press the Tab key, REAL Studio will move from one control to another. This is also the order the user will move through the controls when using the Tab key. For more information, see “Changing The Tab Order” on page 179. Holding down the Shift key while pressing the Tab key selects controls in reverse Tab order. If only one control is selected, REAL Studio draws resize handles at each corner of

the control. You can select several controls by holding down the Shift key as you click on the controls. You can also draw a marquee around a group of controls to select them.

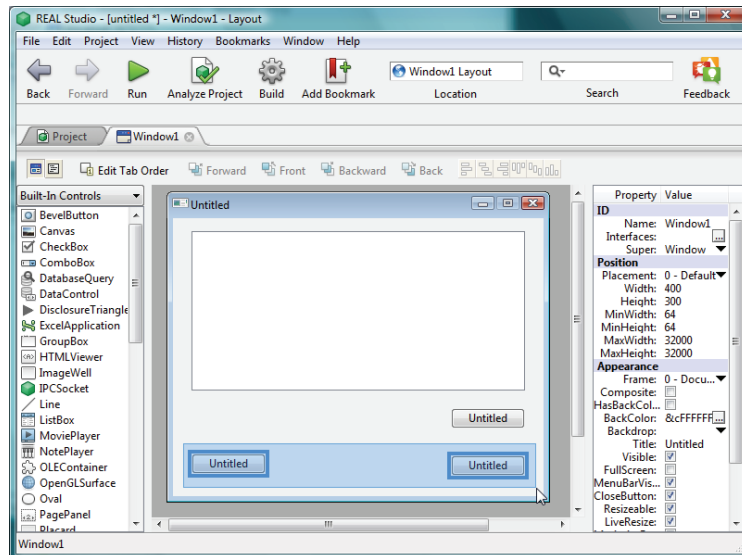


NOTE: You can also choose Edit ► Select All to select all the controls in the window. After all the controls are selected, you can Ctrl+click (Command-click on Macintosh) on a control to remove it from the selection.

You can select two or more controls by clicking on one control and then holding down the Ctrl key (Command key on Macintosh) and continue clicking on other controls. All the controls you click on are then selected.

If the controls you wish to select are next to one another, you can draw a selection rectangle around them using the mouse. Drag diagonally (i.e., from top-left to bottom-right) and release the mouse button. A translucent marquee surrounds the selected controls.

Figure 85. Selecting two controls using a selection rectangle.



Selecting Invisible Controls

It's possible for a control to disappear. For example, if you give a control a large enough negative or positive Left or Top property, it will disappear off the edge of the window. Or, if you give it Width and Height properties of zero, it will remain in its position but become invisible. You would have a tough time clicking on it to select it.

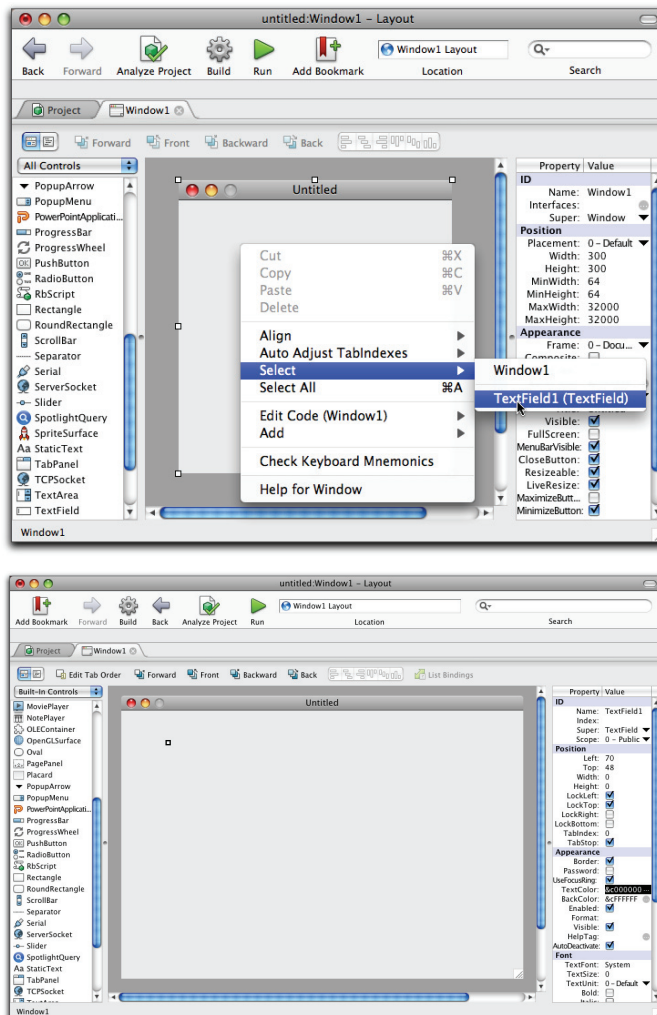
Negative values of the Left property are not recommended. If you want to temporarily move a control off the window, a preferred strategy is to move it to the right of the window and onto the visible pasteboard (the gray area that surrounds the window itself in the Layout editor). Enlarge the IDE window, if necessary, so

that the control is still visible. Do not move the control further to the right than needed, as REAL Studio needs to use memory to remember its exact location.

Another recommended strategy is to set the Visible property of the control to False instead of using the pasteboard.

However, you can always use the window's contextual menu to select an invisible control. The Select submenu will always list all controls that belong to the window even if they are not visible. In Figure 86, a Text that has its Width and Height properties set to zero is selected in this manner. When selected, only a selection handle appears. Once it is selected, the Properties pane changes to show its properties. You can then use the Properties pane to change its position and size properties (see Figure 86. below and the following section).

Figure 86. Selecting a TextField that has zero width and height.



In the second illustration in Figure 86, notice that the Properties pane has changed to show the invisible TextField's properties. Its Width and Height properties are both zero. You can now change its Width and Height properties so that it will become visible.

Changing a Control's Position

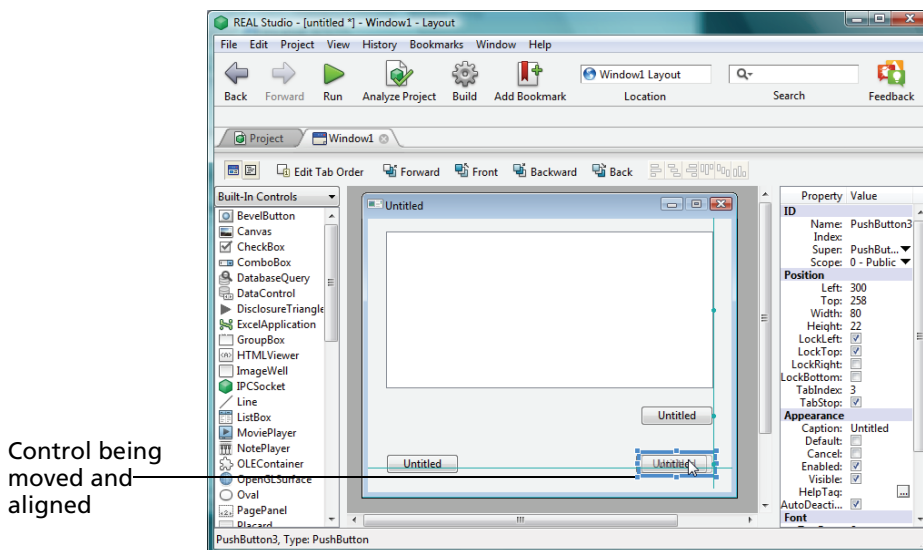
A control's position can be changed by dragging the control using the mouse, by using the arrow keys (to move it one pixel at a time in the horizontal or vertical directions) and by changing the properties in the Position group in the Properties pane.

The Left and Top properties determine the location of the top-left corner of the control, while the Width and Height properties determine its size. You can always use these properties to position and size controls precisely. For example, the invisible TextField in Figure 86 can be repositioned by changing its Left and Top properties.

Using Alignment Guides

When you drag a control, you can align it with other objects in the window by taking advantage of built-in horizontal and vertical alignment guides. When the object you are dragging is near the horizontal and/or vertical side of another object, alignment guides temporarily appear, allowing you to position the object precisely. Figure 87 on page 121 illustrates the process of aligning a PushButton with the baseline of another PushButton and the right side of a TextArea control.

Figure 87. Alignment guides help you position objects.



Using the “Lock” Properties to Set a Control’s Position

Any visible control has four Boolean properties that you can use to “lock” the control’s horizontal or vertical edges to the corresponding horizontal or vertical edges of the window. These properties are `LockLeft`, `LockRight`, `LockTop`, and `LockBottom`. When one of these properties is turned on, the space between the designated edge of the control and the corresponding edge of the window remains the same when the user resizes the window.

Beginning with REAL Studio 2009r5, when you add a new control to a window, `LockLeft` and `LockTop` are `True` by default. By default, `LockRight` and `LockBottom` are `False` for new controls.

You use these properties to tell REAL Studio to resize or move the control when the user resizes the parent window. For example, if you use a `TextArea` as a text processing window, you will want to align the edges of the control to the window and then use the `LockLeft`, `LockRight`, `LockTop`, and `LockBottom` properties to resize the control automatically when the user resizes the window. Figure 88 on page 122 illustrates this:

Figure 88. A `TextArea` used as a text processor.

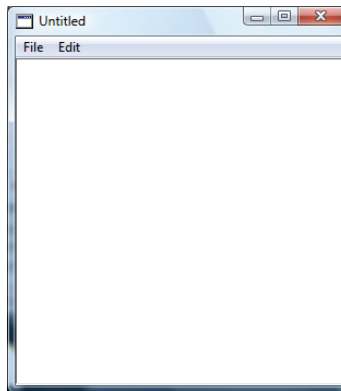
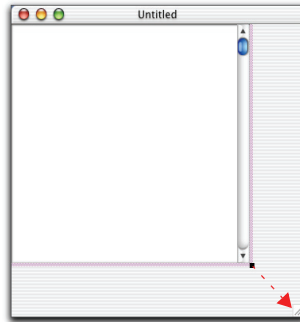
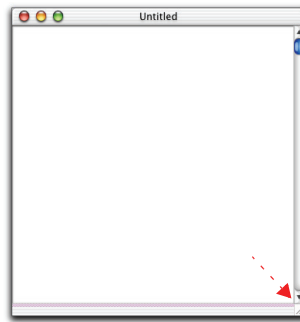


Figure 89 shows how these four properties work.

Figure 89. Resizing the window with the “Lock x” properties on and off.



Lock properties off: the TextArea maintains its size and position when the window is resized.

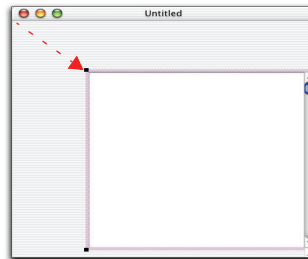


Lock properties on: the TextArea grows and shrinks as the window is resized.

In this example, if only the LockRight and LockBottom properties are set, the TextArea will move down and to the right as the window’s grow box is moved in that direction.

Figure 90. Enlarging a window when LockLeft and LockTop are not set.

Only the TextField’s bottom and right sides are locked.



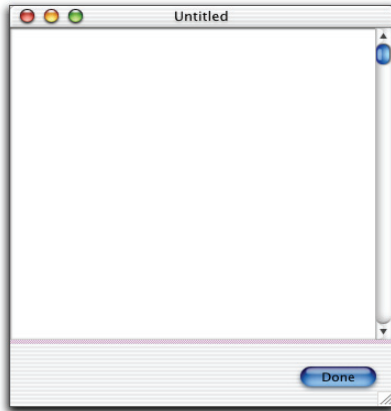
The situation gets more interesting as we add another control. Consider the layout in Figure 91.

Figure 91. A PushButton below the TextArea.

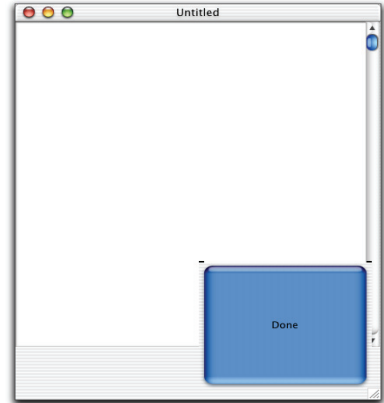


The `PushButton` should be locked to the bottom and right side of the window only; we want it to maintain its relative position to the `TextArea`. However, we do not want to resize the `PushButton` as the window is resized. This is illustrated in Figure 92.

Figure 92. Effects of resizing a window when a `PushButton` is locked.



The `PushButton` is locked to the bottom and right side; it maintains its shape and relative position to the `TextField` as the window is enlarged.



The `PushButton` is locked to all four sides; it maintains the vertical and horizontal distances to the left and top of the window, forcing it to grow as the window is enlarged.

An unusual effect is achieved if the user reduces either the horizontal or vertical dimension of the window when the `PushButton` is locked to the sides that are getting closer to each other. In Figure 93 on page 125 the vertical dimension is decreasing and the control is locked to both the top and bottom.

Since the `PushButton` is supposed to maintain its distance from both the top and bottom, the control can appear to be pushed right out of existence as the height of the window is reduced. In Figure 93 on page 125, the `PushButton` is trying to maintain its distance from both the top and bottom edges of the window simultaneously as the height of the window decreases. Eventually, it will collapse into REAL Studio's version of a Black Hole. In the right image in Figure 93, the `PushButton` has folded up upon itself (i.e. in effect, its bottom edge is above its top edge).

Figure 93. Effect of reducing the height of a window on a PushButton that is locked to both the top and bottom.

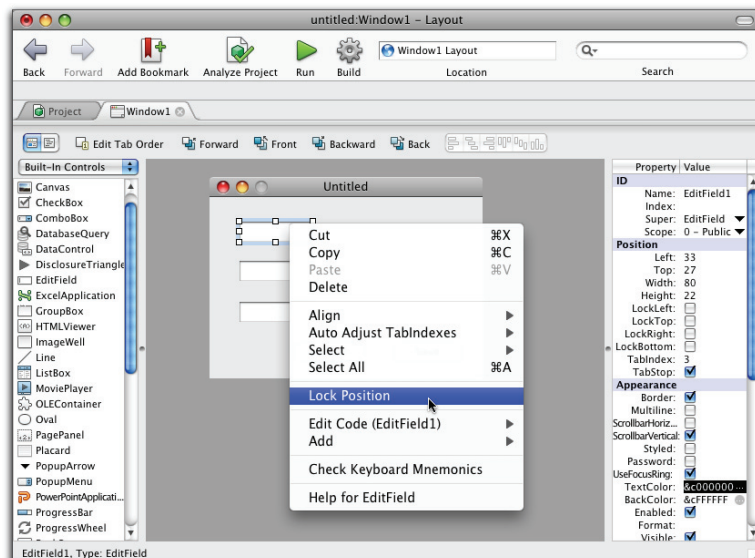


This happens because there is space both above and below the PushButton. The TextArea, on the other hand, doesn't get pushed out of existence because there is no space between the top of the control and the top of the window in Figure 91 on page 123.

Locking a Control to its Absolute Position

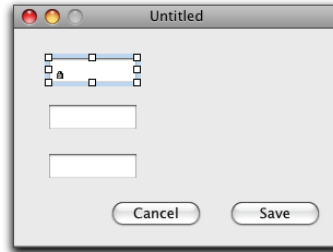
When you are finished adjusting a control's location in a window, you can lock it into place. This will prevent you from accidentally moving it when you select it to read or change its properties. When a control is locked to its position, its size is also locked. To lock a control, right+click on the control and choose Lock Position from its contextual menu (You can also add the Lock/Unlock button to the Window Editor toolbar. For more information, see the section "Customizing the Window Editor Toolbar" on page 105.).

Figure 94. Locking a control into position.



A locked control is shown in the Window Editor with a padlock. You can continue to select the locked control, read and edit its properties, and adjust its Tab Order (if applicable). A locked control can be selected but not moved.

Figure 95. A locked control in a window.



If you lock a child control, it will keep its position relative to its parent, but it will move if you move the parent control. For example, if you lock `RadioButtons` inside a `GroupBox`, you can move the `GroupBox` and the `RadioButtons` will move along with the `GroupBox`. However, you cannot move a locked `RadioButton` inside the `GroupBox`.

You can unlock the control by right+clicking on it and choosing `Unlock Control` from the contextual menu.

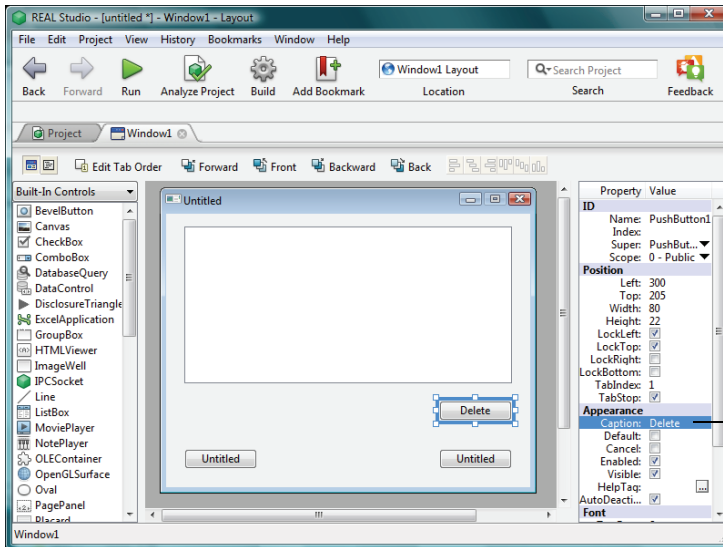
Changing a Control's Properties with the Properties Pane

Some changes to a control can be made without the Properties pane. For example, controls can be rearranged by simply dragging them from one place to another inside the window. However, most of the changes you make to controls will be made using the Properties pane.

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

Some properties are entered by typing, while others with on/off-type values are represented by a `CheckBox`. If the property is set by typing, you can use either the `Enter` or the `Return` key to commit the new value. Some controls' `Caption` or `Text` properties can be set selecting the control in the Window Editor and typing the new text immediately.

Figure 96. Changing the Caption property of a PushButton.



The default caption, "Untitled" was replaced by "Delete".

Numeric Properties

If the property you want to set is numeric — such as the Left, Top, Width, or Height properties or the number of columns in a ListBox — you can either enter a number or an expression that evaluates to a number.

If you want to enter an expression, your available operators are: +, -, *, /, \ (integer division), % (mod), and ^ (power). You can also use parentheses to control the order in which subexpressions are evaluated. You can also use references to property values by name, allowing you to write expressions such as "Top*2" or "Width+Left."

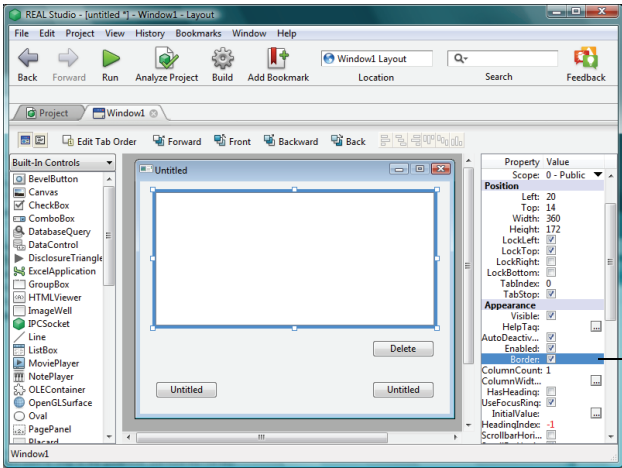
If an expression is invalid on its own, the current value of that property will be prepended; this allows you to (for example) enter "*2" as a handy shortcut for doubling the current value when multiple objects are selected. To add a value to the current value, use "+ +value". For example, to add 10 use "+ + 10", since "+ 10" will be treated as "10".

Boolean Properties

The values of Boolean properties are shown as CheckBoxes in the Properties pane. A value of False is indicated by an unchecked CheckBox and a value of True is indicated by a checked CheckBox.

You change a value from False to True or vice versa by clicking on the CheckBox. Or, you can select the boolean property by clicking on its name and then pressing the Spacebar. In Figure 97 the user has selected the Border property by clicking on its name. Pressing SpaceBar will toggle this property's value.

Figure 97. Selecting a Boolean property.

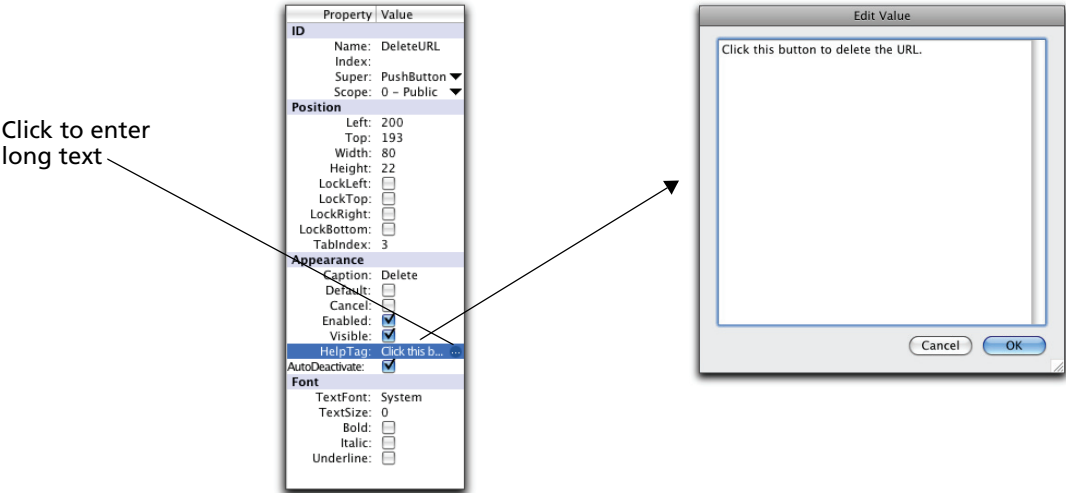


With the HasHeading property selected, you can press Spacebar to change its value from True to False

Text Properties

Properties such as the text that appears in a TextField or the caption of a PushButton are entered simply by typing into the text area. If the text you want to enter is long, you can click the text icon to bring up a modal window into which you can enter more lengthy text. A typical Text window is shown in Figure 98.

Figure 98. Entering text for HelpTag in a separate window.



Constants

You can also choose to enter a constant as a value in the Properties pane. For example, you can create a constant that contains the caption for all of the “Accept” PushButtons in your application. If you want to change the caption, you only need to change the value of the constant rather than edit the values for each PushButton.

Constants are especially useful for applications that are deployed in more than one language. Instead of using literal text as property values, you use constants for all text that the user sees. This includes menus and menu items as well as windows. REAL Studio enables you to use different values of each constant for different languages.

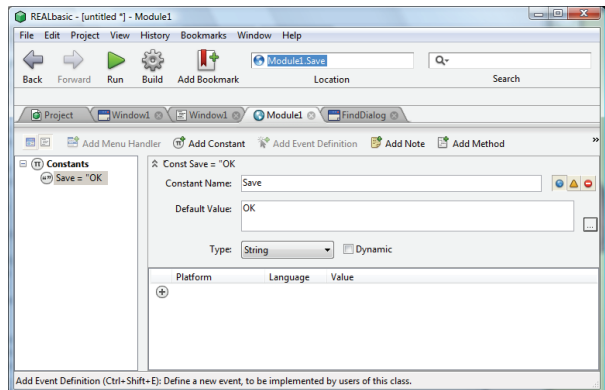
A good place to create constants for this purpose is in a module. For information on how to create a constant, see the section “Adding a Constant to a Module” on page 376. A constant can be given Global scope only in modules. In Figure 99, a global constant is being created in a module and then referred to in the Caption property of a PushButton.

To use a constant, precede the name of the constant by the number sign, #, as the value for a property in the Properties pane. For example, if you want to use a global constant (in a module) named “Save”, you would refer to it in the Properties pane as “#Save”. If it were a public constant in Module1, you would refer to in the Properties pane as “#Module1.Save”.

Here are the Appearance properties of a PushButton that references the Global constant named “Save.”

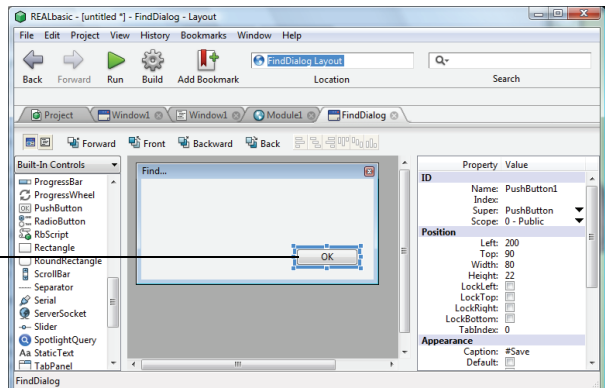
Figure 99. Setting the Caption property of a PushButton using a constant.

The constant “Save” is defined as the string “OK” in a module...



...and is used as the Caption property of a PushButton

The control in the Window Editor shows the value of the constant as defined in the module

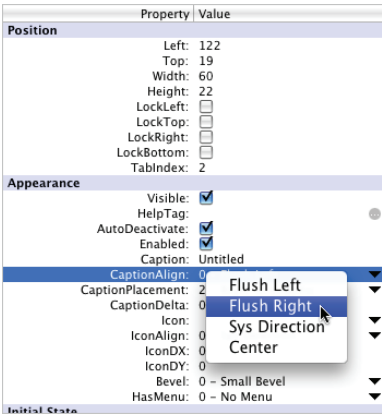


In the built application, the PushButton uses the value of the “Save” constant. For information about creating constants and using them to localize your application, see the section “Using Constants to Localize your Application” on page 378.

Choice Lists

Some properties that require you to choose a value from a fixed list are displayed as pop-up menus. Such properties have a downward-pointing arrow to the right. Simply choose the desired value from the pop-up menu. In Figure 100, the “Right of Graphic” value is being assigned to the CaptionPlacement property of a BevelButton control.


Figure 100. Choosing a value from a Property pop-up menu.



For controls that accept a picture as a property value, you can select the picture by choosing from the pop-up menu associated with the property. All pictures that have been added to the project are listed automatically. In addition, the last menu item is “Browse.” If the desired picture has not been added to the project, you can choose Browse to locate the picture via an open-file dialog box. When you assign a picture to the property in this way, the picture that you select is automatically added to the Project Editor.

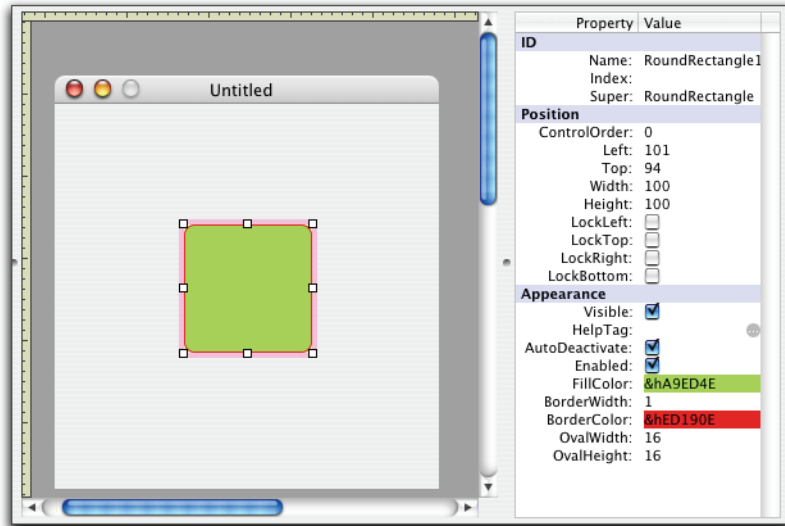
Color Properties

Color properties display the selected color. These colors can be changed by clicking on the color in the Properties pane and using the Color Picker to choose a color. If you wish, you can enter the color value by supplying RGB (Red-Green-Blue) values using the & operator.

For example, the Rectangle and Rounded Rectangle controls have properties that control the border and fill colors. You can set these colors by clicking on the three dots  in the Properties pane for the color property to display the Color Picker. Choose the color using the Color Picker. In Figure 101 the RGB value of the color is also shown in the FillColor and BorderColor properties.

For information on setting the color by specifying the RGB color model, see the section “Color” on page 215.

Figure 101. The Properties pane for a Rounded Rectangle.



Removing Controls

You can remove a control from a window using the control's contextual menu or a menu command.

To remove a control from a window, do this:

- 1 **Bring the window that contains the control to the front. If it's not open, double-click on it in the Project Editor to open it.**
- 2 **Click on the control to select it.**
- 3 **Choose Edit ► Cut (Ctrl+X or ⌘-X on Macintosh), or Edit ► Delete or press the Delete key, or right-click on the control (Control-click on Macintosh) and choose Cut or Delete from the contextual menu.**

If you want a copy of the deleted control on the Clipboard, use Cut instead of Delete.

Understanding Control Layers

Each control in a window has its own layer. This layer is like a sheet of transparent plastic on which each control is placed. It determines whether one control is in front of the other. The Window Editor toolbar provides commands for moving a control forward one layer, to the front, backwards one layer, and to the very back of the layers. These commands are also available in the Edit ► Arrange submenu.

Control layers determine the order in which your application selects the controls as the user presses the Tab key. For more information, see the section "Changing The Tab Order" on page 179 for more details. Control layers may also be important when controls overlap.

Understanding The Focus

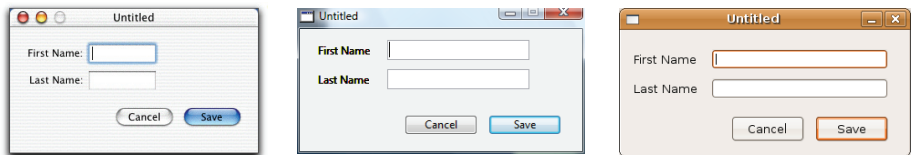
The *focus* is a visual cue that tells the user which control receives keystrokes. TextFields and TextAreas, ComboBoxes, Canvas controls, PushButtons, and ListBoxes can receive the focus on Macintosh. In addition, Sliders, PopupMenus, and CheckBoxes can also receive the focus on Windows. On Linux, TextFields and TextAreas, ComboBoxes, CheckBoxes, PushButtons, PopupMenus, and Sliders can receive the focus.

TextFields

TextFields on any platform display the focus by showing a blinking insertion point and accepting text entry. The behavior of the TextField when the Tab key is pressed is controlled by the AcceptTabs property. If this property is False, pressing the Tab causes the TextField to lose the focus and the next control in the entry order gains the focus. If AcceptTabs is True, the TextField accepts the Tab character for data entry, just as any text character. The TextField keeps the focus. By default, AcceptTabs is False.

In Figure 102, the First Name TextField has the focus. On Windows and Linux, the TextField does not get a focus ring; the focus is indicated only by the blinking insertion point.

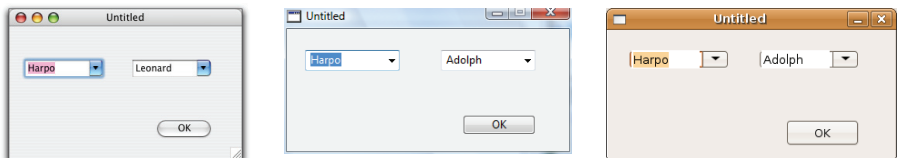
Figure 102. TextFields with and without the Focus.



ComboBoxes

ComboBoxes on any platform display the focus by highlighting the selected item and showing a blinking insertion point. On Macintosh, a ComboBox with the focus also has a focus ring. In other words, a ComboBox with the focus acts like a TextField that has a pop-up menu attached to it.

Figure 103. ComboBoxes with and without the focus.



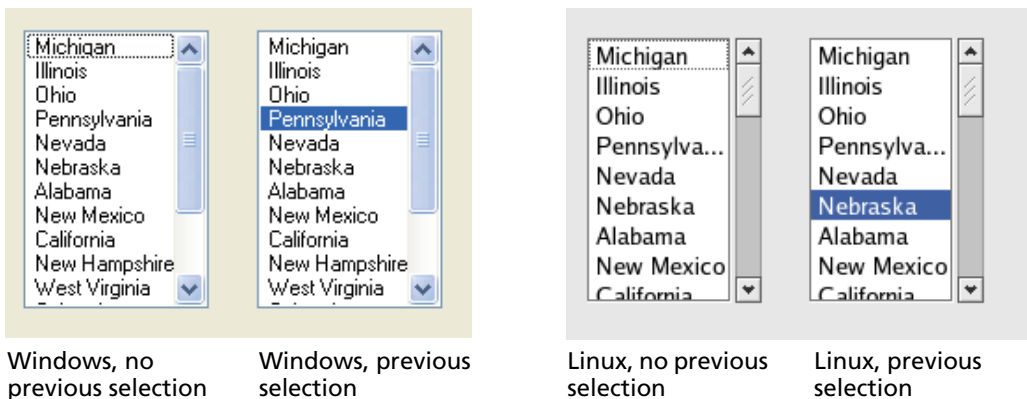
When a ComboBox has the focus, you can scroll through the list of choices with the up and down arrow keys and select an item by pressing the Return key. Of course, you can also edit or replace the selected item.

ListBoxes

When a ListBox has focus on Macintosh, REAL Studio draws a focus ring around the ListBox.

Figure 104. A ListBox on Macintosh.

When a ListBox gets the focus on Windows or Linux, REAL Studio either draws a marquee around the previously selected item (i.e., the highlighted item) or the first item in the ListBox if there is no previously selected item. The marquee may be difficult to see because the item is also highlighted. If the previously selected item is not currently displayed (i.e., the user has scrolled it out of view), there is no visible change in the appearance of the ListBox.

Figure 105. A ListBox with the focus (Windows and Linux).

When a ListBox has the focus, it responds to the Up and Down arrow keys. Pressing either arrow key changes the selected (highlighted) text. It also receives any other keys the user types. This allows you to provide *type selection* functionality where typing selects the item that matches the characters being typed. An example of type selection is provided with REAL Studio.



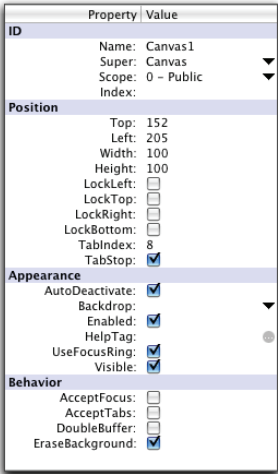
NOTE: If a ListBox is the only control in the window that can get the focus, it will initially get the focus and keep it.

Canvas Controls

Since the Canvas control can get the focus, you can use it to create custom controls of any type that can get the focus. For example, you can use the Canvas to simulate controls that don't get the focus on Macintosh, such as buttons and pop-up menus. Unlike other controls that can get the focus, the ability of a Canvas to accept the

focus is turned off by default. In order for a Canvas control to receive the focus, you must set the AcceptFocus property to True. This can be done either in code or in the Properties pane. The Canvas control also has an AcceptTabs property that indicates whether pressing Tab selects the next control in the window or sends the Tab keystroke to the Canvas control for processing. If AcceptTabs is off, pressing Tab causes the Canvas control to lose the focus. The next control in the entry order gets the focus. If AcceptTabs is on, the Canvas control detects the Tab key as if it were any other key and allows your code to detect and respond to the Tab key.

Figure 106. The Canvas Control’s “Behavior” properties relating to focus.



If the AcceptFocus and UseFocusRing properties are set to True, the Canvas control indicates focus on Macintosh by drawing a border around the control. This is shown in Figure 107.

Figure 107. Canvas controls with and without the focus (Mac OS X).



On Windows and Linux, unfortunately, the UseFocusRing property has no effect. There is no visual indicator of focus that works automatically. However, it is easy to simulate a focus ring using the language. See the Canvas control entry in the *Language Reference* for an example of how to do this.

PopupMenu

On Windows, when a PopupMenu receives the focus the currently selected item is highlighted. Like the ListBox, it also responds to the up and down arrow keys and provides the same type selection functionality. Figure 108 on page 135 shows a PopupMenu loaded with the list of states shown in the ListBoxes in Figure 104 on page 133 with and without the focus. If the user types an “O” while the

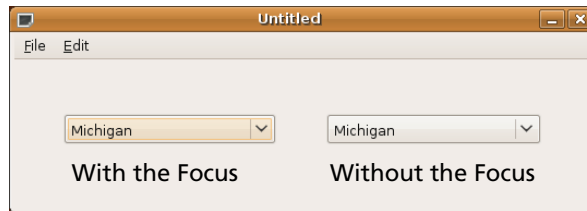
PopupMenu has the focus, the state of Michigan is selected; typing an “O” selects Ohio, and so forth.

Figure 108. A PopupMenu with and without the focus on Windows Vista.



When a PopupMenu gets the focus on Linux, the currently selected item is highlighted. You can change the selected menu item with the up and down arrow keys.

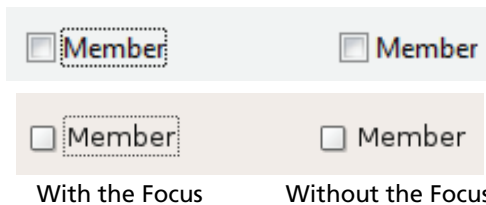
Figure 109. A PopupMenu with and without the focus on Linux.



CheckBoxes

When a CheckBox gets the focus, a marquee surrounds the CheckBox label. Pressing the Spacebar while the CheckBox has the focus toggles the control between its unchecked and checked states.

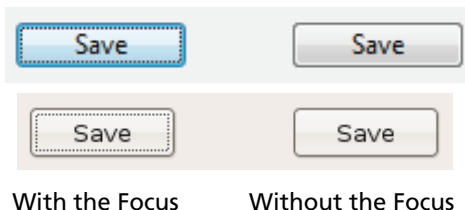
Figure 110. A CheckBox with and without the focus on Vista and Linux.



PushButtons

When a PushButton gets the focus, a marquee surrounds the PushButton’s caption. Pressing the Spacebar while the PushButton has the focus pushes the button, i.e., executes its Action event handler.

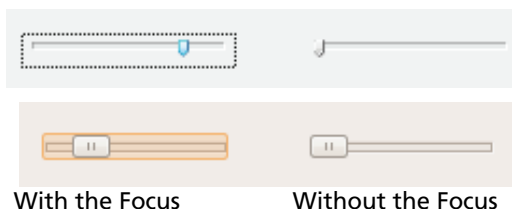
Figure 111. A PushButton with and without the focus on Vista and Linux.



Sliders and Scrollbars

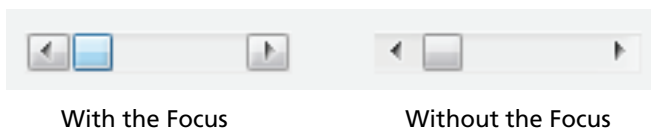
When a Slider gets the focus, a marquee surrounds the control. Pressing the Up or Left arrow key decreases the value of the Slider and pressing the Down or Right arrow key increases the value of the Slider. The amount that the value is changed by each keypress is controlled by the Slider's `LineStep` property. By default, the Slider has a range of 0 to 100 and `LineStep` is 1. The user can also click anywhere along the slider's track to change the slider's value. The amount that the slider moves with each click is controlled by the Slider's `PageStep` property. The default value of `PageStep` is 20.

Figure 112. A Slider control with and without the focus (Vista and Linux).



When a ScrollBar gets the focus on Vista, the thumb's color changes. This is shown in Figure 113.

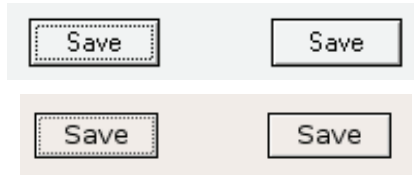
Figure 113. A ScrollBar with and without the focus on Vista.



The ability of a ScrollBar to get the focus can be turned off by deselecting its `AcceptFocus` property.

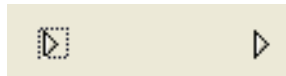
BevelButtons

When a BevelButton gets the focus, a selection rectangle surrounds its label. When it has the focus, the user can press the button by pressing either the Spacebar or the Enter key. BevelButtons can get the focus on Windows and Linux. You must set the `AcceptFocus` property to `True` to enable a BevelButton to get the focus.

Figure 114. A BevelButton with (left) and without the focus.

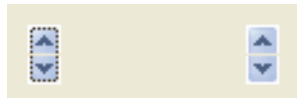
Disclosure Triangle

When a DisclosureTriangle gets the focus, a selection rectangle appears around the control. The user can toggle its state by pressing either the Spacebar or the Enter key. You must set the AcceptFocus property to True to enable a DisclosureTriangle to get the focus. DisclosureTriangles can get the focus only on Windows.

Figure 115. A DisclosureTriangle with (left) and without the focus.

UpDownArrows

When an UpDownArrows control gets the focus, a selection rectangle appears around the control. The user can press the Up and Down arrow keys on the keyboard to press the top and bottom arrows in the control. You must set the AcceptFocus property to True to enable an UpDownArrows control to get the focus. UpDownArrows can get the focus only on Windows.

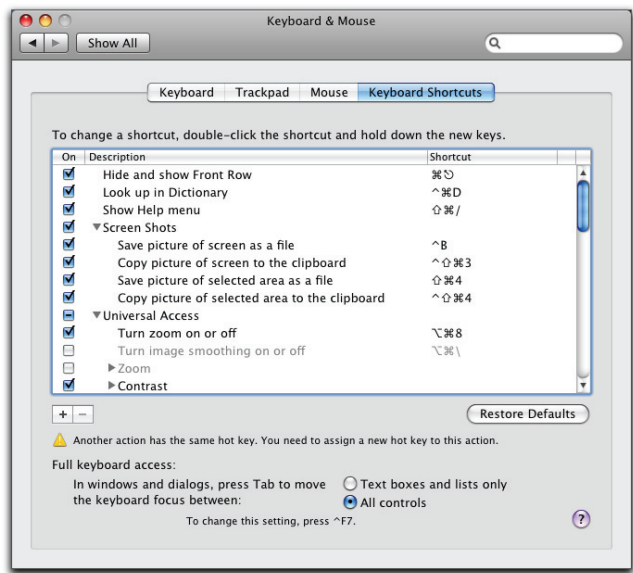
Figure 116. An UpDownArrows control with (left) and without the focus.

Full Keyboard Access

Mac OS X includes a system-wide feature called *full keyboard access*. This feature enables a user to do work with only the keyboard that ordinarily is done using both the keyboard and the mouse. For example, the standard Mac OS X interface calls for mouse gestures to operate the menu system and the dock. With full keyboard access, when the menu bar has the focus, menu items can be highlighted by the up and down arrow keys and an item is selected by pressing Spacebar.

Full Keyboard Access is enabled in the Keyboard Shortcuts panel of the Keyboard and Mouse System Preference. Click the All Controls radio button, shown in Figure 117, to enable Full Keyboard Access.

Figure 117. Enabling Full Keyboard Access (Mac OS 10.4 to 10.5).



In Mac OS X10.6, the dialog has been reorganized so that you choose a topic from the left list and then make changes to the items on the right.

Figure 118. The Full Keyboard access dialog in Mac OS X Snow Leopard.

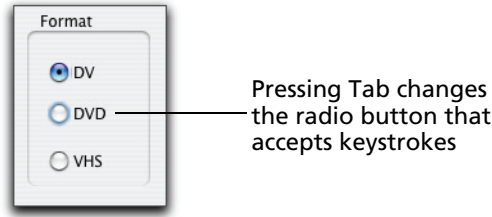


When full keyboard access is on, you can select and set values for controls via the keyboard that normally do not have the focus. When a control accepts keystrokes via

full keyboard access, it has a halo. For example, when full keyboard access is on, the user can select radio buttons via the keyboard only.

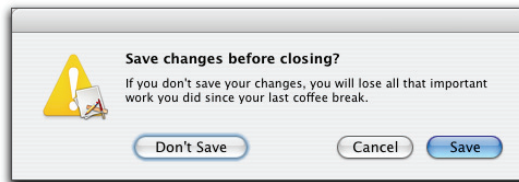
In the example shown in Figure 119, full keyboard access is on and radio buttons can get the focus. The user can press Tab to change the focus. In Figure 119 the “DVD” radio button accepts keystrokes. Pressing the Spacebar key sets the value of the radio button with the focus.

Figure 119. Selecting a radio button with full keyboard access.



When full keyboard access is on, PushButtons can be selected by pressing the Tab key and “clicked” by pressing the Spacebar. In Figure 120, the Don’t Save button has the focus, denoting that it accepts keystrokes but the Save button is the default button. Pressing Spacebar with the dialog box in this state is equivalent to clicking the Don’t Save button.

Figure 120. Selecting the Don’t Save button via the keyboard.



To “click” a MessageDialog button using full keyboard access, press and hold down the Spacebar. Pressing the Spacebar once will highlight the button as if pressed but will not dismiss the dialog.

Please note that full keyboard access is a Mac OS X system-wide option that the end-user must select on his machine. The Mac OS X version of REAL Studio supports full keyboard access if the user chooses to turn it on. However, you cannot turn it on for them, so you can’t assume that all (or any) of your users will be using full keyboard access.

Duplicating Controls

You can duplicate the selected control or controls by choosing Edit ► Duplicate (Ctrl+D or ⌘-D on Macintosh) or by holding down the Control key (Option key on Macintosh) and dragging the selected control.

The Object Hierarchy

Since controls are objects and REAL Studio is an object-oriented language, all controls are derived from other classes. Among other things, this means that each

control inherits properties from the class that it is derived from. Built-in controls that are visible are subclassed from the `RectControl` class. This means that each such control inherits a group of properties from the `RectControl` class automatically. (A few of the built-in controls are not visible in built applications, such as the `Timer` and `TCP Socket`.)

In the *Language Reference*, the parent class for a control is listed as the Super Class and has a hyperlink to the Super Class. To view all the properties and methods for a control, you need to check out the properties and methods of the Super Class as well as the properties and methods that are unique to the control. If a control's Super Class has another Super Class, then you need to keep going up the hierarchy.

For example, the `RectControl` class is the Super Class for visible controls. It has the properties `Width`, `Height`, `Top`, and `Left` which are used to establish the position and size of the `RectControl` in the window. Since all controls that are derived from `RectControl` have these properties automatically, they are not repeated for each subclass of `RectControl` in the *Language Reference*.

To see the object hierarchy in the REAL Studio IDE, use a Window Editor's contextual menu and choose the `Add Menu Item`. The object hierarchy is shown as a hierarchical menu system.

Button Controls for Performing Actions

There are four controls that are commonly used to perform actions when clicked: the `CheckBox`, the `PushButton`, the `BevelButton`, and the `RadioButton`. They are derived from the `RectControl` class.

PushButton

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

Figure 121. A `PushButton` pressed and unpressed.

Windows



Linux



Mac OS X



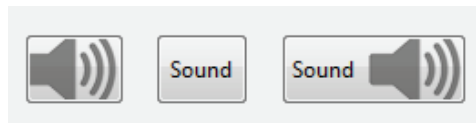
BevelButton The BevelButton control provides very similar functionality as the PushButton and adds several additional powerful features. You can, for example:

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

The usage of a BevelButton control as a pop-up menu is described in the section "BevelButton" on page 154. Note that you can combine an image with a pop-up menu.

Here are several examples of BevelButton options:

Figure 122. Icon, Text, and 'combo' BevelButtons.



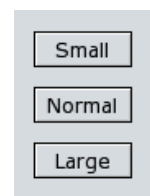
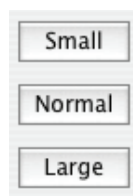
In Figure 122, the speaker image was saved as a transparent PNG file and the BackColor property of the BevelButton was used to provide the neutral gray background. In that way, the BackColor is the background for both the text and the picture (rightmost image). BackColor is supported on Windows and Linux only.

Figure 123. Bevel Sizes.

Windows

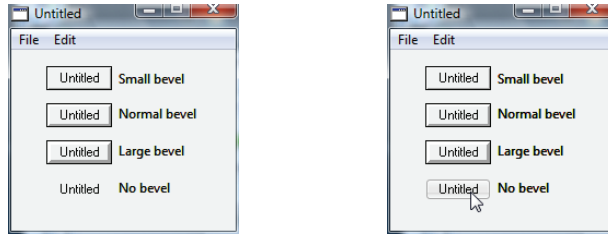


Mac OS X and Linux
(bevel size has no effect)



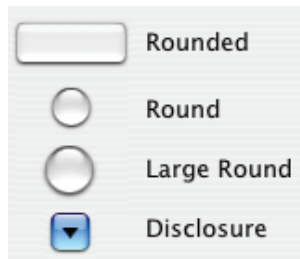
On Windows only, the 'No Bevel' option supports 'mouse-over' effects. When the mouse is not in the region of the button, only the button text is visible. When the mouse enters the region of the button, the button itself is shown. On all non-Windows operating systems, the 'No Bevel' option appears the same as 'Small' in Figure 123. On Windows, a BevelButton with the 'No Bevel' size selected behaves as shown in Figure 124.

Figure 124. A BevelButton on Windows Vista with the ‘No Bevel’ style.



On Mac OS X only, four additional bevel styles are available: Rounded, Round, Large Round, and Disclosure. When clicked, the Disclosure bevel style toggles between two states: upward and downward pointing arrows. The others highlight when clicked. If deployed on other platforms, these bevel styles all look like a standard Small bevel BevelButton. The Mac OS X-only bevel styles are illustrated in Figure 125.

Figure 125. The Mac OS X-only BevelButton bevel styles.

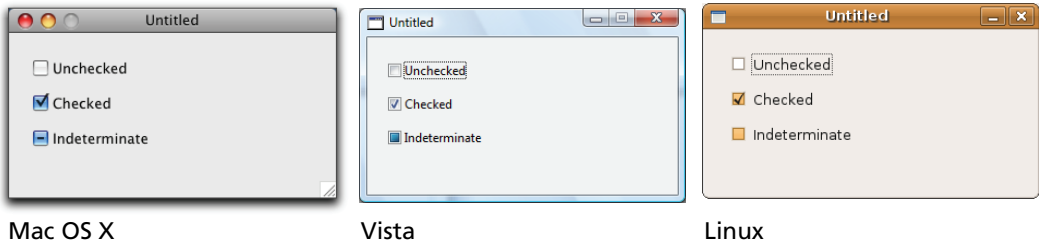


CheckBox

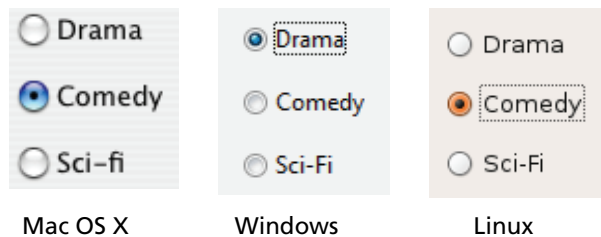
Use CheckBoxes to let the user choose a preference that has three possible states, where one of the states can be selected by default. A CheckBox can be in one of three states: Unchecked, Checked, and Indeterminate. CheckBoxes should not cause an immediate and obvious action to occur except perhaps to enable or disable other controls.

A Checkbox has a State property that stores one of the three values and a Value property that indicates whether the CheckBox is checked or unchecked. An indeterminate state is treated the same as a Value of Checked, but the State property retains the Indeterminate setting.

The three values are displayed as shown in Figure 121 for Macintosh, Windows Vista, and Ubuntu Linux.

Figure 126. Checked, Unchecked, and Indeterminate Checkboxes.**RadioButton**

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

Figure 127. A group of RadioButtons with one selected.

If you are creating a window that will have two or more independent sets of RadioButtons, you will need to use a GroupBox control to make your RadioButton groups respond independently. See “GroupBox” on page 155.

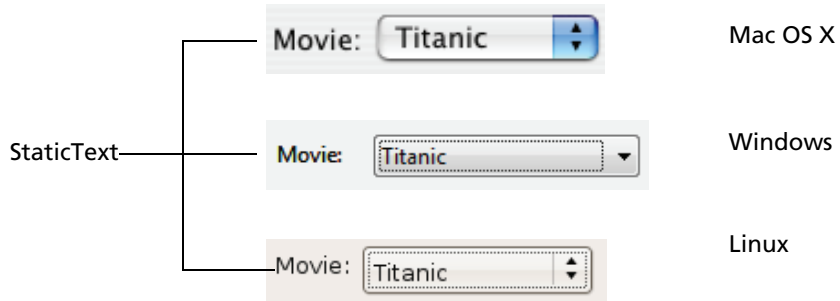
Controls for Displaying and Entering Text

REAL Studio provides controls that let you display text the user can't select, display text the user can select but not edit, and display text the user can both select and edit. These controls are also derived from RectControl.

StaticText

Used to display text that the user cannot select or edit. StaticText controls are commonly used to label other controls (like PopupMenus) or provide titles for groups of controls. The text of the label can be controlled via code (it is the Text property of the control), so you can use it to display dynamic text that is “read only.” For example, read-only fields from a REAL database can easily be displayed with a StaticText control. An easy way to do this is to use the StaticText control in conjunction with a DataControl. The DataControl enables you to “bind” the StaticText to a field in the database and display the contents of the field as the user navigates among records.

Figure 128. A StaticText control used to label a PopupMenu control



TextField

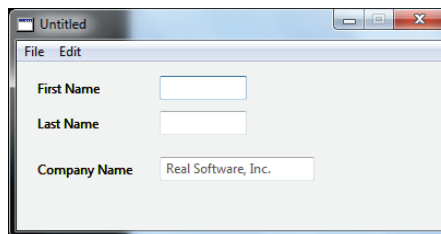
TextFields display text that can be modified by the user. It works as a “single line” field, such as a database field. In contrast, the The TextArea control is a multiline field, such as a word processor field.

The TextField supports the Edit menu’s Cut, Copy, and Paste menu items and keyboard shortcuts automatically. This functionality is built into the default Desktop Application project. If you rename or otherwise modify the Cut, Copy, and Paste menu items, you can break this functionality.

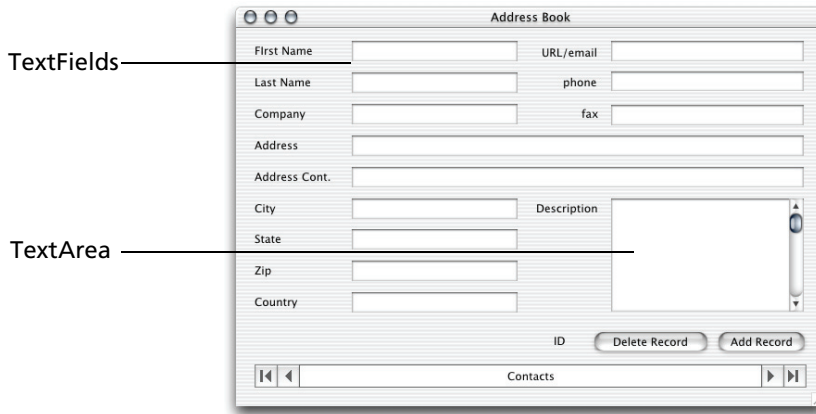
A single-line TextField can be configured as a “password” field — displaying a bullet for each character that is entered. You can also specify a mask for the TextField which filters data entry on a character-by-character basis. For example, if you are using a TextField for a telephone number entry area, you can specify that only numbers can be entered and you can restrict the entry to the correct number of characters.

The CueText property of the TextField enables you to specify a prompt text string that suggests a data entry value. A user can enter the cue text themselves or enter another value. It does not serve as a default value. In the following screen, a company name TextField has the CueText property specified.

Figure 129. The CueText property for a Company Name field.

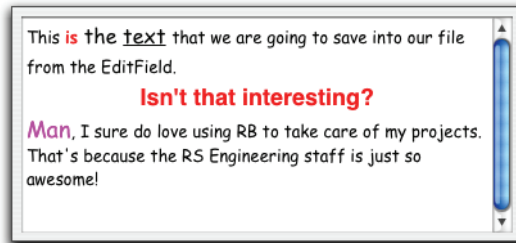


The value for the field must be supplied by the user.

Figure 130. TextFields used for read/write access to database fields.

TextArea

The TextArea control is a multiline text field, such as a word processor field. In contrast to a TextField, it can contain multiple lines and style text. A TextArea field with the Styled property set can display text in multiple fonts, styles, and sizes and have both horizontal and vertical scrollbars. Individual paragraphs can be left, centered, or right aligned via the StyledText class. See the entry in the *Language Reference* for the StyledText class for more information on how to create the example shown in Figure 130.

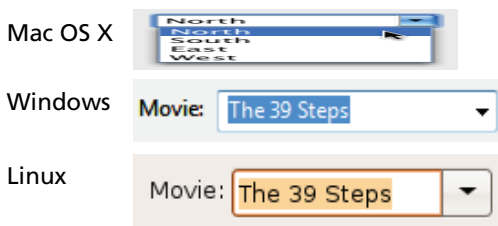
Figure 131. A TextArea with multiple fonts, styles, sizes, colors and paragraph alignments.

The TextArea supports the Edit menu's Cut, Copy, and Paste menu items and keyboard shortcuts automatically. This functionality is built into the default Desktop Application project. If you rename or otherwise modify the Cut, Copy, and Paste menu items, you can break the automatic functionality.

ComboBox

The ComboBox control works like a combination of a TextField and a pop-up menu. The user can either enter text in the ComboBox or choose an item from the attached pop-up menu. Also a menu selection can be selected and modified. Unlike a real TextField, you cannot use a mask to filter data entry or operate it as a Password field.

Figure 132. A ComboBox on Macintosh, Windows, and Linux.



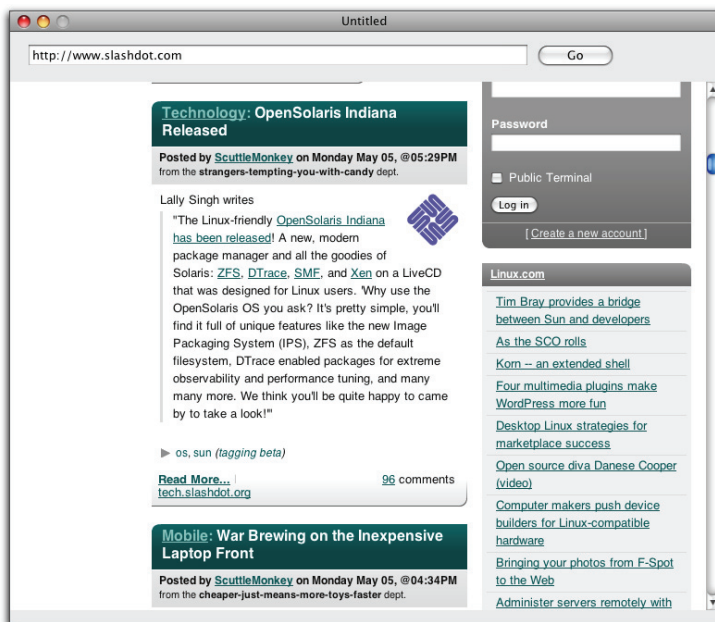
For some examples and information about using a ComboBox to present a list of choices, see the section “ComboBox” on page 154.

HTMLViewer

The HTMLViewer control renders HTML (like any web browser application) and provides basic navigation functions. Using the language, you can pass it an HTML file, the HTML text itself, or tell it to load the HTML specified by a URL. If the HTML is valid, it renders it. The REAL Studio Online Language Reference uses the HTMLViewer control.

The following example is a very simple web browser that uses an HTMLViewer control. The user types in a URL into the TextField at the top of the window and clicks the Go button. If it is a valid URL, the web page appears in the HTMLViewer control.

Figure 133. A very simple web browser that uses an HTMLViewer control.



Controls for Displaying and Entering Numeric Values

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

ScrollBar

ScrollBars can be presented vertically or horizontally. To make a vertical ScrollBar, simply resize the ScrollBar object so that the height is greater than the width.

The default thickness of a ScrollBar is 16 pixels, but you can narrow the short side of the ScrollBar to turn it into a mini-scrollbar. You can either drag the object in the Window Layout Editor or change the control's property. As you narrow the control, it will “snap” to its mini thickness.

Figure 134. Standard and mini ScrollBars on Macintosh.

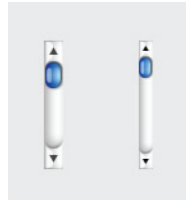


Figure 135. Horizontal and vertical ScrollBars.

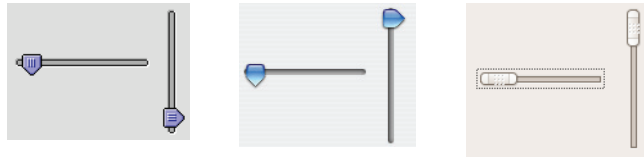


ScrollBars can get the focus on Windows only and have Windows-only properties that you can use to determine when a ScrollBar gets and loses the focus.

Slider

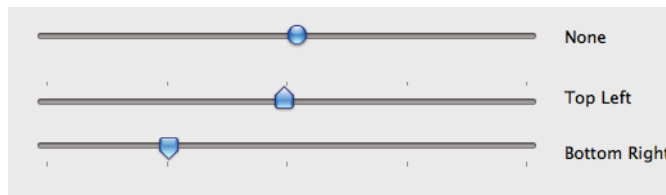
The Slider has the same functionality as a ScrollBar control. However, ScrollBar controls have come to be associated with scrolling text or a picture and less with assigning numeric values. The Slider control provides an interface that is clearly for increasing or decreasing a numeric value. Like the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

Figure 136. Horizontal and vertical Slider controls.



The Slider can have tick marks on the left or right (vertical orientation) or the top or bottom (horizontal orientation). When tick marks are used, the indicator points in the direction of the marks. Fix shows top and bottom tick marks for the horizontal orientation.

Figure 137. Horizontal tick marks for a Slider control.

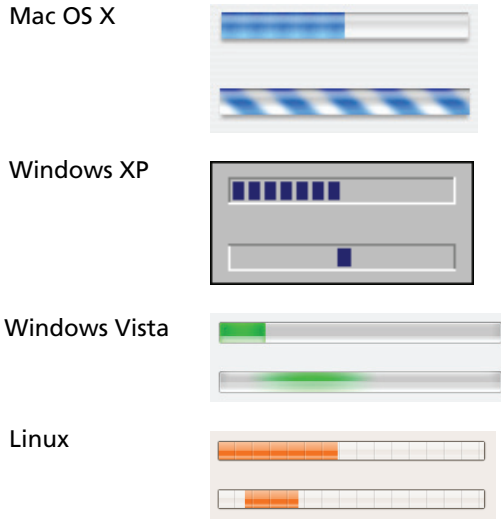


Sliders can get the focus on Linux and Windows. For those platforms, you can use events to determine when a Slider gets and loses the focus.

ProgressBar

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

On Windows and Linux, an indeterminate progress bar takes the form of a single block that moves back and forth.

Figure 138. Determinate and indeterminate ProgressBars.

Controls for Presenting a List of Choices

RadioButton and CheckBox controls can, of course, be used to provide the user with a limited list of choices. There are situations, however, when using these controls is either an inefficient use of space or impossible. Some of these situations are:

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

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

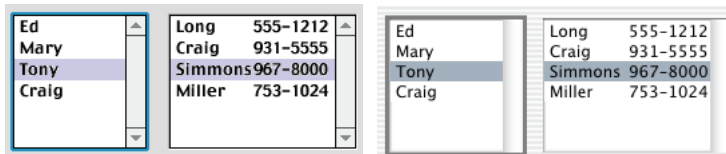
ListBox

ListBox controls display a scrolling list of values. The user can use the mouse or the arrow keys to choose an item. ListBox controls can contain one or more columns of data, can be hierarchical, and can allow one row selection or multiple row selection.

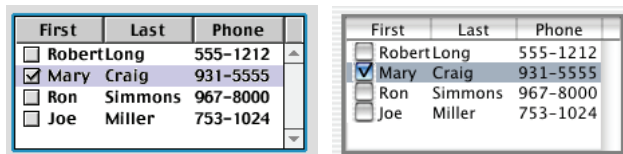
However, you can change the number of columns of any ListBox simply by setting the ColumnCount property in the Properties pane. You can use either icon and modify the number of columns after adding the ListBox to a window.

Figure 139. Examples of ListBoxes.

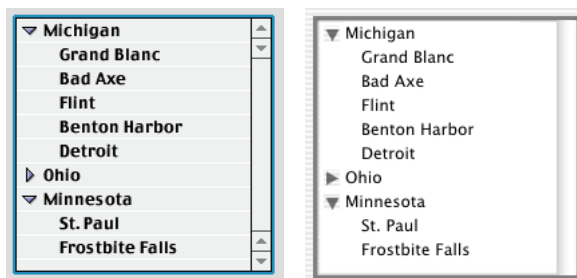
Single column
and multiple
column



Multiple column
with headers and
CheckBoxes

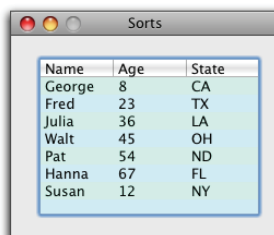


Two column
hierarchical

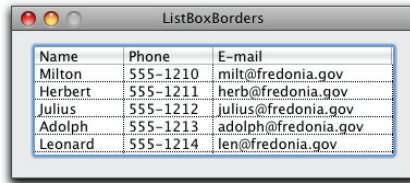


Checkbox-style cells in ListBoxes can store one of three states: Checked, Unchecked (as shown in Figure 139), and Indeterminate. They work the same way as the CheckBox control. For more information, see the section, “CheckBox” on page 142.

You can also shade cells, rows, and columns programmatically. For example, in Figure 140, alternating rows are shaded differently and the selected row has a custom shading to indicate that it is selected.

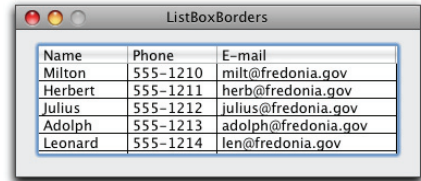
Figure 140. An example of custom shading.

You can also customize row and column borders. Figure 141 on page 151 illustrates the four types of horizontal and vertical rules that can be drawn programmatically. The Default style is “None.”

Figure 141. Four styles of Custom borders.


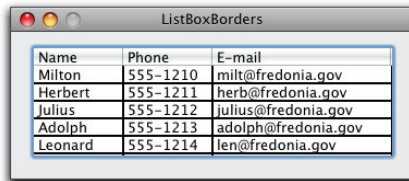
A screenshot of a window titled 'ListBoxBorders' showing a table with a thin dotted border.

Name	Phone	E-mail
Milton	555-1210	milt@fredonia.gov
Herbert	555-1211	herb@fredonia.gov
Julius	555-1212	julius@fredonia.gov
Adolph	555-1213	adolph@fredonia.gov
Leonard	555-1214	len@fredonia.gov

Thin Dotted


A screenshot of a window titled 'ListBoxBorders' showing a table with a thin solid border.

Name	Phone	E-mail
Milton	555-1210	milt@fredonia.gov
Herbert	555-1211	herb@fredonia.gov
Julius	555-1212	julius@fredonia.gov
Adolph	555-1213	adolph@fredonia.gov
Leonard	555-1214	len@fredonia.gov

Thin Solid


A screenshot of a window titled 'ListBoxBorders' showing a table with a thick solid border.

Name	Phone	E-mail
Milton	555-1210	milt@fredonia.gov
Herbert	555-1211	herb@fredonia.gov
Julius	555-1212	julius@fredonia.gov
Adolph	555-1213	adolph@fredonia.gov
Leonard	555-1214	len@fredonia.gov

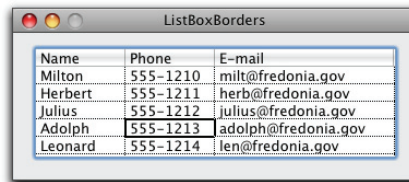
Thick Solid


A screenshot of a window titled 'ListBoxBorders' showing a table with a double thin solid border.

Name	Phone	E-mail
Milton	555-1210	milt@fredonia.gov
Herbert	555-1211	herb@fredonia.gov
Julius	555-1212	julius@fredonia.gov
Adolph	555-1213	adolph@fredonia.gov
Leonard	555-1214	len@fredonia.gov

Double Thin Solid

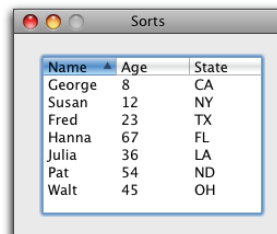
You can also apply these ruling styles to selected cells or even selected cell borders. For example, in Figure 142, a cell containing a phone number is highlighted using Thick Solid borders while the remainder of the ListBox uses Thin Dotted rules:

Figure 142. A custom border around a selected cell.


A screenshot of a window titled 'ListBoxBorders' showing a table where the 'Phone' column for the first row is highlighted with a thick solid border, while the rest of the table has a thin dotted border.

Name	Phone	E-mail
Milton	555-1210	milt@fredonia.gov
Herbert	555-1211	herb@fredonia.gov
Julius	555-1212	julius@fredonia.gov
Adolph	555-1213	adolph@fredonia.gov
Leonard	555-1214	len@fredonia.gov

When you can add a header with column labels to a ListBox, the user can sort the data in the ListBox by clicking on a column header or you can sort the rows of the ListBox programmatically. The sort direction is indicated by a sort direction arrow in the header area.

Figure 143. A ListBox sorted by the Name column.


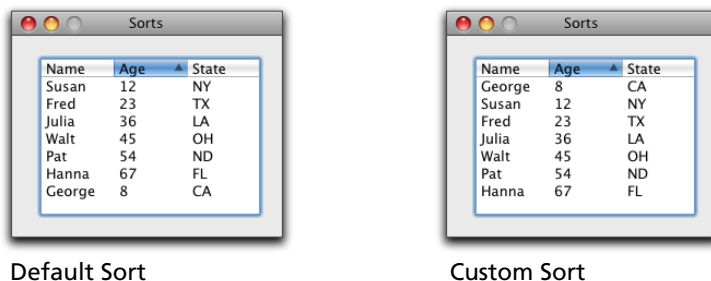
A screenshot of a window titled 'Sorts' showing a table with three columns: Name, Age, and State. The 'Name' column header has a small upward-pointing arrow, indicating it is the current sort key. The data is sorted alphabetically by name.

Name ▲	Age	State
George	8	CA
Susan	12	NY
Fred	23	TX
Hanna	67	FL
Julia	36	LA
Pat	54	ND
Walt	45	OH

The default sorting method works for alphabetic values, but does not produce valid results for numbers and dates. If you need to sort these data types, you can do so

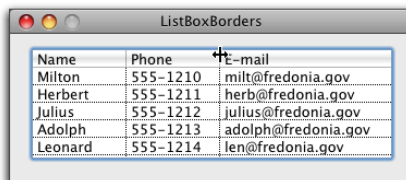
using the `CompareRows` event of the `ListBox`. In the following example, code that compares the values of adjacent rows in the `Age` column is used to get the desired sort order.

Figure 144. Default and custom sorting on a number column.



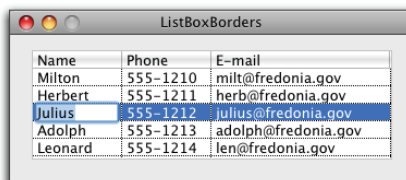
For multicolumn `ListBoxes`, you can control whether the user can resize columns by dragging column borders. You can enable column resizing on a column-by-column basis or for the entire `ListBox`. If user resizing is enabled, you can also specify minimum and maximum column sizes. If resizing is enabled, the pointer turns to a column resizing pointer when it is moved to a column border. For example, in Figure 145 the `Name` column is being resized to make more room for the `Email` column.

Figure 145. A column being resized.



You can also permit data entry in `ListBoxes`. You can programatically switch a cell's mode from normal (i.e., view only) to inline editable. When the cell is editable, it has the focus and has a focus ring around it. The contents of the cell are initially selected and typing replaces the entry. When the user clicks on another cell or tabs out of the editable cell, the cell's contents are saved and its mode reverts to view only.

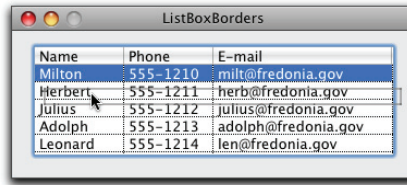
Figure 146. An editable cell in a `ListBox`.



For both single and multicolumn `ListBoxes`, you can allow the user to drag columns to rearrange the rows. When you drag, the target for the drop is indicated by a solid

line between rows. For example, in Figure 147 the row for Milton is being dropped between Herbert and Julius.

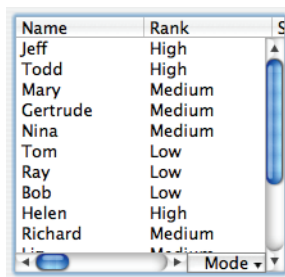
Figure 147. Dragging a row in a ListBox.



A ListBox can be scrolled either horizontally or vertically. The ListBox control has built-in scrollbars. You can choose to add either or both to a ListBox via its Properties pane. If you want to customize the scrollbar area, you can deselect the built in scrollbar and instead place a Scrollbar control and other controls in the scrollbar's area.

For example, the following illustration shows a Scrollbar control that is used as the horizontal scrollbar and a BevelButton control that is used as a pop-up menu. The pop-up menu enables the user to select between single-line and multi-line selection in the Listbox. The vertical scrollbar is the built-in scrollbar that was added via the ListBox's Properties pane.

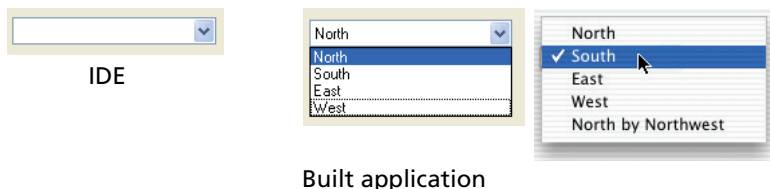
Figure 148. A customized horizontal scrollbar in a ListBox.



See the entry in the *Language Reference* for the ListBox control for the code that is used in this example.

PopupMenu

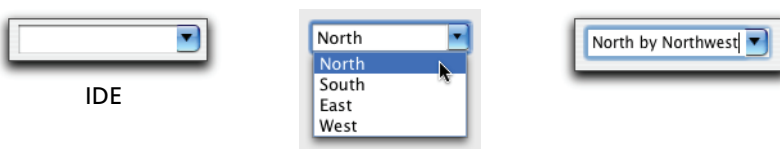
PopupMenu controls are useful when you have a single column of data to present in a limited amount of space. It presents a list of items and the user can choose one item. When the user displays the PopupMenu's items, the selected item is indicated by a checkmark.

Figure 149. A PopupMenu control.

You populate the list by entering them in the InitialValue property in the Properties pane or by calling the AddRow method before the PopupMenu is displayed, for example, in its Open event handler. You can read the value the user has selected (or change the value via code) with the ListIndex property.

ComboBox

A ComboBox combines the features of a PopupMenu and a TextField. The user can either choose an item on the list or enter a different item. Of course, they should be used instead of PopupMenus when you want to allow the user to enter an item that is not on the list.

Figure 150. A ComboBox control.

The user can choose an item on the list (left) or type an item.

You can populate the list the same way as with a PopupMenu control. The ListIndex property indicates which item the user has selected, but it does not change if the user has typed a value into the ComboBox. Read the value of the Text property to get the current text, which can be either an item on the menu or text entered by the user.

The user can use the down arrow key to display the list when text (or nothing) is displayed in the ComboBox. When the ComboBox has the focus, the user can change the selected item on the list using the up and down arrow keys. The Return key selects the highlighted item in the list and the Escape key closes the popup menu without selecting the highlighted item on the list.

BevelButton

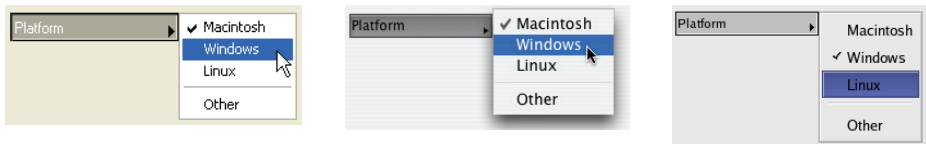
A BevelButton control can be configured to operate as a pop-up menu. Simply set the HasMenu property to 1 or 2 (Normal menu or Menu on Right). See the entry for the BevelButton control in the *Language Reference* for the list of a BevelButton's properties.

The BevelButton menu shown in Figure 151 was created with this code in the Open event of the BevelButton:

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

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

Figure 151. A BevelButton popup menu.



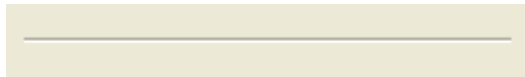
Controls for Visually Grouping Other Controls

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

Separator

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

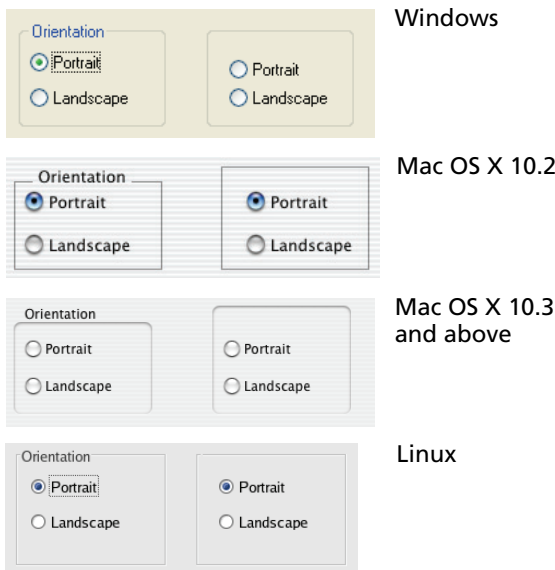
Figure 152. A Separator control



GroupBox

A GroupBox can be displayed with or without a caption. If a window has more than one group of RadioButton controls, one of the groups must be contained within a GroupBox control in order for the RadioButton groups to function independently. In Mac OS X 10.3 Apple gave GroupBoxes a 3D “sunken” look.

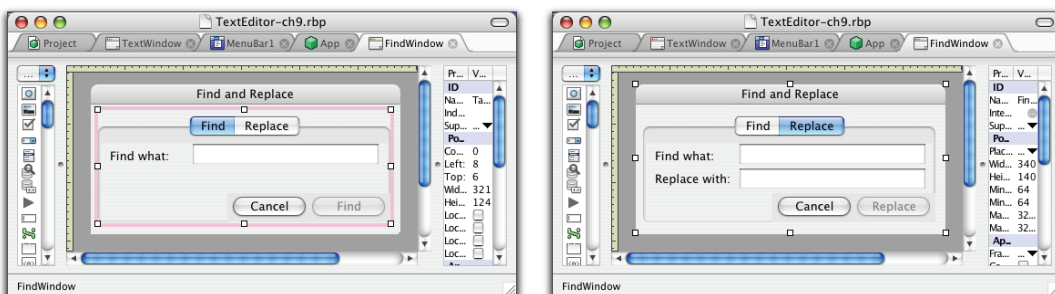
Figure 153. GroupBox controls with and without a caption.



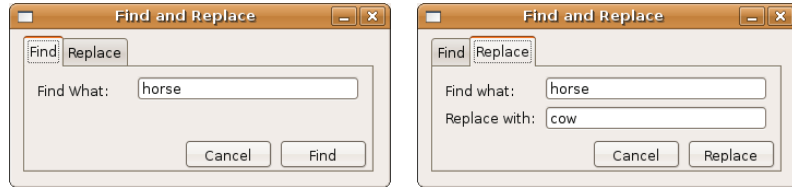
TabPanel

When you have several groups of controls and space is very limited, TabPanels are appropriate. A TabPanel presents each group of controls in a separate panel. When the user clicks on a tab in the TabPanel, REAL Studio automatically hides the controls on the current panel and displays the controls on the panel the user selected. In Figure 154, both “Find” and “Find and Replace” functionality is built into the same dialog box using a TabPanel control.

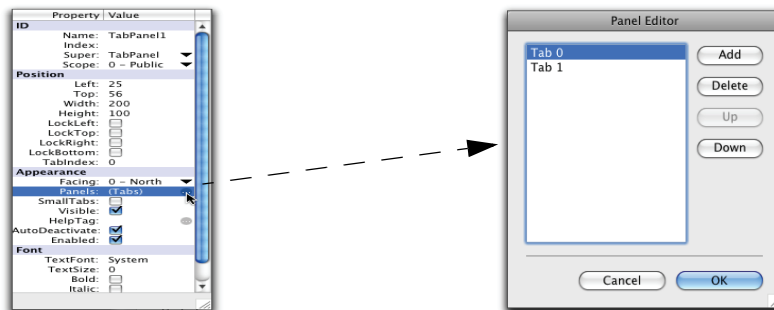
Figure 154. A two-panel TabPanel control.



In the built application, the control looks like this:

Figure 155. The two-panel TabPanel control (Linux).

By default, a TabPanel control has two panels. You add, modify, rearrange, or delete tabs and tab labels using the Tab Panel Editor. Click the value of the Panels property of the TabPanel to display the Tab Panel editor.

Figure 156. Opening the Tab Panel Editor.

With the Tab Panel Editor, you can:

- **Rename a tab:** Click once on its name to select it and then click again to get an insertion point. When you get the insertion point, replace the current label with the new label.
- **Add a new tab:** Click the Add button. A new tab appears in the list with a default name. The default name is selected, so you can rename it by typing.
- **Delete a tab:** Click once on it to select in and then click the Delete button.
- **Rearrange the tabs:** Highlight a tab you want to move and then click the Up or Down buttons.

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

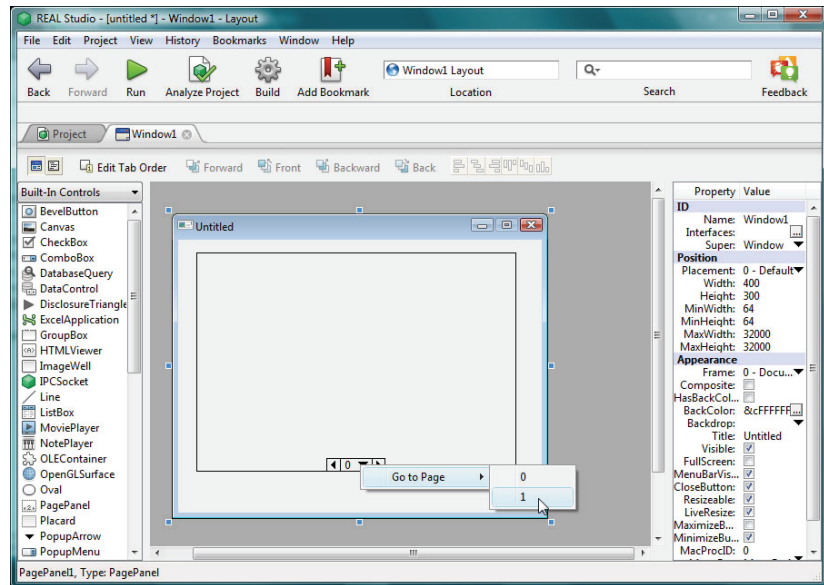
PagePanel

The PagePanel control implements the same idea as the TabPanel, except that the PagePanel control itself is not visible to the end-user. It has no tabs or other navigation widgets, nor does it have a visible border. Only the controls on each page are visible. You are responsible for providing the method of navigating from one page to another. You can do so by setting the value of its Value property programmatically.

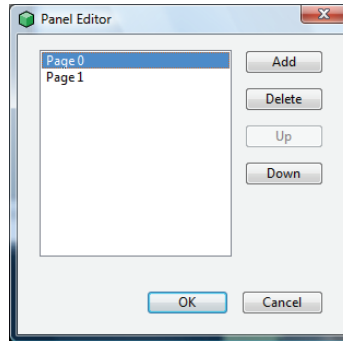
In the IDE, you navigate among pages in a PagePanel control using a widgets at the bottom of the control. This is shown in Figure 157.

There are two ways to navigate among existing pages. The Go to Page command displays a submenu of page numbers. Select a page number from this menu to go directly to a page. Click in the center of the navigation widget to get the drop-down list of page numbers. Or, click the left or right arrows to the left or right of the page number to go to the next or previous page.

Figure 157. The PagePanel's pop-up menu (IDE).

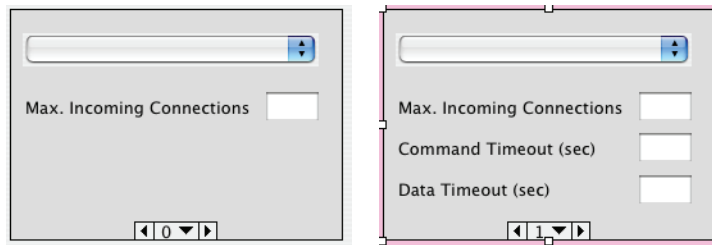


You add, delete, and reorder pages using the PagePanel Editor, shown in Figure 158. You can display the PagePanel editor by clicking the Panels property of the PagePanel in the Properties pane. This is identical to the process illustrated in Figure 156 on page 157 for the TabPanel. The PagePanel editor uses the same interface as the TabPanel editor, except that you cannot name PagePanels. With that exception, the buttons in the PagePanel editor work as described for the TabPanel.

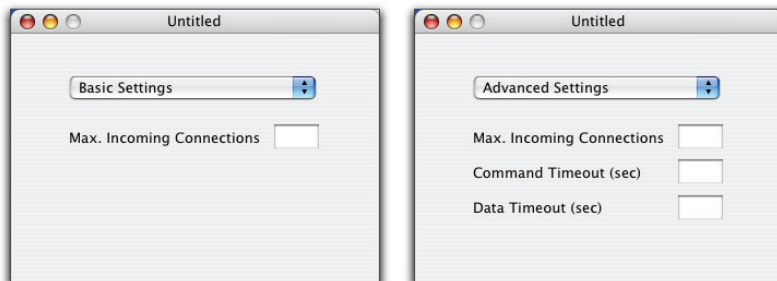
Figure 158. The PagePanel editor.

Here is a simple example of the use of a PagePanel control. The user wants to present two versions of an interface — the ‘Basic’ version, which is shown by default — and an Advanced version, which is shown only if the user chooses to reveal it. A PopupMenu control is used as the widget that controls which page is shown.

Figure 159 shows the two pages in the IDE.

Figure 159. The ‘Basic’ and ‘Advanced’ pages in the IDE.

In the built application, the two pages look like this.

Figure 160. The Basic and Advanced settings pages in the built application.

Code in the Change event of each pop-up menu controls the page that is displayed and the setting of the pop-up menu on the other page. For example, the code for the pop-up menu on the ‘Basic’ page is:

```
pagePanel1.value=me.listindex
popupMenu2.listindex=me.listindex
```

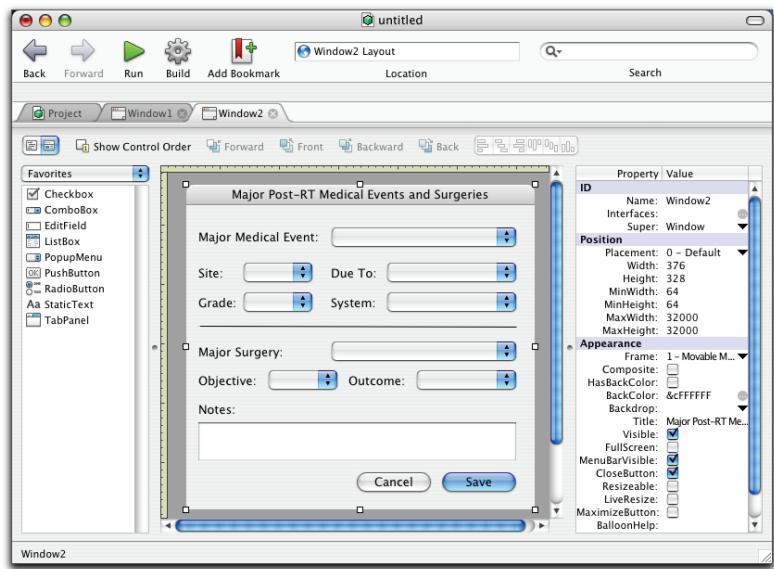
Controls for Displaying Graphics and Pictures

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

Line

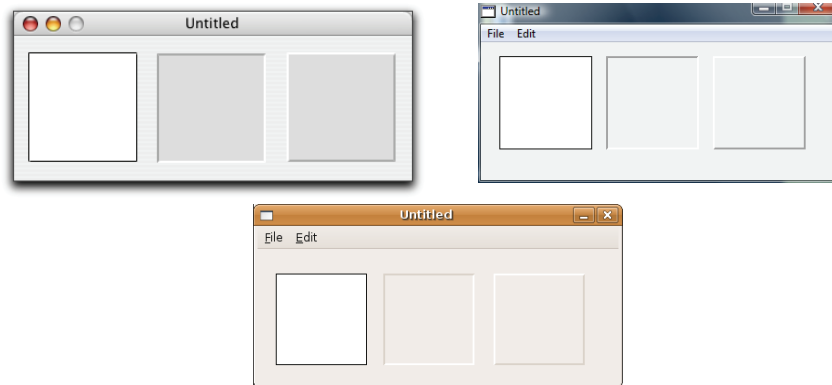
A Line control draws a line that can be of any length, width, color, and direction. By default, lines are 100 pixels in length, 1 pixel in width, black, and horizontal. In Figure 161, a Line control is used to divide two areas of a complex dialog box used for data input.

Figure 161. A Line control used to divide two sections in a dialog box.



Rectangle

A Rectangle control draws a rectangle that can be of any length, width, border color, and fill color. By default, rectangles are 100 pixels in length and width, with a black border that is 1 pixel thick and a white center. Because you can control the color of the left and top borders independently from the right and bottom borders, you can easily create rectangles that appear to be sunken or raised. The example code is in the entry for the Rectangle control in the *Language Reference*.

Figure 162. A Rectangle with default, sunken and raised appearances.

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

Figure 163. A RoundRectangle control.

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

Figure 164. An Oval control.

Canvas A Canvas control can be used to display a picture from a file or a picture drawn using REAL Studio's programming language. If your application requires a type of control that is not built-in, you can use a Canvas control and REAL Studio drawing commands to create the controls you need. The Canvas control has access to the drawing tools belonging to the Graphics class; with these tools you can programmatically draw objects within the Canvas.

The Canvas control can get the focus, so you can emulate any other type of control that you would like to get the focus.

Canvas controls can be used to create extremely sophisticated controls. In Figure 165, a Canvas control is used to provide a table of data with rows that can be selected and columns that can be sorted by clicking on the column title.

Figure 165. A sophisticated control created using a Canvas control.

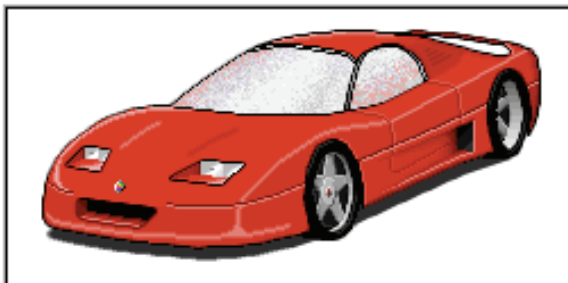
Name	Age	City	
Smith	20	London	
Jones	10	Paris	
Blake	30	Paris	
Clark	20	London	
Adams	30	Athens	
Björn	22	Reykjavik	

The Table control above was created by Björn Eiríksson.

ImageWell

The ImageWell control provides an area in which you can display a BMP or PNG image on Windows and Linux or a PICT image on Macintosh. You can easily program the ImageWell control to accept a dragged picture.

Figure 166. An ImageWell.



OpenGLSurface The OpenGLSurface control enables an OpenGL programmer to create an OpenGL object in a REAL Studio application. It provides an interface for OpenGL drawing. You will need to be familiar with the OpenGL language in order to program this control. Information on OpenGL is found at <http://www.opengl.org>.

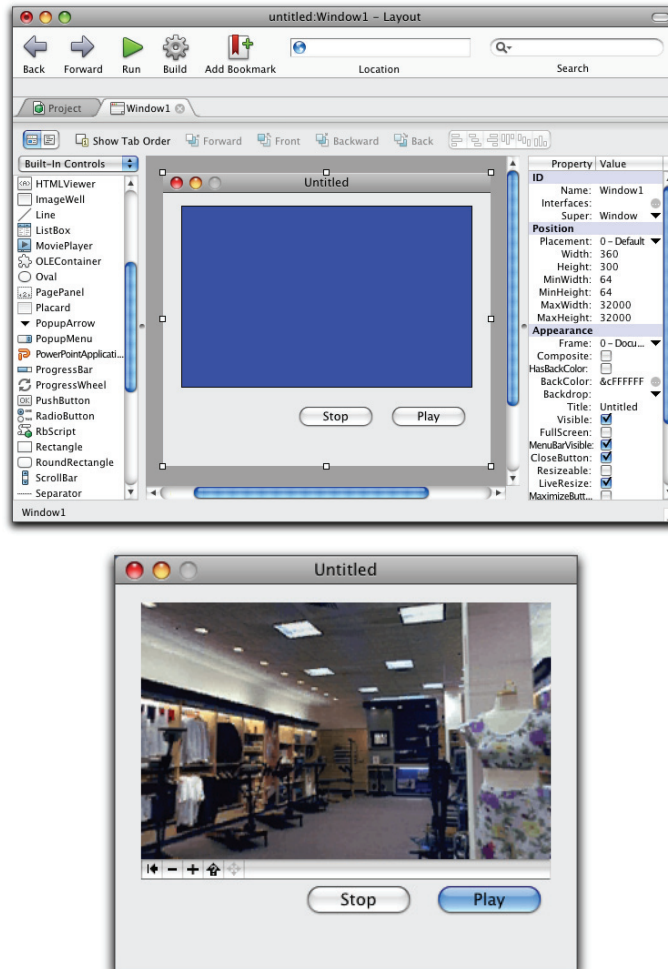
Controls for Playing Movies, Music, and Animation

The MoviePlayer control plays movies using a built-in movieplayer. QuickTime™ is the default player on Macintosh and Windows Media Player is the default on Windows. On Linux, the MoviePlayer uses GStreamer by default (it requires version 0.10+) and uses Xine if GStreamer is not available.

MoviePlayer

The MoviePlayer control displays the standard movie controller for your platform.

Figure 167. A MoviePlayer control in the IDE and an application (QT3D movie shown).



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

Assigning a movie to a `MoviePlayer` control is easy. The `Movie` property of the `MoviePlayer` stores the movie that the `MoviePlayer` will play. It has a pop-up menu that lists all the movies that have been added to the Project Editor. You can add a movie to the Project by either dragging it from the desktop or with the `File ► Import` menu command.

If you did not add the movie to the Project Editor, you can locate it from the Window Editor. Choose `Browse` from the `Movie` property's pop-up menu. An open-file dialog box appears. Select the movie you want to play. When you do so, the movie is also added to the Project Editor.

You can then add `Stop` and `Play` `PushButtons` to the window or let the users use the built in controls in the player.

If you wish to provide `Stop` and `Play` controls, add two `PushButtons` to the window and label them “`Stop`” and “`Play`,” as shown in Figure 167. In the `Action` event for the “`Stop`” button, use the code:

```
MoviePlayer1.Stop
```

The `Action` event for the “`Play`” button is:

```
MoviePlayer1.Play
```

The result is a fully functional `MoviePlayer` application shown below the IDE in Figure 167 on page 163.

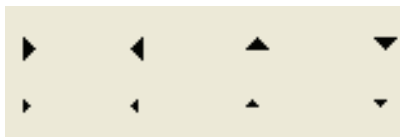
Miscellaneous Controls

PopupArrow Control

The `PopupArrow` controls places an arrow in the window that points in any of four directions. Two sizes of arrows are available.

You control both the direction and size of the popup arrow using one property, the `Facing` property. You get a choice of four orientations (North, South, East, and West) and two sizes, standard and small. Typically, you would use a `PopupArrow` control as part of a custom control. The orientation of the arrow indicates whether the custom control can display additional information or options. Figure 168 shows all four orientations and both sizes.

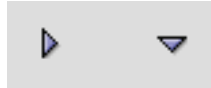
Figure 168. Examples of the `PopupArrow` Control.



Disclosure Triangle Control

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

Figure 169. Disclosure Triangles.



UpDownArrows Control

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

Figure 170. The UpDownArrows Control (Windows, Linux, and Macintosh).



ProgressWheel Control

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

Figure 171. The ProgressWheel Control (Windows, Linux, and Macintosh).



RBScript Control

The RBScript control allows the end user to write and execute REAL Studio code within a compiled application. Scripts are compiled into machine code.

You pass the REAL Studio code that you want to run via the Source property and execute it by issuing the Run method. Please see the *Language Reference* for details on the functions, control structures, and commands supported by the RBScript control.

IDE Scripts

You can also automate the IDE itself via IDE scripts. Any menu command or toolbar button can be automated. You can also get and set property values, navigate among panels in the IDE, manipulate code in the Code Editor, and automate the build process among other things. To script the IDE, use the File ► IDE Scripts ► New IDE Script command to open the IDE Script editor. See the entry for IDE Scripts in the *Language Reference* for the commands used in IDE scripting. IDE scripting is a Studio-only feature of REAL Studio.

Controls for Handling Communications

REAL Studio provides controls that allow your application to communicate through the serial port (for communicating via a modem or through a serial cable to another

device) and over a network to other computers using TCP/IP, the Internet's communication protocol.

Serial

Although the Serial control displays an icon when placed in a window in the Window Editor, it is not visible in the built application. It is designed only for executing code to communicate via the serial port. For more information, see the Serial control in the *Language Reference* for more details.

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

TCPSocket

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

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

For more information, see the section on the TCPSocket control in the *Language Reference*.

SSLSocket

The SSLSocket control is similar to the TCPSocket. The SSLSocket implements Secure Sockets Layer communication via TCP/IP. It supports SSL versions 2 and 3 as well as TLS (Transport Layer Security) version 1.

However, it does not have an icon of its own in the Controls pane. You can also add it to a window via the window's contextual menu or create an instance of an SSLSocket via code.

To establish an SSL connection with the SSLSocket, set the Secure property to True and use the Connect method.

The SSLSocket is available in the Professional and Studio versions of REAL Studio; the Personal version of REAL Studio does not compile references to the SSLSocket.

For more information, see the section on the SSLSocket control in the *Language Reference*.

ServerSocket

The ServerSocket class enables you to support multiple TCP/IP connections on the same port. When a connection is made on that port, the ServerSocket hands the connection off to another socket, and continues listening on the same port. It includes the ability to replenish its supply of TCPSockets as connections are made. Without the ServerSocket, it is difficult to implement this functionality due to the latency between a connection coming in, being handed off, creating a new listening socket, and restarting the listening process. If you had two connections coming in at

about the same time, one of the connections may be dropped because there was no listening socket available on that port.

The `ServerSocket` is available in the Professional and Studio versions of REAL Studio. For more information, see the section on the `ServerSocket` class in the *Language Reference*.

UDPSocket

The `UDPSocket` supports communications via a UDP (User Datagram Protocol) connection. It also can be created via code because it is not derived from the `Control` class.

UDP is the basis for most high speed, highly distributed network traffic. It is a connectionless protocol that has very low overhead, but is not as secure as TCP. Since there is no connection, you do not need to take nearly as many steps to prepare when you wish to use a UDP socket.

UDP sockets can operate in various modes, which are all very similar, but have vastly different uses. Perhaps the most common use is “multicasting.” Multicasting is a lot like a chat room: you enter the chatroom, and are able to hold conversations with everyone else in the chatroom.

For more information, see the section on the `UDPSocket` class in the *Language Reference*.

IPC Socket

The `IPCsocket` performs interprocess communications between two REAL Studio applications running on the same computer. Use it to send and receive messages.

Like other sockets, the `IPCsocket` control displays an icon when placed in a window in the Window Editor but has no interface.

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

For an example of the use of the `IPCsocket`, see the example application and the section on the `IPCsocket` control in the *Language Reference*.

“Easy” Communications

The REAL Studio language includes “easy” versions of the `UDPSocket` and `TCPsocket` classes that are designed for communication among REAL Studio applications on a network. They are called the `EasyUDPSocket` and `EasyTCPsocket` classes. Also, `AutoDiscovery` class can automatically poll the network and return the IP addresses of the computers that are logged in for “easy” communications. Once this is done, it is very easy for members to send messages to one another.

These classes make it easy to set up a network of REAL Studio users, but they are not designed for more generic communication with other applications, such as an FTP or HTTP server.

For more information on the “Easy” communication classes, see the section “Making Networking Easy” on page 679 and the entries for the classes in the *Language Reference*.

Toolbar Control

The Toolbar control enables you to create cross-platform toolbars. Support for toolbars consists of the Toolbar control itself and two supporting classes, ToolItem and ToolButton. Each item in a toolbar is a ToolButton. It can be a pushbutton, a button that toggles, a vertical separator, a drop-down list, or a fixed or flexible space. A pushbutton returns to its unpressed state when the user releases the mouse button, but the toggle button remains depressed until clicked again.

You create one ToolButton for each item that appears in the toolbar. For buttons and drop-down lists, you can pass the picture that it will use as its icon and enter the label that will appear below the button. It is convenient to add all the pictures that will be used by the toolbar to the Project Editor beforehand.

When you are finished creating all the toolbar items, you add the finished toolbar to each window that uses it. You then must write an event handler for the toolbar’s Action event.

In the IDE, you can create a complete toolbar via the Toolbar Editor. You can also create a toolbar via the language, using the Toolbar, ToolItem, and ToolButton classes. For an example of a toolbar that was created via code, see the entry for the Toolbar control in the *Language Reference*.

To create a toolbar in the IDE, follow these steps:

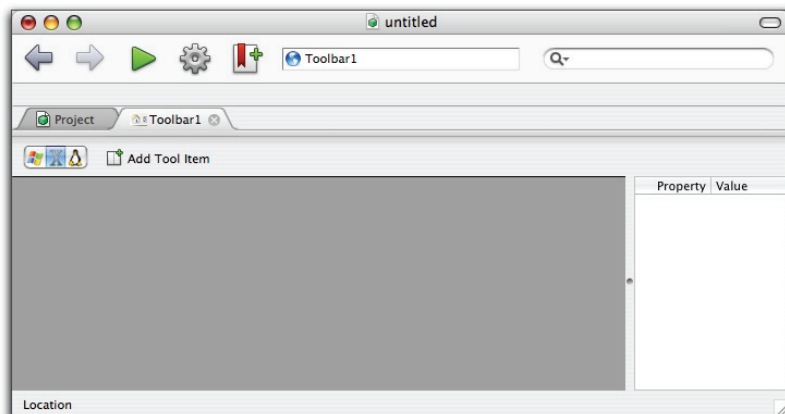
- 1 **In the Project Editor, click the Add Toolbar button in the Project Editor toolbar or choose Project ► Add ► Toolbar.**

REAL Studio adds a class based on the Toolbar class to the project.

- 2 **Double-click the Toolbar class in the Project Editor.**

The Toolbar Editor appears.

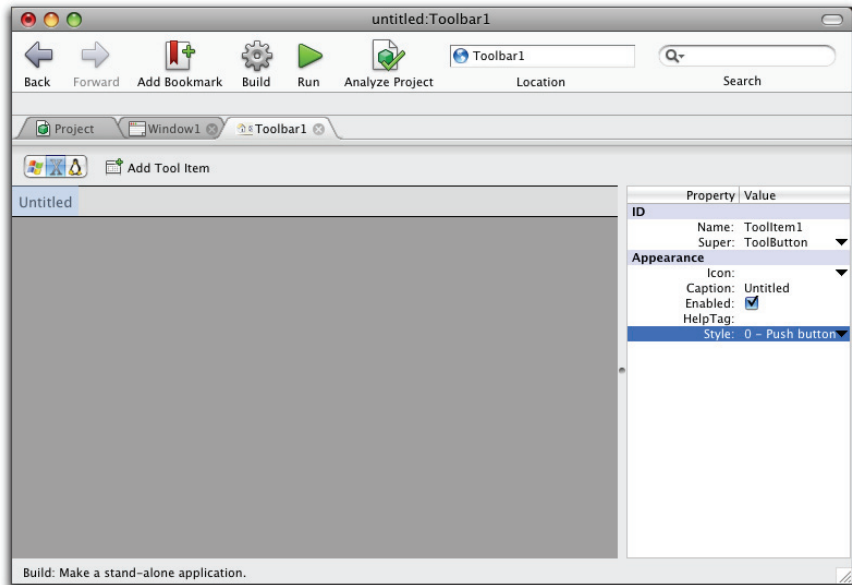
Figure 172. The Toolbar Editor.



3 Click Add Tool Item to add the first toolbar item.

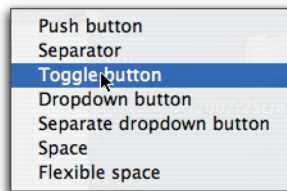
The Toolbar Editor adds the toolbar to the editor and places a new blank item in the first position.

Figure 173. The Toolbar Editor with the first item added.



The Properties pane changes to show the properties of this item. You create the finished item by specifying its properties. The type of button is determined by its Style property. Your choices are shown in Figure 174.

Figure 174. The Style drop-down list.



Here are descriptions of each button style:

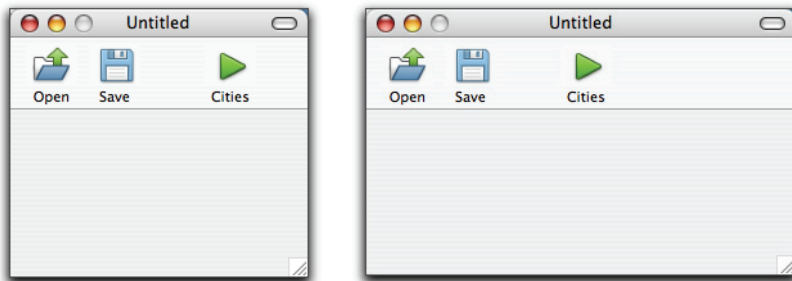
- **Push Button:** The ToolButton is a standard PushButton. Use the Caption and Icon properties to assign the PushButton's label and icon. To set the Icon property, add the image of the icon to the Project Editor. This will populate the Icon drop-down menu in the Properties pane.
- **Separator:** The ToolButton is a separator. It draws a vertical line in the toolbar.

- **Toggle Button:** The `ToggleButton` is a button that toggles between its normal and depressed state. Use the `Caption` and `Icon` properties to assign the Toggle button's label and icon. Toggling a Toggle button does not affect the state of other toggle buttons in the toolbar. That is, a group of Toggle buttons behave like Checkboxes, not RadioButtons.
- **Drop-Down Button:** The `ToggleButton` is a drop-down list. On Windows, an arrow is drawn by default. Use the `Caption` and `Icon` properties to assign the button's label and icon. If the Drop-down button has no arrow, the user can click on the text or icon to display the drop-down menu.

To specify the menu, you need to create a menu and assign it to the `DropDownMenu` property of the `ToggleButton` class. Handle the selected menu item in the `DropDownMenuAction` event of the `Toolbar` class. See the example for the `ToggleButton` class in the *Language Reference*.
- **Separate Drop-Down button:** The `ToggleButton` is a drop-down menu with a separate down arrow on its right. Use the `Caption` and `Icon` properties to assign the Toggle button's label and icon.

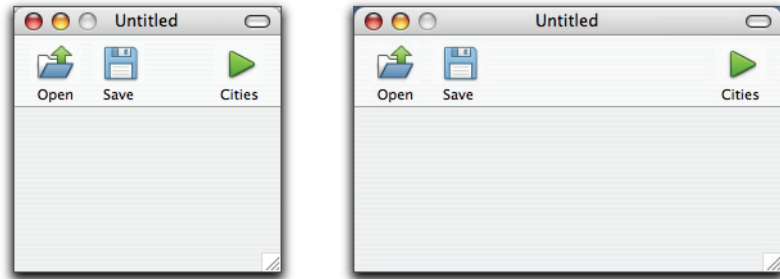
To specify the menu, assign it to the `DropDownMenu` property of the `ToggleButton` class. Handle the selected menu item in the `DropDownMenuAction` event of the `Toolbar` class. See the example for the `ToggleButton` class in the *Language Reference*.
- **Space:** The `ToggleButton` is a fixed-width space between ToolButtons. This is not supported on Windows, so no extra button or space appears.

Figure 175. A Fixed-Width Space button between the Save and Cities Toolbuttons.



- **Flexible Space:** The `ToggleButton` is a variable-width space between ToolButtons. It right-aligns the buttons to its right as the window is resized. This is not supported on Windows, so no extra space or button will be inserted.

The difference between fixed and flexible spaces is illustrated in Figures 175 and 176. In both cases, the window has been enlarged by the user. As the window is enlarged, the fixed-space maintains the spacing between the buttons to its left and right. The flexible space expands as the window is enlarged.

Figure 176. A Flexible Space button between the Save and Cities ToolButtons.

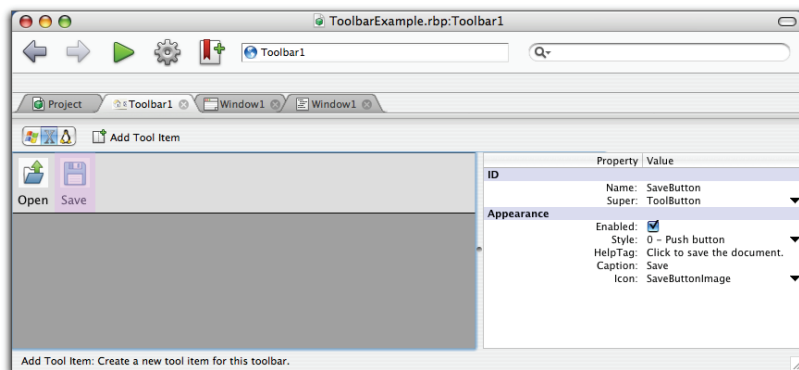
- 4 Use the Properties pane to choose the Style of the item and, if necessary, the values of the Caption and Icon properties.

You do not need to specify an Icon for the Separator style.

- 5 (Optional) Change the Name of the item (you use the Name property to refer to the item in code) and the HelpTag. Deselect the Enabled property if you want the item to be disabled when the window opens.

The HelpTag is displayed when the mouse hovers over the item but the mouse button is not pressed.

Figure 177 shows a completed toolbar with Open and Save buttons.

Figure 177. A completed toolbar with two Push buttons.

In Figure 177, the Save button is selected, so the Properties pane shows its properties.

After adding items to the toolbar, you can reorder them by dragging an item to the left or right. You can preview the toolbar on other platforms by clicking one of the Toolbar preview mode buttons in the Toolbar's toolbar.

Figure 178. The Toolbar preview mode buttons.

Adding the Toolbar to a Window

The completed toolbar in the Project Editor will also appear in the list of Project Controls in the project's Window Layout Editors. Choose Project Controls from the Controls drop-down list in a Window Editor and drag the toolbar to the desired position in the window.

To make the toolbar work, you need to handle the toolbar's Action event. It occurs when the user interacts with buttons or drop-down lists. If the user clicks a button or a menu, the item that was clicked is passed to the Action event handler of the Toolbar. In the Action event, you can test which item was clicked and take appropriate action. For example, here is the Action event handler for the simple toolbar that is shown in Figure 177. It tests the Name property; this was set in the Properties pane for each item.

```
Sub Action (Item as ToolItem)
  Select Case Item.Name
    Case "OpenButton"
      MsgBox "You clicked the Open Button."
    Case "SaveButton"
      MsgBox "You clicked the Save Button."
  End Select
```

If you have used either type of Drop-down list button, you need to handle the user's menu selection. You do this in the ToolButton's DropDownMenuAction event handler. It is passed the item in the toolbar and the menu item that was selected. For example, here is an event handler for a drop-down menu with three items.

```
DropDownMenuAction(item as ToolItem, hitItem as MenuItem)
  Select Case hitItem.Text
    Case "Grand Blanc"
      MsgBox "You chose Grand Blanc from the "+item.Name+" ."
    Case "Flint"
      MsgBox "You chose Flint from the "+item.Name+" ."
    Case "Bad Axe"
      MsgBox "you chose Bad Axe from the "+item.Name+" ."
  End Select
```

Please see the sections in Chapter 5 on events and event handlers and the *Language Reference* entries for the Toolbar, ToolButton, and ToolItem classes.

The completed toolbar in the Project Editor will also appear in the list of Project controls in the project's Window Layout Editors. Choose Project Controls from the Controls drop-down list in a Window Editor and drag the toolbar to the desired position in the window.

To make the toolbar work, you need to handle the toolbar's Action event. It occurs when any item is clicked. The item that was clicked is passed to the Action event handler, so you can test which item was clicked and take appropriate action. For example,

here is the Action event handler for the simple toolbar that is shown in Figure 177. It tests the Name property; this was set in the Properties pane for each item.

```
Sub Action (Item as ToolItem)
  Select Case Item.Name
    Case "OpenButton"
      MsgBox "You clicked the Open Button."
    Case "SaveButton"
      MsgBox "You clicked the Save Button."
  End Select
```

The Timer

The Timer executes some code once or repeatedly after a period of time has passed. Although the Timer displays an icon when placed in a window in the Window Editor, it is not visible in built applications. It is not a control and can be created with code. See the section on the Timer object in the *Language Reference* for more details.

Controls for Working With Databases

There are two special-purpose controls that are relevant only in projects that access databases: the DatabaseQuery control and the DataControl control. For more information about both controls, see “Creating Databases with REAL Studio” on page 603.

DatabaseQuery

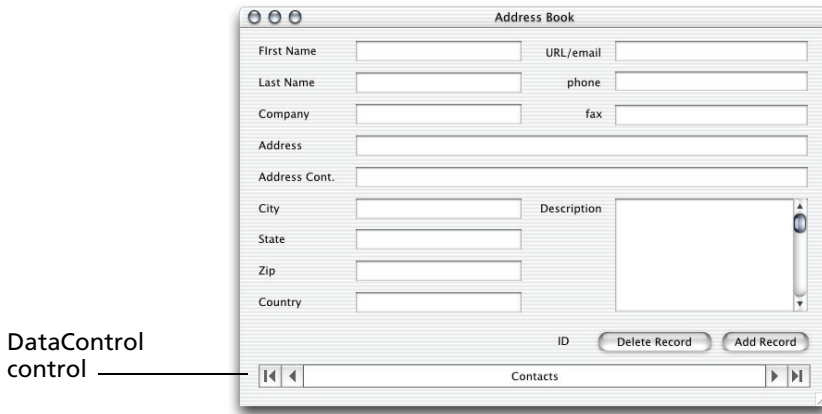
The DatabaseQuery control can be used to send SQL queries to the database. This function can also be done with the language (without using the DatabaseQuery control at all). When you add a DatabaseQuery control to a window, it is not visible in the built application.

To use the DatabaseQuery control, you write a SQL statement and assign it to the SQLQuery property of the control. The SQLQuery is executed automatically when its window appears. It also has one method, RunQuery, which runs the query stored in the SQLQuery property.

DataControl

The DataControl control is a “composite” control that provides a very easy way to build a front-end interface to a database. It consists of four record navigation buttons (First Record, Previous Record, Next Record, and Last Record) and a caption.

Figure 179. A DataControl control used for navigating among records.



When linked to a database and controls that display data, you can create a fully-functional database front-end with no programming. See the example in the section “The DataControl Control” on page 618.

The Spotlight Query Control

Mac OS X 10.4 introduced a sophisticated search engine known as “Spotlight.” It relies on metadata attributes of items to do its searches. The SpotlightQuery class in the REAL Studio language provides access to the Apple Spotlight API so that you can incorporate Spotlight searches in your REAL Studio applications. You can pass queries to Spotlight, determine when it has results, and then display those results in standard REAL Studio controls such as ListBoxes and TextFields.

Of course, these capabilities are supported only for users who are running Mac OS X 10.4 and above. On all other operating systems, the SpotlightQuery control and your Spotlight-related code will do nothing.

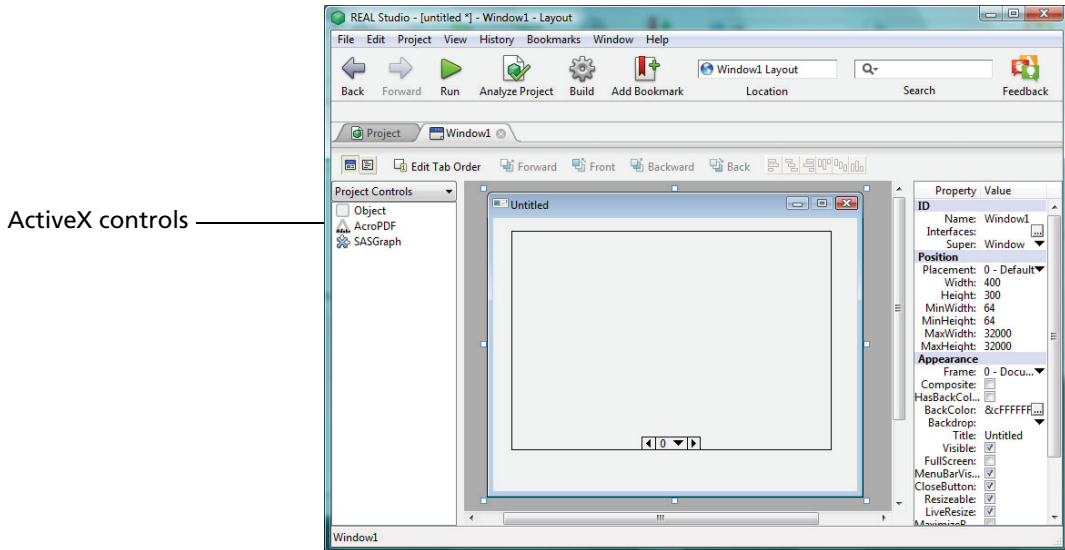
You use the SpotlightQuery class to perform these searches. Use the Query property of this class to pass your Spotlight query to the API. The SpotlightQuery class is listed in the Built-in Controls list to make it convenient to add it to a window, but it is not subclassed from the Control class. The SpotlightQuery class is not based on the Control class so you can also create SpotlightQuery objects via code and implement your Spotlight support that way.

The entry for SpotlightQuery in the *Language Reference* has examples of Spotlight support both with and without a SpotlightQuery control in a window.

ActiveX Controls

The Windows version of REAL Studio allows you to add ActiveX controls to the Controls pane. They are added as Project controls. Once added to the Controls pane, an ActiveX control can be added to your application as if it were a part of REAL Studio.

Figure 180. ActiveX controls in the Project Controls list.

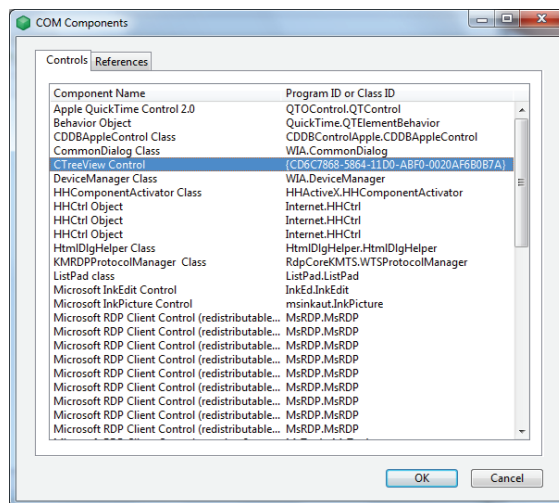


To add an ActiveX control, do this:

- 1 In the Windows version of REAL Studio, click the **Add ActiveX Component** button in the Project Editor toolbar or choose **Project ► Add ► ActiveX Component**.

The COM Components dialog appears, listing all the ActiveX controls and automation objects that are currently installed on your computer.

Figure 181. The ActiveX Components dialog box.



- 2 Select the controls or automation objects that you wish to add to your Project Controls list and click **OK**.

The selected controls or objects are added to the items in the Project Editor. If they are ActiveX controls, they are also added to the list of Project controls in the all the Window Editors in your project. Controls generally have their own icon, while programmable objects use the icon for the OLE Container control.

The ActiveX components that you add in this manner become part of the current project rather than your copy of REAL Studio. To add the ActiveX components to other projects, repeat this process.

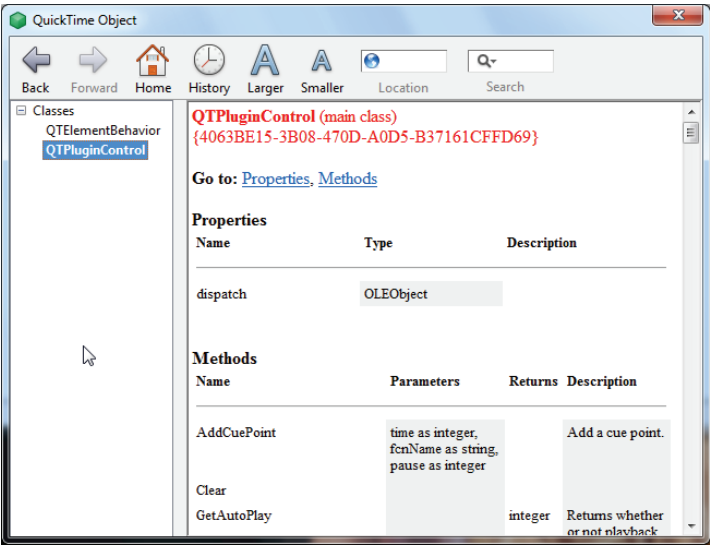
You add an ActiveX control to a window just as any built-in control. Drag it to a window and use the Code Editor to add code to the ActiveX control. Please refer to Microsoft documentation for information about programming ActiveX components and the specific ActiveX components you want to use. Microsoft documentation is in the MSDN library at:

<http://msdn.microsoft.com/library/>

COM
Component
Help

When you add a COM Component to your project, a COM Component menuitem is added to the Help menu. It has submenus for each ActiveX component that you have added to your project. This is the ActiveX online Language Reference. For example, Figure 182 is the page for the QT Plugin component:

Figure 182. The QuickTime Component reference page.



OLEContainer
Control

The OLEContainer control enables you to embed ActiveX controls in your interface. ActiveX is a Windows-only feature. The ActiveX controls that you add to your project are derived from OLEContainer class.

The Container Control

The Container Control is a special type of control that can contain all other types of controls and other Container Controls. The ContainerControl is available only in the Professional and Studio versions of REAL Studio.

A Container Control itself is invisible in built applications. Think of it as a blank canvas onto which you can place visible controls. It enables you to organize several controls into groups and manage dynamic layouts. Container Controls have multiple uses, You can:

- Organize groups of controls into reusable interface components,
- Create custom controls made up of several constituent controls,
- Increase encapsulation of complex window layouts,
- Create dynamic layouts.

Unlike other controls, the Container Control does not appear in the list of controls in the Window Editor. It appears instead in the Project Editor toolbar or in the Project ► Add submenu item in the Project Editor.

To create a Container Control, add a Container Control to the Project Editor. Double-click it to open its Container Control Editor. Its editor looks like a Window Editor except that the Container Control is represented by a rectangle. A Container Control has no frame and no window widgets, such as the Minimize, Maximize, and Close buttons of a document window.

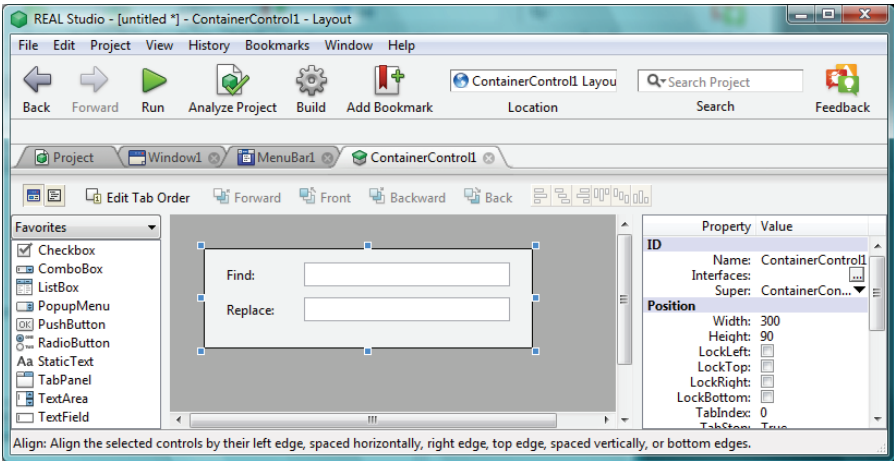
You place controls in a Container Control just as if it were a window in a Window Editor. Controls that are placed in a Container Control have a Tab Order that works inside the parent Container Control. If you wish, you can convert a ContainerControl to a Window by changing its Super class to Window in the Project Editor. Conversely, you can convert a window to a ContainerControl by changing its Super class to ContainerControl. However, you cannot convert the default window of a project.

To use a Container Control in your project, follow these general steps:

- Add a Container Control to your project by clicking the Add Container Control button in the Project Editor toolbar or using the Project ► Add ► Container Control menu item,
- Rename it using its Properties pane (optional),
- Double-click the Container Control in the Project Editor to open its editor and then design the instance of the Container Control in its Container Control Editor,
- Display the window in which you want to display the Container Control in its Window Editor. Use the Control pane's drop-down list to switch to Project Controls. Your Container Controls will be listed in the Project Controls, not the Built-in Controls.
- Add the Container Control to the window.

Figure 183 is an example Container Control in its editor that will be used in a Find/Replace dialog box. It consists of two StaticText controls and two TextFields.

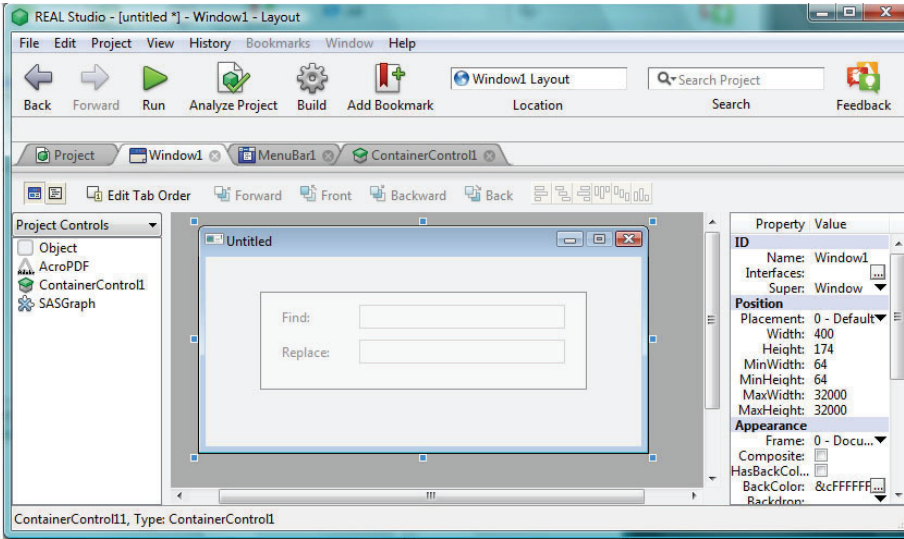
Figure 183. The Container Control in its Container Control Editor.



You can also place and position it via the language. This feature enables you to create dynamic screens.

In the Figure 184, the Container Control shown in Figure 183 has been placed inside a TabPanel control in a Find and Replace dialog box.

Figure 184. A Container Control in a window.



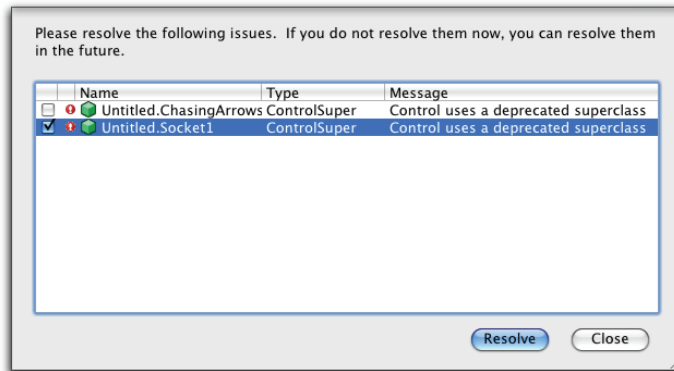
Opening an Old Project

When you open a project that was built with an early version of REAL Studio, it may contain controls and other project items that have since been deprecated. In

general, the deprecated items have been replaced with new items that provide equivalent or superior functionality. For example, the ChasingArrows control has been replaced with the ProgressWheel.

If you open an old project that contains a deprecated control, REAL Studio detects the deprecated control and offers you the opportunity to convert it to a supported control. As the project is about to be opened, you will see a dialog such as this.

Figure 185. A Convert Control dialog.



In this case, the project contains a ChasingArrows control and a Socket control.

Highlight the control to be converted to enable the Resolve button and then click Resolve. The icon on the left becomes a check. In this example, the Socket control is converted to a TCPSocket.

Resolve each item and then click Close to put away the dialog and open the project. When you do so, you will find that the deprecated controls have kept their original names but their super class has been updated.

Changing The Tab Order

The order in which the user moves through controls that receive the focus when he presses the Tab key is called the *Tab Order*. When a window opens, REAL Studio gives the focus to the control that is farthest back that can also receive the focus. A control is said to be in the back when another control can cover it up when it is moved to the same location. It is the first control in the Tab Order.

You can change the Tab Order in several ways in the Window Editor.

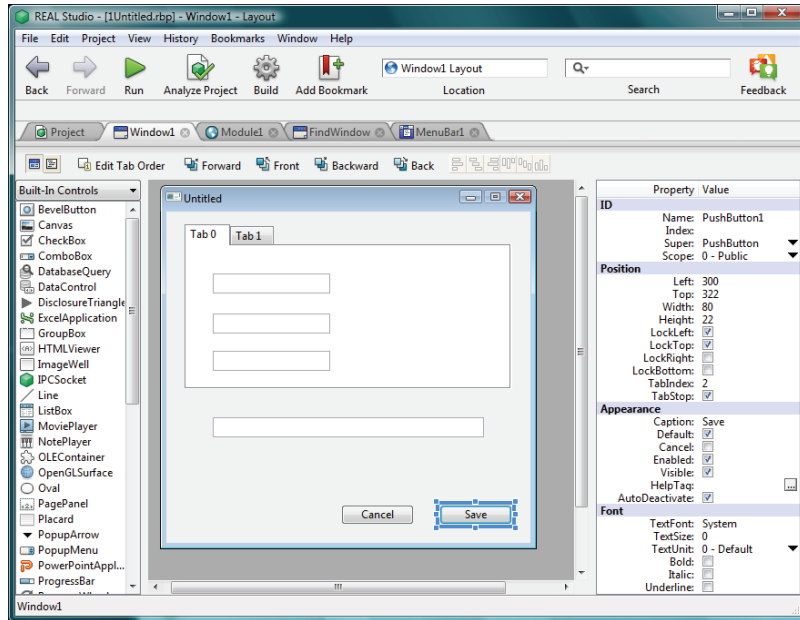
Using the Window Editor Toolbar

The Window Editor toolbar has four buttons that you can use to move the selected control backwards or forwards in the tab order: Forward, Front, Backward, or Back. All visible controls also have a property, `TabIndex`, which denotes its position in the tab order. A control's position in the Tab Order is also indicated by the value of its `TabIndex` property.

The control with a `TabIndex` value of 0 is first in the tab order. It's the one in the back. The Front and Back buttons move the control to the front or back in the order, while Forward and Backward move it one position up or down.

For example, in Figure 186, the Save button in the layout is selected. The four buttons in the toolbar are enabled and the Properties pane for the Save button shows that its `TabIndex` is 2.

Figure 186. A selected control in the Tab Order.



You can modify the Tab Order by changing the value of the `TabIndex` property in the IDE or via code. For example, you can use the line:

```
Me.TabIndex=0
```

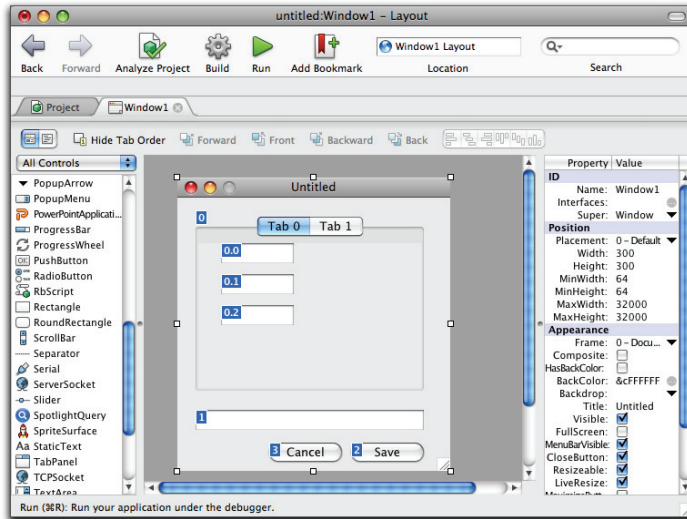
in a control's Open event handler.

Using the Edit Tab Order Mode

The Window Editor toolbar menu contains an Edit Tab Order button that changes the way in which you can modify the Tab Order. When you click Edit Tab Order, the editor changes to a mode that temporarily prohibits normal Window Editor actions such as adding and deleting controls or modifying their properties. It also disables the toolbar buttons.

If the Edit Tab Order button is not shown, add it to the Toolbar by right-clicking on the Toolbar and choosing Customize.

In this mode, each control that can get the focus on any platform gets a badge that indicates its current position in the Tab Order, with zero being the first control in the order.

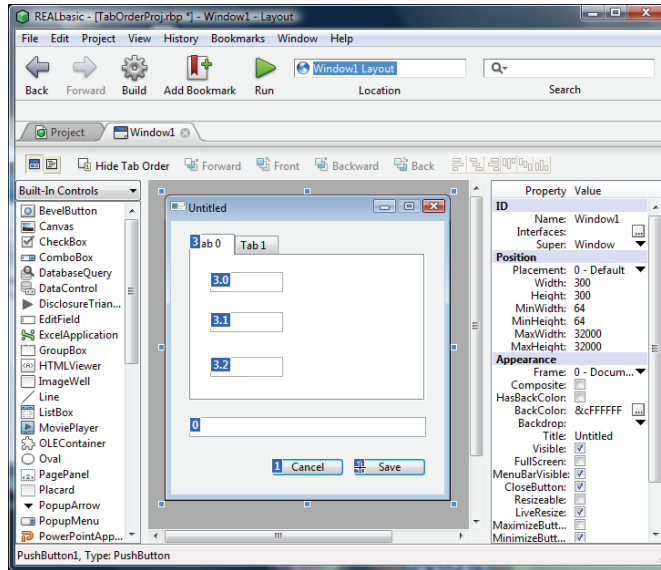
Figure 187. A Window Editor in its Edit Tab Order mode.

The displayed Tab Order assumes that the control is capable of getting the focus. Some controls do not receive the focus on Macintosh, but are included in the Tab Order if they can get the focus on any platform on which REAL Studio can run. For example, PushButtons, CheckBoxes, and RadioButtons do not get the focus on Macintosh, but are shown in the Tab Order within the Macintosh IDE.

The displayed Tab Order is valid for Windows and Linux builds of the application. For Macintosh builds, you need to ignore the Tab Order of a control that will not receive the focus. The `TabIndex` property, however, is valid on Macintosh if the user has the Full Keyboard Access option enabled at the Mac OS X system level.

When Edit Tab Order is on, the pointer changes to a crosshairs. You can change the Tab Order for a control by clicking on its badge to change its value. This is shown in Figure 188. The Tab Order for the Save button is being changed.

Figure 188. Changing the Tab Order in Edit Tab Order mode.



Click on a control to change its Tab Order. Continue clicking on controls until the resulting Tab Order is satisfactory. You don't necessarily need to click on all the controls.

To leave Edit Tab Order mode, click on the Hide Tab Order button. The window will then return to the normal editing mode.

If any of the controls in the window are within other controls, then the Tab Order numbering system uses “dot” notation to describe the Tab Order within a control.

Tab Panels, Page Panels, and GroupBoxes generally serve as parent controls. Controls that are nested within the parent get a decimal tab order number.

That is, the parent control gets an integer denoting its Tab Order relative to other controls, while the nested controls get a decimal number indicating the Tab Order within the parent. The dot notation indicates the control hierarchy. In Figure 188, for example, the Tab Panel has a badge of 3 and the three child TextFields have badges that display 3.0, 3.1, and 3.2. The Properties panes for the three child controls will show TabIndex values of 0, 1, and 2.

That is, the Properties pane displays the local Tab Order within the Parent control. This indicates shows how the nested controls are selected when the parent gets the focus.

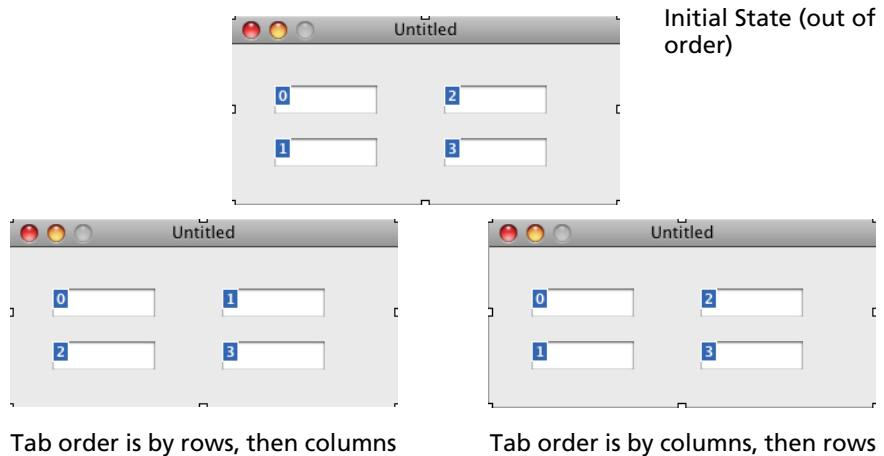
When you are in Edit Tab Order mode, you can switch tabs on a Tab Panel or pages on a Page Panel by holding down the Control key (Windows and Linux) or Option key (Macintosh) while clicking on the tab or page indicator.

Auto-Adjustment of the Tab Order

If you only want to set the Tab Order to a simple Top-left to Bottom-right order, REAL Studio offers a convenient way to do this. Two automatics sort orders are available. One favors the rows and the other favors the columns. That is, one order starts at the top-left control and next chooses the control to its right; the other chooses the top-left control and then chooses the control below it.

You can assign either of these orders via the Edit ► Auto Adjust TabIndexes menu item. It offers two choices: Top-down Left-right and Left-right Top-down. The choices are illustrated below.

Figure 189. Setting the Tab Order automatically.

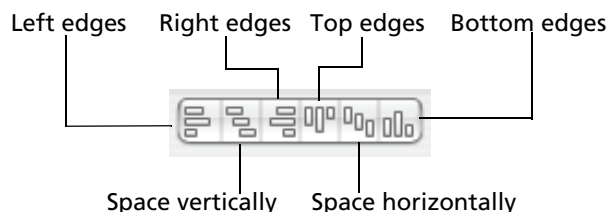


Aligning Controls with Other Controls

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

To align controls that have already been placed in the window, you use the Alignment icons on the right side of the Window Editor's toolbar. You can also use the equivalent menu command in the Edit ► Align submenu.

Figure 190. The Window Editor's Alignment and Spacing icons.



The icons align the selected objects as shown in Figure 190. The Edit ► Align submenu has the same commands.

The Space Vertically and Space Horizontally commands distribute three or more objects evenly. They are described in the section that follows immediately.

If you need to align several controls, do this:



- 1 Click on the control whose position is already correct to select it.**
- 2 Click the Back button in the Window Editor toolbar or choose Edit ► Arrange ► Send to Back to insure that the selected control remains in place while the other controls move to align with it.**

You can also move the control to the back by setting its ControlOrder property to zero using the Properties pane.

- 3 While holding down the Ctrl key (Command key on Macintosh), select each of the controls you wish to align or draw a selection rectangle around the controls to be aligned.**
- 4 Click the desired alignment icon in the Window Editor toolbar or choose the equivalent menu command from the Edit ► Align submenu.**

Spacing Controls Evenly

REAL Studio provides an easy way to reposition controls to evenly distribute empty space between them. The row of alignment icons (Figure 190) also contain icons for distributing controls evenly along the horizontal and vertical axes.

To distribute the controls evenly, do this:



- 1 Click on a control to select it.**
- 2 Hold down the Ctrl key (Command key on Macintosh) and select at least two other controls or draw a selection rectangle around the additional controls.**
- 3 To distribute the controls horizontally, click the Space Horizontally icon or choose Edit ► Align ► Space Horizontally.**
- 4 To distribute the controls vertically, click the Space Vertically icon or choose Edit ► Align ► Space Vertically.**

Your selected controls will now be equally spaced in either the horizontal or vertical axis.

The Control Hierarchy

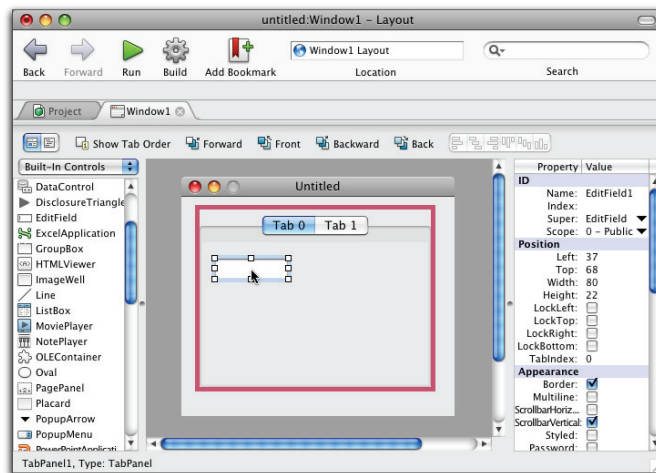
In some cases you build portions of your interface by placing some controls within another control. For example, you use the GroupBox control to organize other controls, usually RadioButtons, CheckBoxes, PopupMenus, and TextFields/TextAreas. The TabPanel and PagePanel controls also designed to enclose other controls. The Control Hierarchy in the IDE enables you to work with all the controls as a group.

The control that encloses the others is known as the *parent* control and the controls that are entirely within its borders are the *child* controls. This works automatically if you create the controls in the order of: parent-child. That is, first create the control that encloses the others, then create the child controls or duplicate existing child controls and keep them inside the parent control.

If you create the control that is supposed to be the parent after creating some of the other controls, you can convert it to the parent by moving it to the back in the Tab Order. Do this by selecting the control that is supposed to be the parent and click the Back button in the Window Editor toolbar or choose Edit ► Arrange ► Send to Back, or change the value of the TabIndex property to zero in the Properties pane.

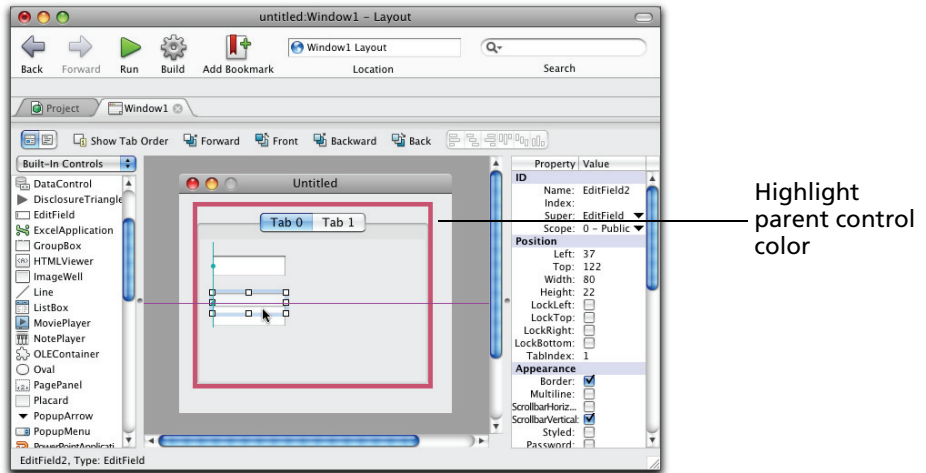
When you drag a child control to a parent and then drop it into place, a marquee surrounds the parent. This gives you visual confirmation that you are, in fact, adding a child to a parent. For example, in Figure 191 a StaticText control is being dropped onto a TabPanel.

Figure 191. Adding a child control to a parent.



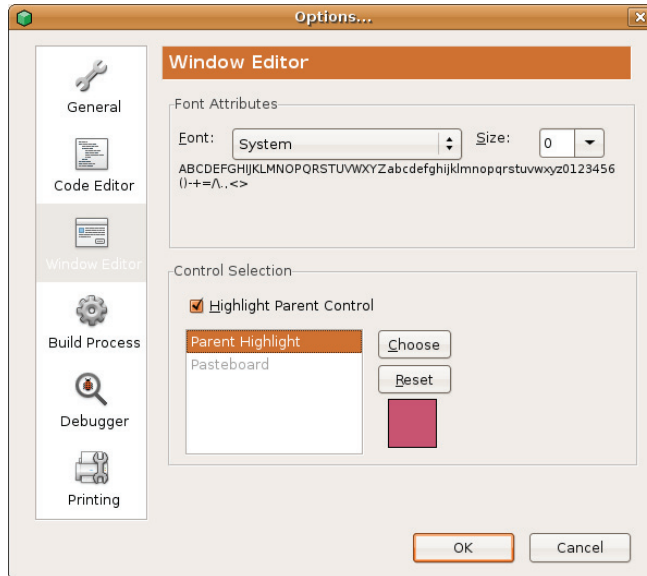
By default, a marquee also surrounds the parent control whenever you select an existing child control. For example, in Figure 192 the user clicked the second StaticText control and is aligning it with the top StaticText. A marquee surrounds its parent (TabPanel) control as well.

Figure 192. Selecting and aligning a child control.



An option in the Window Editor options panel turns this feature on or off. By default, this feature is enabled; to disable the feature, choose **Edit ► Options (REAL Studio ► Preferences on Macintosh)** and select the **Window Editor** panel and deselect **Highlight Parent Control**.

Figure 193. The Window Editor options.



You can additionally select the color of the marquee. Highlight Parent Highlight in the Control Selection list and click the color sample to display the Color Picker to select another color. Click Reset to revert to the original color.

If a control is not completely enclosed by another control, then it doesn't automatically become a child; it simply overlaps the other control.

If you move a child outside its parent, it is no longer a child of that control. If you move it completely inside another control, it becomes the child of that control.

When you copy a parent control, you copy all its child controls as well.

You can create more than one level of nesting. For example, you can place a `GroupBox` within a `TabPanel` and then place several `RadioButtons` within the `GroupBox`. In this case, the `GroupBox` is the parent of the `RadioButtons` and the `TabPanel` is the parent of the `GroupBox`. To make this work automatically, you must create them in the order that respects the hierarchy: First create the `TabPanel`, then the `GroupBox`, and then the `RadioButtons`.

However, you cannot nest a `TabPanel` in another `TabPanel` or a `PagePanel` and you can't nest a `PagePanel` in either a `TabPanel` or another `PagePanel`.

In the Control Selection area of Window Editor options (Figure 193 on page 186), you can also set the color of the Window Editor's *pasteboard*. The pasteboard is the area that surrounds the window itself in the Layout editor. In Figure 192, for example, the pasteboard is gray.

Clicking the Reset button in the Control Selection area resets the colors to the OS's default colors.

Control Hierarchy Features

If you duplicate a child control and leave the duplicate within the parent, then it is automatically a child. However, if you move the duplicate outside the parent, it is no longer a child. A control must be fully enclosed by the parent to be considered a child.

In Figure 194 the bottom control was duplicated from the top one, but has been moved out of the `TabPanel`. It is no longer a child.

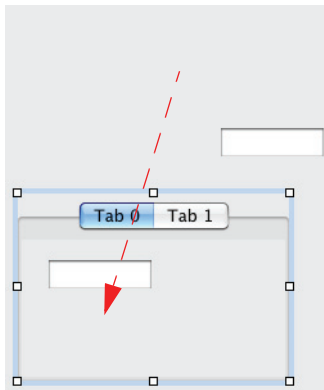
Figure 194. A child and an 'unrelated' control.



If you move the bottom `TextField` back within the `TabPanel`, it becomes a child of the `TabPanel` once again.

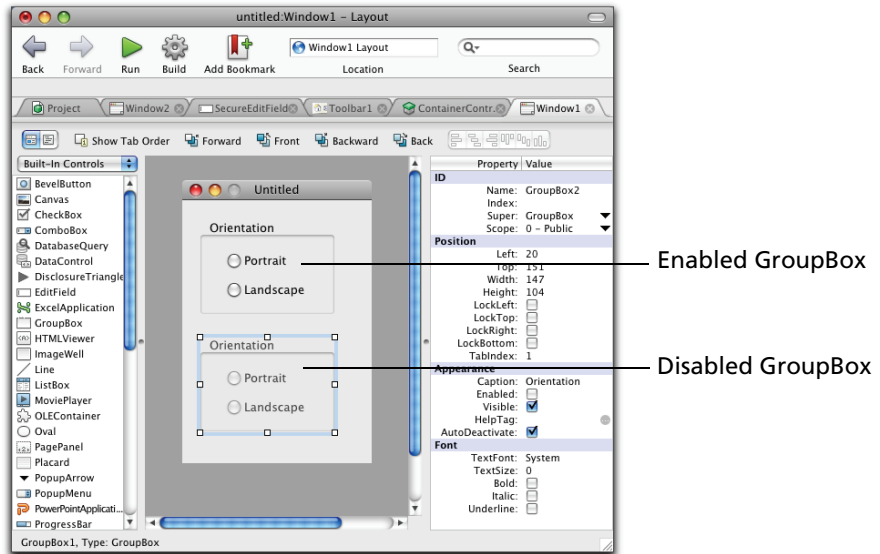
Moving a parent control moves its child controls as well. In Figure 195, the `TabPanel` shown in Figure 194 was moved down. It takes the top `TextField` with it, but leaves the bottom one orphaned.

Figure 195. Moving a parent control.



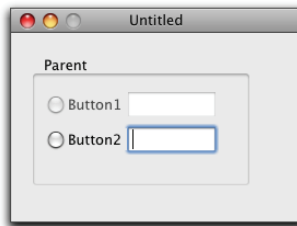
- Deleting a parent control deletes all child controls.
- Hiding a parent hides all child controls, but retains the previous visibility status of all children.
- Showing a parent control shows only the child controls whose visibility is set.
- Disabling a parent control disables all its child controls, but retains the previous visibility status of all children. In the IDE, disabling a container visually disables all the child controls, but does not update the Enabled property.

In Figure 197, a disabled GroupBox in which two RadioButtons as children is shown in the IDE. The top GroupBox is enabled and the bottom GroupBox is disabled.

Figure 196. A GroupBox disabling its children.

When the GroupBox is disabled, all of its children are disabled.

Enabling a parent control enables only the child controls whose Enabled property is set. In Figure 197, only the bottom row of child controls have the Enabled property set. When the GroupBox is enabled, the top row of controls remains disabled.

Figure 197. A enabled GroupBox control.

If this behavior is not desirable, you can break the control hierarchy by setting the Parent property of any child control to Nil in the Open event of the window. You can also break the control hierarchy by moving the child control in back of the parent control in the tab order. The enclosed control will then behave independently of its former “parent” in the built application.

Adding Menus and Menu Items

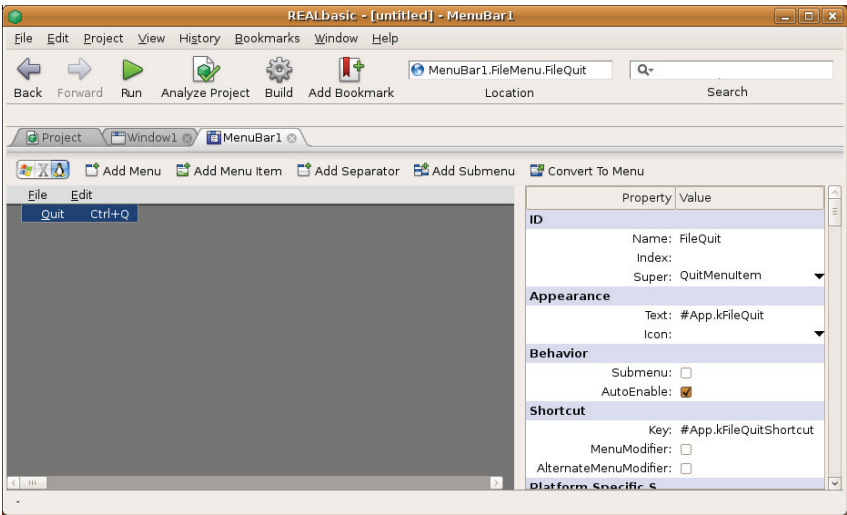
REAL Studio has a built-in Menu editor that makes adding menubars, menus, and menu items to your project easy. The Desktop Application template includes a menubar that is used as the default menubar for the entire application. You can

accept the default or create other menubars that are used for certain windows. In this case, you can assign a menubar to a window.

The Default Menubar

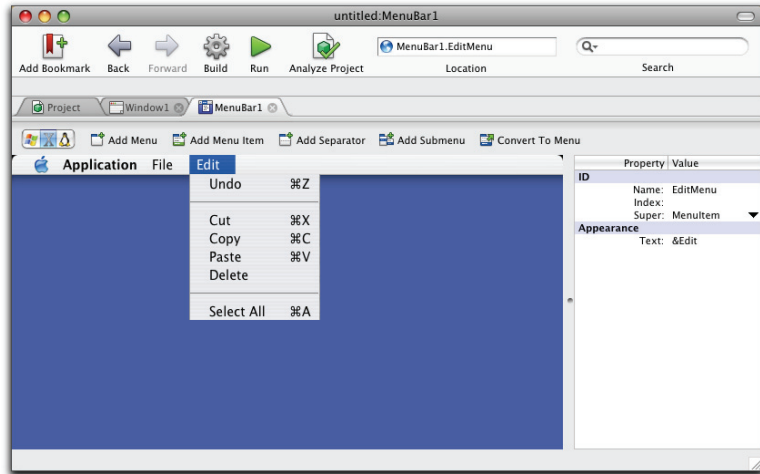
The default menubar for a Desktop Application, Menubar1, includes File and Edit menus and the standard File and Edit menu items. The File menu has one menu item, Exit (on Windows) or Quit (on Macintosh and Linux). The properties of the Quit/Exit menu item are supplied, so that the menu item works automatically. You don't need to modify or add to the menu item's properties in order to enable it.

Figure 198. The Quit Menu item's Properties (Linux).

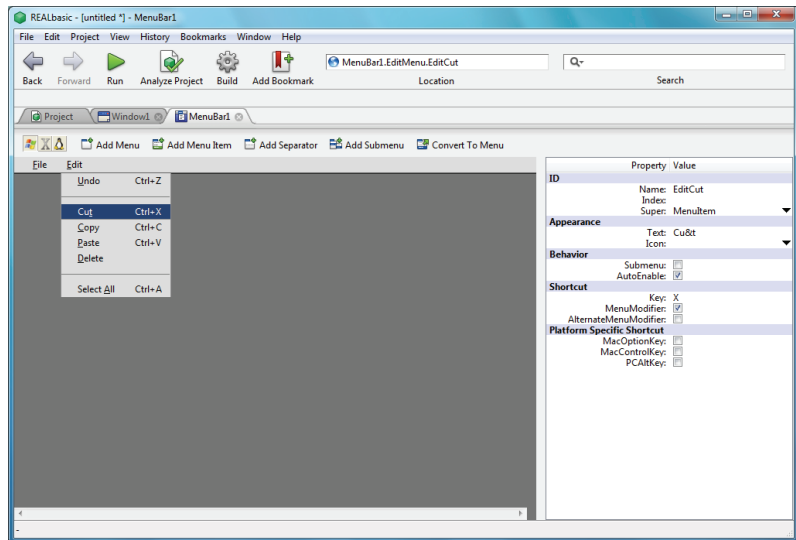


The menu item's text is specified via a constant of the App class, as is the keyboard shortcut. The menu item is subclassed from the QuitMenuItem class rather than the MenuItem class. This means that it will automatically call the Quit method and quit the application when it is called.

Similarly, the Edit menu is populated with Undo, Cut, Copy, Paste, Delete, and Select All menu items.

Figure 199. The default Edit Menu (Macintosh).

The Cut, Copy, Paste, and Select All menu items work automatically with TextFields and TextAreas. To take advantage of the built-in functionality, you should not rename the automatic menu items. You do not need to add code to use the keyboard equivalents or accelerators (Windows and Linux) or change any other defaults.

Figure 200. The Cut menu item's properties.

In Figure 200, the “t” is defined as the accelerator key and Ctrl+X (Command-X on Macintosh) is defined as the keyboard equivalent.

Adding Menubars

When you first create a Desktop Application project, the project includes one window and one menubar, named MenuBar1. This is the application's default menubar. By default, it is used for your entire application. However, you can add additional menubars to your project and associate a menubar with a window. On Linux and Windows, each window has its own menubar, so each window's menubar is always used for that window. On Macintosh, when a window becomes active, the window's menubar replaces the current menubar.

To add an additional menubar to your project, do this:

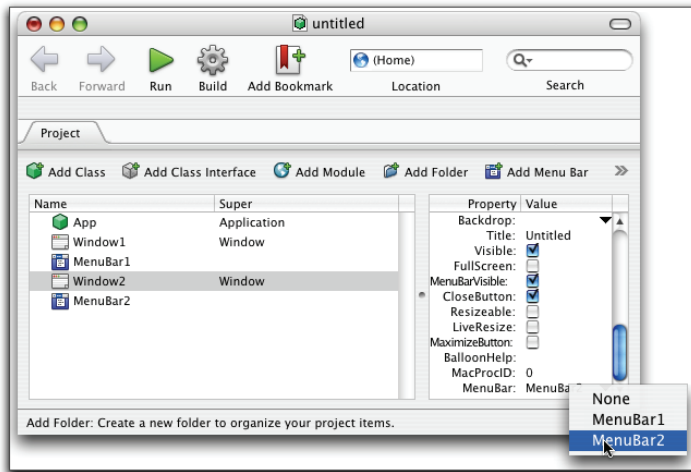
- 1 If it is not already displayed, click the **Project** tab to view the **Project Editor**.
- 2 Click the **Add Menu Bar** button or choose **Project ► Add ► Menu Bar**.

A new menubar appears in the Project Editor, named MenuBar2.

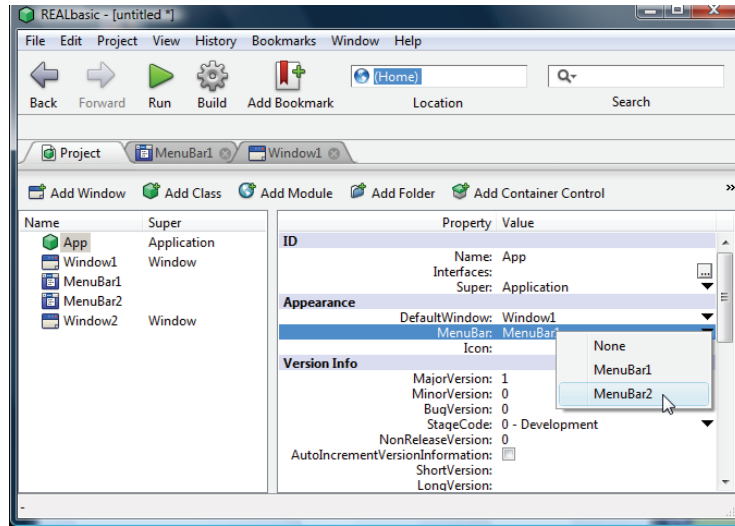
- 3 Follow the steps in the following section, “Adding Menus” on page 193 to add menus and menu items to the menubar.
- 4 To assign a menubar to a window, use the window's **Properties** pane to set the window's **MenuBar** property to the name of the menubar.

The window's MenuBar property is listed in the Appearance group in the Properties pane. All the menubars that belong to the project appear in this pop-up menu.

Figure 201. Changing a window's menubar.



- 5 (Optional) To change the default global menubar, set the **App** class's **MenuBar** property to the desired menubar.

Figure 202. Changing the default global menubar after adding a second menubar.

On Windows MDI applications, the global menubar is used as the enclosing frame's menubar.

Adding Menus There are four steps for implementing a menu and its items. These steps are the same for the default global menubar and any additional menubars that you add to the project.

- Adding the menu to a menubar using the Menu Editor,
- Adding the menu's items using the Menu Editor,
- Enabling the menu item. A disabled menu item is grayed out and is not selectable.
- Adding a menu handler. A menu handler is the method that tells REAL Studio what to do when the user chooses the (enabled) menu item. An enabled menu item without a menu handler does nothing.

There are two ways to enable a menu item. If the menu item should be enabled all the time, you should select the menu item's `AutoEnable` property in the menu item's Properties pane (see Figure 200 on page 191). The `AutoEnable` property is selected by default. When you add the menu handler, the menu item becomes functional. For example, you might want to leave the `AutoEnable` property selected for the New menu item in the File menu which creates a new document in the application.

If the menu item should not be enabled all the time, deselect the `AutoEnable` property. In this case, the menu item is disabled by default. You need to tell REAL Studio when to enable it. For example, you might have an Export menu item that should be enabled only if the user has selected an item to export.

When `AutoEnable` is `False`, the menu item is disabled until the user attempts to pull down the menu containing the item. At that moment, you can decide if conditions are right for the menu item to be enabled. For example, a “Save” menu item would need to check that the document has never been saved or that changes have been made to the document since the last save operation. For information about enabling menu items, see the section, “Enabling Menu Items” on page 360. For information about creating a menu handler, see the section, “Adding Code To a Menu Handler” on page 359.

Note You can also add or remove menu items via code. For more information, see the entry for the `MenuItem` class in the *Language Reference*.

Desktop Applications normally have a File menu with a Exit (Quit on Macintosh) menu item. You can remove the Edit menu only if your application has no controls that could be edited by the Edit menu items.

By default, the Menu Editor previews the menubar and menus for the platform on which the REAL Studio IDE is running. You can change the preview to another platform by clicking one of the Preview Mode buttons in the Menu Editor toolbar.

Figure 203. The Preview Mode buttons (Linux selected).

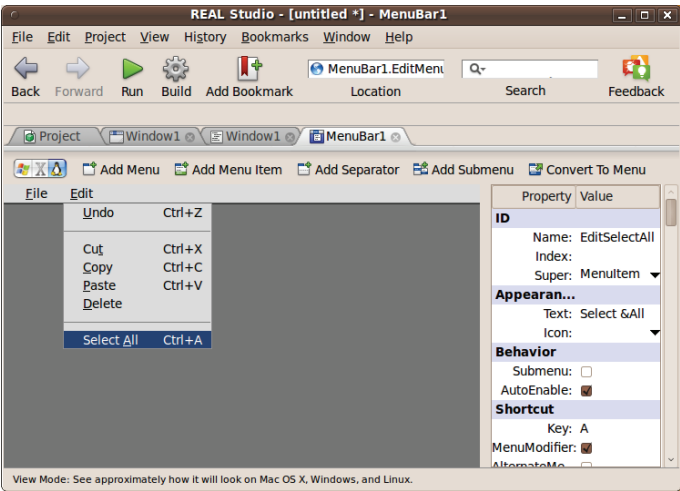


To add a menu to a menubar, do this:

- 1 If it is not already displayed, click the Project tab.
- 2 Double-click on a Menubar to open its Menu Editor.

The Menu Editor appears.

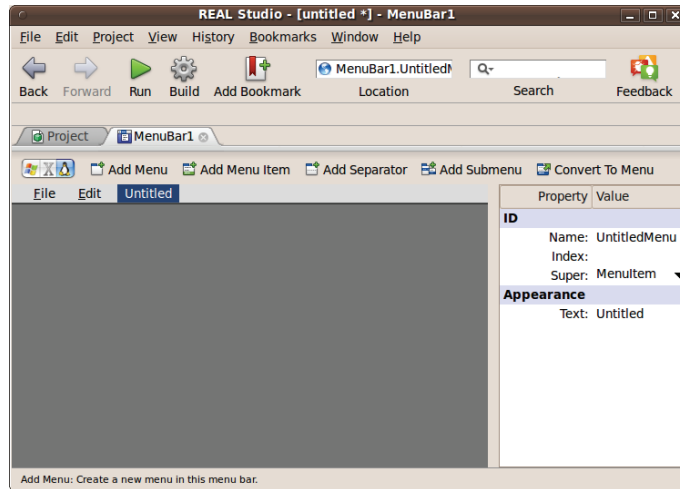
Figure 204. The Menu Editor.



3 Click the Add Menu button or choose Project ► Add ► Menu.

A new menu is added to the menubar as the last menu, or, if a menu is selected, to the right of the selected menu. Its properties are shown in the Properties pane.

Figure 205. The Menu Editor after adding a menu.



4 (Optional) If the new menu does not appear in the desired position in the menubar, drag it horizontally to its new position.

By default, the new menu is added to the right of the existing menu bar menus.

5 In the Properties pane for the new menu, enter the Name of the menu and the Text that will appear in the menubar.

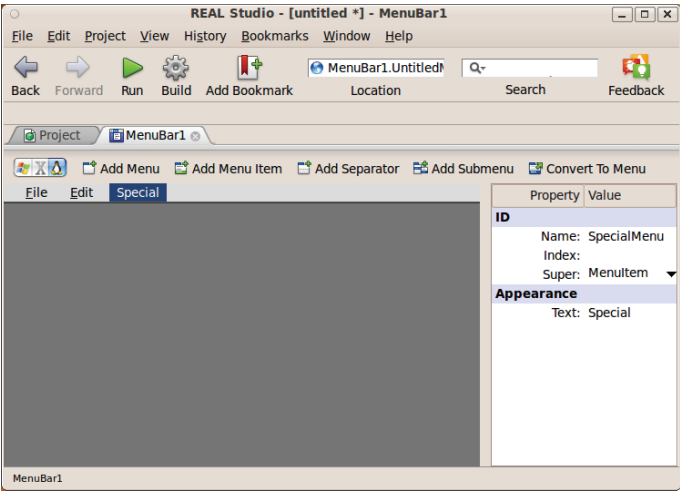
Table 1: Properties of a MenuItem.

Name	Description
Name	The internal name of the Menu used to identify it in programming code.
Super	The class of object the Menu control is based on. This will normally be MenuItem. You can also use PrefsMenuItem, AppleMenuItem, or QuitMenuItem.
Text	The text that will appear in the Menu bar. To designate a keyboard accelerator for the menu place the "&" prior to the accelerator key. For more information, see the section "Accelerator keys" on page 198.

If you enter the Text property first and then press Return, REAL Studio will suggest a name.

Figure 206 shows a new menu and its properties in the Properties pane.

Figure 206. A new menu in the Menu Editor.



Adding a Help Menu

Many applications have a Help menu that is the right-most menu in the application. The Help menu may contain menu items that give the user access to an application-specific help system. You can add a Help menu to your project.



To add a Help menu, do this:

- 1 Add a menu to the end of your menubar.
- 2 Set the Text property of the menu to Help.

Adding Menu Items

The Menu Editor makes it easy to add menu items to your menus. You can specify shortcut keys and accelerator keys via the IDE and also build submenus.

Keyboard Shortcuts

You can assign keyboard shortcuts to menu items, but remember that the operating system looks for a shortcut starting from the leftmost menu. That means that if you assign the same keyboard shortcut to two different menu items, one of them won't work. There are also several specific keyboard shortcuts that are reserved for specific functions. These are:

Table 2: Reserved Keyboard Shortcuts

Menu	Keys (Windows and Linux)	Keys (Macintosh)	Command
File	Ctrl-N	⌘-N	New
File	Ctrl-O	⌘-O	Open...
File	Ctrl-W	⌘-W	Close
File	Ctrl-S	⌘-S	Save
File	Ctrl-P	⌘-P	Print...
File	Ctrl-Q	⌘-Q	Quit

Table 2: Reserved Keyboard Shortcuts (Continued)

Menu	Keys (Windows and Linux)	Keys (Macintosh)	Command
Edit	Ctrl-Z	⌘-Z	Undo
Edit	Ctrl-X	⌘-X	Cut
Edit	Ctrl-C	⌘-C	Copy
Edit	Ctrl-V	⌘-V	Paste
Edit	Ctrl-A	⌘-A	Select All
Edit	Esc	⌘-period	Terminate an operation
Edit	Ctrl-M	⌘-M	Minimize

Windows has the following system-wide keyboard equivalents that you should be aware of:

Table 3: Windows keyboard equivalents.

Key Combination	Description
Alt+Esc	Switches to the next application.
Alt+F4	Closes an application or window.
Alt+hyphen	Opens the window menu for a document window.
Alt+Print Screen	Copies an image in the active window onto the Clipboard.
Alt+Spacebar	Opens the window menu for the application's main window.
Alt+Tab	Switches to the next application.
Ctrl+Esc	Switches to the Start menu.
Ctrl+F4	Closes the active group or document window.
F1	Starts the application's Help file, if it exists.
Print Screen	Copies an image of the screen onto the Clipboard
Shift+Alt+Tab	Switches to the previous application. The user holds down both Shift and Alt while pressing Tab.

The MenuItem class has the following properties that you use to specify the keyboard shortcut:

Table 4: Properties for specifying a keyboard shortcut.

Property	Data Type	Description
Key	String	The key that the user presses to trigger the menu item's event handler. Enter a single uppercase letter or the name of a non-printable key, as described below. It is pressed in combination with one or more modifier keys, specified with the other properties.

Table 4: Properties for specifying a keyboard shortcut.

Property	Data Type	Description
MenuModifier	Boolean	If True, the user must hold down the Ctrl key (Windows and Linux) or Command key (Macintosh) when pressing the Key key. When a value is entered for the Key property, this property is set to True by default.
AlternateMenuModifier	Boolean	If True, the user must hold down the Shift key when pressing the Key key.
PCAltKey	Boolean	If True, the user must hold down the Alt key when pressing the Key key. This property is for Windows and Linux only.
MacOptionKey	Boolean	If True, the user must hold down the Option key when pressing the Key key. This is a Macintosh-only property.
MacControlKey	Boolean	If True, the user must hold down the Control key when pressing the Key key. This is a Macintosh-only property.

For example, to specify the keyboard shortcut Ctrl+H, enter “H” as the Key property and select the MenuModifier property. To specify Shift+Ctrl+H, you would select both the MenuModifier and AlternateMenuModifier properties. The keyboard shortcut Alt+Ctrl+H requires the PCAlt and MenuModifier key properties.

When you make your selections, the keyboard equivalent is shown to the right of the MenuItem’s text in the Menu Editor.

You can also specify non-printable keys. The following choices are available: F1 to F15, Tab, Enter, Space, Del (Delete), Return, Bksp (Backspace), Esc, Clear, PageUp, PageDown, Left, Right, Up, Down, Help, and Ins (Insert). When you specify a non-printable key in the IDE, simply enter its name from this list into the Key property area and select the desired modifier key by selecting its checkbox.

Accelerator keys

The Windows and Linux platforms also have the concept of keyboard accelerators. In addition to the keyboard equivalent, which work on all platforms, you can also add a keyboard accelerator for each menu and menu item. When you designate a key as the keyboard accelerator, it is underlined in the menu name or menu item. The user can display the menu or invoke the menu item by holding down Alt and pressing the accelerator key. With a comprehensive system of accelerators, a user can use the menu system without using the mouse at all.

To designate the accelerator key, precede the letter by an ampersand (“&”) in the menu or MenuItem’s Text property. For example, if you are creating a menu named “Actions” and you want to make the keyboard accelerator the “a”, you would enter “&Actions” as the menu’s Text property. To make the “t” the accelerator key, enter

“Ac&tions”. Keep in mind that accelerators are not shown and do not work on Macintosh.

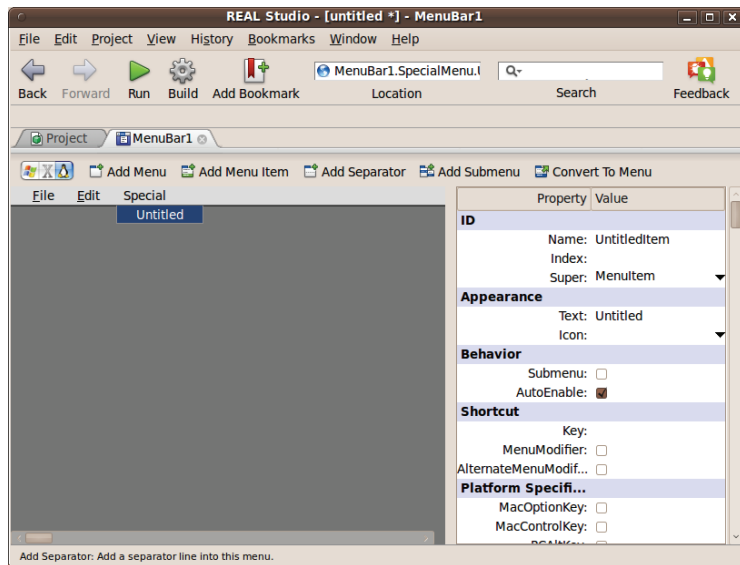
To add a menu item to a menu, do this:



- 1 In the Menu Editor, select the menu you wish to add a menu item to by clicking on it.**
- 2 Click the Add Menu Item button or choose Project ► Add ► Menu Item.**

A new menu item is added to the selected menu and its properties are shown in the Properties pane.

Figure 207. A new menu item added to the menu.



- 3 In the Properties pane, enter the Text property for the menu item and press Enter.**

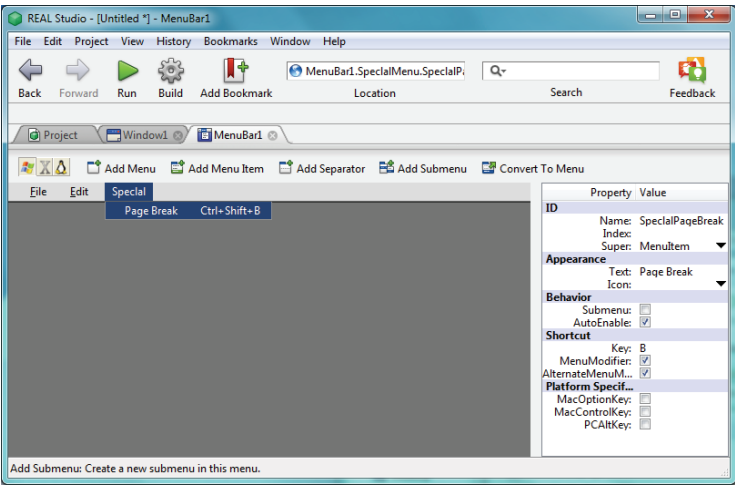
To create a keyboard accelerator for Windows and Linux, precede the accelerator key with the “&” symbol.

The Menu Editor will supply a default Name. If you wish, you can replace the suggested Name with a name of your own.

- 4 If desired, add a keyboard shortcut by assigning a letter (or the name of a non-printable key) to the Key property and select at least one of the modifier key properties listed in Table 4 on page 197.**
- 5 If desired, disable the AutoEnable property if you need to enable the menu item only under certain conditions.**

Figure 208 on page 200 shows a Page Break menu item added to the Special menu with the Keyboard equivalent Shift-Command-B (Shift-Ctrl-B on Windows and Linux).

Figure 208. A new menu item in the Special menu.



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

Adding a Submenu



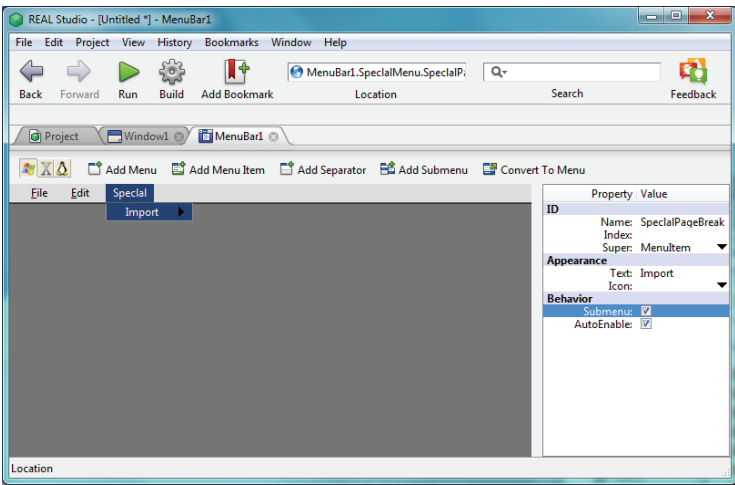
Submenus are menu items that display an additional menu to their right. The menu item itself is not selectable. It acts only as a title for the submenu.

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

- 1 Click on the menu item in the Menu Editor to select it.
- 2 In the Properties pane, place a checkmark in the Submenu property.

The Menu Editor adds an arrow to the right of the menu item to indicate that it is the parent of a submenu.

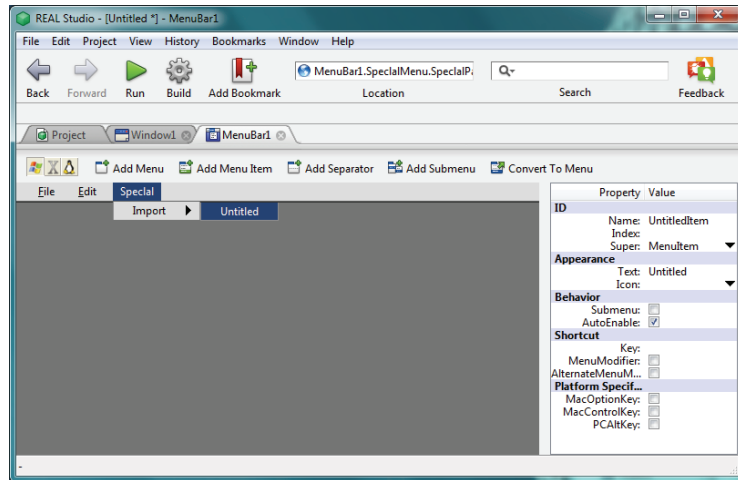
Figure 209. A menu item marked as the parent of a submenu.



3 Click the Add Menu Item button or choose Project ► Add ► Menu Item.

In the Menu Editor, an item is added to the submenu menu item you selected in step 1.

Figure 210. A menu item added to the Special ► Import menu item.



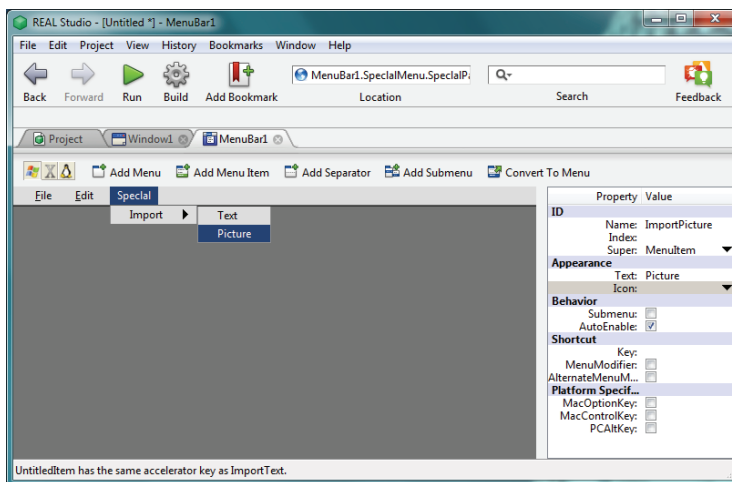
4 In the Properties pane, enter the Text for the submenu Item and press Enter.

5 To add an additional submenu item to this submenu, select this submenu item and click the Add Menu Item button in the Menu Editor toolbar.

In the Menu Editor, REAL Studio adds the new submenu item to the current submenu.

6 Use the Properties pane to enter the Text of this submenu item and press Enter.

REAL Studio suggests a default name for the new submenu item. If you wish, you can change it.

Figure 211. A submenu with two submenu items.

If you wish, you can continue to add a submenu to an existing submenu, creating a three-level hierarchical menu system. However, submenus can be difficult to navigate for the new computer user. They also hide menu items from view. If a user scans through the menu items looking for a particular menu item, he may not look at the submenus. Consider the audience for your application before using submenus. If many of your users will be new computer users, consider displaying a dialog box to choose the functions you could put in a submenu.

Adding a Menu Item to the Mac OS X Apple and Application Menus

Mac OS X applications add a new menu between the Apple menu and the standard File menu. It automatically takes on the name of the application. When you create a standalone Mac OS X application, the menu's name is the name of the Macintosh version of the application that you entered in the App class's Properties pane. (see Figure 506 on page 700).

Although the Application menu appears in the Menu Editor when you use the Mac OS X preview, you cannot add menu items directly to it or to the Apple menu. Instead, you use two special MenuItem classes to install menu items onto those menus, PrefsMenuItem and AppleMenuItem. Use the PrefsMenuItem class to manage your Preferences menu item.

For your preferences menu item, put the menu item where you want it to appear under Windows and Linux and set its Super class to PrefsMenuItem. A MenuItem based on the PrefsMenuItem class will be moved automatically to the Application menu for the Mac OS X build.

According to Apple's Mac OS X user interface guidelines, you should use ⌘-, (Command-comma) as the keyboard shortcut for an application's Preferences menu item.



Similarly, use the `AppleMenuItem` class for any menu items that should appear in the application's menu on Mac OS X. Items subclassed from `AppleMenuItem` will move to the application's menu in the Mac OS X build.

In theory, you could also base a `MenuItem` on the `QuitMenuItem` class. `MenuItem`s that are based on `QuitMenuItem` automatically call the `Quit` method and quit the application. Since a `Quit MenuItem` is included in the File menu by default, you are not required to add such an item to your project.

The Exit (or Quit) menu item

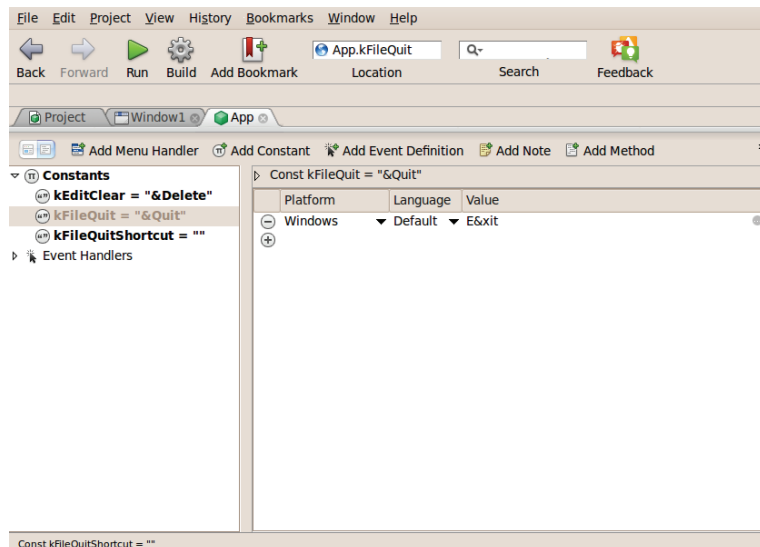
The `QuitMenuItem` class is intended only for the Quit (or Exit) menu items and causes the menu item to move to the application's menu for Mac OS X builds. Its default Text and Key properties use constants rather than text literals.

Figure 212. The Quit menuitem's properties.

Property	Value
ID	
Name:	FileQuit
Index:	
Super:	QuitMenuItem
Appearance	
Text:	#App.kFileQuit
Icon:	
Behavior	
Submenu:	<input type="checkbox"/>
AutoEnable:	<input checked="" type="checkbox"/>
Shortcut	
Key:	#App.kFileQuitShortcut
MenuModifier:	<input type="checkbox"/>
AlternateMenuModifier:	<input type="checkbox"/>
Platform Specific S	

Each constant is defined in the App class's Code Editor.

Figure 213. The menu item constants in the App class.



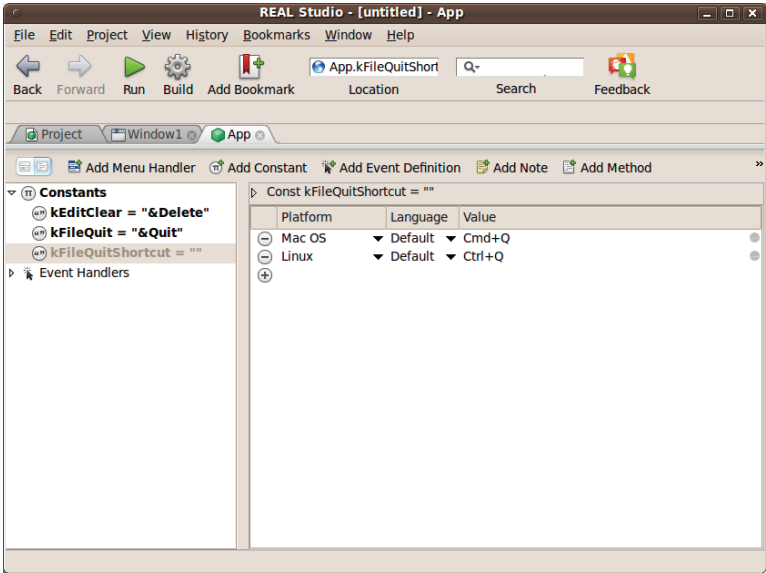
The `kFileQuit` constant's `Text` is defined as "Quit" by the line:

```
Const kFileQuit = "&Quit"
```

The interior of the table has one row, which redefines the `Text` for the `Windows` platform. If the platform is `Windows`, then the `Text` is "Edit."

The keyboard shortcut for this menu item is specified by the `kFileQuitShortcut` constant of the `App` class.

Figure 214. The `kFileQuitShortcut` constant.



For Macintosh, the shortcut is "Cmd+Q"; for Linux, it's "Ctrl+Q." For `Windows`, no shortcut is defined. This is indicated by the blank default value field in the constant definition area.

The use of constants for this purpose is discussed in the section "Using Constants to Localize your Application" on page 378.

Moving
Menus and
Menu Items



To move a menu item, do this:

- 1 Click on the menu item you want to move to select it.
- 2 Drag the menu item towards the position on the menu where you want it.
- 3 When the menu item is in the desired position, release the mouse button.

You can also move menus within the menu bar. In a similar fashion, drag a menu in the menu bar and move it to the left or right. Drop the menu when it is between the desired menus.

Converting a Menu Item to a Menu

You can also convert a menu item to a menu. To do so, select the menu item and then click the Convert To Menu button in the Menu Editor toolbar. The menu item is then removed from its menu and appears in the menubar. From there you can drag it to another position in the menu bar if you wish.

Removing Menu Items

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

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

Adding A Menu Item Separator

Menu item separators are lines that appear in between menu items to logically group items together. To add a menu item separator, simply select a menu item and click the Add Separator button in the Menu Editor toolbar or choose the Project ► Add ► Separator command. The separator will appear just below the selected menu item. If you wish, you can drag it vertically to a new location.

Creating MenuItems on the Fly

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

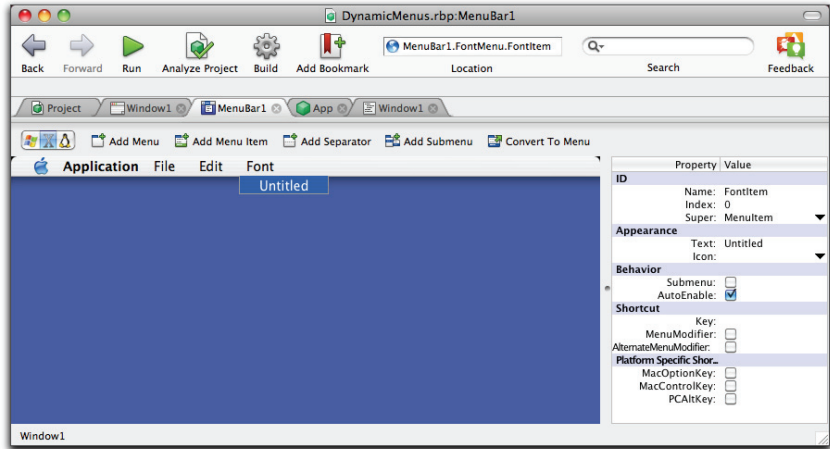
A common example of this is the Font menu that is normally included in any application that supports styled text. The programmer has no way of knowing in advance which fonts happen to be installed on the user's computer. The application must read the user's fonts and build the Font menu from that information.

The process is similar to creating an array of controls. A menuItem that can act as a template must already exist. This menu item will be "cloned." You can then change the clone's properties such as the Text, keyboard shortcut, etc.

The initial menuItem must have a value in its Index property in order to be used as a template. Assign a zero to the Index property of the menuItem. This signifies that it is the first element of an array.

You begin by creating the Menu normally. You then create a menuItem, name it, and set its Index property to zero. It doesn't matter what its Text property is. Figure 215 shows this menuItem after it has been created.

Figure 215. The initial Font menu item.



You then must write code to populate the array with the names of the fonts installed on the user's computer. If we assume that fonts won't be added or deleted while the application is running, we can build the Font menu's menuitems when the application starts up. To do this, place the code in the App class's Open event handler. This runs when the application starts up. The App class that is added to your project by default is the place for this.

The following code populates the Font menu when the application opens. It uses the built-in Font function which returns the name of the i^{th} font on the user's computer. It assumes that the Font menu item that was added manually is named FontItem and has the Index of zero.

```
Dim m as MenuItem
Dim nFonts as Integer

nFonts=FontCount-1
//build the font menu
FontItem(0).text=Font(0) //name the first Font menuitem
For i as Integer = 1 to nFonts //loop through the remaining fonts
    m=New MenuItem //create new menu item
    m.Text=Font(i) //obtain name of  $i^{\text{th}}$  font
Next
```

That builds the Font menu on the user's computer. Next, the menuitems must be enabled or else they will be dimmed.

You can add code for the EnableMenuItems event handler for the dynamic menuitems, such as:

```
Dim nFonts as Integer = FontCount-1
For i as integer = 0 to nFonts
    FontItem(i).Enabled=True
Next
```

If you wish to be able to programmatically remove menu items you have created dynamically, you need to store the reference that was returned when you created the menu item. You can then use this reference to remove the menu item by calling the Close method. Suppose you are storing references to the menu items in a module property array called “WindowRefs.” You can then remove a particular dynamically created menu item (the item stored in the fourth array element in this case) using this syntax:

```
Window Refs(4).Close
```

There is another approach to this problem. It uses a custom MenuItem class. When you create dynamic menus using this approach, you actually create a custom class based on the MenuItem class and add each menu item using this subclass. Since this technique does not rely on the Menu Editor, it will be covered in the chapter on events and objects.

For more information on this technique, see the section “Creating New Menu Items On The Fly” on page 362.

Importing and Exporting Menus



Menubars can be exported to the desktop and imported into other projects. When you export a menubar, only the menubar is exported, not the menu items’ menu handlers.

To export a menubar, do this:

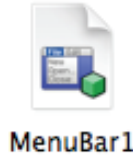
- 1 In the IDE, click the tab of the menubar you want to export or, if the menubar is not open, click on it in the Project Editor.**
- 2 Choose File ► Export *MenubarName*, where *MenubarName* is the name of the menubar you have selected in the previous step.**

A save-file dialog box appears.

- 3 Navigate to the desired directory, enter a filename, and click Save.**

REAL Studio saves the menubar as a .rbm file. Its desktop icon looks like this:

Figure 216. An exported menubar.



To import a menubar, do this:

1 From any IDE editor, choose File ► Import.

A standard open-file dialog box appears.

2 Navigate to the directory that contains the exported menubar, select it, and click Import.

REAL Studio adds the imported menubar to the Project Editor. It uses the name it had in the project from which it was exported.

User Interface Guidelines

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

Each platform that REAL Studio supports has its own conventions. User interface guidelines are available from the following sources:

- **Windows:** Microsoft's User Interface guidelines at:
<http://msdn.microsoft.com/en-us/library/aa894348.aspx>
- **Macintosh:** Apple Human Interface guidelines:
<http://developer.apple.com/documentation/UserExperience/Conceptual/AppleHIGuidelines/XHIGIntro/XHIGIntro.html>
- **Linux KDE Desktop:** KDE user interface guidelines:
<http://developer.kde.org/documentation/design/ui/>
- **Linux Gnome Desktop:** Gnome user interface guidelines:
<http://library.gnome.org/devel/hig-book/stable/>

KDE and Gnome are the most popular Linux desktops and are used by default in several major Linux distributions. However, there are others. Linux supports a greater degree of desktop customization than Windows and Macintosh.
- REAL Studio's Interface Assistant™ helps you create a nice interface by making it easy to align controls with other controls. For example, when you drag a control

near the edge of the window, REAL Studio displays an alignment line to let you know where to stop. For the left, right and bottom edges, it stops at 20 pixels and for the top edge, it stops at 14 pixels. When dragging a control towards another control, REAL Studio displays an alignment line at 12 pixels (the recommended distance between controls).

But there is more to a professional, polished interface than simply aligning controls. We all think we know how to create a nice interface because we have used lots of applications. But using an interface is a lot different from designing on.

BASIC Programming Concepts

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

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

Contents

- Data Types
- Storing Values in Properties, Variables, and Constants
- Executing Instructions with Methods
- Executing Instructions Repeatedly with Loops
- Decision Making

BASIC versus REAL Studio

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

For a long time BASIC was considered less powerful than other languages, but this was mostly due to the way it was implemented rather than the language itself. Spoken languages wouldn't be considered to be very powerful if you could only speak one word every 10 minutes. Computers actually only understand two things, 1 and 0. That's it. That's all they know. The rest of what a computer does all breaks down to that fundamental concept. Every command you give your computer is eventually translated into a series of instructions that consist only of 0's and 1's. These 1's and 0's that computers understand are referred to as *machine language*.

Most versions of BASIC used an interpreter program to execute the code. This means that each time a program ran, the BASIC interpreter had to turn the BASIC code into machine language. Other languages had compilers which are special programs that translate the programming language into machine language all at once. This makes programs execute faster because the real-time interpretation is removed.

REAL Studio has a compiler built in to it. That means your code runs as fast as possible. BASIC is a traditional interpreted programming language that starts with the first line of programming code and continues until the last line. REAL Studio is a modern, object-oriented version of BASIC. If you are new to programming that might not mean much now but it will. REAL Studio takes the simplicity of the BASIC language and adds the power of modern programming through its object-oriented implementation and compiler. Also, most programming languages require you to know quite a bit about how to communicate with the computer's operating system. REAL Studio abstracts you from all of that making it easier for you to learn and easier to run your application on computers running operating systems that are different from the one you created your application on.

Storing Values in Properties and Variables

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

What are Properties?

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

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

Variables

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

Data Types

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

One important programming convention in REAL Studio is that you must tell REAL Studio the data type of each variable and property that you create. It does not "infer" the data type from the way you use your variables and properties.

String

A String is just a series (or string) of characters. Basically any kind of information can be stored as a string. "Jeannie", "3/17/98", "45.90" are all examples of strings.

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

You can combine two strings with the addition symbol (+). For example, the statement “Big” + “Dog” results in the string “BigDog”. That is really the extent of the “mathematics” you can perform on strings. However, REAL Studio has many built-in functions that make processing strings easy. For example, the Lowercase function takes a string and converts all the characters to lowercase. The Trim function trims off any leading and trailing whitespace characters.

When you create a string literal by enclosing the string with quote marks, a null character automatically terminates the string, no matter where the null character appears. For example, if the string literal begins with a null character, the string will appear to be empty. If you may have unprintable or null characters in a string literal, you should filter them out before executing the line of code.

Integer

An integer is a whole number. It cannot accept a decimal or fractional value. REAL Studio offers both signed and unsigned integer data types that use one, two, four, or eight bytes of memory. The more bytes that are used, the larger the value that can be stored. A Signed integer can be negative, zero, or positive, while an Unsigned integer can only be zero or positive. The following table summarizes these data types.

Table 5. Integer data types in REAL Studio.

Data Type	Number of Bytes	Range
Int8	1	-128 to 127
Int16	2	-32,768 to 32,767
Int32 or Integer	4	-2,147,483,648 to 2,147,483,647
Int64	8	-2 ⁶³ to 2 ⁶³ -1
UInt8 or Byte	1	0 to 255
UInt16	2	0 to 65535
UInt32	4	0 to 4,294,967,295
UInt64	8	0 to 2 ⁶⁴ -1

REAL Studio’s Integer data type is a signed integer that uses the word length for the target platform. Currently this is four bytes on all supported platforms.

Because integers are numbers, you can perform mathematical calculations on them. Unlike strings, integers do not have quotes around them in your code.

Unsigned and signed integer operations, including comparisons, promote themselves to 64-bit signed integers when performing the operation. This fixes the problem where an unsigned 32-bit integer compared with a signed 32-bit integer would produce the wrong result.

Single

A Single is a number that can contain a decimal value. The range of a Single is shown in Table 6. In other languages, REAL Studio's Single may be referred to as a single precision real number. Because Singles are numbers, you can perform mathematical calculations on them. Single numbers use 4 bytes of memory.

Double

A Double is a number that can contain a decimal value. The range of a Double is shown in Table 6. In other languages, REAL Studio's Double may be referred to as a double precision real number. Because Doubles are numbers, you can perform mathematical calculations on them. Doubles use 8 bytes of memory. Since Doubles use more decimal places to represent a number, you would want to use Doubles rather than Singles if the extra precision is important in your calculations. Calculations on Singles are generally faster than Doubles, but the difference may not be meaningful on modern computers.

Table 6 gives the upper and lower limits of integers, singles, and doubles:

Table 6. Minimum and Maximum values for Integers, Singles, and Doubles.

Data Type	Smallest Value	Largest Value
Integer (Int32)	-2147483648	2147483647
Single	1.175494 e-38	3.402823 e+38
Double	2.2250738585072013 e-308	1.7976931348623157 e+308

Currency

This is a 64-bit (8-byte) fixed-point number format that holds 15 digits to the left of the decimal point and 4 digits to the right. It is scaled by 10,000 to give 4 digits to the right of the decimal point. It is always accurate to four decimal places. It is useful for calculations involving money and for calculations where accuracy is very important. It is compatible with the Currency data type offered in some versions of Visual Basic.

Boolean

A Boolean can take on the values of either True or False. Boolean values are False by default but can be set to True using REAL Studio's True function and back to False using the False function, or by any expression that returns a boolean value. Some of the properties of objects in REAL Studio are boolean values. For example, most of the controls have an Enabled property that is boolean.

Color

A Color is an intrinsic data type that stores the value of a REAL Studio color. A Color "value" actually consists of three numeric values that can be set using any of the three popular color models, Red-Green-Blue, Hue-Saturation-Value, or Cyan-Magenta-Yellow. Each value is stored as a byte. That means that each number can take on 256 possible values.

A Color can also be set to a value via one of the three color functions that correspond to the three color models. Table 7 summarizes your options.

Table 7. REAL Studio functions for specifying a color.

Function	Color Model	Data type and range of Parameters
RGB	Red-Green-Blue	Integer (0-255)
HSV	Hue-Saturation-Value	Double (0-1)
CMY	Cyan-Magenta-Yellow	Double (0-1)

For example, the following code assigns a color to the FillColor property of a rectangle:

```
Rectangle1.FillColor=CMY(.35,.9,.6)
```

Rectangle1 is the name of a rectangle object in a window and FillColor is one of its properties. The data type of FillColor is color, so it only takes values that correspond to a color specification. If you tried to assign a value of a different data type, you would get an error message.

You can also assign a color using the Red-Green-Blue model using the format:

```
&cRRGGBB
```

where *RR* is the value of Red in hexadecimal, *GG* is the value of Green in hexadecimal, and *BB* is the value of Blue in hexadecimal. Each value ranges from 00 to FF (FF is 255 in hexadecimal). The expression “&c” signals that the next six characters make up a color value in hexadecimal.

For example, you can write an expression such as:

```
Rectangle1.FillColor=&cFF594A
```

You can use any of the three color functions interchangeably. After you assign a color, you can read the value of any of its nine properties.

Variant

A Variant is a special data type that can store the value of any data type, including objects and arrays. The Variant’s Type method returns an integer that identifies the data type that the variant is storing:

Result	Description
0	Nil
2	Integer types of 32 bits or less, signed or unsigned.
3	Int64 or UInt64
4	Single
5	Double
6	Currency

Result	Description
7	Date
8	String
9	Object
11	Boolean
16	Color
18	CString
19	WString
20	PString
21	CFStringRef
22	WindowPtr
23	OSType
26	Ptr
36	Structure
4096, logically OR'ed with the element type	Array

If a variant stores an array, Type returns the result shown in the table logically OR'ed with the data type of an array element. You can get the element type by calling the `ArrayElementType` function.

Depending on the data type of the value stored by the Variant, you can use one of the following properties to convert the value to another data type:

Name	Description
<code>BooleanValue</code>	Returns the value of the Variant as a Boolean.
<code>ColorValue</code>	Returns the value of the Variant as a Color.
<code>CStringValue</code>	Returns the value of the Variant as a CString.
<code>CurrencyValue</code>	Returns the value of the Variant as a Currency.
<code>DateValue</code>	Returns the value of the Variant as a Date. When retrieving the string value of a date, the string is returned in the SQL date-time format: <code>YYYY-MM-DD HH:MM</code> .
<code>DoubleValue</code>	Returns the value of the Variant as a Double.
<code>Int32Value</code>	Returns the value of the Variant as an Int32.
<code>Int64Value</code>	Returns the value of the Variant as an Int64.
<code>IntegerValue</code>	Returns the value of the Variant as an Integer.
<code>ObjectValue</code>	Returns the value of the Variant as an Object.
<code>OSTypeValue</code>	Returns the value of the Variant as an OSType.
<code>PStringValue</code>	Returns the value of the Variant as a PString.
<code>PtrValue</code>	Returns the value of the Variant as a Ptr.

Name	Description
SingleValue	Returns the value of the Variant as a Single.
StringValue	Returns the value of the Variant as a String. If the Variant holds a Date, it is converted to a SQLDateTime format.
UInt32Value	Returns the value of the Variant as an UInt32.
UInt64Value	Returns the value of the Variant as an UInt64.
WindowPtrValue	Returns the value of the Variant as a WindowPtr.
WStringValue	Returns the value of the Variant as a WString.



Although you use a Date object to store date and date/time values, it is not a data type. Technically, Date is a class. See the section “Declaring Objects” on page 227 for information on using the Date class.

- Ptr** A Ptr is a 4-byte pointer to a chunk of memory. You can pass a MemoryBlock object via this data type and it will be treated as a pointer to the memory contained within the MemoryBlock.
- Delegate** A Delegate is a pointer that enables you to refer to a method or function. It encapsulates the memory address of a method in your code. It behaves as a class with a single method, named Invoke, whose parameters and return value match the delegate’s parameters and return type. The Invoke method calls the method the delegate instance represents. The Delegate data type has an implicit conversion operator to Ptr.
- WindowPtr** A WindowPtr is a 4-byte integer that is a synonym for the Handle property of a Window. You can pass a Window object to this data type and it will be treated as a pointer to the Window.
- CString** A CString is an ANSI C string, which is terminated with a Null character. There is an implicit (automatic) conversion between the REAL Studio String data type and the CString. When this is done, the Null character is automatically appended to the CString.
- WString** A WString is an ANSI Wide C string, which is also terminated with a Null character. There is also an implicit conversion between the REAL Studio String data type and the WString. A Null character is automatically appended to the WString.
- PString** A PString is a Pascal string. It includes a one-byte length identifier in the first position, followed by up to 255 characters. There is also an implicit conversion between the REAL Studio String data type and the PString.

CFStringRef	A CFStringRef is a Mac OS Core Foundation string object. There is also an implicit conversion between the REAL Studio String data type and CFStringRef.
OSType	The OSType is for passing OSType parameters to Mac OS and QuickTime for Windows functions. If you pass a REAL Studio string of four characters via this data type, it is automatically converted into a four char code Integer. Also, OSType does an implicit type conversion to String when you assign an OSType to a String variable.
Structures	<p>A structure is a compound data type. It consists of a list of fields, where each field can have its own data type and each field is of fixed length. It is similar to Visual Basic's "user-defined types" and Visual Basic .net's structures.</p> <p>Structures can be created only in REAL Studio modules, while all the other data types discussed in this section can be declared anywhere in REAL Studio. For information on structures, see the section "Structures" on page 394 in the chapter on Modules.</p>
Other Data Types	There are many other data types. You will learn about these in the next chapter.
Changing a Value From One Data Type to Another	<p>There may be times when you need to change a value from one data type to another. This is usually because you want to use the value with something that is designed to work with a different data type. For example, you might want to include a number in the title of a window. The title of a window is a string, not a number. Consequently, if you try to assign a number to the title of a window, REAL Studio will display an error message when you run your application. The error will tell you that the two data types are not compatible (they are different). Since the window title is a string, you will need to change the number into a string before you can assign it to the window title.</p> <p>Fortunately, REAL Studio has a built-in function called Str (which stands for String) that can change a number into a string. See the Str function in the <i>Language Reference</i> for more information. There is also a built-in function called Val (which is short for Value) that changes strings into numbers. See the Val function in the <i>Language Reference</i> for more information.</p>
Assigning Values to Properties	<p>The basic syntax for assigning a value is:</p> <pre>objectName.propertyName=value</pre>

For example, if you have a PushButton named PushButton1, and you want to set its Caption property to “OK”, you would use the following code:

```
PushButton1.Caption="OK"
```

You can read this as *change PushButton1's caption property to “OK”*. This syntax is used when you want a control in a window to change a property of a control in the same window. This syntax is sometimes called *dot syntax* since the dot is used to indicate that the property to the right of the dot belongs to the object to the left of the dot.

If you want a control to change a property of a control in another open window, you must include the target window's name (not its title) in the syntax. That syntax is:

```
windowName.objectName.propertyName=value
```

For example, suppose you have two open windows whose names are Window1 and Window2 respectively. You want a PushButton on Window1 to set the value of PushButton1's caption on Window2 to “OK”. The syntax looks like this:

```
Window2.PushButton1.caption="OK"
```

If you didn't specify the window, REAL Studio would assume you meant the control called PushButton1 in the window that contains the object executing the code. If you specify a window that is not open, REAL Studio will open the window and make the change. If you have more than one copy of the window open that contains the control you are trying to change, this syntax won't work because you won't be able to tell REAL Studio which copy of the window you are referring to. You will learn how to deal with this issue in the next chapter.

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

```
Title="Hello World"
```

REAL Studio will understand that “Title” is a property of the window.

In some cases, properties are, in effect, grouped hierarchically. That is, one property gives you access to properties and methods of another object. One example of this occurs with the Canvas control. This control comes equipped with a set of drawing tools that allow you to completely customize the appearance of the control. All the drawing tools are accessible via the Graphics property of the Canvas control. For example, the line;

```
Canvas1.Graphics.DrawRect 0,0,100,50
```

draws a 100 x 50 rectangle starting at the 0,0 coordinates of Canvas1. DrawRect is actually a method belonging to the Graphics class—not the Canvas class. The Graphics property of the Canvas class allows you to access this method.

Using Class Constants to Assign a Value

In some cases you need to choose a value from a list of specific values. For example, the Serial class has properties that specify the Baud rate, number of Stop bits, and Parity for serial communications. The Baud rate should be chosen from the list of possible values. The *Language Reference* gives the specific values that each of these properties can accept. In each instance, it also gives class constants for those values that are equivalent to the numeric values. Your code will be much more readable if you use the class constants instead of the numeric values. For example, here are the class constants for specifying the Baud rate.

Baud Rate	Value	Constant	Baud Rate	Value	Constant
300	0	Baud300	9600	8	Baud9600
600	1	Baud600	14400	9	Baud14400
1200	2	Baud1200	19200	10	Baud19200
1800	3	Baud1800	28800	11	Baud28800
2400	4	Baud2400	38400	12	Baud38400
3600	5	Baud3600	57600	13	Baud57600
4800	6	Baud4800	115200	14	Baud115200
7200	7	Baud7200	230400	15	Baud230400

To refer to a class constant, use the syntax:

ClassName.ClassConstant

For example, the expression **Serial.Baud19200** returns the value of 10.

If you want to assign a baud rate of 57600 to the Serial control named Serial1, you can write:

Serial1.Baud=Serial.Baud57600

instead of

Serial1.Baud=13

When you use the class constant, the line of code is much more readable. No one has to go and look up what the 13 means.

Me and Self

The REAL Studio language also has two very useful pronouns, *Me* and *Self*. *Me* refers to the control that you are working with. For example, if you are trying to set the

Caption property of `PushButton1` to “OK” in one of the control’s event handlers, you could write either:

```
PushButton1.Caption="OK"
```

or

```
Me.Caption="OK "
```

The advantage of using `Me` is that it makes your code generic. You could use it with any other `PushButton` control and it would work without change.

The pronoun *Self* refers to the control’s parent object. For a control in a window, it refers to the parent window. You can use it in an event handler for any control within the window. For example, the lines:

```
FindDialog.Title="Find"
```

and

```
Self.Title="Find"
```

and

```
Title="Find"
```

are all equivalent. In the first example, the `Name` property of the window is used; that is, the name of the window is `FindDialog`. In the second example, the `Self` pronoun accomplishes the same thing without “hardcoding” the window’s name. The last example takes advantage of the fact that the parent window is assumed when you are coding within one of the window’s controls.

`Me` refers to the control that fired the current event. Outside an event handler, it refers to the current object (`Self`).

Getting Values From Properties

You can get a value from a property in almost the same way you store values in properties. The only difference is that the target of the value (where you want the value of the property stored) goes on the left side of the equals sign and the object and property names go on the right. For example, if you had a variable named `X` and you wanted to assign `PushButton1`’s caption to it, the syntax would be:

```
x=PushButton1.Caption
```

And just as in setting properties, you can get the property of a control in another window by including the window’s name. For example, if you want to assign the

variable `x` to the `Caption` property of `PushButton1` in `Window2`, you would use this syntax:

```
x=Window2.PushButton1.Caption
```

And just like setting properties, if you include only the property name, REAL Studio assumes you are referring to a property of the window that contains the control that is executing the code. For example, if you have a `PushButton` called `PushButton1` and you want it to assign the window title to the variable `x` when it is clicked, you would use this syntax:

```
x=Title
```

Getting and Setting Values in Variables

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

You can store values in variables and get values from variables in the same way you do with properties. To get a value from a variable, it must be on the right side of the assignment operator (`=`). Suppose you want to set the caption of a `PushButton` to the value in a variable called “ButtonTitle”.

The first thing you do is create the variable and give it a data type. Like the built-in properties of objects, variables have data types. Before you can use a variable, its data type must be made known using the `Dim` statement. `Dim` is short for Dimension, which means to make space for the variable in the computer’s memory. You use the `Dim` statement to declare the name and data type of the new variable. For that reason, it is often called a *declaration statement*.

In this example, you’d use the statement:

```
Dim ButtonTitle as String
```

This creates the variable “ButtonTitle” in memory and tells REAL Studio that it can only store a string value. It can store one string, not several distinct strings.

After you’ve created it, you can give it a value and assign its value to another variable or a property. For example, you can write:

```
ButtonTitle="Save"
```

After it has a value, you can then assign its value to a property of an existing object. That is, you can assign it to a property that is also of type String.

The example below accomplishes that:

```
PushButton1.Caption=ButtonTitle
```

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

```
Buttontitle=PushButton1.Caption
```

In the example below, the variable `i` is dimensioned (or “dimmed”) as an integer:

```
Dim i as Integer
```

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

```
Dim i,j,k as Integer
```

If you want to declare variables of different types, you can also declare them in one Dim statement. For example, the following Dim statement is valid:

```
Dim Name,Address as String, ShoeSize as Single
```

When you create a variable with the Dim statement you can also assign it a value. You do so by following its data type with an equals sign and the value. For example, the following are valid statements:

```
Dim i as Integer =1  
Dim Name as String = "Igor",Address as String = "15 Rue de Vallee"
```

You can also mix variables with and without initial values, as in:

```
Dim a as Integer, b as Integer = 15
```

In this case, the variable `a` is declared as an integer but is not assigned a value. If you examine the value of `a`, it will be zero. The variable `b`, on the other hand, gets the value of 15.

Notice also that the following is valid:

```
Dim a,b,c as Integer = 15
```

This statement declares three integer variables and assigns all of them the value of 15. Although this is valid, you should be careful about assigning values to more than one variable in a single Dim statement because you could easily lose track of some of the variables.

When you assign initial values in a Dim statement you can either use literal values (as shown in the previous examples), constants, or enumerations. You will learn about constants in the section “Constants” on page 236 and enumerations in the section “Adding an Enumeration to a Module” on page 397. A constant is like a variable except that it is assigned its value when it is created and retains its value for its life. An Enumeration holds a list of constant values.

You can use an Enumeration or a constant using the same syntax as you use for a literal. For example, if you define the global constant “InvalidIndex” in a module as the value -1, you can use it in a Dim statement like this:

```
Dim b as Integer = InvalidIndex
```

Enumerations work the same way. Suppose you add a global Enumeration named “SecurityLevel” in a module with values None, Minimum, Maximum, and Forced. You can then use any of the enums in a declaration statement such as:

```
Dim testEnum as Integer = SecurityLevel.Maximum
```

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

```
Dim x,z as Integer
Dim y as String
x=1
y="1.75" //this is the string "1.75" not the number 1.75
z=x+y //error here
```

In the example above x is a number but y is typed as a String. An error is generated because you can’t add different data types together. When you try to compile a project that contains this code, REAL Studio will stop the compilation process and report that it has found a Type Mismatch Error. It will show you the line of code that caused the error. You cannot compile the project until you fix such errors.

The Scope of Variables

A variable that has been declared via the Dim statement exists in memory as long as the particular method is running. It can be accessed only within the method in which it was declared. When the method is finished, the variable is no longer accessible and the memory that was used to store the value becomes available for other uses. This means that another method in the application cannot access the value of the variable.

For that reason, variables that are declared with the Dim statement in any method are considered *local variables*. They exist only locally, inside the “neighborhood” of the method in which they are declared. That is, the *scope* or access scope of the variable is local.

A Dim statement can be placed anywhere in a method, as long as it precedes the first usage of the variables that it dimensions. If you place a Dim statement inside an If statement, its scope is limited to that If statement, not the entire method. For example, you can write:

```
If x > 5 then
    Dim y as Integer
    y = 10
end if
//y goes out of scope here; the variable name y can now be reused.
//It is redeclared as a string in the following If statement
If x < 5 then
    Dim y as String
    y = "hello"
End if
```

If you need a variable whose scope is greater than local, you need to create a *property* of an object or a property in a module. Its scope should match the size of the “neighborhood” that needs to read and/or write the value of the property. For example, if the variable needs to be available to some of the controls in a window, consider making it a property of the window. A property that you create can have one of four possible levels of scope:

- **Global:** The property is accessible throughout the application. Global properties can be created only in modules. A global property can be accessed by its name only anywhere in the application. For more information, see the section “Scope of a Module’s Items” on page 370.
- **Public:** The property is accessible throughout the application. Within the object that owns the property, it is accessible by name only; outside the object, with the syntax *objectName.propertyName*. For example, a Public property of a window named Window1 would be referred to outside the window as Window1.*PropertyName*, where *PropertyName* is its name. Inside Window1, it can be referred to by name only: *PropertyName*.
- **Protected:** The property is accessible only within the object that owns it, by name only. If you try to access the property outside the object that owns it, REAL Studio will display an informative error message.
- **Private:** The property is accessible only within the object that owns it. Other objects based on the owning object cannot access the property (This is not true of Protected properties). If you try to access the property outside the object that owns it, REAL Studio will display an informative error message.

Generally, you will want to choose a scope that is as narrow as possible. This eliminates the possibility that some code outside the object that owns the property might inadvertently read or set the value of the property. Making all properties

global can create bugs that are hard to pin down. For more information about scope, see the section “The Scope of a Property” on page 315.

Declaring Objects

You already know about the data types Integer, Currency, String, Boolean, Single, Double, and Color. But variables can also be declared as specific object types. For example, REAL Studio has an object called Date that you use to store dates and times.

You might think that REAL Studio should be storing a date in an ordinary variable, but it actually makes sense to make it an object that has several properties. Technically, “Date” is a REAL Studio class. You will learn more about classes in Chapter 10.

When you want to use a Date variable, you start with the built-in class as a template for your variable. A template comes with places to store properties of the object and, sometimes, its own methods that perform operations related to the class of objects.

You then create an instance of the class for your use. The instance is the variable that you refer to in your code, not the class itself.

For example, the Date class has separate properties for the Year, Month, and Day, as well as Minute and Second. It has additional properties that format the date as a ShortDate, AbbreviatedDate, or a LongDate.

When you create a new variable based on a class, you start with the same syntax as you use for creating other types of variables. For example, the statement:

```
Dim BirthDate as Date
```

creates a new Date object. However, this statement does not create the instance of the Date template for your use. For example, if you tried to set a property of the variable, BirthDate, to a value, REAL Studio wouldn’t allow you to do it. For example, Year is an Integer property of a Date object, but you can’t read or set it yet. For example:

```
Dim BirthDate as Date  
BirthDate.Year=1996
```

doesn’t work.

Before you can access or set any of the Date variable’s properties, you need to use the New operator to create an instance of the Date object. In programming lingo, creating an instance of an object is called *instantiating* the object.

The following two lines creates and instantiates a Date object:

```
Dim BirthDate as Date  
BirthDate=New Date
```

Now, you are ready to access or set any of the Date variable's properties. For example, you can use assignment statements to establish the value of the date.

```
Dim BirthDate as Date
BirthDate=New Date
BirthDate.Year=1996
BirthDate.Month=6
BirthDate.Day=23
```

If you were to access the ShortDate property of the BirthDate variable, it would now be: "6/23/96" (using the US format for ShortDate).

There is a shortcut syntax for declaring and instantiating objects. You can place the New operator in the Dim statement itself. Using this shortcut, you can rewrite this example as:

```
Dim BirthDate as New Date
BirthDate.Year=1996
BirthDate.Month=6
BirthDate.Day=23
```

This can also be done in one line of code using the Date class's Set method. It takes the Year, Month, and Day as integer parameters. That is:

```
BirthDate.Set(1996,6,23)
```

is equivalent to the three separate assignment statements in the previous example.

An even more concise way of doing this is to do the assignment in the declaration statement. That is, you can write:

```
Dim BirthDate as New Date(1996,6,23)
```

This statement illustrates one of the Date class's *constructors*. A constructor allows you to instantiate the class and assign its value in one line of code. This constructor takes three Integer parameters: Year, Month, and Day. For more information on constructors, see the section "Constructors and Destructors" on page 569.

If you omit the instantiation step, the REAL Studio compiler will issue an error message when you first try to read or set one of the object's properties or run one of its methods.

If you use the optional New operator in a Dim statement, then you cannot declare more than one variable in that statement. For example, if you wrote:

```
Dim n,m as New Date
```

the REAL Studio compiler would generate an error. Instead, you can use one Dim statement to declare the variables and New statements to instantiate them separately.

```
Dim n,m as Date
n=New Date
m=New Date
```

Your other option is to declare and instantiate the two objects separately:

```
Dim n as New Date
Dim m as New Date
```

Using Arrays

An array is a special type of variable that contains several values of the same data type. Each variable in the array is called an *element* of the array. To refer to a particular element, you use an *index* number.

The Dim statement also lets you create and type arrays. When you declare an array, you specify the number of elements it has. Later on, you can change the number of elements.

Creating Arrays

You declare an array by specifying the index of the last element of the array. The index that you specify in the Dim statement is actually one less than the number of elements in the array because REAL Studio arrays always have an element zero. Such an array is sometimes referred to as a *zero-based array*. If the first element of an array is element 1, then it would be called a one-based array.

Since REAL Studio arrays are zero-based, the statement:

```
Dim aNames (10) as String
```

creates a string array with eleven elements.

The statement

```
Dim aNames (0) as String
```

creates a string array with one element, element zero. You can read and write to this element just as with any element with a positive index.

In other words, you declare a variable as an array simply by adding the index of the last element to the Dim statement. The index of the last element must be either a number or a REAL Studio constant. REAL Studio does not accept variables in this syntax.

If you don't know the size of the array you need at the time you declare it, you can declare it as a null array, i.e., an array with no elements. You do this by using an

index of -1 in the Dim statement or leave empty parentheses. This means “an array of no elements.” For example, the statements:

```
Dim aNames (-1) as String
```

and

```
Dim aNames () as String
```

creates the array aNames with no elements. Later on, you can resize the array using the ReDim statement or “grow” it element-by-element using the Append or Array methods.

You can create multi-dimensional arrays in REAL Studio. For example, a spreadsheet layout can be thought of as a two-dimensional array, rows by columns.

Each dimension is referred to by its own index. For example, the elements of an array with two dimensions are referred to by one index for the rows and the other for the columns. The first element in the upper-left corner is element 0,0.

You create a multi-dimensional array by specifying an index for each dimension. For example, the statement:

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

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

You can use constants in Dim statements to set the size of an array. For example, the following declaration refers to a global constant, nLanguages, that is defined in a module:

```
Dim aControl(10,nLanguages) as String
```

In this example, “nLanguages” is the number of languages supported by the application and is used in numerous places in the application. When it changes, you only need to change its definition in the module.

For information on creating constants, see the section “Adding Constants to Modules” on page 375.

Referring to Array Elements

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

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

places the string “Frank” in array element (1,1)

Getting the Index of the Last Element

The **Ubound** function returns the index of the last element of a one-dimensional array. For example, the expression **Ubound(aNames)** returns this value. The number of elements is one greater than this number, since the array has an element zero.

For multi-dimensional arrays, **Ubound** returns the index of the last element of the dimension you specify, or, if you do not specify a dimension, it returns this value for the first dimension. The first dimension is numbered 1. If you pass a -1, it returns the number of dimensions in the array.

For example, the following example returns 5 in the variable *i* and 3 in the variable *j*.

```
Dim i,j as Integer
Dim aNames (5,3) as String
i=Ubound(aNames)
j=Ubound(aNames,2)
```

You can also get the last element of an array using dot notation syntax. The following code is equivalent.

```
Dim i,j as Integer
Dim aNames (5,3) as String
i=aNames.Ubound
j=aNames.Ubound(2)
```

Initializing Arrays

After you have declared an array, you can assign initial values to the elements with the **Array** function as well as individual assignment statements, such as shown above. The **Array** function takes a list of values and assigns the values to the elements of the array, beginning with element zero. In other words, it provides the same functionality as separate assignment statements for each element of the array.

For example, the following statements initialize the array using separate assignment statements for each array element:

```
Dim aNames(2) as String
//using separate assignment statements
aNames(0)="Fred"
aNames(1)="Ginger"
aNames(2)="Stanley"
```

The following statements use the **Array** function to accomplish the same thing. Note that you don't have to declare the exact size of the array in the **Dim** statement. The **Array** function will add elements to the array as needed.

```
Dim aNames() as String
aNames=Array("Fred", "Ginger", "Stanley")
```

If you declare the array as a fixed size but don't specify as many values as elements, The Array function will start with element zero and use as many elements as are specified.

Array Assignment

If you have two arrays of compatible data types, you can assign one array to the other array. Simply use the assignment statement without the parentheses. Here is a simple example:

```
Dim aNames(2),bNames(2) as String
aNames=Array("Fred","Ginger","Stanley")
bNames=aNames
```

The last statement assigns the values of all three elements of aNames to the first three elements of bNames. If bNames had fewer elements than aNames, then additional elements would first be added to bNames and the assignment of all the elements of aNames to bNames would be completed. For example, the following is valid:

```
Dim aNames(3),bNames(2) as String
aNames(0)="Fred"
aNames(1)="Ginger"
aNames(2)="Tommy"
aNames(3)="Woody"
bNames=aNames
```

After the code runs, the bNames array has a fourth element for storing the value "Woody".

Resizing Arrays Several methods in the language resize arrays.

- **Append:** The Append method adds an element to the array, increasing its size by one. You pass the value you want to add to the array when you call Append. For example, the following statement:

```
aNames.Append "Dave"
```

adds an element to the array aNames and sets the value of this element to the string, "Dave".

Append works on one-dimensional arrays only.

- **Insert:** The Insert method creates an additional element and inserts it in the place you specify. It takes two parameters, the index of the element to be inserted and the value of the new element. For example:

```
aNames.Insert 9,"Hal"
```

After this statement runs, the value of `aNames(9)` would be “Hal”; the old element(9) would be shifted up to element (10) and so forth. The size of the array would be increased by one, as with `Append`.

`Insert` also works only on one-dimensional arrays.

- **Remove:** The `Remove` method deletes the element whose index you specify. For example:

```
aNames.Remove 9
```

removes the element with index 9, decreasing the size of the array by one and shifting the array elements after the removed element down by one. It also works on one-dimensional arrays only.

- **Redim:** The `Redim` statement resizes an existing array. You pass the new values of the array’s indexes but you don’t specify the data type, which is set by the initial `Dim` statement. For example, the statement:

```
Redim aNames (100)
```

resizes the array `aNames` to 101 elements.

`Redim` works on both one- and multi-dimensional arrays. For multi-dimensional arrays, you can only resize the existing dimensions; you cannot add or reduce the number of dimensions themselves.

A difference between `Dim` and `Redim` is that `Dim` accepts only integers or constants, while `Redim` accepts any expression that returns an integer. This includes, for example, a user-written function that returns an integer value. Using this feature, you can dimension your arrays on-the-fly.

If you don’t know the size of the array you need at the time you declare it, you can declare it as a null array, i.e., an array with no elements, and use the `Redim` command to resize it later. If your program needs to load a list of names that the user enters, you can wait to size the array until you know how many names the user has entered. You can write a function to figure out what that number is and use it with `Redim` or use the `Append` method to add the required elements to the array one-by-one.

Converting to and from an Array to Variables

Two functions enable you to take an array and break it up into separate variables and take a single string variable and convert it into an array.

- **Split:** The `Split` function takes a `String` variable and creates a one-dimensional array by dividing the string up into elements. It uses a delimiter—a character or character string that signals the end of one element and the start of the next element—to do this task. By default, the delimiter is a space, but you can specify another delimiter.

The Split function takes the string as its first parameter and the delimiter as the second parameter.

Here is an example that divides up the contents of a string into array elements. It specifies the comma as the delimiter.

```
Dim aNames(-1) as String //resize using the Split method
Dim s as String
s="Juliet,Taylor,Casting"
aNames=Split(s, ",")
```

After this call, the resulting array, aNames, will have three elements:

```
aNames(0)="Juliet"
aNames(1)="Taylor"
aNames(2)="Casting"
```

- **Join:** The Join function takes a one-dimensional array and creates a String variable that contains all the elements in the array, separated by a delimiter character or character string. By default, the delimiter character is a space, but you can specify your own delimiter by passing the delimiter as the second parameter. The delimiter is passed as the second parameter.

For example, if you have an array that contains elements that store a person's first and last names and phone number, the Join function will create one String variable that contains all the information. Here is an example:

```
Dim aNames(2) as String
Dim s as String
aNames(0)="Anthony"
aNames(1)="Aardvark"
aNames(2)=" (406) 737-8946"
s=Join(aNames, ", ") //creates "Anthony,Aardvark,(406) 737-8946"
```

In this example, the call to the Join function specifies the name of the array and the comma as the delimiter. If the call were:

```
s=Join(aNames)
```

the result would be “Anthony Aardvark (406) 737-8946”.

Dictionaries

A *dictionary* is a REAL Studio object that is made up of a list of *key-value* pairs. That is, each value is paired with an identifying key. The interesting feature of Dictionaries is that both the key and the value are variants. This means that a dictionary can store a mixture of data types—and that the key doesn't have to simply be an integer. You can look up a value in a dictionary by specifying either its key or its sequential position in the dictionary. For example, the *Language Reference* entry for Dictionary has an interesting example in which colors are used as keys.

Pairs

The Pair class is similar to the Dictionary. It has two properties: Left and Right. Thus, each Pair instance consists of a *key-value* pair. As it the case with the Dictionary, the values in the pair are variants. You use the “:” operator to assign the Left and Right values when the Pair is declared, e.g.,

```
Dim p as Pair = "Telephone Number" : "(406) 737-8946"
```

This assigns “Telephone Number” to the Left property of the pair and the value of the number to the Right property.

It also can store a linked list of pairs when it is passed a list of items, such as:

```
Dim p as Pair = 1 : 2 : 3 : 4 : 5
```

The first pair consists of the pair “1” (Left property) and the second pair object (Right property); the next pair consists of the “2” and the third pair, and so forth.

Mathematical Operators

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

Operation Performed	Operator	Example
Addition	+	2 + 3 = 5
Subtraction	-	3 - 2 = 1
Multiplication	*	3 * 2 = 6
Floating Point Division	/	6 / 4 = 1.5
Integer Division	\	6 \ 4 = 1
Modulo	Mod	6 Mod 3 = 0 6 Mod 4 = 2
Exponentiation	^	2^3 = 8

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

REAL Studio supports standard mathematical precedence. This means that equations surrounded by parentheses are handled first. REAL Studio will begin with the set of parentheses that is embedded inside the most other sets of parentheses. Next, exponentiation is performed, followed by any multiplication or division from left to right. Finally any addition or subtraction is performed. In the example below, the three expressions return different results because of the placement of parentheses:

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

Operator Precedence

Operator Precedence determines the order in which operators execute when there is more than one operator in an expression. For example, the expression:

5+2/3

contains both the division and addition operators. It matters whether the division or the addition takes place first. You can force a particular precedence via parentheses, as described above. By default, division has precedence over addition, so REAL Studio would evaluate this expression as 5 plus 2/3. You can force it to do the addition first by writing:

(5+2)/3

The order from highest precedence to lowest is shown in the following table:

Operator	Description
.	Dot operator
AddressOf	Delegate creation operator
IsA	Type checking operator
^	Exponentiation operator
-	Negation or the unary minus operator
Not	Logical not operator
* /\ Mod	Multiplication and division arithmetic operators
- +	Subtraction and addition arithmetic operators
= > < >= <= <> Is	Comparison operators
And	Bitwise and logical operator
Or Xor	Bitwise and logical operator
:	Pair creation operator

If there is more than one operator in an expression, the precedence goes from left to right. For instance, multiplication is higher than floating-point division, which is higher than integer division, which is higher than modulus.

All operators are left-associative except for pairs (:) and exponentiation (^).

Left association means that (foo or bar or baz) will evaluate like ((foo or bar) or baz) instead of (foo or (bar or baz)). Conversely, right association means that (foo bar : baz) will evaluate like (foo : (bar : baz)) instead of ((foo : bar) : baz).

Constants

A constant is like a variable but it holds a fixed value for its entire “life.” When you create a constant, you give it its value. You can read the constant’s value in your code, but you cannot use an assignment statement to change the value of a constant. You cannot create array constants.

You can create constants in REAL Studio for windows, modules, and objects based on classes that are added to the Project Editor (You’ll learn more about windows,

modules, and classes later on in this manual.) You can also create a local constant inside any method you write.

Each constant has a *Scope*. The Scope determines which parts of your application can “see” the constant and read its value. When you create a constant in any method, its scope is automatically Local to that method: Only the code within that method can read the value of the constant. A Local constant is assigned its value within a method and can be referred to anywhere within that method. If another method has a constant with the same name, it is, in effect, a completely different constant.

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

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

You do not have to indicate the data type of a constant. For example, the following statement is valid:

```
Const BackGroundColor=&cFF0000
```

REAL Studio will “figure out” that BackGroundColor is a Color. You can then assign the constant BackGroundColor to any property or variable that accepts a Color data type.

The following code is acceptable:

```
Const Accept="OK"  
BevelButton1.caption=Accept
```

When the application runs, the PushButton’s caption reads “OK”.

The Scope of Constants

The Const statement has local scope, just like the Dim statement (see the section “The Scope of Variables” on page 225). When you declare a constant inside a method (as has just been described), it exists only as long as the method is running and can be accessed only inside that method. Other methods can’t read its value. It means that the constant is local to the method in which it is declared.

You can also declare constants for windows, modules, or classes. You will learn about managing these objects later in the *Users Guide*.

Attributes

Attributes are compile-time properties. They can be added to both Project and Code Editor items. An attribute consists of its Name and its Value. The Name property is mandatory and the Value property is optional.

Attributes are created in the IDE with the Attributes Editor. Items that can have attributes include constants, methods, properties, classes, modules, windows, containercontrols, class interfaces, and toolbars.

For classes, a subclass inherits the attributes of its parent class and attribute values can be overridden if redefined by the subclass.

The names “Deprecated” and “Hidden” are reserved for internal use and should not be used as an attribute’s Name.

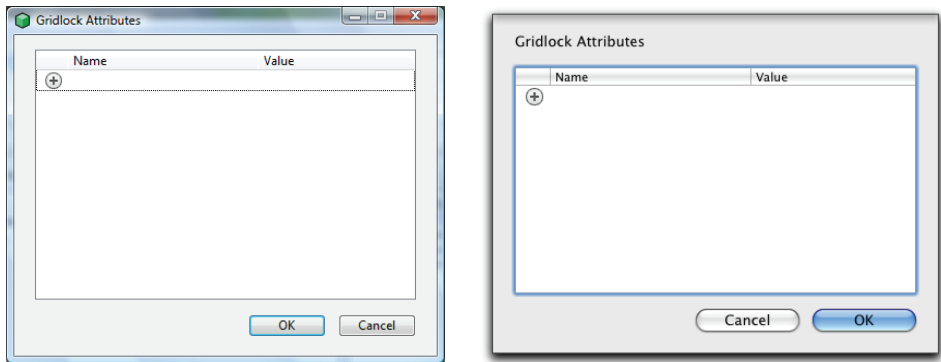
Each Code or Project Editor item has a contextual menu, from which you can add, modify, or delete the item’s attributes.

To add an attribute, do this:

- 1 **Right+Click (Command-Click on Macintosh) on a Project Editor or Code Editor item and choose Attributes... from the contextual menu.**

The Attributes Editor appears.

Figure 217. The Attributes Editor



- 2 **To add an attribute, click on the Plus sign.**

An entry area for the attribute appears. The first column is for the (mandatory) attribute name; the second column is for the optional attribute value.

- 3 **Enter the name and, if desired, its value.**
- 4 **Repeat this process for each additional attribute.**
- 5 **Click OK to put away the Attributes Editor.**

To modify or delete an attribute, do this:

- 1 **To modify an existing attribute, open the Attributes Editor for the item and click twice in its name field to get an insertion point.**

The Attributes Editor appears, with the item’s attributes listed.

- 2 **Edit the Name or Value as desired.**

To delete an existing attribute, do this:

- 1 **Open the Attributes Editor for the item.**
The Attributes Editor appears, with the item’s attributes listed.
- 2 **Click on an attribute’s minus sign.**
- 3 **Click OK to put away the Attributes Editor.**

Accessing an Attribute

Attributes are accessed at runtime via the Introspection system. The `AttributeInfo` class is designed to hold the *Name-Value* attribute pairs of a particular item. See the entry for the `AttributeInfo` class in the *Language Reference* for an example of how to retrieve an object's attributes.

Reserved Words

The following words should not be used as variable or object names because they are used as part of the REAL Studio language itself:

Reserved Word	Reserved Word	Reserved Word
#bad	Exception	Private
#else	Exit	Property
#Elseif	Extends	Protected
#endif	False	Public
#if	Finally	Raise
#pragma	For	RaiseEvent
#tag	Function	Rect
AddressOf	Global	Redim
Aggregates	GoTo	Rem
And	Handles	Return
Array	If	Select
As	Implements	Self
Assigns	In	Shared
Attributes	Inherits	Soft
Break	Inline68k	Static
ByRef	Interface	Step
ByVal	Is	Structure
Call	IsA	Sub
Case	Lib	Super
Catch	Loop	Then
Class	Me	To
Const	Mod	True
Continue	Module	Try
Declare	Namespace	Until
Delegate		Wend
Dim	New	While
Do	Next	With
DownTo	Nil	_ (as the first character of a variable, method, event definition, or property name.)
Each	Not	
Else	Object	
Elseif	Of	
End	Optional	
Enum	Or	
Event	ParamArray	

Executing Instructions with Methods

A *method* is one or more instructions that are performed to accomplish a specific task. REAL Studio has many built-in methods. For example, the Quit method causes your application to quit. Most REAL Studio classes have built-in methods. For example, the ListBox class has a method called AddRow for adding rows to a ListBox (as the name implies).

You can also create your own custom methods. Just like variables, methods are given names to describe them and the same rules apply: the name can be any length, but must start with a letter and can contain only alphanumeric values (a-z, A-Z, 0-9) or an underscore (_).

The following is an example of a simple method that calculates how many days old a person is in 1998 who was born in 1960:

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

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

Passing Values to Methods

Some methods require one or more pieces of information to perform their function. These pieces of information are called *parameters*. Parameters are passed to a method placing them to the right of the method name. In the following example, the AddRow method of a ListBox called ListBox1 is being called. AddRow adds one row to the end of the ListBox and writes text in the new row. In order to do this, you need to pass the text to AddRow as a parameter. Here is an example:

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

You can also put the parameter in parentheses. The following is equivalent:

```
ListBox1.AddRow ("January")
```

When you pass parameters, you must pass values of the correct data type. AddRow, for example, requires a String. If you needed to add a number to the ListBox, you must pass it as a String. For example, if you need to add the number 12.7 to the ListBox, you can write:

```
ListBox1.AddRow "12.7"
```

Or, you can use the built-in Str function which converts a number passed to it into a string. The following also works:

```
ListBox1.AddRow Str(12.7)
```

The Str function takes one parameter, a value, and returns a string. So, this expression takes the number you want to display, converts it to a string, and then passes the string to the AddRow method.

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

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

As is the case with one parameter, you can place the list of parameters in parentheses:

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

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

```
Dim Month as String  
Month="January"  
ListBox1.InsertRow 3, Month
```

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

Passing Arrays as Parameters

An array can be passed as a parameter in a call to a method or function. You can pass both one and multi-dimensional arrays. To specify that a parameter is a one-dimensional array, put empty parentheses after its name in the declaration. For example,

```
Names () as String
```

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

When you pass an array to the method or function, omit the parentheses. For example, use

```
PrintLabels (Names)
```

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

You can pass multi-dimensional arrays without specifying the number of elements in each dimension, but you need to indicate the number of dimensions. Do this by placing one fewer commas in the parentheses than dimensions. For example, if `aNames` were a two-dimensional String array, you would declare the array in the following manner:

```
aNames(,) as String
```

When you pass a multi-dimensional array to a method or function, you can include the parentheses but not any commas. For example,

```
PrintLabels aNames()
```

Returning Values from Methods

Some methods return values. This means that a value is passed back from the method to the line of code that called the method. For example, REAL Studio's built-in method, `Ticks`, returns the number of ticks (60th's of a second) that have passed since you turned on your computer. You can assign the value returned by a method the same way you assign a value. In the example below, the value returned by `Ticks` is assigned to the variable `x`:

```
x=Ticks
```

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

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

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

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

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

Passing Parameters by Value and by Reference

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

Parameters passed by value are treated as local variables inside the method—just like variables that are created using the Dim statement. This means that you can modify the values of the parameters themselves rather than first assigning the parameter to a local variable. For example, if you pass a value in the parameter “x”, you can increment or decrement the value of x rather than assigning the value of x to a local variable that is created using Dim.

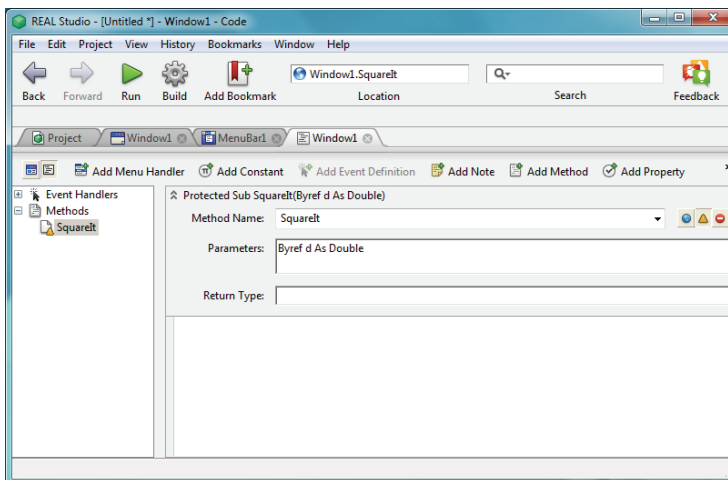
For example, the following method is valid:

```
Sub SquareIt(a As Integer)
    a=a*a
    MsgBox str(a)
```

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

You use the keywords ByVal or ByRef to specify the type of parameter passing (ByVal is assumed and does not need to be used). To pass a parameter by reference, use the ByRef keyword in the method declaration when you declare a parameter that is to be passed by reference. For example, Figure 218 shows a parameter that is declared ByRef. The method can replace the parameter with the computed value.

Figure 218. Declaring a parameter ByRef.



Suppose the code that calls this method is:

```
Dim a as Integer
a=3
SquareIt a
TextField1.text=str(a)
```

and the method is simply:

```
a=a*a
```

The TextField will display the number 9. If you don't specify ByRef, the method can change the value passed to it, but it cannot return the changed value.

Arrays are passed by reference. The method can change all or some of the elements of the array.

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

Using the Meta-Constant

The built-in meta-constant `CurrentExecutingMethodName` is available in all methods and events. It automatically contains the fully-qualified name of the method or event. It is the same as if the user had declared:

```
Const CurrentExecutingMethodName="Methodname"
```

where *MethodName* is the fully-qualified name of the method.

For example, if you create a method, `MyNewMethod`, belonging to `Window1` that has the code:

```
MsgBox CurrentExecutingMethodName
```

A call to this method will display the name “`Window1.MyNewMethod`”.

The `CurrentExecutingMethodName` constant is also available in Events. For example, if you call it in the `MouseEnter` event of `Rectangle1` in `Window1`, then it contains “`Window1.Rectangle1.MouseEnter`”.

Documenting Your Code

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

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

By default, comments in your code appear in red. You can set a different color for comments using REAL Studio options (Preferences on Macintosh). For more information, see the section “Configuring the Code Editor” on page 275.

If you have several consecutive lines that you want to convert to comments, select the lines and click the Comment button in the Code Editor toolbar. You can also use the Edit ► Comment menu item (Ctrl+' on Windows and Linux or ⌘-' on Macintosh). If the line of code that contains the insertion point is a comment, then the Comment button changes to Uncomment and the corresponding menu item changes to Edit ► Uncomment. Uncomment changes the lines back to executable code.

For that reason, its best to use “//” for lines of documentation (lines that cannot be executed) and reserve the use of the up-down quotes for commenting out lines of code that you may want to convert back to executable code.

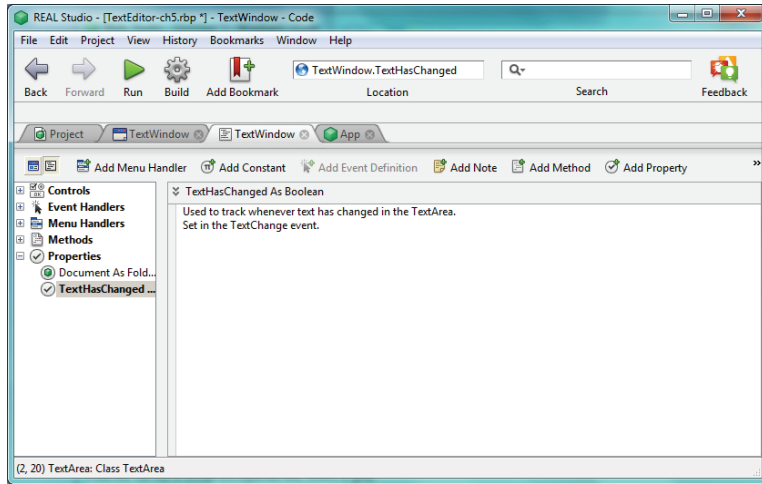
Documenting Properties

When you create a property of a window, class, or module, you can use the Code Editor to document the property. In the declaration screen, REAL Studio states the property’s declaration in the Code Editor and you can add any text underneath it.

For information about properties, see the section “Adding Properties to Windows” on page 314.

Any text you enter in the code editing area is automatically non-executable, even if it is REAL Studio code.

Figure 219. Documenting a window property.

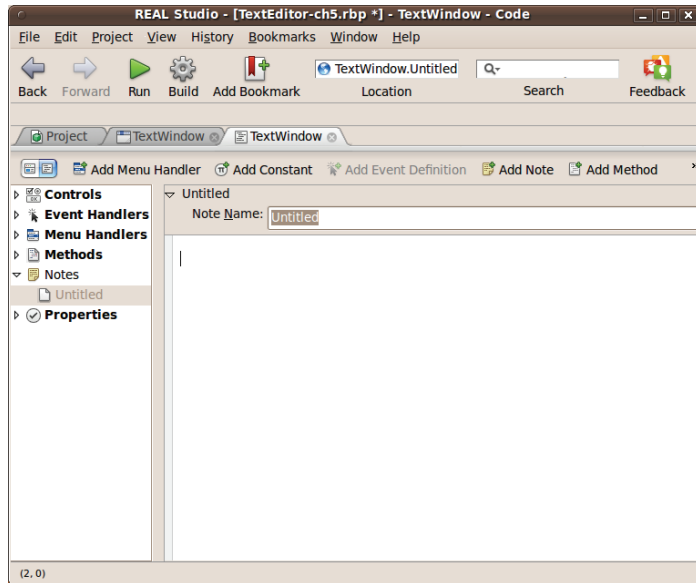


Using Notes

You can also document your code (or application) at a more global level using *Notes*. Each Code Editor has an item called Notes in its browser area that you can use to store comments about your application (or anything else, for that matter). Like comments in your code, Notes are not compiled and are not included in a built application. Although they appear in the Code Editor, you shouldn't try to reference them in your code.

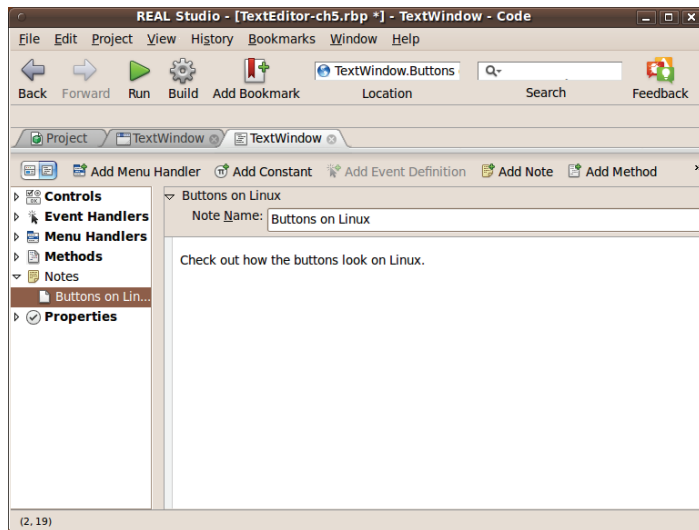
You add a note to a Code Editor just as you would add an object that is compilable. To add a note to a Code Editor, click the Add Note button or choose Project ► Add ► Note. The Code Editing area changes to an entry area for the note. Enter your note in the entry area.

Figure 220. The Add Note screen.



After you click OK, the name of the note appears in the Notes group and the editing area of the Code Editor opens to that note, enabling you to enter your comments.

Figure 221. The text of a Note.



Comparison Operators

There are many times when you need to compare two values to determine whether or not a particular condition exists. When making a comparison, what you are really

doing is making a statement that will either be True or False. For example, the statement “My dog is a cat” evaluates to False. However, the statement “My dog weighs more than my cat” may evaluate to True. The table below shows examples of the comparison operators that are available:

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

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

In addition, REAL Studio includes a floating point equals operator. It allows you to determine whether two floating point numbers are close enough to be considered equal. Use it to account for the imprecision of operations such as floating point division. The floating point equals operator is the Equals keyword. There is no symbol for it.

The syntax is:

```
result=expression.Equals(NumValue,x)
```

Expression is the floating point value to be compared, *NumValue* is the value it is being compared to, and *x* is the number of units in the last position that denotes the acceptable range. *Result* is either True or False, depending on the result of the comparison.

For example, if you are comparing 10000 to a value and specify *x*=1, then the acceptable values are 10000.0000000000002, 10000.0 and 9999.999999999998.

Logical Comparisons

You can test more than one comparison at a time using the And, Or, and Not operators. When passed boolean values, these operators determine whether the expression is true or false.

And Operator

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

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


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

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

Xor Operator Use this operator when you need to know whether two boolean expressions are not equal. In the example below, if the variable x is 5 then the expression evaluates to True.

```
x>1 Xor x<5
```

Not Operator Use the Not operator to reverse the value of a boolean variable. For example:

```
Not (x < 0)
```

tests whether x is equal to or greater than 0.

The following table summarizes the results of logical comparisons.

Expression1	Expression2	Xor	Or	And
True	True	False	True	True
True	False	True	True	False
False	True	True	True	False
False	False	False	False	False

Bitwise Comparisons

You can also compare the individual bits of two integers to determine whether or not they are equal. This is done by re-expressing each integer as a binary number. Then you compare the bits in the pair of integers place-by-place. You get a new integer which is the result of each comparison.

You can do this with the And, or, and Xor operators. They are considered *overloaded*. This means that they can accept either booleans (as was shown above) or integers. If they are passed boolean expressions, they “know” that you want the logical comparisons that are described in the previous section. If they are passed integers, they “know” that you want to compare the bits that make up the two integers. In this case, they each return an integer made up of the results of all the bit comparisons. The following table summarizes the results for the comparison of each

bit in the passed integers.

Bit in Integer1	Bit in Integer2	And	Or	Xor
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

For example, this expression does a bitwise Xor on the two integers passed.

```
i=5 Xor 3 //returns 6
```

The Not operator is also overloaded. If you pass an integer to Not, it simply reverses each bit value. The corresponding bit in the result is set according to the following table.

Bit in Expression	Bit in Result
0	1
1	0

You can also do these bit comparisons using methods of the Bitwise class. Please see the entry for the Bitwise class in the *Language Reference* for more information.

Executing Instructions Repeatedly with Loops

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

```
Beep
Beep
Beep
```

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

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

REAL Studio offers the following types of loop structures:

- **While...Wend:** The loop runs until the condition specified in the While statement is satisfied.
- **Do...Loop:** The loop runs until the condition specified in the Loop statement is satisfied.
- **For...Next:** The loop runs a specified number of times given in the For statement. A local counter variable controls the execution of the loop.

- **For...Each:** The loop runs repeatedly for each element in an array.

You can declare local variables inside a loop structure. When you define a local variable inside a loop structure, its scope is local to the structure itself, not the entire method. It goes out of scope after the condition of the loop is satisfied. If you need to use the variable after the loop executes, define it outside the loop, so that it is local to the method.

While...Wend

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

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

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

Do...Loop

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

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

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

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

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

```
Do  
  i=i+1  
Loop
```

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

Endless Loops Make sure that the code inside your While and Do loops eventually causes the condition to be satisfied. Otherwise, you will end up with an endless loop that runs forever. Should you do this accidentally, you can switch back to the IDE by clicking the Stop button in the Run screen or closing the Run screen in the IDE to stop the loop. If this doesn't work, you will need to force REAL Studio to quit by pressing Ctrl+Alt+Del or ⌘-Option-Escape (Macintosh).

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

- If the user should wait until the loop is finished before doing anything else (e.g., if a user action might invalidate the results of the loop), you can signal that a lengthy operation is in progress by changing the mouse cursor to a watch cursor until the loop ends. Keep in mind that, with this solution, the user can't even stop the loop prematurely because the loop has 'taken over' the application. See the section on the `MouseCursor` class in the *Language Reference* for more information.
- If the user is permitted to do other tasks while the loop is running, you should place the code for the loop in a separate thread. A thread runs concurrently with the main application (the one that handles user input), but as a background task, allowing user operations in the foreground. Please see the entry in the *Language Reference* for the `Thread` class for information on placing code in threads.

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

variable, a starting value for that variable and an ending value. The basic construction of a For loop is:

```
Dim Counter As Integer
For counter=0 to 100
  // [your code goes here]
Next
```

Notice that the Dim statement declares the counter as an Integer. This is the most common way to define the counter, but it is not required. You can also declare the counter variable as a Single or Double.

In this example, the counter variable was declared in the usual way, via the Dim statement. Since counter variables are rarely needed outside the For loop, REAL Studio also allows you to declare the counter variable right inside the For statement. In other words, you can redo this example in the following way:

```
For Counter As Integer=0 to 100
  // [your code goes here]
Next
```

Notice that the Dim statement has been removed from the example. If you declare the counter variable this way, you can use it only within the For loop. It disappears after the For loop is finished. If you need to read or change the value of the counter variable outside the For loop, you should use the Dim statement instead.

In this case, the starting value and the ending value are specified as numbers. You can also use variables, as shown in this example:

```
Dim StartingValue, EndingValue As Integer
StartingValue=0
EndingValue=100
For counter As Integer=StartingValue to EndingValue
  //your code goes here
Next
```

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

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

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



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

Note: The letter “*i*” is commonly used as the loop counter for historical reasons. In FORTRAN, the letters *I* to *N* are typed as integers by default. Therefore, FORTRAN programmers began the practice of using those letters as counters, and in the order they appear in the alphabet. That is, if a FORTRAN programmer needed to nest one loop in another (as is described on page 257), he would use *j* as the counter for the inner loop. This convention made it easy for FORTRAN programmers to follow the logic of code that processed multi-dimensional arrays.

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

```
For i as integer=5 to 100 Step 5
  ListBox1.AddRow Str(i)
Next
```

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

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

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

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

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

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

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

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

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

```
Dim row, column, count As Integer
For row=0 to ListBox1.ListCount-1
  For column=0 to ListBox1.ColumnCount-1
    If ListBox1.Cell(row,column)="hello" then count=count+1
  End if
Next
Next
MsgBox Str(count)
```

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

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

The For...Each statement

Another situation in which you want to loop through a group of values is array processing. Rather than looping through a set of statements for each value of a counter,

the For...Each statement processes each element of an array that is passed to it. Here is a simple example:

```
Function SumStuff(values() as Double) as Double
    Dim sum as Double
    For Each element as Double In values
        sum=sum+element
    Next
    Return sum
```

“Values” is a one-dimensional array of numbers that is passed to the function SumStuff. In the For Each statement, role of the counter variable is taken by the variable “element”, which refers to an element in the array. The For Each loop executes the statements between the For Each statement and the Next statement for each element in the array.

Since the array doesn’t necessarily have to be numbers, this statement enables you to process a group of objects of any type. They could be pictures, colors, documents, sets of database records, and so forth.

The For Each statement identifies the counter variable (element) and the array to be processed. To call the function, pass an array of doubles in a statement such as:

```
s=SumStuff(MyNumbers)
```

where s is declared as a Double and MyNumbers is an array of Doubles.

As is the case for the For...Next loop, you can declare the data type of the counter inside the For Each statement rather than in a separate Dim statement. For example the previous example could be rewritten like this:

```
Function SumStuff(values() as Double) as Double
    Dim sum as Double
    For Each element as Double In values
        sum=sum+element
    Next
    Return sum
```

Adding Loops to your Code

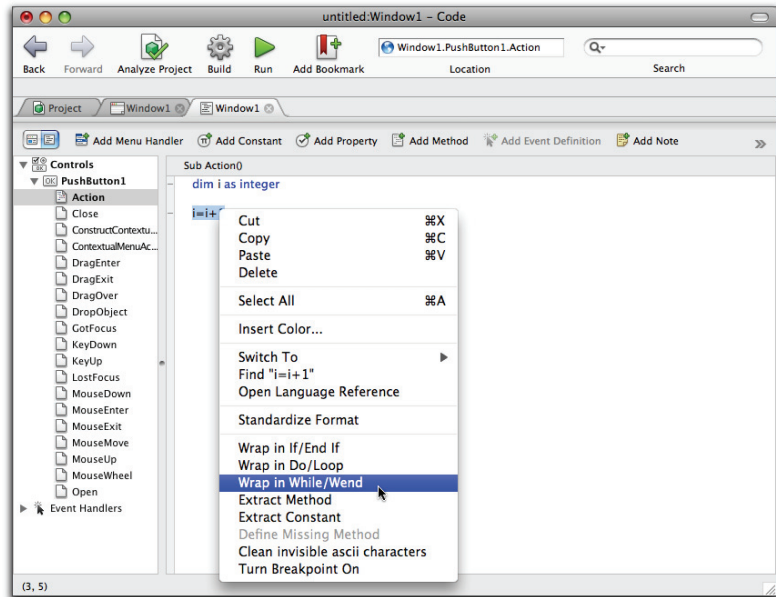
The Code Editor’s contextual menu offers an especially convenient way of adding loops to your code. The last three items in the contextual menu wrap the selected lines of code inside a type of loop. To use these menu items, simply write the code that goes inside the loop, select the lines, and then choose the type of loop from the contextual menu. Your choices are If...End If, Do...Loop, and While...Wend.

To take the example of the While...Wend loop shown on page 253, suppose the user has typed only the lines:

```
Dim i As Integer  
i=i+1
```

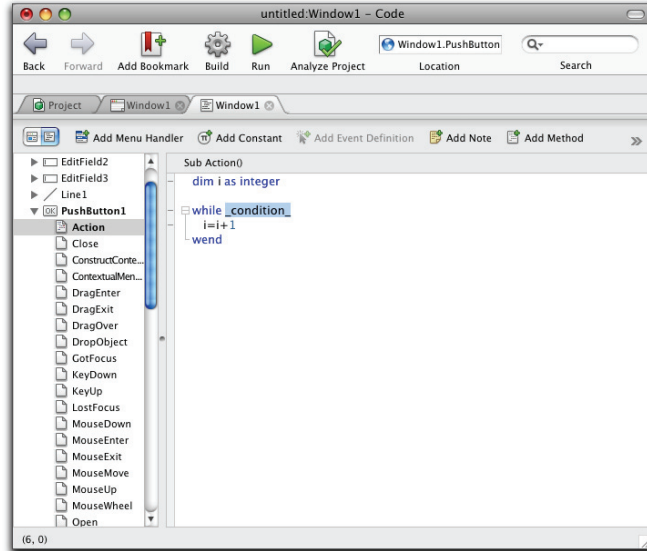
and selected the two lines that go inside the loop before choosing the Wrap in While...Wend contextual menu item.

Figure 222. Wrapping lines of code inside a loop.



The result is shown in Figure 223:

Figure 223. The Code Editor after wrapping lines inside a loop.



REAL Studio can wrap the selected lines in the loop but it doesn't know what condition should terminate the loop. Therefore, it has inserted the placeholder text `_condition_` in the While statement. It has already been selected, so all you need to do is replace it with the condition you want. In this example, it's `i < 10`.

This process is the same for the Do...Loop and If...End If loops, described in the next section.

Making Decisions with Branching

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

If...Then...End If

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

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

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

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

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

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

```
If CheckBox1.value Then
  //[Your code goes here]
End If
```

You can declare local variables using the `Dim` statement inside an `If` statement. However, such variables go out of scope after the `End If` statement. This is similar to the feature for the loop statements. For example:

```
If error=-123 Then
  Dim a As String
  a="Whoops! An error occurred."
End if
MsgBox a    //out of scope
```

If you need the variable after the `End If` statement, you should declare it local to the entire method, not the `If...End If` statement.

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

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

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

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

If...End If

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

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

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

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

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

If...Then...Else An If statement can be written on one line, provided the code that follows the Then and (optionally) the Else statements can all be written on one line. When you use this syntax, you omit the End If statement. For example, the following statements are valid:

```
If error=123 Then MsgBox "An error occurred."
If error=123 Then MsgBox "An error occurred." Else MsgBox "Success"
If error=103 Then Break //breaks into the debugger
```

#If...#Endif The #If...#Endif statement is designed to handle a very special case of conditional compilation. You use #If...#Endif when you need to compile different versions of code for different platforms. That's its only use. The boolean condition that the #If statement uses will accept only special boolean constants that can determine the

type of code that is being compiled. Here is the list of boolean constants that the #If statement accepts.

Boolean Constant	Description
DebugBuild	The application is running within the REAL Studio application, i.e., from clicking the Run button in the IDE Main toolbar.
RBVersion	Returns the version of REAL Studio that is being compiled. You can use this in an expression that evaluates to True or False to determine which version of REAL Studio is compiling the application.
TargetBigEndian	The compiled application is running on a machine that uses the Big Endian byte order. Macintosh PPC uses the Big Endian byte order.
TargetCarbon	The compiled application is currently running Carbon/Mac OS X code.
TargetHasGUI	The application has a graphical user interface, i.e., it is not a ConsoleApplication or a ServiceApplication.
TargetLinux	The compiled application is running Linux code.
TargetLittleEndian	The compiled application is running on a machine that uses the LittleEndian byte order. PCs and Intel Macintoshes use the Little Endian byte order.
TargetMachO	The compiled application is running on Mac OS X, running object code in the format for the Mach kernel. This code runs only on Mac OS X.
TargetMacOS	The compiled application is currently running Macintosh code. As of REAL Studio 2007 Release 4, all Macintosh builds are also TargetMachO.
TargetMacOSClassic	The compiled application is currently running Macintosh code within the "classic" Mac OS. As of REAL Studio 2007 Release 4, REAL Studio no longer supports Mac OS classic. This constant was initially called "TargetPPC."
TargetPowerPC	The compiled application is currently running on a machine that uses PowerPC hardware, regardless of operating system.
TargetWin32	The compiled application is currently running Win32 code. It will run on any version of Windows, from Windows 2000 to Vista.
TargetX86	The compiled application is running on a machine that uses x86 hardware, regardless of operating system.

For example, if you want to use code that manages the Mac OS X dock, you can include it only in your MachO build of the application with a statement such as:

```
#if TargetMachO
    //code goes here
#endif
```

If you need to include different code for different platforms rather than just including code only for one platform, you can use the `#Elseif` keyword in this manner:

```
#If TargetWin32
//Windows specific code here
#Elseif TargetMacOS
//Macintosh code goes here.
#Elseif TargetLinux
//Linux code goes right here.
#Endif
```

You can also include or exclude code based on the version of REAL Studio that is running. You use the `RBVersion` function to get the version of REAL Studio that's running.

```
#If RBVersion >= 2007
//include version 2007 code here
#endif
```

This example only includes the code if the version of REAL Studio that is compiling the code is at least 2007.

Note that the “then” keyword is not needed in the `#If` statement. It is optional.

For more information on conditional compilation, see the entries in the Language Reference for the `#If` statement and the boolean constants and Chapter 15, “Building Stand-Alone Applications” on page 693.

Select...Case

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

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

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

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

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

You can create local variables using the Dim statement inside a Case statement. However, such variables go out of scope at the conclusion of the statement. For example:

```
Select Case dayNumber
Case 2
    Dim day as String
    day="Tuesday"
Else
    MsgBox "It's NOT Tuesday!"
End Select
MsgBox day //day is out of scope
```

The variable “day” should be declared prior to the Select...Case statement so that it is available after the End Select statement executes.

The Select...Case statement works with variables of any data type. It works for strings, integers, singles, doubles, booleans, and colors. For example, you can compare colors, as in the following example:

```
Dim c as Color
c=&cFF0000 //pure red
Select case c
  Case &c00FF00 //green
    MsgBox "Green"
  Case &cFF0000 //red
    MsgBox "Red"
  Case &c0000FF //blue
    MsgBox "Blue"
End select
```

A Case statement can accept more than one value, with different values separated by commas. For example, the following is valid:

```
Dim c as color
c=&cFF0000 //red
Select case c
  Case &c00FF00,&cFF0000 //green, red
    MsgBox "Green or Red"
  Case &cFF0000 //red
    MsgBox "Red"
  Case &c0000FF //blue
    MsgBox "Blue"
End select
```

In the preceding example, the first Case statement is True, so its code executes. Although the color passed to Select...Case is Red, the code for the second case does not execute because it is not the first matching case.

The Select Case statement accepts an “Else” clause. The code in the Else clause executes only if none of the preceding cases matches. The Else clause can be written

as either “Else” or “Case Else”. In the following example, the Case Else clause executes because the color FFFF00 was passed:

```
Dim c as Color
c=&cFFFF00 //pure red
Select case c
  Case &c00FF00 //green
    MsgBox "Green"
  Case &cFF0000 //red
    MsgBox "Red"
  Case &c0000FF //blue
    MsgBox "Blue"
  Case Else
    MsgBox "None of the above"
End select
```

The Case statement can also accept a range of consecutive values that you pass using the “To” keyword. For example:

```
Dim i as Integer = 53
Select case i
  Case 1 to 25
    MsgBox "25 or less"
  Case 26 to 50
    MsgBox "26 to 50"
  Case 51 to 100
    MsgBox "51 to 100"
End Select
```

In this example, the third case, “51 to 100”, is true.

You can combine ranges with nonconsecutive values, by separating them with commas, such as:

```
Case 0, 26 to 50, 75, 100 to 200
```

You can write inequalities with the “Is” keyword and an inequality operator. The syntax is:

```
Is inequalityOperator
```

For example:

```
Dim i as Integer = 10
Select Case i
  Case Is <= 10
    //this case selected
  Case Is > 10
    //this case not selected
End Select
```

You can combine inequalities with values, as in:

```
Dim i as Integer = 75
Select Case i
  Case 0, Is <=10,100
    //case not selected
  Case Is > 10, Is < 99
    //case selected
End Select
```

You can even use functions that return a value of the specified data type in a Case statement. Here is a simple example:

```
Dim i as Integer = 4
Dim a as Integer = 2
Select Case i
  Case Functx(a)
    //case 1
  Case a
    //case 2
Else
  //no match
End Select
```

The function in the first Case statement is:

```
Function Functx(a As Integer) as Integer
Return a*a
```

In this example, the function squares the value passed to it, so the first Case statement matches.

In the case of a simple function like this, you can write the expression in the Case statement itself. That is, the following is an equivalent matching Case statement:

```
Case a*a
```

The Select Case statement can also compare variables of type Object. The following example uses a Select...Case statement to determine which button the user pressed

in a `MessageDialog` box. This example was shown in the section “The `MessageDialog` Class” on page 109 and the resulting dialog box is shown in Figure 79 on page 112. The `Select Case` statement compares objects of type `MessageDialogButton` to determine which of three possible dialog buttons was pressed.

```
Dim d as New MessageDialog //declare the MessageDialog object
Dim b as MessageDialogButton //for handling the result
d.icon=MessageDialog.GraphicCaution //display warning icon
d.ActionButton.Caption="Save"
d.CancelButton.Visible=True //show the Cancel button
d.AlternateActionButton.Visible=True //show the alternate action
button
d.AlternateActionCaption='Don't Save'
d.Message="Save changes before closing?"
d.Explanation="If you don't save your changes, you will lose " _
    + "your work."
b=d.ShowModal //display the dialog
Select Case b //the MessageDialogButton returned by d
    Case d.ActionButton //determine which button was pressed.
        //user pressed Save
    Case d.AlternateActionButton
        //user pressed Don't Save
    Case d.CancelButton
        //user pressed Cancel
End Select
```

You can also use the `IsA` operator to determine whether an object is of a particular class. The syntax is:

```
Case IsA ClassName
```

Here is a simple example. The code in a pushbutton in a window is:

```
Select Case Me
Case IsA PushButton
    MsgBox "I'm a pushbutton."
Case IsA TextField
    MsgBox "Nope!"
End Select
```

The term “`Me`” refers to the pushbutton, so the first `Case` statement returns true.

Programming with Events and Objects

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

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

Contents

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

Understanding Event-Driven Programming

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

Each object you create in REAL Studio can include, as part of itself, the code you write that executes in response to the various events that can occur for that type of object. For example, a PushButton can include the code you wish to execute when the PushButton is pushed. An object can even respond to events you might not have thought it could — such as responding as the user moves the pointer over a button. When the user causes an event, REAL Studio checks to see if the object the event was directed towards has any code that needs to execute in response to that event. If the object has code for the event, REAL Studio executes that code and then waits for the user to cause another event to occur. This continues until something causes the application to quit, usually the user's choosing Exit from the File menu (Quit on Macintosh).

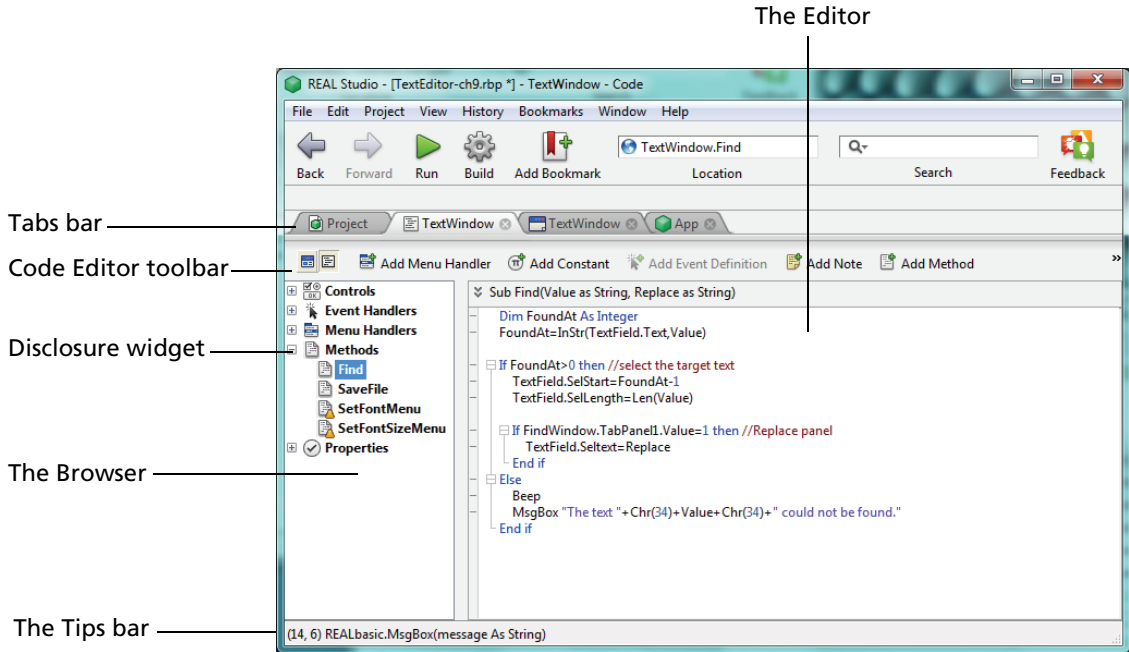
As mentioned earlier, the user can also indirectly cause events to occur. Buttons, for example, have an event called Action which occurs when the user clicks the button. The code that handles the response to an event is called (appropriately enough) an *event handler*. Suppose the button's Action event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

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

Using The Code Editor

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

Figure 224. The Code Editor.



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

- Controls
- Event Handlers
- Menu Handlers
- Methods
- Properties
- Notes

If there are no items in a category, the category does not appear.

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

Opening the Code Editor

Use the Code Editor to edit the code for controls, windows, classes, and modules. There are several ways to open the Code Editor for a specific window.



To open the Code Editor for a window, do this:

- 1 **Click the tab in the Tab bar belonging to the window's Window Editor or, if it does not yet have a tab, double-click the name of the window in the Project Editor.**

By default, the Layout Editor for the window appears. The Edit Mode buttons on the left side of the Window Editor toolbar toggle between displaying the Layout Editor and the Code Editor for the window.

The keyboard equivalents for switching between these two editors is Ctrl+` (Windows and Linux) and Command+` (Macintosh). You can also switch editors with the View ► Show Layout and View ► Show Code menu items. These menu items are available only when a Window Editor is displayed.

- 2 **Click the Code Editor icon or use the keyboard equivalent or menu command to switch editor views.**

The Window Editor switches to the Code Editor for the Window.

If the Project Editor is open, you can go the Code Editor for a window directly by holding down the Ctrl+Shift keys (Windows and Linux) or Option (on Macintosh) when you double-click or press Enter (Return on Macintosh) on the Window's name.



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

- 1 **If it is not already open, click on the tab for the window that contains the control.**

The Window Layout Editor for the window appears.

- 2 **Double-click on the control.**

This will open the Code Editor for the control's parent window. REAL Studio will then automatically expand the Controls group in the browser area, expand the control you double-clicked on, and select the default event handler (e.g., the Action event handler for a PushButton).

Each control has its own default event handler. For example, a PushButton's default event is the Action event, a ScrollBar's default event is the ValueChanged event, and a ListBox's default event is the Change event.



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

- 1 **Right-click on a control in the window (Control-click on Macintosh) and choose Edit Code from the hierarchical menu.**

A submenu of events will appear.

- 2 **Choose the desired event from the submenu.**

To open the Code Editor for a module or class, right-click the name of the item in the Project Editor (Control-click on Macintosh) and choose Edit Source Code from its contextual menu. Modules are stand-alone objects in which you can manage constants, methods, and properties. Classes are reusable objects that are based on exist-

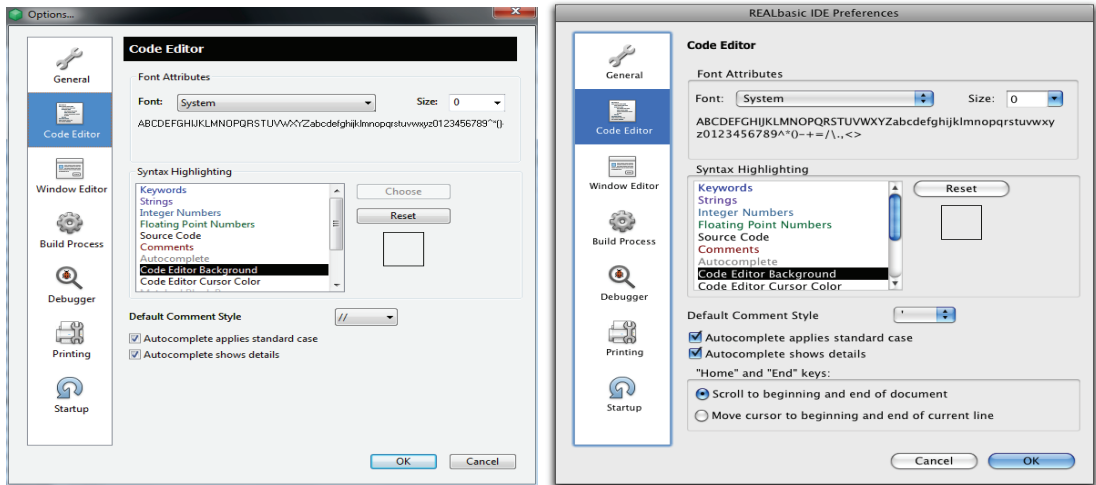
ing objects in REAL Studio. To open a window's Code Editor directly from the Project Editor, you can select it and press Option-Return on Macintosh.

You will learn more about modules in Chapter 6, "Adding Global Functionality with Modules" on page 367 and classes in Chapter 10, "Creating Reusable Objects with Classes" on page 531.

Configuring the Code Editor

You can specify various options for the Code Editor. On Windows and Linux, choose Edit ► Options to display the Options dialog box. On Mac OS X, choose REAL Studio ► Preferences. The Code Editor Options is shown in Figure 225

Figure 225. The Code Editor Options screen.



With the Code Editor Options or Preferences screen, you can set preferences for the following items:

- **Font:** Controls the font and font size used for your code. The first two choices, System and SmallSystem, tell REAL Studio to use the current system or small system font for the platform on which REAL Studio is running. Setting a font size of zero tells REAL Studio to choose a font size that looks best for the current platform.
- **Syntax Highlighting:** This set of preferences sets the colors the Code Editor uses to make your code more readable.

You can assign colors to the following items:

- **Keywords:** REAL Studio language elements, such as control structures (If, while, etc.), and data types (integer, color, double, etc.).
- **Strings:** Literal strings used in code, such as literal text passed to the MsgBox function to display a message to the user.
- **Integer Numbers:** Numbers with no decimal point.

- **Floating Point Numbers:** Numbers that use a decimal point. The distinction between Number and Real Numbers here doesn't depend on the declared data type. For example, if you declare the variable `i` to be an Integer but set `i=5.76`, the "5.76" uses the color assigned to Floating Point Numbers.
- **Source Code:** The text of your variables, controls, classes, operators, methods, properties, and so forth.
- **Comments:** The text of comments inserted in your code. Comments are preceded by two slashes (`//`), the `REM` keyword, or a single quote mark (`'`).
- **Autocomplete:** The text that appears when you use the AutoComplete feature while writing code. For more information on how to use Autocomplete, see the section "Autocomplete" on page 291.
- **Code Editor Background:** The default color for the background in the code editing area of the Code Editor. The default color is the current OS background color.
- **Matched Block Braces:** The default color in the Code Editor for correctly matching lines of code that begin and end a loop of any type, such as "If" and "End if".
- **Unmatched Block Braces:** The default color in the Code Editor for incorrect lines of code that are supposed to begin and end a loop of any type. Matched and unmatched block braces are illustrated in the section "Entering Your Code in the Code Editor" on page 283.
- **Debugger Highlight:** The default color that the Debugger uses to highlight the line of code that is executing. This feature is described in the section "The Debugger" on page 636.
- **URLs:** The default color that is used in the Code Editor for URLs that are entered as strings.

To change a color, select the item that you want to modify in the Syntax Highlighting list and click on the color patch to display the Color Picker, select a new color, and click OK. To revert all color preferences to their default settings, click the Reset button.

- **Default Comment Style:** This preference enables you to control whether the Code Editor will use two slashes (`//`) for commented lines or the single quote (`'`). Your preference is used when you click the Comment button in the Code Editor Toolbar.
- **Autocomplete applies standard case:** If you select this preference, REAL Studio uses the standard uppercase/lowercase conventions when you use the Code Editor's autocomplete feature. It automatically capitalizes terms and uses the internal capitalization shown in the *Language Reference*. If this preference is deselected, autocomplete uses whatever capitalization you've typed. For more information on the Autocomplete feature, see the section "Autocomplete" on page 291.
- **Autocomplete shows details:** If you select this preference, each item in the autocomplete drop-down list includes an icon that denotes its data type. If the

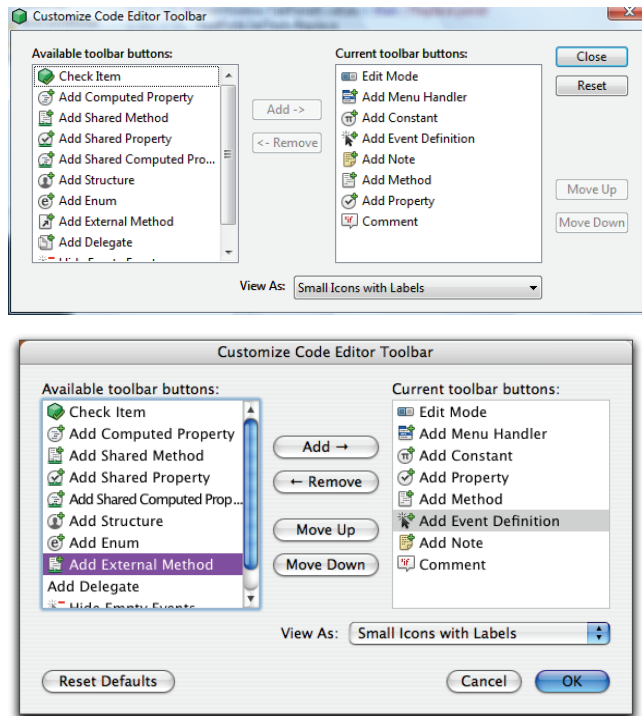
preference is deselected, then only the names of the items are shown. By default, Autocomplete Shows Details is selected.

- **Home and End Keys:** This preference enables you to specify the behavior of the Home and End keys. Your choices are: Scroll to the beginning and end of the document or to move the cursor to the beginning and end of the current line.

Customizing the Code Editor Toolbar

The Code Editor Toolbar (just below Tabs bar) has buttons for adding items to the code for the object being edited. By default, it has buttons for adding methods, properties, event definitions, constants, menu handlers, notes, and comments. If you like, you can modify the Code Editor toolbar with the View ► Editor Toolbar ► Customize submenu. Note that a Code Editor pane must be selected to customize the Code Editor toolbar rather than another editor's toolbar. The Customize Code Editor Toolbar dialog appears:

Figure 226. The Customize Code Editor Toolbar dialog box.



The Customize Code Editor Toolbar dialog box uses a “mover” interface to configure the toolbar. Listed in the right panel are the current items in the toolbar. The left panel contains any available items, but all Code Editor toolbar items are displayed by default.

The following operations are available:

- To add an item, highlight it in the left panel and click the Add button (assuming that an item is available).
- To remove an item, highlight it in the right panel and click the Remove button. This moves the item to the list on the left.
- To reorder an item, highlight it in the right panel and click either Move Up or Move Down. The order in which the items are listed is the left-to-right order in the toolbar.
- To change the appearance of the items in the toolbar, choose an item from the Display As drop-down menu. Your choices are:
 - Big icons with labels.



- Small icons with labels,



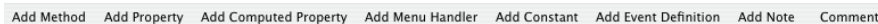
- Big icons (no labels),



- Small icons (no labels),



- Labels only.



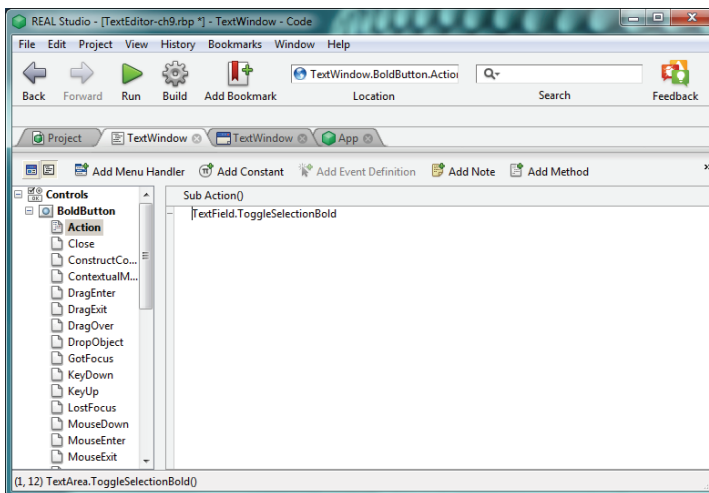
- To reset the toolbar to the default toolbar, click the Reset button (Windows and Linux) or Reset Defaults button (Macintosh).

The Browser

To view the items in each category, click the plus sign (Windows) or disclosure triangle (Macintosh and Linux) to the left of the category name. For example, to view all of the controls for the window, click the plus sign or disclosure triangle next to the Control's category name in the window's editor. When you do this, the list of controls will appear below and to the right. Each of the controls can then be expanded in the same way to display a list of the event handlers for that control.

For example, in Figure 227 you can see that the window named TextWindow has a BevelButton named BoldButton.

Figure 227. A BevelButton's Action event handler.



The Location area gives the name of the screen as `TextWindow.BoldButton.Action`. That is, it's the name of the window, followed by the name of the control, followed by the name of the event handler, separated by dots:

WindowName.ControlName.EventHandlerName.

This control has the following event handlers:

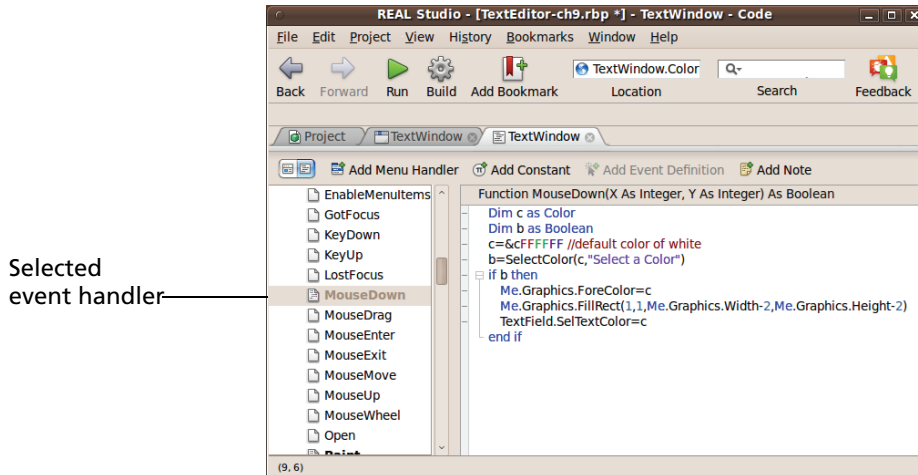
- Action
- Close
- DropObject
- MouseDown
- MouseEnter
- MouseExit
- MouseMove
- MouseUp
- Open

The control's event handlers are listed in alphabetical order within a control or object. When you first open the event handlers for a control (for example, by double-clicking a control in a window), its default event handler is selected. It may be the first event handler in the list (i.e., the Action event handler for a PushButton) but this is not always true. For example, the Paint event is the default event handler for a Canvas control.

Clicking on a control's event handler in the Browser list displays the code associated with that event handler in the Code Editor.

You will learn more about these event handlers later in this chapter.

Figure 228. Some code associated with a control's MouseDown event handler.



Event handlers in the Browser that have code associated with them appear in bold. If one of a control's event handlers has code in it, the event handler's name, the control's name, and the Controls category will all appear in bold. For example, in Figure 227 on page 279, only the Action event has any code associated with it. Controls that have no code appear in plain text. When you are trying to find some code, the bold style acts as a visual cue to let you know if there is any code you might need to look at.

Showing and Hiding Empty Events

When you have code in only a few event handlers, it is often easier to work with the browser when you hide the event handlers that have no code in them. By hiding the empty event handlers in the browser, you can reduce the amount of time that you spend scrolling the browser up and down.

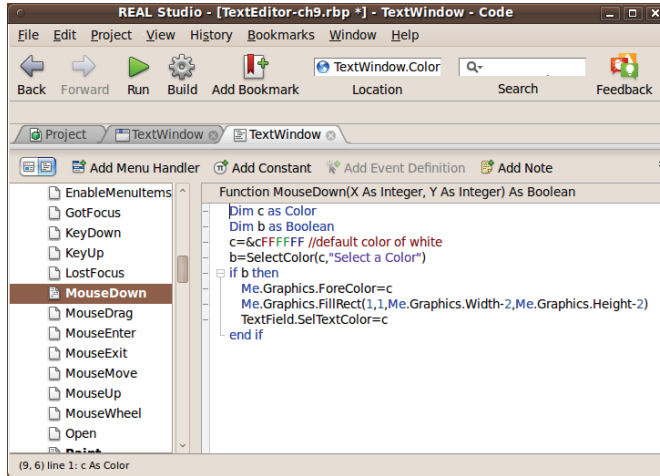
The View menu and the contextual menu in the Code Editor contain menu commands for hiding (or showing) empty events.

When the empty event handlers are hidden, more objects and their event handlers can be brought into view. Figure 229 on page 281 compares the two Code Editor states.

Figure 229. The Code Editor with shown and hidden empty events.

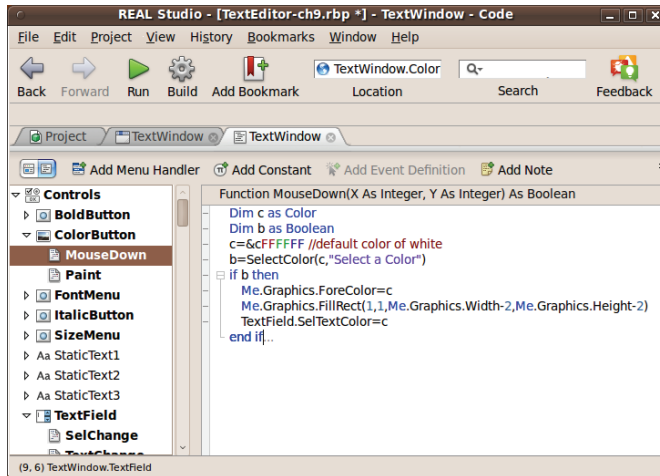
Empty Events Shown

You need to scroll the Browser to bring all objects and event handlers into view. However, you can add code to an empty event handler by clicking on it.



Empty Events Hidden

You have access to all objects and event handlers that have code without scrolling the Browser.



To hide empty events, choose View ► Hide Empty Events or right+click in the browser area (Control-click on Macintosh) and choose Hide Empty Events from the contextual menu. When the empty events are hidden, these menu items change to Show Empty Events.

When you want to add code to an empty event handler while empty events are hidden, you can use the Switch To contextual menu item. Right+Click (Control-click on Macintosh) and choose the Switch To hierarchical menu item. It has submenus that let you switch the Code Editor to any event handler, including ones that are hidden in the Browser tree.

Of course, you can also use either the View ► Show Empty Events menu item or the contextual menu to display the empty event handlers and then navigate to the desired empty event handler in the usual way.

For more information on the Code Editor's contextual menu, see the section "The Code Editor's Contextual Menu" on page 297.

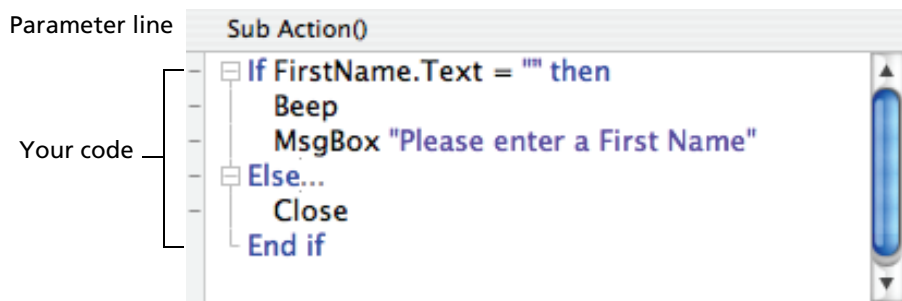


NOTE: When you add new controls to a window, REAL Studio gives them default names. For example, the first `PushButton` you add to a window will be named "PushButton1" by default. A name like that describes the type of object but not what it does. The Browser displays small icons next to each control's name to make the control type clear. Its icon indicates the class from which the control was derived. You should use the control's Properties pane to give the control a meaningful name. For example, in Figure 227 on page 279 a `BevelButton` control has been renamed `BoldButton` to indicate its function in the application.

Understanding Methods in the Code Editor

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

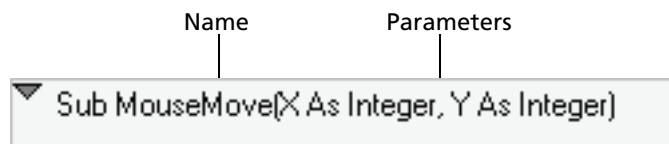
Figure 230. The parts of a method.



The Parameter Line

The parameter line displays `Sub` (short for subroutine) if the method does not return any values, followed by the name of the method or event handler, and then any parameters surrounded by parentheses. The example in Figure 231 shows the parameter line of a `MouseMove` event handler. This event handler is called whenever the mouse is moved inside the control. It is passed two parameters that describe the current mouse location. The parameter `X` represents the horizontal measurement and `Y` represents the vertical measurement. Your code for the `MouseMove` event handler can use the values of `X` and `Y`.

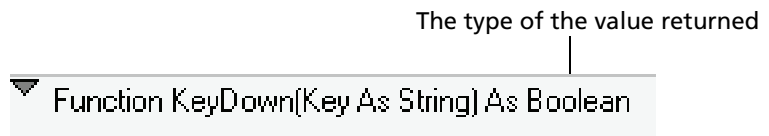
Figure 231. The parts of the parameter line.



For more information on parameter passing, see "Passing Values to Methods" on page 241 of chapter 4.

If the method returns a value, it's called a *function*. A function's parameter line begins with the word *Function* instead of *Sub* and has an additional parameter; the data type of the value that will be returned by the function. The declaration of the value returned by the function follows the parameters. Figure 232 shows the parameter line for a TextField's KeyDown event handler. This event handler is called when the user types a key in a TextField. It is passed the key that was pressed in the parameter *key*. The value returned is a boolean. If you return True from the function, the event is discarded as if it never happened at all and the key that was pressed will not appear in the TextField.

Figure 232. The parameter line of a function



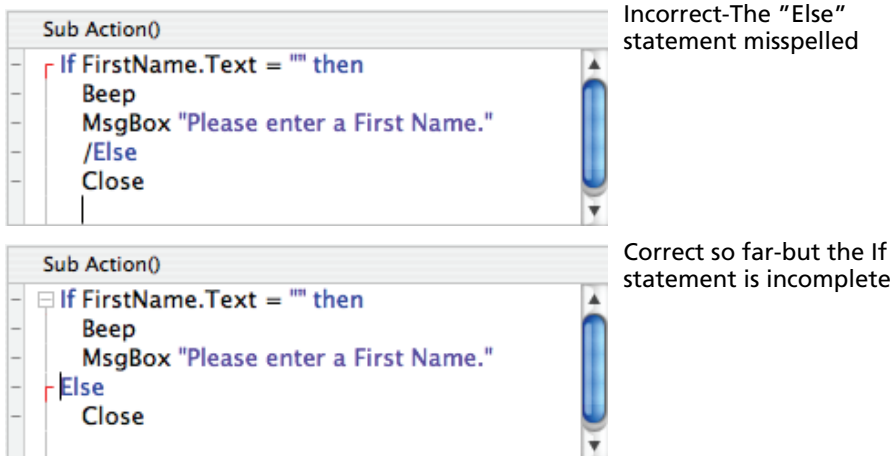
For more information on functions, see “Returning Values from Methods” on page 243 of chapter 4.

Entering Your Code in the Code Editor

As you enter your code, REAL Studio does a few things for you automatically. First, it indents your If...Then, Select...Case, and loops as you type them to make it easier to see which lines of code fall inside a particular statement. It also indicates the scope of each level of indentation with gray brackets to the left of your code. For example, Figure 230 on page 282 shows how the lines indicate the scopes of the If and Else clauses in an If...Then statement. If the If...Then statement were incomplete, then the top corner of the bracket would be red rather than gray and the bottom section of the bracket would be missing.

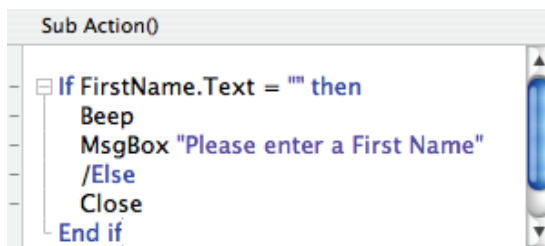
For example, in the top illustration in Figure 233, the Else statement is misspelled, so the bracket does not enclose the If...Else portion of the statement. The red corner indicates that the If statement is open. In the bottom illustration, the spelling error is corrected, but the If statement is still open because the End If statement hasn't been added.

Figure 233. Errors in If...Then statements.



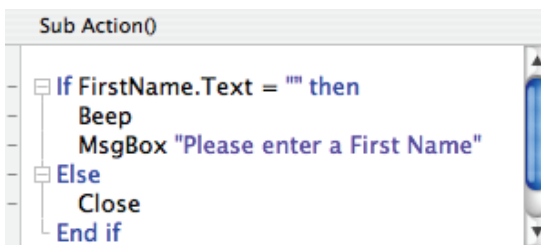
In Figure 234, the If statement has been closed, but the Else statement is still misspelled. Notice that Else isn't indented as it should be and the bracket does not show that it is part of the If...Else...End if statement

Figure 234. A closed If...End if statement with Else misspelled.



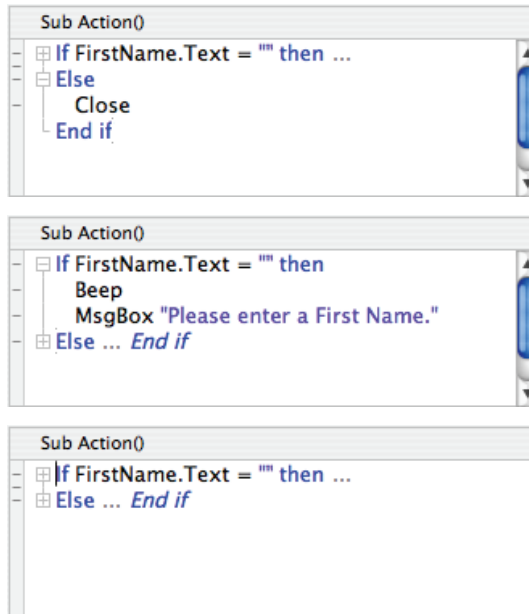
When the errors in the If.End If statement are fixed, the brackets around each clause are completed and the error indicators disappear. The bracket then looks as shown in Figure 235.

Figure 235. A closed If...End If statement.



When the If...End If statement is correct, the If and Else lines each acquire a widget, indicating that the statement be collapsed and then expanded. Each widget can collapse and then expand its own clause. This is shown below.

Figure 236. Collapsed If and Else statements.



The "If" clause is collapsed, indicated by the plus sign on the left and the three dots on the right. Dots indicate that there's hidden code up to the "Else."

The "Else" clause is collapsed, indicated by the plus sign on the left and the italic "End If".

Both the "If" and "End If" clauses are collapsed. Dots indicate where there's hidden code.

If a clause in an If...End If statement is collapsed, the widget changes to a plus sign; click it to expand the clause.

If your code uses two or more levels of nested If statements, the indentation and bracketing shows the scope of each statement. Errors are indicated in the same way as for single-level If statements.

Figure 238 on page 286 shows a correct nested For...Next statement and two types of errors. In the middle illustration, the inner (nested) For...Next statement is incomplete, so its set of brackets has a red top corner and no bottom corner. In the bottom illustration, both loops are complete, but the outer loop is incorrect. It is ended by an (incorrect) "End For" statement, which doesn't exist in the language. The brackets for the outer loop are complete but both the top and bottom corners are red.

All of the errors that are indicated by incomplete or red brackets indicate errors that will prevent the application from being compiled. They must be fixed before you can test the application.

These illustrations use the default colors for incomplete or complete brackets. If you prefer other colors, you can choose different colors in the Code Editor panel of the Options dialog box (Preferences on Macintosh). The color choices are listed as "Matched Block Braces" and "Unmatched Block Braces" in the Syntax

Highlighting listbox. To change a color, simply highlight the item and click the color patch to the right of the listbox. Choose the color in the Color Picker. When you close the Color Picker, your choice will be shown in the Syntax Highlighting groupbox.

Figure 237. Changing the color for Unmatched Block Braces.

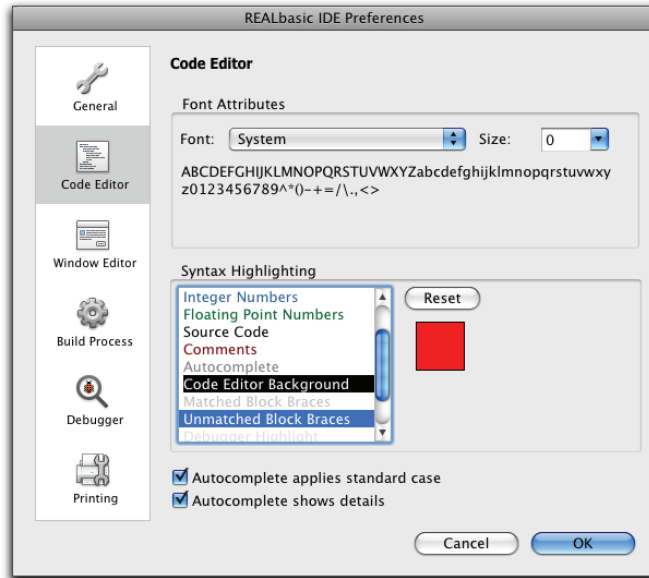


Figure 238. Nested For statements.

```
Dim i,j as Integer
For i=1 to 10
  For j=1 to 10
    Next
  Next
```

Correct: Each For...Next loop has its own brackets

```
Dim i,j as Integer
For i=1 to 10
  For j=1 to 10
  Next
Next
```

Incorrect: The inner loop is incomplete, indicated by a red incomplete bracket

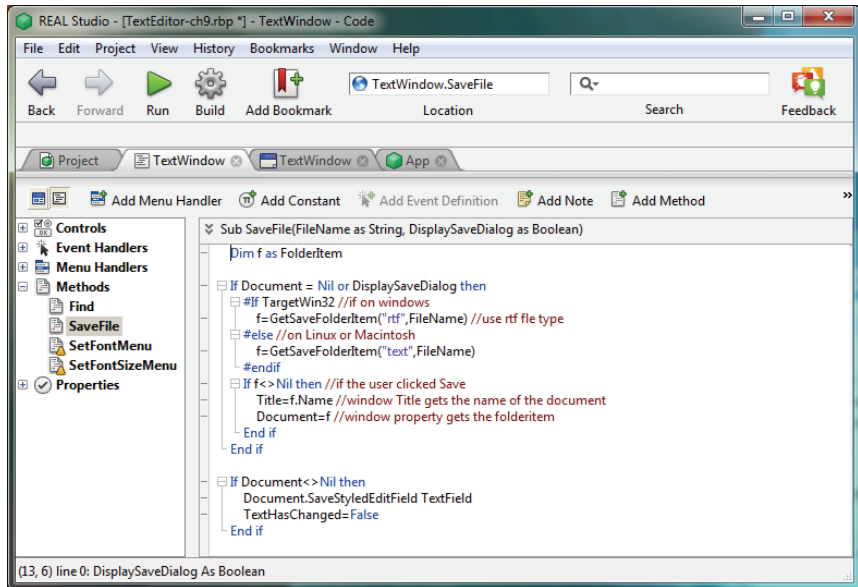
```
Dim i,j as Integer
For i=1 to 10
  For j=1 to 10
    Next
  End For
```

Incorrect: The outer loop is complete but incorrect. The "End For" must be replaced by "Next". Both top and bottom corners of the outer loop are red.

Second, it uses colors to indicate different types of keywords in your code. It uses blue text for keywords and data types (You can change this color by selecting another color using the Code Editor Options (Preferences on Mac OS X). See “Configuring the Code Editor” on page 275.) Figure 230 on page 282 shows an example of these features.

If you insert comments in your code, they appear in red type by default. The color applied to comments can also be changed via Code Editor Options. A text string is treated as a non-executable comment if it is preceded by double slashes (//), a straight quote mark ('), or the REM keyword. Comments do not necessarily have to appear on separate lines, as is illustrated in Figure 239.

Figure 239. Comments at the end of lines of executable code.



A different color is also used for text strings that appear as part of executable code. This text is in red, as shown in Figure 239.

Inserting a Color into the Code Editor

The REAL Studio language includes three functions that return a color, RGB, HSV, and CMY. You can always use them to insert a color value into the Code Editor. You can also specify a color literal by writing its RGB value in hexadecimal and preceding it with the &c symbol. For example, &cFF0000 specifies the color red. However, none of these ways display the color that you specify. If you want to choose a color from a color palette, you have the following option.

When you need to insert the value of a color into the Code Editor, you can take advantage of a shortcut that is in the Code Editor contextual menu. Place the insertion point where you want to insert the color value. Right-click (Control-click on Macintosh) and choose Insert Color from the contextual menu. The Color Picker for your operating system appears. Use it to select a color from its color palette and

click OK. When you click OK, the value of the color in hexadecimal is inserted at the insertion point. It uses the format &cRRGGBB, where RR is the value of red in hex, GG is the value of green, and BB is the value of blue. Each value ranges from 00 to FF.

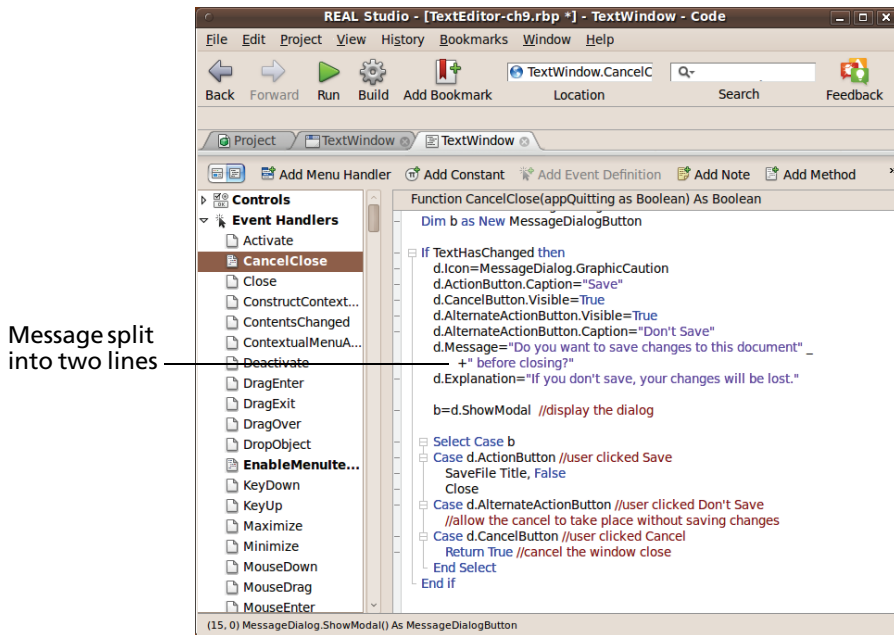
Breaking up a Line of Code in the Code Editor

If you are writing a very long line of code, you can continue it onto a second “line” in the Code Editor. End the first line with an underscore character and continue typing on the next line. The REAL Studio compiler will treat the continuation line as part of the first line of code.

For example, if you need to assign a long string to a property or variable, you can split it up into two lines in the Code Editor. You do this by placing an underscore character as the last character on the line to be continued. When you use the underscore for this purpose, the continuation lines will be indented automatically, as shown in this following example. The text assigned to the Message property of a MessageBox is split into two lines in the Code Editor, but the compiler treats it all as one string.

```
d.Message="Do you want to save changes to this document" _
    +" before closing?"
```

Figure 240. Using the underscore character to continue a long line.



Notice that this example places the underscore character outside the literal text string and uses the + operator to append the second string to the first. You need to do this so that the compiler doesn't think that you are trying to include the under-

score character as part of the literal text string. If the line that you are splitting up does not involve a quoted string, you don't have to do this.

To type the underscore, enter Ctrl+Enter (Option-Return on Macintosh). This enters the underscore and moves the text insertion point to the next line. The next line is automatically indented, indicating that it is a continuation line.

Standardizing the Format of your Code

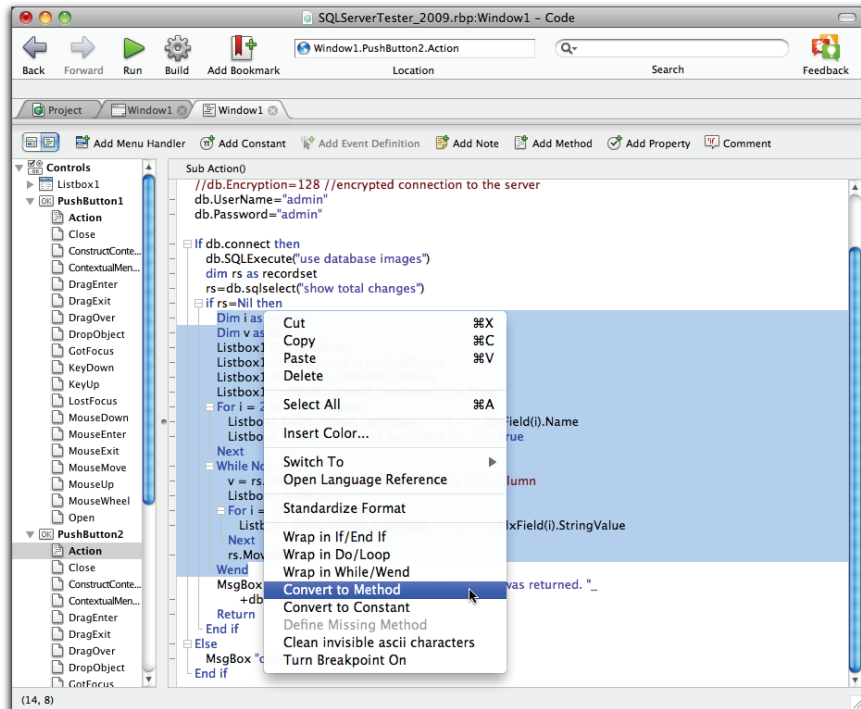
The Code Editor contextual menu offers a menu command that uniformly capitalizes the REAL Studio terms in your code. If you would like your code to be in this format, highlight the code and then right+click (Command-click on Macintosh) and choose Standardize Format from the contextual menu.

Converting a New Method

When you find that a block of code in a method can be reused elsewhere, it is best to refactor the existing method. You can move the reusable code to its own method and then call it from the existing method and any other point from which it should be executed.

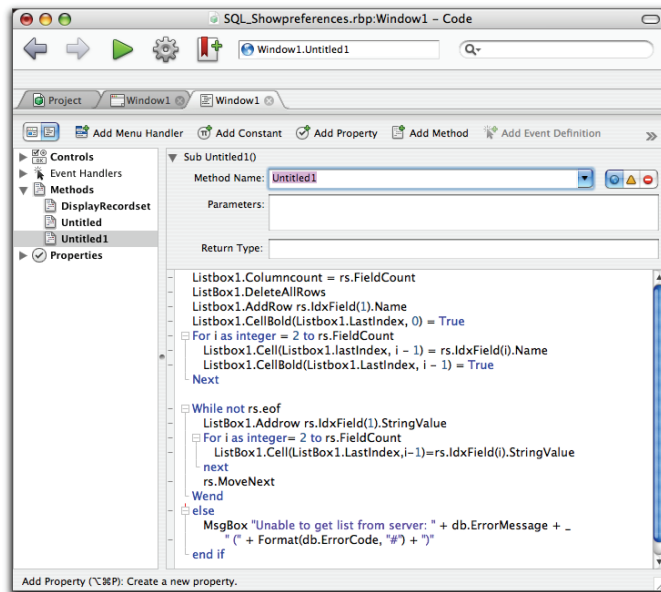
The Code Editor's contextual menu provides a simple way of doing this. Select the block of code that you need to reuse and then Control+click (Command-click on Macintosh) to display the contextual menu. Choose Convert to Method.

Figure 241. The Convert to Method contextual menu.



In Figure 241, a block of code in an If statement is being converted. When the Convert to Method command is chosen, REAL Studio cuts the selected code and pastes it into a new Untitled method, as shown below.

Figure 242. The converted method.



Rename the method and add any parameters that the new method requires. Then go back to the method from which the code was converted and add a call to the new method.

Copying all items in a Group

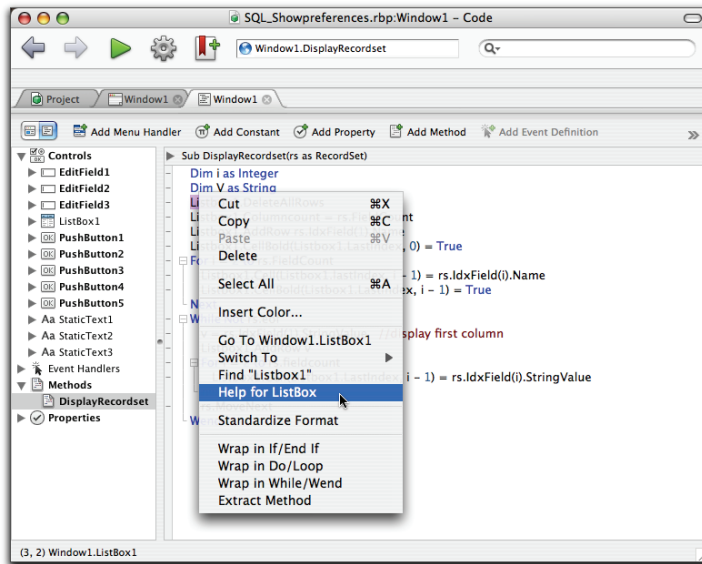
The Code Editor supports dragging and dropping a group of items from the Code Editor in one window to another. For example, suppose you have a large group of menu handlers for a window and you wish to reuse those menu handlers for another window.

To do so, drag and drop the group heading (e.g., menu handlers, methods, properties, or constants) while holding down the Option key for Macintosh. Drag to the browser area and release the mouse button.

Getting Help in the Code Editor

The Code Editor's contextual menu contains a Help menu command that you can use to get help for any REAL Studio item. Select the item for which you want help and then right-click (Command-click on Macintosh). Choose Help for *ItemName* from the contextual menu.

Figure 243. Getting help for the ListBox class using the Code Editor’s contextual menu.



The Help menu command will open the online reference to the item that you requested. Of course, help is not available for user-written methods. You can, instead use the *Go To MethodName* or the *Find MethodName* to open the desired method.

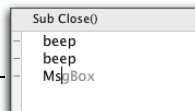
If no item is selected, this menu item changes to “Open Language Reference.” It opens the online reference to the home page.

Autocomplete

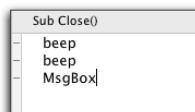
As you type, REAL Studio also attempts to guess what you are typing. If you type the first few characters of a REAL Studio language object — either built in or a variable, method, or property that you created — it shows its guess in light gray type. If the guess is correct, complete the entry by pressing the Tab key. This process is illustrated in Figure 244.

Figure 244. REAL Studio proposes “MsgBox” when the user types “Ms”

As the user types “Ms”, REAL Studio proposes “MsgBox”



When the user presses Tab, REAL Studio completes the



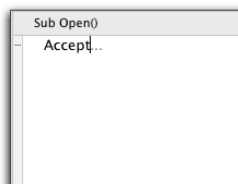
If REAL Studio finds several matching objects, it instead displays an ellipsis (“...”). Press the Tab key to display a contextual menu of choices. You can select an item on the contextual menu with the mouse or navigate up or down in the menu with the Up and Down Arrows. When using the Down arrow keys and you reach the end of the list, the selection will wrap back to the start of the list (same if you use the Up arrow key and reach the top of the list). When the desired item is highlighted, you can select it by pressing the Spacebar, Return, Enter, or Right Arrow keys.

While the contextual menu is displayed, you can continue typing to narrow down the list of choices or cancel the contextual menu by pressing the Left Arrow, Esc, Delete, Backspace, or Clear keys.

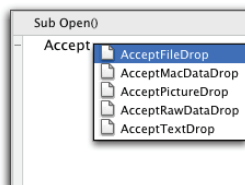
The process of choosing from the contextual menu is illustrated in Figure 245.

Figure 245. Multiple choice autocomplete options.

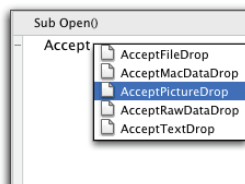
The user types “accept”
and an ellipsis appears...



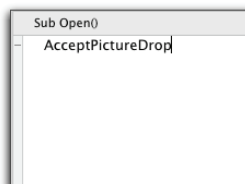
He presses Tab to display
the contextual menu...



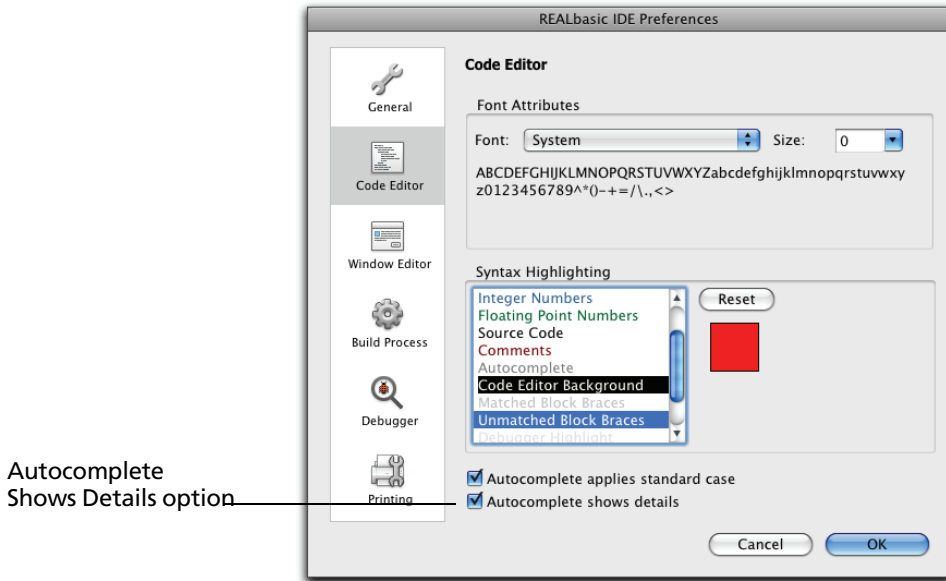
He uses the Up and Down
Arrow keys to highlight
the desired choice and
presses Enter or clicks on it
with the mouse.



REAL Studio completes
the entry.



The Autocomplete popup includes a small icon to the left of each item that denotes its type. This feature is on by default, but it can be turned off by deselecting the “Autocomplete shows details” preference in the Code Editor options panel.

Figure 246. The Autocomplete Shows Details” option.

Autocomplete works in the middle of an expression. With the insertion point anywhere in an expression, you can press Tab to see a pop-up menu of acceptable choices. Auto-code completion also works for user-defined properties, methods, functions, and events. Autocomplete works in the Location and Search areas in the Main toolbar as well.

It also works in the search window in the *Online Reference*.

Using the Edit Menu

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



Note: The Comment button may not be shown; if you wish to use it, you can add it to the Code Editor toolbar by choosing View ► Editor Toolbar ► Customize or the Customize menu item in the toolbar's contextual menu and adding it to the toolbar.

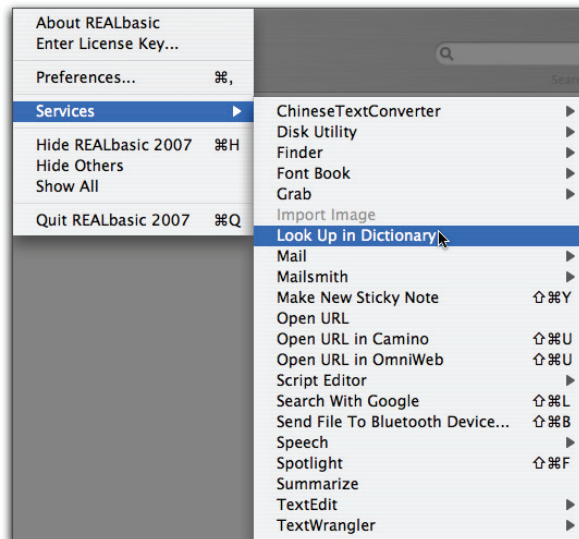
Using Text Services

Mac OS X provides 'hooks' that integrate a variety of services into existing applications. Text Services are provided by the OS and utilities that are written to provide them. Mac OS X Text Services are supported by REAL Studio. A variety of services are available from the Code Editor.

The exact services that are available to you depend on the utilities on your Macintosh. You should have the ability to open a URL in your browser, send mail, copy text into text processors, search with Spotlight, and google a term.

To request Text Services while working in the Code Editor, choose REAL Studio ► Services. A hierarchical menu such as this will appear.

Figure 247. A Text Services menu (some items depend on specific applications).



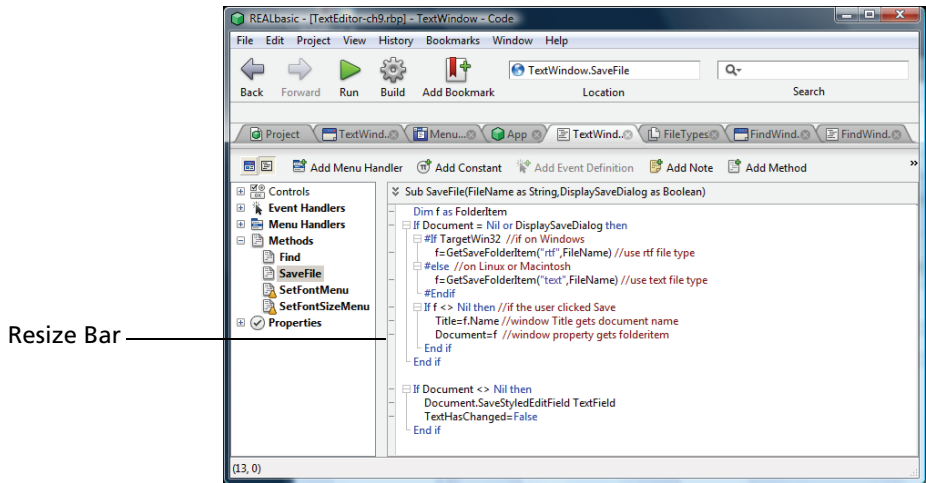
Some services assume that you have selected text and want the service to handle it in some way.

Getting More Usable Space in the Code Editor

There may be times when you need more vertical or horizontal space in the Code Editor. You can, of course, resize the IDE window to get more space, but this isn't always an option. One way to get more space is to use a smaller font. You can set the font and font size for the Code Editor by choosing Edit ► Options (REAL Studio ► Preferences on Mac OS X) and selecting the Code Editor font and font size in the Source Code Editor pane.

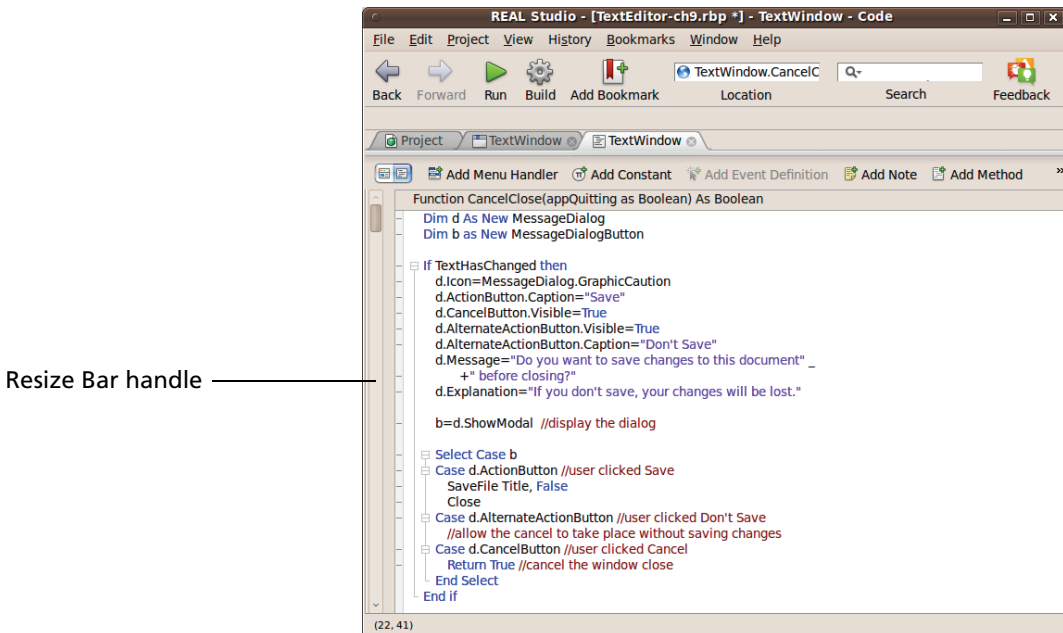
You can also minimize the Browser when you don't need it. Use the View ► Editor Only command to minimize the size of the browser area and maximize the code editing area. You can also hide the Browser by dragging the divider (the bar between the Browser and the Editor) all the way to the left side of the Code Editor window. However, dragging the divider to this position does not change the state of the Editor Only menu command. If Editor Only is off and you drag the divider to the extreme left, Editor Only is still off even though only the editor is visible.

Figure 248. The Code Editor's Resize Bar.



When you do this, the Browser is hidden and the divider is reduced to a button to the left of the Code Editor.

Figure 249. The Code Editor with the Browser hidden.



As you can see in Figure 249, this gives you quite a bit of horizontal space to work with in the Code Editor. You can show the Browser again by dragging the Resize Bar handle towards the right side of the Code Editor window.



NOTE: The Browser will expand and collapse the categories (Controls, Events, Menu Handlers, Properties) when they are selected by pressing Ctrl+Left Arrow (to collapse) or ⌘-Left Arrow and Ctrl+Right Arrow or ⌘-Right Arrow (to expand).

Opening a Window from its Code Editor

If you are working in a Code Editor that belongs to a window, you can switch to the window's Window Editor (a.k.a., Layout Editor) by pressing Ctrl+` (Command-` on Macintosh) or choosing the View ► Show Layout menu command. When the Window Editor is shown, the menu command changes to View ► Show Code. Pressing the keyboard shortcut will toggle back to the Code Editor for the window.

You can also switch views using the pair of icons just above the browser area. They toggle between the Layout Editor and the Code Editor for the window. The icon on the right displays the Code Editor and the icon on the left displays the window in the Layout Editor.

Figure 250. View Mode Buttons for switching between the Layout Editor and the Code Editor.



On Macintosh and Linux, the selected view mode is highlighted (as shown in Figure 250) and on Windows it is depressed.

If you prefer, you can configure the Window Editor so that both the window's Layout and Code Editors can be open at the same time. You can do this in either of two ways. You can set up the IDE so that the Layout and Code Editors each get separate tabs in the Tabs bar. You can also open up the Code and Layout Editors in separate IDE windows.

To give each type of editor separate tabs, open the Options dialog (Edit ► Options on Windows and Linux and REAL Studio ► Preferences on Macintosh), select General preferences, and then deselect the "Code and Window Editors share a Tab" option. To use separate windows for all your editors, deselect the "Enable Tabbed Browsing" option.

Finally, you can drag the tab for either the Code or Layout editor out of the IDE window and it will open in a separate window. For more information on these options, see the section "Dragging a Tab" on page 54 and "Configuring the IDE for Multiple Windows" on page 51.

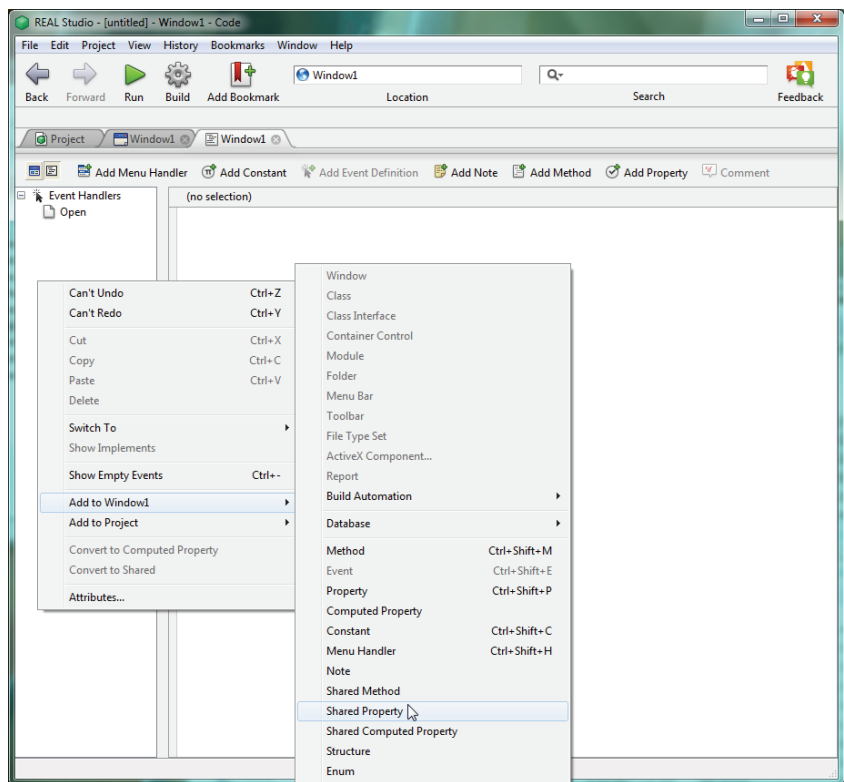
You can also open up a new IDE window for the project and use it for any purpose you like. Choose File ► New Window to get a new IDE window for the current project. In the new window, you can use one IDE window to display the window in its Window Editor and the other IDE window to display its Code Editor.

The Code Editor's Contextual Menu

Contextual menus are context-sensitive pop-up menus that appear when you right+click the mouse button on an item (Ctrl+click within the Code Editor on Macintosh). This displays a contextual menu with all of the items from the Browser. This is especially handy when you are using the View ► Editor Only option to provide more horizontal space in the Code Editor or when you have the Hide Empty Events option on (see “Showing and Hiding Empty Events” on page 280).

The Add contextual menu item has a submenu that allows you to add a new item to the Code Editor. For Code Editors belonging to windows, you can add a new method, shared method, property, shared property, computed property, shared computed property, constant, menu handler, or note. For Code Editors belonging to classes, you can also add a new event definition.

Figure 251. The Code Editor's Add contextual menu.



These menu commands are also available under the IDE's Edit and Project menus. The contextual menu in the code editing area has special items for searching on the current item and to wrap selected items in If, Do, and While statements.

Searching your Project

The REAL Studio IDE offers three types of interfaces for finding project items. They are:

- The Search area in the Main toolbar,
- The Find and Replace dialog box,
- The Code Editor contextual menus.

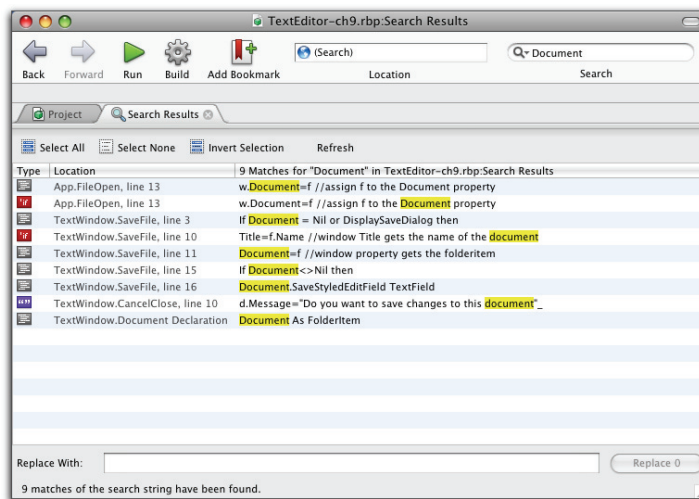
The Search Area

The Search area contains a menu in which you can specify that you want to search the entire project, the current item (for example, the current window, menubar, class, or module), or the current method only. On Mac OS X 10.4 and above, it also offers to search the entire computer using the operating system’s search engine, Spotlight. You display the pop-up menu by pressing the mouse button on the magnifying glass in the Search area.

To do a search, choose the scope of the search from the pop-up menu and then type in the item to be searched. Press the Enter key (Return on Macintosh). When it finishes searching your project, it will display the results of the search in a new IDE screen called Search Results. If you do a new search, those results will replace the current results.

For example, Figure 252 shows a search on the string “Document”. In the project, it is the name of a window’s property. REAL Studio has found all occurrences in the project and listed them on the Search Results page. The Type column in the list shows the type via an icon and the Location column gives the location in the form *Classname.EventHanlder* or *WindowName.EventHandler*. The Matches column gives the line of code that contains the term being searched.

Figure 252. The Search Results for a search on “Document”.



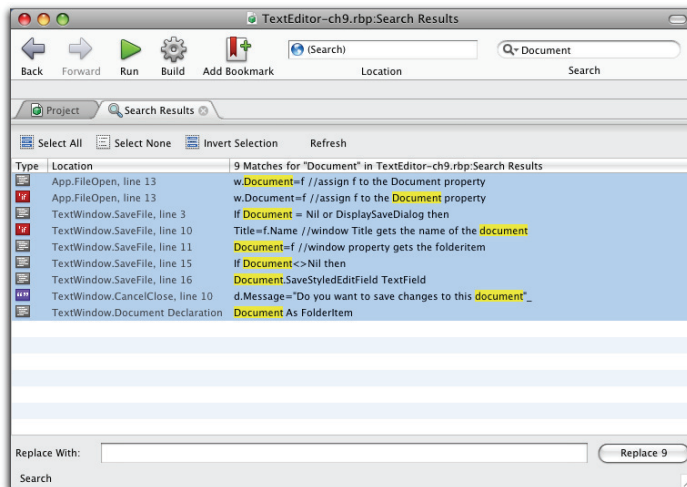
The columns in the Search Results list are sortable. Click on the column header to sort on that column.

In this search, the search string is the name of a property. The icon in the left column indicates whether its usage is in executable code, in a text string, or in a non-executable comment.

The Search Results screen gives you the option of replacing the highlighted string. If you want to do a replace, enter the replacement string into the Replace With area at the bottom of the window. Then highlight one or more lines in the list. You can use the Select All, Select None, and Invert Selection buttons in the Search Results toolbar to make or change the selection.

The replacement will be done only in a highlighted line. As you highlight the lines, the Replace button will indicate how many replacements will be done. For example, in Figure 252, no lines are highlighted, so no replacements will be done. In Figure 253 all the lines are highlighted so all the instances will be replaced.

Figure 253. The Search Results screen with all rows selected.



If you don't want to do a replace, you can navigate to any of the search results by double-clicking the line. The Search Results screen remains accessible via the Tab bar after you have visited the editor containing the searched string. When you are finished dealing with the occurrence of the search string, you can return to the Search Results screen by clicking on its tab.

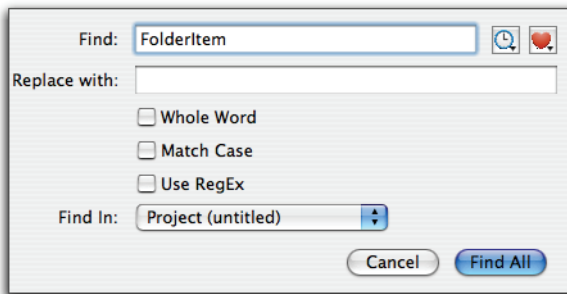
When you do more than one search, a Refresh button appears in the Search Results screen. Click it to redisplay the first search.

The Find in Project and Find dialogs

The Edit ► Find menu item has submenu items for finding or replacing for the current item and for the whole project.

Find All displays a Find and Replace dialog in which you can specify the scope of the Find.

Figure 254. The Find in Project dialog box.

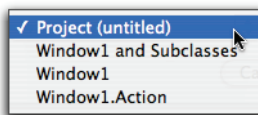


The Find In pop-up menu lets you set the scope of the search. You can choose to search in:

- **The entire project:** Searches all of the project items in your Project Editor. This is not limited to classes and windows.
- **The current window or class and their subclasses:** Searches only the current window or class and all subclasses of that window or class.
- **The current project item only:** Searches only the current class, window, or module, and omits any subclasses derived from the current class or window.
- **The current method only:** Searches only the current method in the Code Editor. Available only when a method is displayed in its Code Editor.

In this case, the Action event of a Pushbutton in a window was displayed, so the pop-up offers the following choices:

Figure 255. The Find scope pop-up menu.



The checkboxes in the center of the Find in Project dialog offer three options:

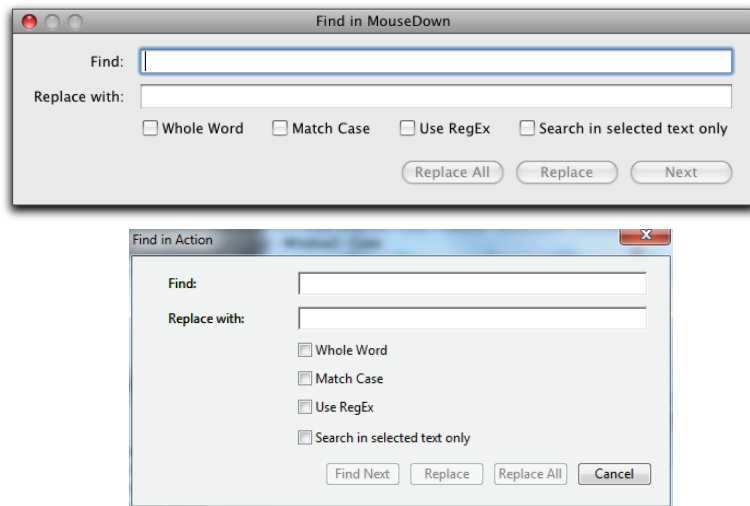
- **Whole Word:** Find will search only for the entire word or words specified in the Find area. It will ignore matching text strings that are embedded within words.
- **Match Case:** Find will search only for text strings that match the uppercase/lowercase specifications entered into the Find area. For example, if you entered “document”, and checked Match Case, it will not find “Document”.
- **Use RegEx:** If RegEx is specified, you can use regular expressions in your find and replace specifications. REAL Studio will interpret them according to the rules for regular expression searches. If you use regular expressions but do not check this CheckBox, REAL Studio will assume that you mean the literal text you entered. See

the entry for the RegEx class in the *Language Reference* for information about regular expressions.

When you click Find All, it will find all the instances of the string in the Find scope that you specified. It opens a Search Results tab in the IDE and lists all the items that it found. An example Search Results screen is shown in Figure 259 on page 303.

The Find dialog offers the same find and replace options but the scope of the search is limited to the current event handler. Optionally, you can limit the scope further to the selected text. Click the “Search in selected text only” checkbox to limit the scope.

Figure 256. The Find dialog box.



The Title bar gives the name of the event handler or method that the search is limited to. Unlike the Find in Project dialog, the Find dialog does the find or find/replace one item at a time. Click Next to find the next item or click Replace to do the next replace.

The Edit ► Find submenu gives you the ability to find the next occurrence of the item you are searching for, replace the highlighted text in the Code Editor with the text in the Find window’s Replace field, and replace all occurrences within the chosen scope. The keyboard equivalents are shown in Table 8:

Table 8: Keyboard equivalents for Find commands.

Command	Keyboard Equivalent
Find	Ctrl+F or ⌘-F
Find Next	F3 or ⌘-G
Replace	Ctrl+= or ⌘-=
Replace and Find Next	Ctrl+Shift+H, or ⌘-Shift-=

Recent and Favorite Searches

The two icons in the top-right corner of the Find in Project dialog box enable you to store and recall previous searches. The icon on the left represents recent searches and the icon on the right represents favorite searches. REAL Studio adds your searches to the Recent searches list automatically and you can add searches to your Favorites list with its Add to Favorites menu item.

Figure 257. The Recent searches and Favorite searches icons.



To recall a recent search, simply display the Recent searches pop-up menu and choose it from the list. To add a search to your Favorites list, display the Find or Find and Replace specification in the Find dialog and then choose Add to Favorites from the Favorites pop-up menu. REAL Studio will then present a dialog box enabling you to name the favorite. You can then recall the stored search at any time by choosing its name from the Favorites pop-up menu.

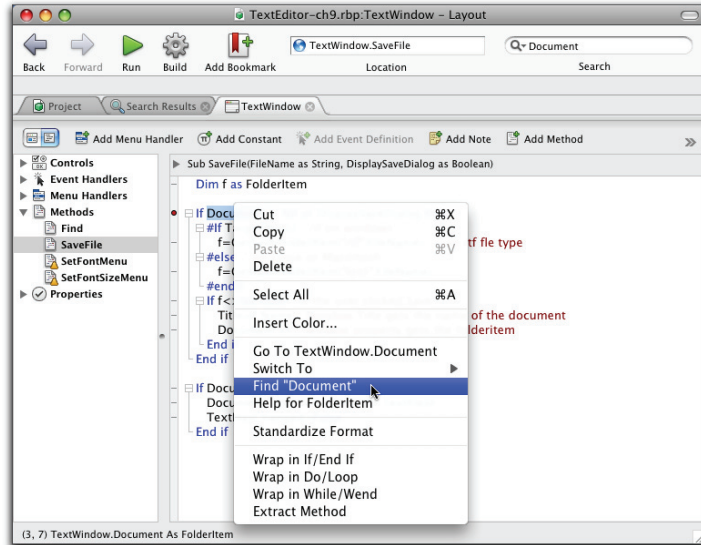
Finding using the Contextual Menu

If you are specifically looking for the occurrences of a variable, constant, property, method or other item, you can use the Code Editor's contextual menu to do the search. The "Find *Item*..." menu item finds the occurrences of *Item*. It is especially convenient for items other than local variables (which are declared with Dim statements within the method) because it searches through the code belonging to other objects in the project. For example, you can easily locate the declarations for the constants you use in localizing your application or properties that belong to other windows, modules, or classes. Or, you can use Find Item to locate the source code for one of your methods.

To use the contextual menu, simply select the item whose occurrences you wish to find. Right+click (Control-click on Macintosh) to display the contextual menu. Choose the "Find *itemName*" menu item. The Find contextual menu item is available in both the code editing area and the Code Editor browser.

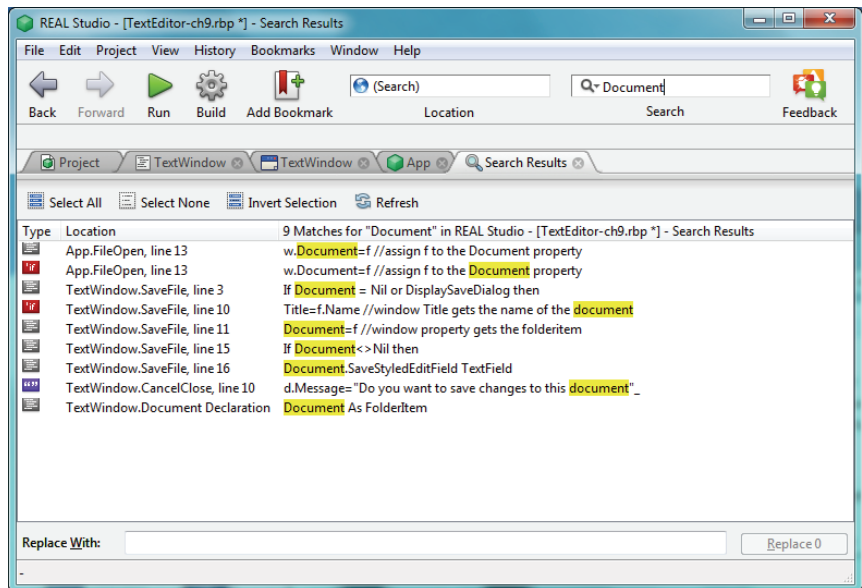
In the following example, the occurrences for of item "Document" in the Code Editor is located. It turns out to be a property of the window.

Figure 258. Using the “Find Item” contextual menu.



REAL Studio opens a new search results screen and lists all occurrences, with the item being searched for highlighted. Figure 259 shows the results of the search that is specified in Figure 258.

Figure 259. The search results screen.



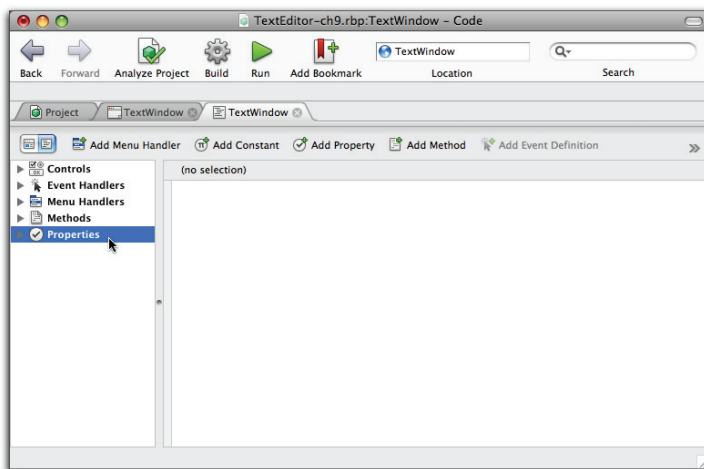
The standard Search Results screen offers the option of replacing the selected text in any number of lines. Also, you can double-click any row of the table to display that occurrence.

Copying and Pasting Code

If you are working with two Code Editor windows and want to move properties, methods, events, or constants in one window in another window, you can copy and paste them. You can select the group header in one window, copy it, and then paste it into another editor.

In this screen shot, the window properties are being selected for copy. To paste the properties into another Code Editor window, chose Edit ► Copy, switch to the other window and choose Edit ► Paste.

Figure 260. Selecting a group header for copying.

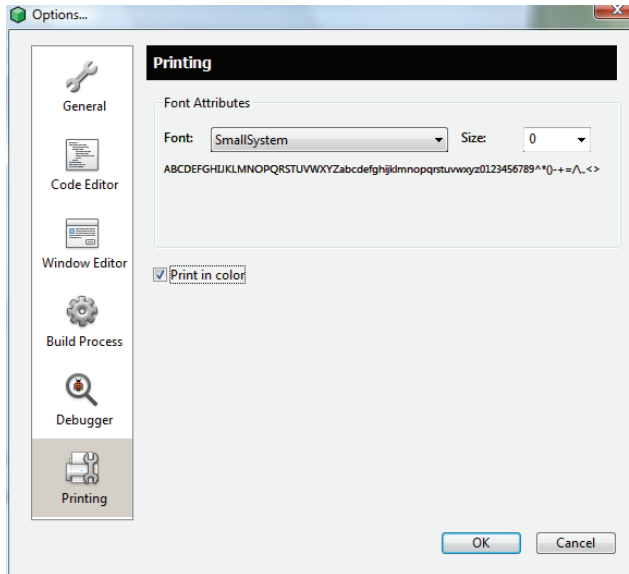


Printing Your Code

When you need to print your source code, choose File ► Print (⌘-P or Ctrl+P).

Using Edit ► Options (REAL Studio ► Preferences on Mac OS X), you can specify the font and font size used for printing, as well as whether you wish to see keywords in bold, and print the colors that appear in the Code Editor. Use the System or SmallSystem fonts to tell REAL Studio to use the system font for the platform on which REAL Studio is running and a font size of zero to tell REAL Studio to choose the optimal font size for your operating system. To print in color, select the “Print in Color” option.

The Printing options are shown in Figure 261.

Figure 261. Options for printing.

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

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

External Project Items

If you want to share an object among several projects, you can import it as an external project item. The item remains on disk and changes that are made to it from another project are automatically reflected in the current project when you re-open it. However, you can't open more than one project simultaneously that references the same external project item. To add an external project item to a project, hold down the Ctrl+Shift keys (⌘ and Option keys on Macintosh) while you drag the item from the desktop to the Project Editor. In the Project Editor, the external project item's name will be shown in italics.

For more information, see the section “External Project Items” on page 80.

Importing

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

For code, open the method that you wish to import code into and drag the text clipping into the body of the method. You cannot use the File ► Import command to import text files into the body of a method.

To delete a file that has been added to the Project Editor, highlight it and press the Delete key on the keyboard or choose Edit ► Delete. You can also delete an item in the Project Editor by Control-clicking on the item and choosing Delete from the contextual menu.

Note: When you import a REAL Studio object by dragging it into the Project Editor, it automatically replaces an item with the same name, without asking you whether you want to replace it with the dragged item.

Some of the items you import are copied into your project. Some types of objects are not copied but instead an alias to the original file is stored inside your project. In the Project Editor, aliases are shown in italics.

When you build a stand-alone version of your project, most of these files are then copied into the stand-alone application. Table 9 on page 306 shows how all of the different file types are handled.

Table 9: How REAL Studio handles imported files.

File Type	Copied Into Project?	Copied into stand alone applications?
Bitmap, PICT, JPEG, GIF	N	Y
Cursors	Y	Y
PowerPC Shared Libraries	N	N
QuickTime and WMP Movies	N	Y
REAL SQL Databases ^a	N	N
REAL Studio Classes	Y	Y
REAL Studio Menubars	Y	Y
REAL Studio Modules	Y	Y
REAL Studio Windows	Y	Y
Resources	N	Y
Sounds	N	Y
AppleScripts	N	Y

a. Data sources, such as a REAL SQL database, appear in plain text in the Project Editor even though the data is stored outside the project. The connection information to the data source is stored within the project.

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

When you import a picture into your project by dragging it into the Project Editor or choosing Import from the File menu, the picture will be loaded into memory when you run your application regardless of whether or not the picture is used.

If you have several large pictures, your application may not behave very well or it might even run out of memory. If you have large pictures, consider storing them externally and loading them with the `GetFolderItem` method.

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

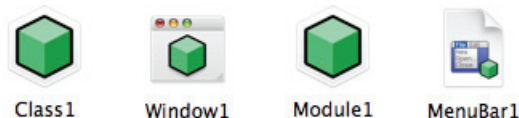
Exporting

The code for methods, events, constants, properties, and so forth can be dragged to the desktop as text clippings or into a text or word processor (Macintosh only). On Windows and Linux, you can copy code to the Clipboard and paste into a text editor or word processor.

You can also export your source code to a text file or in REAL Studio's native format using an `Export...` menu command. Which method you use depends on what you will be doing with the exported code. If you are going to be including code in some kind of documentation, drag the code to the other application or export your code to a text file.

On the desktop, the exported objects have their own REAL Studio icons, shown in Figure 262.

Figure 262. Icons for exported REAL Studio objects.



You can export REAL Studio objects using the `Export` command.

To export using the `Export...` command, do this:



- 1 Open the item so that it is displayed on the screen or select it in the Project Editor.**
- 2 Choose `File ► Export Item` or display the item's contextual menu and choose `Export`.**
- 3 When the `Save As` dialog box appears, type a name and click the `Save` button.**

On Windows and Macintosh, the `Save As` dialog gives you the option of saving as a REAL Studio object, as XML, or in Version Control System (VCS) format.

Encrypting Your Source Code

If you want to distribute a copy of a window, menu bar, module, or class for others to use but you do not want them to be able to view or edit your code, use the `Encrypt` command to protect the object prior to exporting it. The icon belonging to an encrypted item in the Project Editor has a small key in its lower-right corner.

Encryption is supported only in the Professional and Studio editions of REAL Studio. Decryption is supported in all editions.



When a protected object is exported, its icon is no different from an unprotected object.

You can encrypt (protect) or decrypt (unprotect) a window while it is in your project. An encrypted window cannot be opened in a Window Editor and no one can access any code associated with the window or any of the controls on the window.

When encrypting an item, you supply a password that can be used to decrypt it later.



To encrypt an item, do this:

- 1 **Right+click (or Control-click on Macintosh) on the item in the Project Editor and choose Encrypt from the contextual menu or choose Edit ► Encrypt.**

You can optionally add an Encrypt button to the Project Editor toolbar. If you do so, you can highlight an item in the Project pane and click the Encrypt button to encrypt the selected item.

The Encrypt dialog box appears, as shown in Figure 263.

Figure 263. The Encrypt Dialog box.

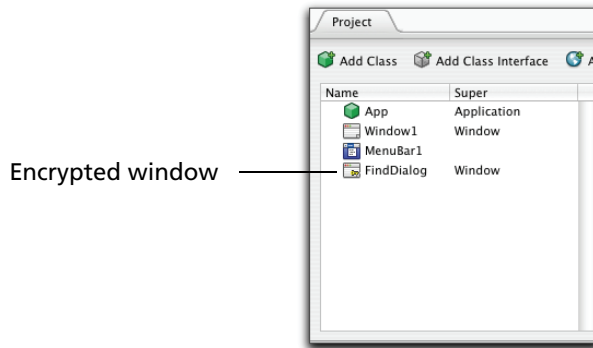


- 2 **Enter and confirm a password for decrypting and click the Encrypt button.**

Important Note: Don't forget the password.

- 3 **If you want the item to be accessible only to REAL Studio 2003r3 and above, check the "Use REAL Studio 2006r3 Encryption" checkbox.**

An encrypted item appears in the Project Editor with a small key in the lower right corner of its icon, such as this icon for the encrypted SaveChanges window.

Figure 264. A project with an encrypted window.

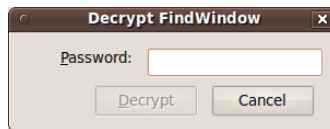
When a programmer (including the original author) tries to open an encrypted item, the Decrypt Object dialog box appears, shown in Figure 265.

To edit the object and/or its code, you must decrypt it using its encryption password.

To decrypt an encrypted object, do this:

- 1 **Right+click (or control-click on Macintosh) on the item in the Project Editor and choose Decrypt from the contextual menu or choose Edit ► Decrypt.**

The Decrypt *Object* dialog box appears.

Figure 265. The Decrypt window dialog box.

- 2 **Enter the password that was entered when the item was encrypted and click Decrypt.**

If the password is correct, the key will disappear from the item's icon in the Project Editor, indicating that it has been successfully decrypted. If you entered an incorrect password, a dialog box will inform you of that fact.

If you have exported an encrypted item and then imported it into another project, you can decrypt it with the original password.

Responding To User Actions with Event Handlers

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

PushButton occurs when the user clicks the PushButton. Other events are indirectly caused by the user, such as the Open event of a window that occurs when the window opens.

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

Object-Oriented Programming

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

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

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

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

Windows Events

Windows get many different events. Table 10 describes these events in general. If you need specific information about window events, see the `Window` class in the *Language Reference*.

Table 10: Window class events.

Event	Description
Activate	The window is being made the active window of the application.
CancelClose	The <code>Quit</code> method or the <code>Window.Close</code> method has been called. Returning <code>True</code> from this method will cancel the quit and the window will remain open.
Close	The window is about to close but hasn't closed yet. Controls also receive <code>Close</code> events. A window receives its <code>Close</code> event after all of the controls have received their <code>Close</code> events.
ConstructContextualMenu	Occurs when it is appropriate to display a contextual menu. Build and display the contextual menu in this event for the window or the <code>ConstructContextualMenu</code> event for a <code>RectControl</code> (any visible control).
ContextualMenuAction	The user has chosen an item from the contextual menu but it has not been handled by the menu item's <code>Action</code> event or its menu handler. Handle a contextual menu selection from a contextual menu created and displayed in the <code>ConstructContextualMenu</code> event handler.
Deactivate	The window is being deactivated.
DropObject	A file, piece of text, or a picture has been dropped on the window itself (not on a control in the window). This event handler is passed a parameter that gives you access to the item dropped.
EnableMenuItems	While the window is front of all other windows, the user has clicked in the menu bar to select a menu item or pressed a menu item's keyboard equivalent. This event handler gives you a place to decide which menu items should be enabled before the user can actually choose one.
KeyDown	A key has been pressed that has to be handled by the window. For example, the tab key is never sent to any control. It is instead handled by the window itself. If the window has no controls that can receive the focus, any keys that are pressed will generate <code>KeyDown</code> events for the window. This event handler is passed a parameter that tells you which key was pressed.

Table 10: Window class events. (Continued)

Event	Description
MouseDown	The mouse button has been pressed and has not yet been released. You can return False in this event handler to filter the event causing the window to act as if the mouse button was never clicked. This event handler receives parameters that indicate where the mouse was clicked in local window coordinates.
MouseDrag	The user has moved the mouse inside the window (but not over a control) while the mouse button is held down. This event handler receives parameters that indicate where the mouse is in local window coordinates.
MouseEnter	The user has moved the mouse inside the window from a location outside the window.
MouseExit	The user have moved the mouse outside the window from a location inside the window.
MouseMove	The user has moved the mouse inside the window. This event handler receives parameters that indicate where the mouse is in local window coordinates.
MouseUp	The mouse button has been released inside the window. This event will not occur unless you return True in the MouseDown event handler. The idea behind this is that if the mouse was never down, it can't be up. This event handler receives parameters that indicate where the mouse was released in local window coordinates.
Moved	The window has been moved by the user or by code that changes the window's Left or Top properties.
Open	The window is about to open but hasn't been displayed yet. Controls also receive Open events. A window receives its Open event after all of the controls have received their Open events.
Paint	Some portion of the window needs to be redrawn either because the window is opening or it's been exposed when a window in front of it was moved or closed. This event handler receives a Graphics object as a parameter which represents the graphics that will be drawn in the window. Graphics objects have their own methods for drawing graphics. See the Graphics class in the <i>Language Reference</i> for more information.
Resized	The window has been resized by the user or by code that changes the window's Width or Height properties.

Table 10: Window class events. (Continued)

Event	Description
Resizing	The user is in the process of resizing the window.
Restore	The window is being restored from its minimized state to its state prior to being minimized.

Opening Windows

There are two different techniques you can use to open windows. The technique you use depends on what you are going to do with the window once it's open. If your application will never have more than one copy of a particular window open at a time, you can open the window simply by making reference to any of the window's properties or by using the window's Show method.

The following example opens a window by accessing one of the window's properties, its Title property:

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

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

```
aboutBoxWindow.Show
```

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

These techniques are available if the `ImplicitInstance` property of the window is set to `True` in the IDE. This property enables you to instantiate an instance of the window implicitly. It is set to `True` by default, so there is no reason to change it unless you want to disable these two ways of opening windows. If `ImplicitInstance` is set to `False`, you must explicitly create a new instance of the window using the following technique. If you try to open a window by accessing one of its properties or calling `Show` and `ImplicitInstance` is `False`, then you will get a compiler error. Turn on `ImplicitInstance` to cure the errors.

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

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

A shorthand way of accomplishing the same thing is to create and instantiate the reference to the window in the Dim statement. Do this by using the New operator as a modifier in the declaration:

```
Dim w as New aboutBoxWindow
```

This single line of code opens the aboutBoxWindow.

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

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

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

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

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

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

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

There is less of a need to dimension a window variable as type Object than there is to use type "Window." However, you might use this technique when you are creating new instances of controls on the fly. With controls, you can have a variable storing a reference to many different kinds of controls. See "Creating New Instances of Controls On The Fly" on page 346 for more information. See "Accessing Items of Other Windows" on page 342 for more information on how to use window references.

Adding Properties to Windows

The properties of an object are pieces of information that help define the object. They're variables that are accessible in more than one event handler or method. Windows have many pre-defined properties such as their title, width, height, etc. You can also add your own properties to windows that allow you to store informa-

tion that is specific to the instance of the window. Window properties work like variables except that they belong to the window and can be accessed by any of the window's event handlers and any of the window's controls. You also have the option of making a property of a window accessible to code outside the window. In contrast, a variable that is defined by a `Dim` statement inside an event handler is local. It cannot be accessed outside that event handler.

For example, if you have a window that displays the contents of a document, you might need to keep track of whether the user has modified the data to determine if he should be given a chance to save changes when he quits your application. Where do you keep track of this? Since the window is effectively a representation of the document, you can add a boolean property called *Changed* to the window. When the user makes a change in the window that affects the document, your code can change the value of the *Changed* property from `False` to `True`. Later, when the user closes the window, the code in the window's `Close` event handler can check the *Changed* property to determine if the user needs to be given the opportunity to save his changes. The syntax for accessing the properties you add to windows is the same as the syntax you use to access a window's pre-defined properties. For example to set the *Changed* property of a window called "myDocumentWindow" to `True`, you use the following syntax:

```
myDocumentWindow.Changed=True
```

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

The Scope of a Property

When you define a property of a window, you must decide whether the property will be accessible only to code belonging to the window and its controls or to other event handlers and methods outside the window. Your choices are:

- **Public: Accessible from Anywhere:** A Public property can be read or set elsewhere in the project, not just within the window that "owns" the property. A Public property can be accessed by event handlers and methods outside the window using the "dot" notation: *windowName.propertyName*. Within the window's own event handlers and methods, you can access the property by its name.
- **Protected: Accessible from the current window and its subclasses:** A Protected property can be read or set only by the event handlers and methods of the window and the event handlers of the controls in the window. A Protected property is accessed by its name. Code outside the window that owns the property cannot "see" the property. If any code outside the window tries to access a Protected property, REAL Studio will display an informative error message.

- **Private: Accessible from the current window only:** A Private property behaves like a Protected property but it cannot be accessed by other windows that are subclassed from the current window. A window that is subclassed from another window inherits its Public and Protected properties. For information about creating subclasses, see the section “Understanding Subclasses” on page 533.

Declaring an Array as a Property

A property can be an array. For example, if you want to declare a four-element integer array of properties called `WindowParameters`, you would write:

`WindowParameters(3)` as Integer

in the Declaration area. You can also declare a property as an array with no elements and add elements later. In that case, you would declare `WindowParameters` using empty parentheses:

`WindowParameters()` as Integer

This denotes that `WindowParameters` is being declared as an array. You can use the `Append` method to add elements or the `Array` method to convert a list of items into an array to add elements.

To add a property to a window, do this:

- 1 **Display the Code Editor for the window by clicking on its tab or, if it has no tab, double-click its name in the Project Editor.**
- 2 **If necessary, click the Code Editor button in the window’s Editor Toolbar to access the Code Editor.**

Figure 266. The Code Editor button in the Window Editor toolbar.

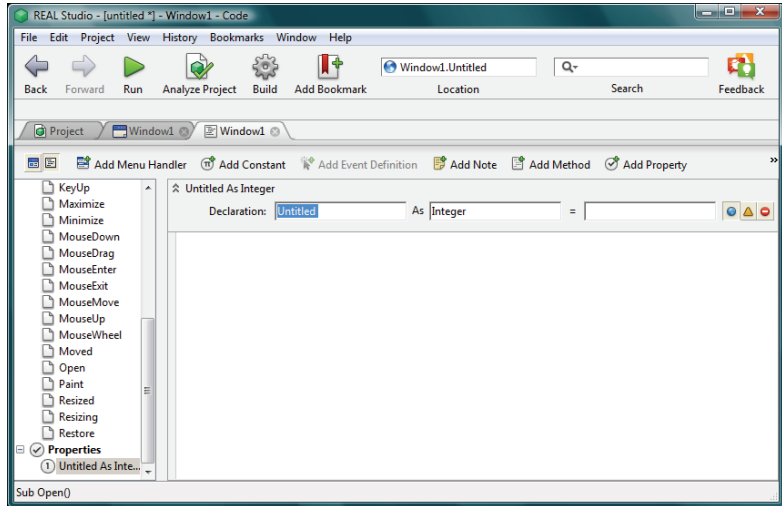


On Macintosh and Linux (top), the selected mode is highlighted; on Windows, it is depressed.

- 3 **Click the Add Property button or choose Project ► Add ► Property.**

A Property declaration area opens just above the code editing area.

Figure 267. The Property Declaration area.



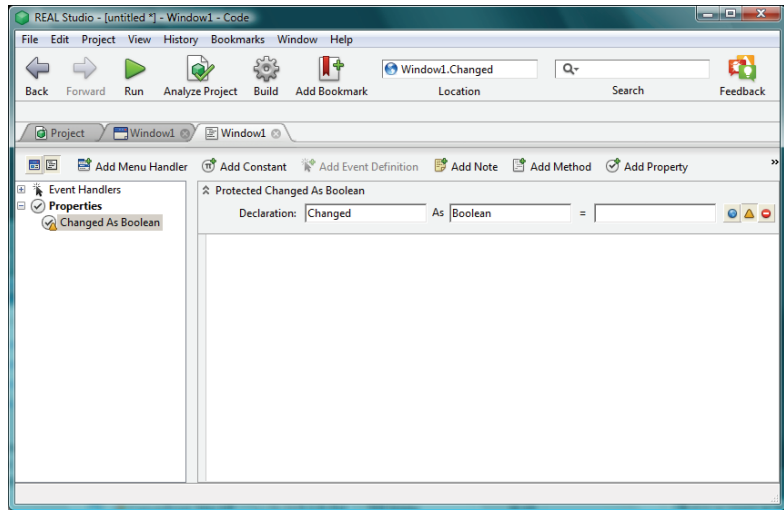
The Property Declaration area has three fields. They are for the name of the property, its data type, and its default value. The first two are required. If you do not provide a default value, the new property will take the default value for the data type that you choose. Strings have a default value of an empty string, numbers have a default value of zero, booleans have a default value of False, colors have a default value of black, and objects have a default value of Nil.

4 Fill in the Name and Data Type fields and, if desired, provide a default value.

For example, the Changed property would be entered with the name **Changed** and a data type of **Boolean**.

An example of a property declaration is shown in Figure 268.

Figure 268. A completed property declaration.



The data type can be a built-in data type, a built-in class, or a user-defined class. For example, the declaration `f as FolderItem` declares a new property, `f`, as an instance of the `FolderItem` class. The `FolderItem` is the REAL Studio class for representing files and folders (a.k.a., directories). You could use this property to refer to the document that is displayed in the window.

You will learn about classes in Chapter 10, “Creating Reusable Objects with Classes” on page 531.

If you want to use a built-in class as the data type, simply enter it into the data type field. The same is true for a user-defined class that is added to the project.

You can also create classes that belong to modules. A module is a stand-alone item that serves as a container for classes, class interfaces, methods, properties, constants, and other modules. For more information about module classes, see the section “Adding Classes to Modules” on page 384.

If you want to use a module class as the data type, you need to use the “dot” syntax to refer to it, i.e., `moduleName.className`. This refers to the class `className` in the module `moduleName`.

5 Choose a Scope for the property by clicking on one of the three Scope buttons.

Your choices, from left to right, are Public, Protected, and Private. See the section “The Scope of a Property” on page 315 for information on Scope.

Figure 269. The Scope buttons (Public selected).

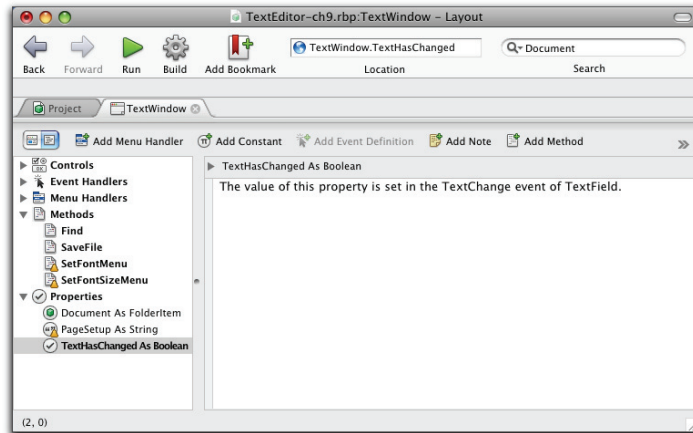


6 (Optional) In the Code Editor area, add notes and comments about the property.

The text entered into the Code Editor for a property is automatically non-executable, even if you write valid REAL Studio code. Add any comments you wish, including code samples.

You do not assign a value to the property in its Code Editor area. It is accessible from the event handlers and methods of the window, the window's controls, and, if the property is Public, event handlers and methods outside this window.

Figure 270. Property notes and comments in the Code Editor area.



Any text that you enter in the Code Editor for a property is not executable, even if it is code. When you add a comment to a property, the property declaration in the Browser area changes to boldface.

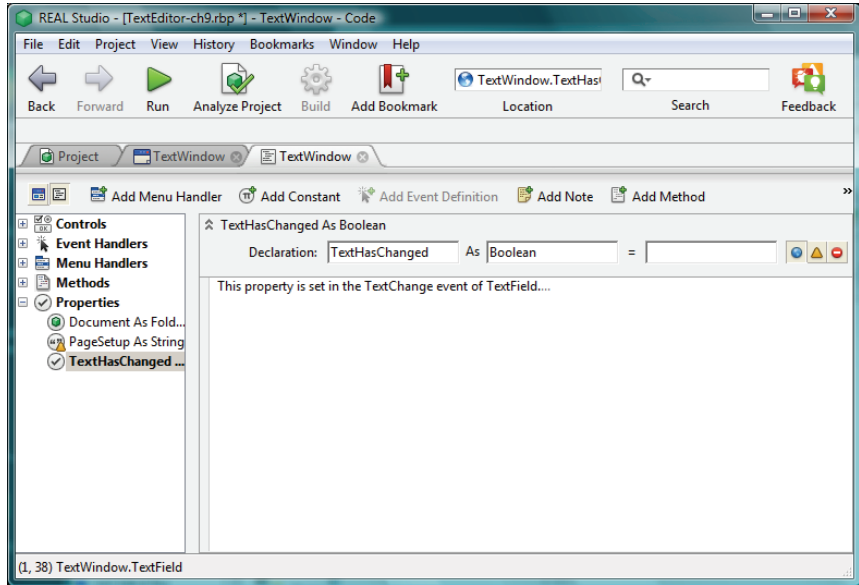
If a property is Protected or Private, the name of the property is preceded by either “Protected” or “Private.”

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

- 1 **Display the Code Editor for the window that contains the property.**
- 2 **In the Browser, expand the Properties category to display the list of properties for the window.**
- 3 **Click on the property to display the Property Declaration pane above the Code Editor.**
- 4 **If necessary, expand the Property Declaration pane by clicking its disclosure triangle (to the left of the property’s name).**



Figure 271. The Edit Property pane.



5 Make any necessary changes to the declaration.



To delete a property from a window, do this:

- 1 **Open the Code Editor for the window that contains the property.**
- 2 **In the Browser, expand the Properties category to display the list of properties for the window.**
- 3 **Click on the property you want to delete to select it.**
- 4 **Choose Edit ► Delete or right+click (Control-click on Macintosh) and choose Delete from the contextual menu.**

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

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

In the absence of the window name, the current window is assumed. If you wish, you can always use the built-in function `Self` to indicate that you are referring to a window's property. When used in a control's event handler, `Self` refers to the parent object, which would be the window in which the control is located. In other words, the statement:

```
Self.Title="My New Window"
```

would also refer to the parent window's Title property.

Computed Properties

A computed property is made up of a pair of methods called Get and Set. The Set method sets the value of the property and Get returns its value. You can implement either or both, making the computed property Read Only, Write Only, or Read/Write.

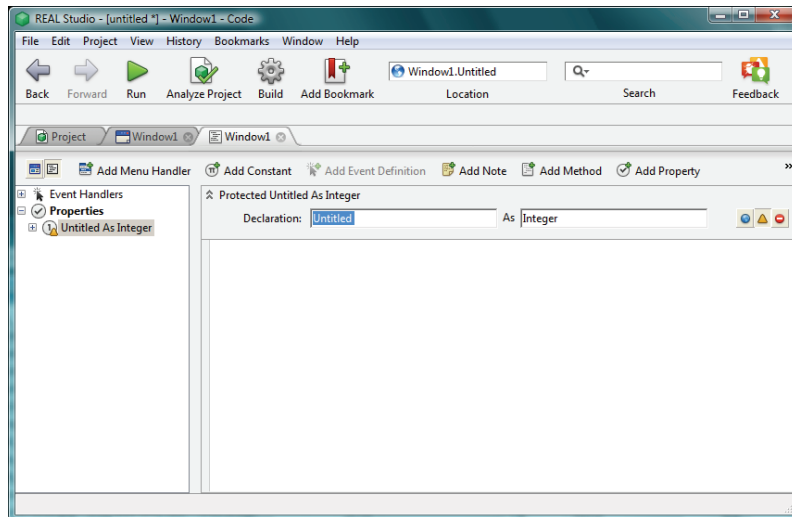
Computed properties can be declared as shared. See the following section, “Shared Methods and Properties” on page 322 for information on shared properties and methods.

To create a computed property, do this:

- 1 **Click the Add Computed Property button in the Code Editor toolbar or choose Project ► Add ► Computed Property.**

REAL Studio adds an untitled property to the window and displays a Property Declaration area above the Code Editor.

Figure 272. The Declaration Area for a Computed Property.



- 2 **Enter the Name and Data Type for the computed property and set its scope.**

Notice that the browser area shows that you can expand the computed property.

- 3 **Click the plus sign (Windows) or the disclosure triangle (Macintosh and Linux) to reveal the Get and Set methods that belong to the computed property.**

The Get method returns a value of the declared data type. The Set method is passed the value of the property. You do not have to implement both methods. You can make the computed property Read Only, Write Only, or Read/Write.

- 4 **Write either the Get or the Set method or implement both methods.**

Converting a Property to a Computed Property



You can have REAL Studio convert an existing “regular” property to a computed property. It will rename the existing property and create getter and setter methods that return the property’s value and set its value to the passed value.

To convert an existing property to a computed property, do this:

- 1 **In the Code Editor browser area, expand the Properties item (if needed) and right-click (Control-click on Macintosh) on the property that you want to convert.**
- 2 **Choose Convert to Computed Property from the contextual menu.**

REAL Studio creates the getter and setter methods for the property and enters the code for them.

Suppose the name of the property was `myProperty`, It renames it “`mmyProperty`”. The computed property is called “`myProperty`” and its Get method is:

```
Return mmyProperty
```

Its Set method has one parameter, **value as *DataType***, where *DataType* is the declared data type of `mmyProperty`. The code is:

```
mmyProperty = value
```

In other words, the computed property is all set up for the original “regular” property. It holds the value that the getter and setter methods manage.

For an example of a computed property, see the section “An Example Computed Property” on page 551.

Shared Methods and Properties

A shared method or property is like a ‘regular’ method or property, except it belongs to the class, not an instance of the class. A shared method or property can be accessed without instantiating an instance of the class or from any instance of the class.

In contrast, “regular” methods and properties are considered *instance* methods and properties. This means that they belong to a particular instance of the class.

The `Self` keyword is not available in a shared method or shared computed property and you cannot access instance methods or instance properties inside a shared method unless you are doing so via an instance.

To create a shared property or method, do this:

- 1 **Pull down the Project ► Add submenu and choose Shared Method, Shared Property, or Shared Computed Property.**

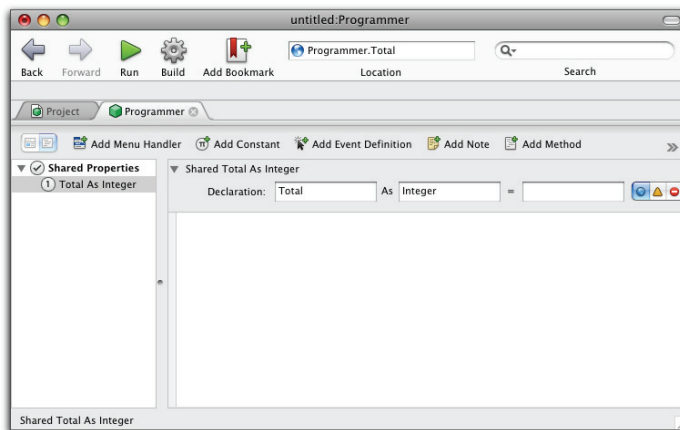
REAL Studio displays the declaration area for the type of item you chose. If the category of item does not already exist, it is added to the Code Editor browser area.

- 2 **Declare the property or method and set its scope the normal way.**
- 3 **If it is a method, write the method in the Code Editor.**



Figure 273 illustrates a shared property in the Code Editor.

Figure 273. A Shared property in the Code Editor.



Here is an example that illustrates the difference between instance and shared methods. Create a class, myClass, in the Project Window and create the following simple method of myClass:

```
Sub WelcomeMe(a As String)
    MsgBox a
```

If you create WelcomeMe as an instance method, you can only call it from an instance of myClass, e.g.

```
Dim greetMe as myClass
greetMe = New myClass
greetMe.WelcomeMe "Hello World"
```

If you create WelcomMe as a shared method, you can call WelcomeMe with the line:

```
myClass.WelcomeMe "Hello World"
```

You can also call a shared method or property from any instance of the class, just like an instance method or class. The key advantage of shared properties and methods is that you can share them among all the instances of the class. For example, if you are using an instance of a class to keep track of items (e.g., persons, merchandise, sales transactions) you can use a shared property as a counter. Each time you create an instance of the class, you can increment the value of the shared property in its constructor and decrement it in its destructor. When you access it, it will give you the current number of instances of the class.

For example, consider the example in the section “Using Classes in Your Projects” on page 577. The local variable “person” stores a reference to the instance of the custom class “Programmer”.

```
Dim person as Programmer  
person=New Programmer  
person.name="Jason"
```

Suppose the Programmer class contains a shared Integer property, Total, that gets incremented each time a Programmer instance is created. For example, give the Programmer class a Constructor of:

```
Programmer.Total=Programmer.Total+1
```

The Destructor is:

```
Programmer.Total=Programmer.Total-1
```

Each time a Programmer instance is created or destroyed, the value of the Total shared property is changed to reflect the current count. The value of Total can be accessed from any Programmer instance or from the Programmer class itself.

Adding Constants to Windows

A constant acts like a property but it holds a fixed value for its entire “life.” When you create a constant, you give it its value. You can read the constant’s value in your code, but you cannot use an assignment statement to change the value of a constant.

You can create constants in REAL Studio for windows, modules, and classes. You can also create a local constant inside any method you write. For information about local constants, see the section “Constants” on page 236.

The Scope of Window Constants

A constant for a window has a Scope, just like properties. Scope determines which parts of your application can “see” the constant and read its value.

A constant added to a window can be Public, Protected, or Private in Scope.

- **Public:** The constant can be read by all of the code in your project. To read the value of a Public constant inside the window that owns it, you simply use its name, *constantName*. To read a Public constant from another object’s method or event handler outside the window, you use the “dot” notation, i.e., *windowName.constantName*. For example if the window “SaveChangesWindow” has a Public constant called “Accept”, you would refer to it as “Accept” from any method or event handler belonging to SaveChangesWindow. However, code belonging to other windows, modules or classes that don’t belong to SaveChangesWindow refer to the constant as “SaveChangesWindow.Accept”.
- **Protected:** The constant can be read only by code owned by the window and its controls. To read the value of a Protected constant, refer to it by name. For example,

a StaticText control that gets its Text property from a window constant can refer to it by `StaticText1.Text=constantName`. If another window, a class, or a module tries to access a Protected constant, REAL Studio will display an informative error message.

- **Private:** A Private constant is like a Protected constant except that any windows subclassed from this window will not be able to access a Private constant. Protected and Public constants are inherited by windows subclassed from the current window. For information about creating subclasses, see the section “Understanding Subclasses” on page 533.

Localizing an Application using Constants

If you need to build separate copies of your application for different platforms and language combinations, you should use constants instead of literal text strings for all your interface items that present text to the user. REAL Studio allows you to define different values for a constant for every platform and language combination that you need.

To do this, you use the Localization table in the New Constant declaration area to enter different values for the constant for every platform/language combination that you support. Often these constants are given Global scope—so that they can be referred to by name only—and that is possible only in a module. If you put all of your localization constants in one place it will be easier to maintain them.

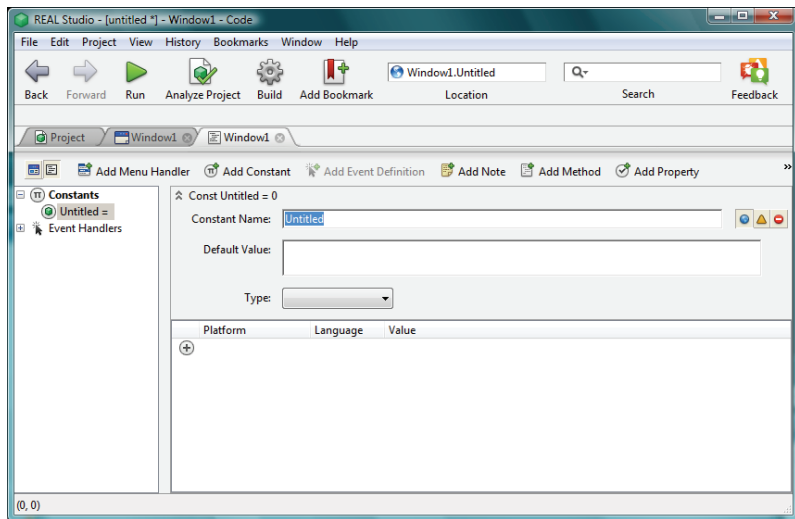
The process is described in detail in the chapter on modules, but you can also use the same process with constants for windows. If you want to use the Localization table for window constants, follow the steps in the section “Adding a Constant to a Module” on page 376.



To add a constant to a window, do this:

- 1 Display the Code Editor for the window.**
- 2 Click the Add Constant button in the Code Editor toolbar or choose Project ► Add ► Constant.**

The Add Constant declaration area appears above the Code Editor area (Figure 274).

Figure 274. The Add Constant declaration area.

3 Enter the name of the constant, its value, and its data type.

When you enter a value, REAL Studio guesses the data type and sets the Type drop-down list accordingly. Any number sets the data type to Number, a string other than “True” or “False” sets it to String, and a hex value that starts with “&c” sets it to “Color.” Entering “True” or “False” sets the Type to Boolean.

If its guess is incorrect, set its data type by selecting a data type from the Type drop-down list, Number, String, Boolean, or Color. The data type of the constant will be indicated by the small icon to the left of the constant’s name in the browser area.

If you chose Color, a color patch appears to the right of the Type drop-down list with the default color of black. Click it to display the Color Picker to choose the color constant. When you choose a color, its value in hexadecimal is added to the Default Value area.

If you chose string, a “Dynamic” checkbox appears to the right of the Type drop-down list. Dynamic constants are used to facilitate localization. For more information about Dynamic constants, see the section “Dynamic Constants” on page 379.

4 Set the Scope of the constant by clicking one of the three Scope buttons.

Your choices for a constant belonging to a window are Public, Protected, and Private, from left to right.

Figure 275. The Scope buttons.

5 (Optional) Use the Localization table at the bottom of the pane to define different values for the constant for different platforms and language combinations.

See the section “Using Constants to Localize your Application” on page 378 for details on the Localization table.

Your application can then access the constant according to the Scope that has been assigned to it. When you want to use the constant as the value of a property in the Properties pane, precede its name with the number sign, #. If you’ve defined different values for different platforms or languages, the correct value will be used automatically for the version of the application built for that combination of platform and language. You set the default region and language with the Build Settings dialog box. For more information, see the section “Default Language” on page 707.

Converting a Literal to a Constant

As a shortcut, you can convert a literal in your code to a constant. When you do so, REAL Studio automatically creates a window constant and replaces the literal with the name of the constant. By default, all converted literals are given a data type of String. If needed, you can change the scope and the data type of the new constant.

To create a constant from a literal, do this:

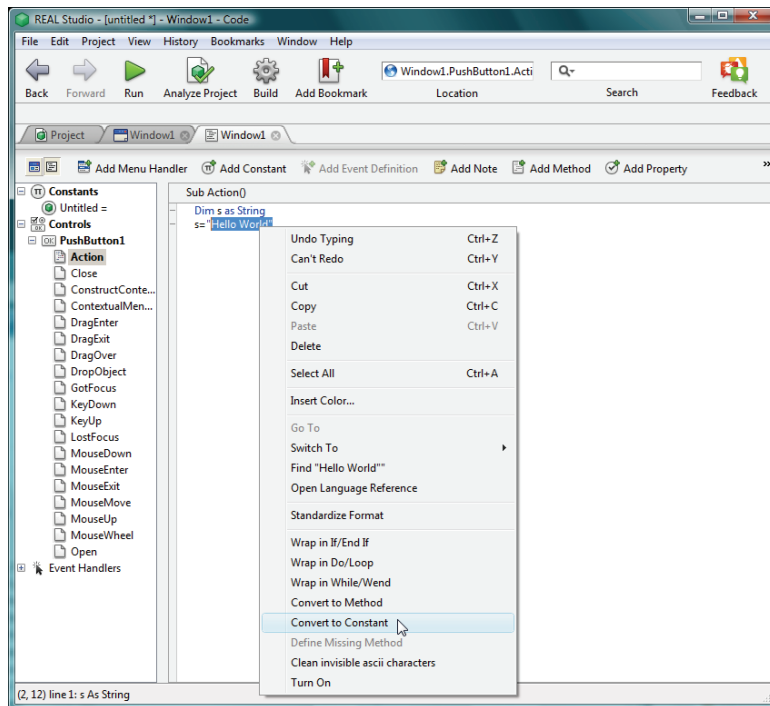
1 Select the literal to be converted in the Code Editor.

If it is a string literal, be sure to include the quote marks.

2 Right+click (Command-click on Macintosh) and choose Convert to Constant from the contextual menu.

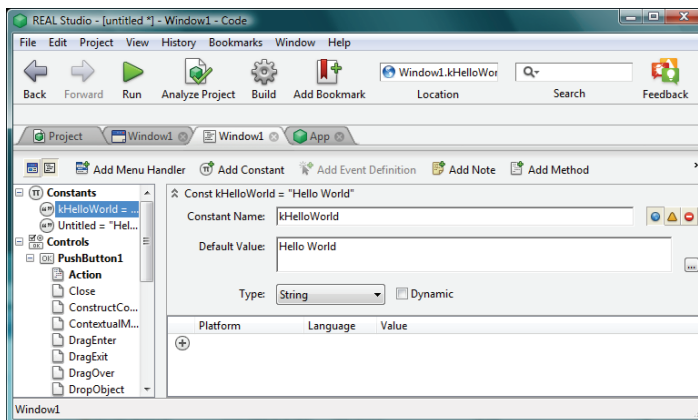
In the following screen shot, the literal “Hello World” is being converted to a constant.

Figure 276. Converting a literal to a window constant.



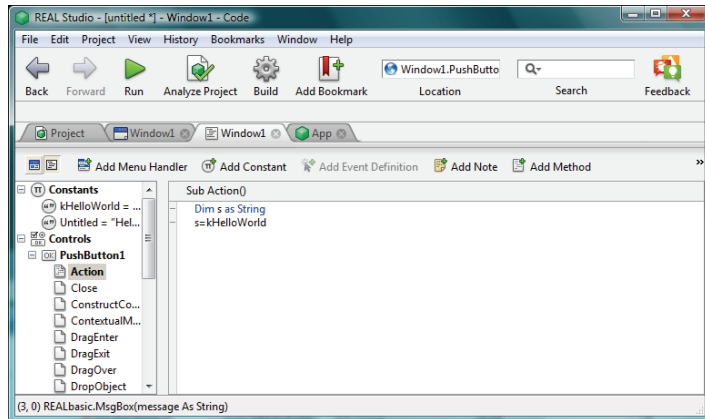
REAL Studio replaces the literal with the name of the constant and adds the new constant to the Constants group in the window's Code Editor. It uses the literal text as the basis of the constant's name and precedes it with the letter "k."

Figure 277. The converted constant defined in the Code Editor.



In the Code Editor, the literal has been replaced by the constant name, "kHelloWorld."

Figure 278. The constant in the Code Editor.



Adding Methods to Windows

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

Passing Parameters to Methods

You can pass parameters to methods. Parameters are variables that the method uses internally. When you call the method, you pass values to the method via its parameters. The method can then use these values. Optionally, parameters return new values.

Parameters are declared the same way that properties are (e.g., “Age as Integer”), except that you write out the declaration instead of entering the name and data type in separate fields. To indicate that the method will require parameters when it is called, name and declare the data type of each parameter. If the method requires multiple parameters, the parameter definitions should be separated by commas. For example, if a method requires an integer and a string, you would declare two parameters in the following manner:

```
myInt as Integer, myString as String
```

Parameters are treated as local variables inside the method. This means the method can change the value of a parameter—just as if it was created as a local variable inside the method. In this example, the name “myInt” becomes the name of the

local Integer variable inside the method and “myString” becomes the name of the local String variable inside the method.

Passing Arrays A parameter that you pass to a method can be an array. To pass an array, you follow the name of the array with empty parentheses when you declare the parameter. For example, if you need to pass an array of real numbers, you would declare the array in the following manner:

aNums() as Double

Since the declaration does not specify the number of elements of the array, the method will accept arrays of any size. You can figure out the number of elements in a particular array by calling the Ubound function inside the method. When you call the method, you simply pass the name of the array you want to pass, without any parentheses.

You can pass multi-dimensional arrays without specifying the number of elements in each dimension, but you need to indicate the number of dimensions. Do this by placing one fewer commas in the parentheses than dimensions. For example, if aNames were a two-dimensional String array, you would declare the array in the following manner:

aNames(,) as String

Finally, there is one more way of passing a series of values to a method. You can use the ParamArray keyword in the parameter declaration. The ParamArray keyword signifies that any number of values of the specified data type will be passed as parameters to the method. For example, if the method will accept a list of integers, you would declare one Integer parameter using the ParamArray keyword:

ParamArray nums as Integer

When you call the method, you pass a list of integers, with the integers separated by commas (just as if they had been declared as separate parameters). For example, you can write:

myMethod(5,7,2,10)

Inside the method, “nums” is an array rather than an integer variable. You can process the values as an array.

Returning Values from Methods

A method can also return a value. A method that returns a value is called a *function*. You create a function using the same process as you use for defining a method; the only difference is that you declare the data type of the value being returned (see Figure 279 on page 332).

The value that a function returns can be an array or a single value. If you want to return a single value, you specify the data type of the value. If you want to return an array, write empty parentheses after the data type. For example, if you want to return an array of integers, write **Integer ()** as the Return Type. If you need to return a multi-dimensional array, place one fewer commas in the parentheses than dimensions. For example, to return a two-dimensional array of Doubles, write **Double(,)** as the Return Type.

When you declare a Return Type, your method needs to use the keyword **Return** to indicate the value to be returned. For example, if the method takes an array of numbers, adds them up, and returns the sum into a local variable called “Total”, you would need to include the line:

```
Return Total
```

in the function. For more information, see the section, “An Example Method” on page 335.

The Scope of Methods

When you create a method, you indicate how much of your application can “see” the method. You can choose to make the method available only to code belonging to the window that “owns” the method or make it available globally, so that any method or event handler can call the window’s method. This is called the *scope* of the method. For windows, you have the following choices:

- **Public:** The method can be called from any event handler or method in the application. For example, you would use this if you want a menu handler that belongs to App class to be able to call the method. To call a Public method outside the window that “owns” it, use the syntax *windowName.methodName*. In the window’s own event handlers, methods, and controls, you can call it by referring to its name only, *methodName*.
- **Protected:** The method can only be called by the window’s event handlers and methods, its controls, and windows subclassed from this window. You would choose Protected if the method performs operations on information in this window and you do not want these operations to be started from outside the window itself. Call a protected method by its name only, *methodName*. If you try to call a Protected method from outside its window, REAL Studio will display an informative error message.
- **Private:** The method can be called only by the window’s event handlers and methods and controls in the window. It cannot be called by other windows that are subclassed from this window. Windows that are subclassed from this window automatically inherit all its Public and Protected methods. Call a private method by its name only, *methodName*. If you try to call a Private method from outside its window, REAL Studio will display an informative error message.

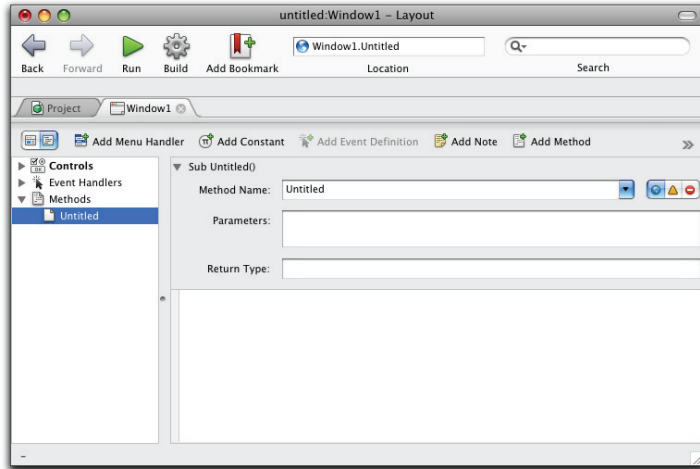


To add a method to a window, do this:

- 1 Open the Code Editor for the window.**
- 2 Click the Add Method button or choose Project ► Add ► Method.**

The Method declaration area appears above the Code Editor area.

Figure 279. The Method Declaration area.



- 3 Enter the name of the method, its parameters, and, if it will return a value, its return type.**

When naming the method, be sure not to use a reserved word. A reserved word is a term that REAL Studio uses elsewhere (see “Reserved Words” on page 240). Use the Parameters and Return Type areas to declare the data type of each parameter and the value to be returned (if the method will return a value).

For example, if you were going to pass the value of the TextChanged property declared in Figure 271 on page 320, you would write, “TextChanged as Boolean” in the Parameters area. If you want to pass several parameters, separate each parameter declaration by a comma.

The Return Type is the data type of the value to be returned if your method will be returning a value. The pop-up menu to the right of the Return Type field has a list of common data types, but any valid data type can be defined in the Return Type field.

There are several advanced options available in the parameter declarations area. For more information, see the sections “Passing a Parameter by Value or Reference” on page 336, “Setting Default Values for a Parameter” on page 338, “Setter Methods” on page 340, “Accessing Items of Other Windows” on page 342, and “Constructors and Destructors” on page 341.

- 4 Select the Scope of the method by clicking a Scope icon.**

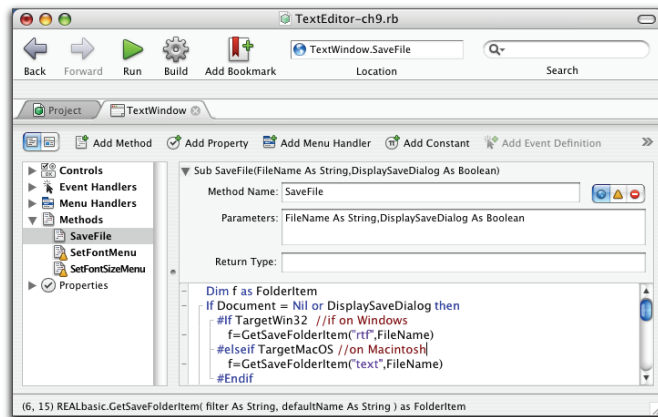
Your choices for a method belonging to a window, from left to right, are Public, Protected, and Private.

Figure 280. The Scope icons.



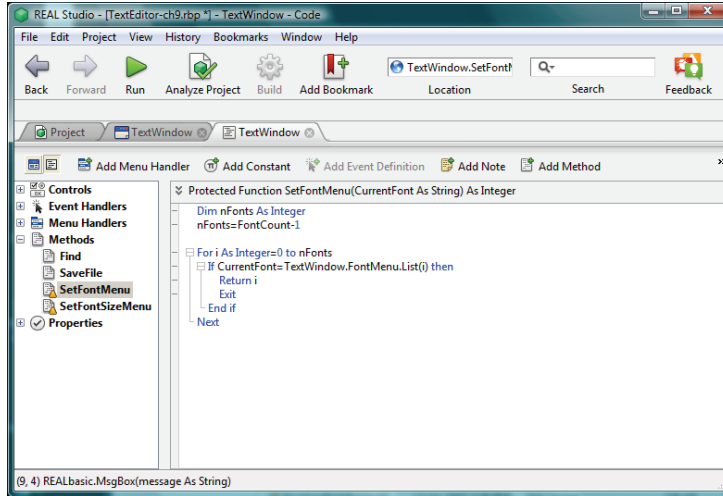
See the previous section, “The Scope of Methods” on page 331 for information on Scope. A completed method declaration looks like Figure 281.

Figure 281. The SaveFile method declaration.



If you have declared the method as Private or Protected, the Sub or Function statement indicates the scope of the method, as shown in Figure 282. If it is a Public method or function, then the Sub or Function statement just says “Sub” or “Function.”

Figure 282. A Protected method in the Code Editor.



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



- 1 **Open the Code Editor for the window that contains the method.**
- 2 **In the Browser, expand the Methods category to display the list of methods for the window.**
- 3 **Click on the method to display it in the Code Editor pane.**
- 4 **If necessary, expand the Method declaration area above the code by clicking its plus sign (Windows) or disclosure triangle (Macintosh or Linux) to the left of the method declaration.**
- 5 **Edit the name, parameters, return type, or scope as needed.**

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



- 1 **Open the Code Editor for the window that contains the method.**
- 2 **In the Browser, expand the Methods category to display the list of methods for the window.**
- 3 **Click on the method you want to delete to select it.**
- 4 **Choose Edit ► Delete or right+click (Control-click on Macintosh) and choose Delete... from the contextual menu.**

Dynamic Method Creation

When you are coding your application, you will sometimes reach a point where you decide that you need call a method that you haven't written yet! What you can do is write pseudo-code in which you do a proper call to the method and then write the method later. This is how Dynamic Method creation works.

In the midst of your code, you can type the method that you intend to write and pass the parameters that it requires. Then select the complete call to the method and right+click (Command-click on Macintosh) and choose Define Missing Method from the contextual method. REAL Studio will use the declaration to create the new method with the parameters that you specify. It is best to declare the parameters' data types so that REAL Studio knows how to write the declaration.

To use Dynamic Method Creation, do this:



1 Write pseudo-code for your new method where you wish to call it.

For example, you can write:

```
Dim i as Integer
Dim s as String
myNewMethod(i,s) //undefined method
```

2 Select “myNewMethod(i,s)” and right+click (Command-click on Macintosh) to display the contextual menu.

3 Choose Define Missing Method.

REAL Studio creates a new method according to the pseudo-code you’ve written.

4 Write the code for the new method.

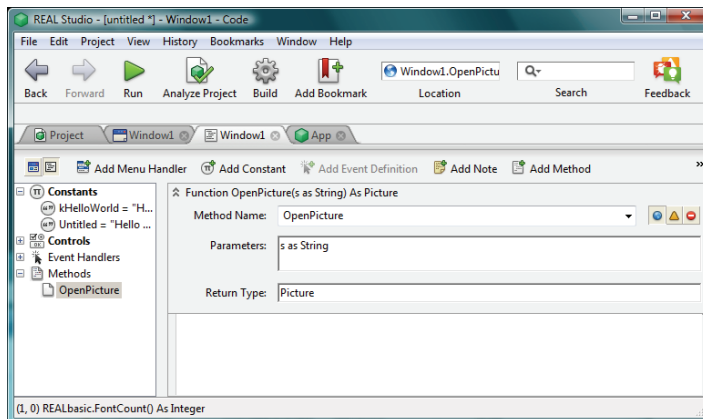
An Example Method

The following example shows how to write a method and call it from any event handler in the window.

Suppose your application needs to allow the user to open pictures and assign the pictures to various objects. You can use a simple method to open the file and return the picture. The event that calls the method can then assign the object to an interface object.

The method will take the name of the file to open as its parameter and return the picture object. Therefore, you declare it as a function, as shown in Figure 283.

Figure 283. The openPicture declaration.



The method consists of the following code (the function declaration is editable in the Method declaration area):

```
Function openPicture (s as String) as Picture
    Dim f as FolderItem
    Dim p as picture
    f=GetFolderItem(s)
    if f.exists then
        p=f.openAsPicture
    else
        MsgBox "Invalid file!"
    end if
    Return p
```

With this method, you can then open a picture file by simply passing it the path to the file as a string. For example, the following line of code displays an image in an ImageWell control by assigning the picture to the Image property of the ImageWell:

```
ImageWell1.image=openPicture("BackgroundImage")
```

Passing a Parameter by Value or Reference

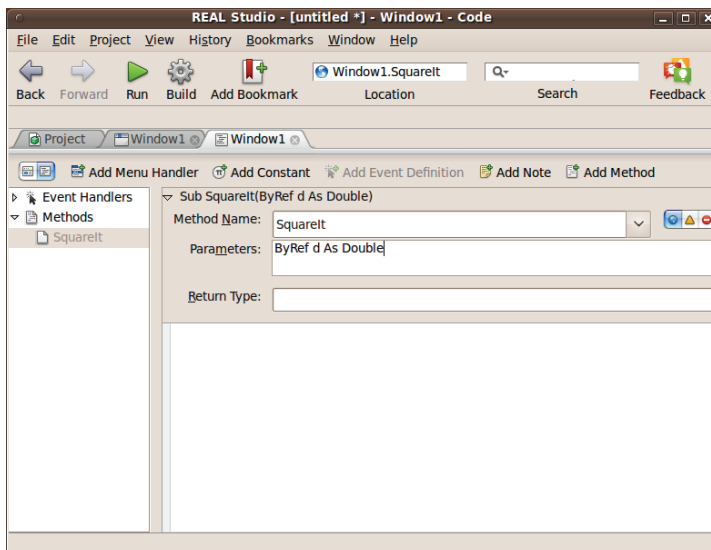
When you define the method's parameters, you can choose between two types of parameter passing: passing by value and passing by reference.

The default is passing parameters by value. With this option, the method receives the value of the parameter and can perform operations on it. The parameter is treated as a local variable within the method. You don't have to do anything special to pass a parameter by value. Passing by value is illustrated in the previous example: a string parameter that contains a pathname is passed to the method and the method uses the passed parameter as a parameter in another function call.

When you use passing by reference, you instead pass a reference to the parameter (i.e., a pointer to the parameter's value). When the method receives a parameter passed by reference, it is free to change the value of the parameter and return the changed value to you after completing the call to the method.

For example, the following method declaration uses one parameter that is passed by reference.

Figure 284. Passing a parameter by reference.



The ByRef keyword in the parameter declaration indicates that the parameter is being passed by reference rather than by value. The method itself consists only of the line:

```
d=d*d
```

That is, the method changes the value of the parameter being passed. Such a method can be called like this:

```
Sub Action ()
  Dim Label as String
  Dim i as Double
  If TextField1.text <> "" then
    i=Val(TextField1.text)
    Label="The square of "+TextField1.text+" is "
    SquareIt(i) //passed by reference
    StaticText1.Text=Label+Str(i)+". "
  Else //no value entered
    Beep
    MsgBox "Please enter a number into the text box."
  End if
```

When you run this method, you will see that the value of *i* changes after the call to SquareIt. This is not possible when you pass by value.

You can also pass arrays by reference using the same syntax. Simply precede the name of the array with the ByRef keyword. When you pass an array ByRef, the method can change all or some elements of the array.

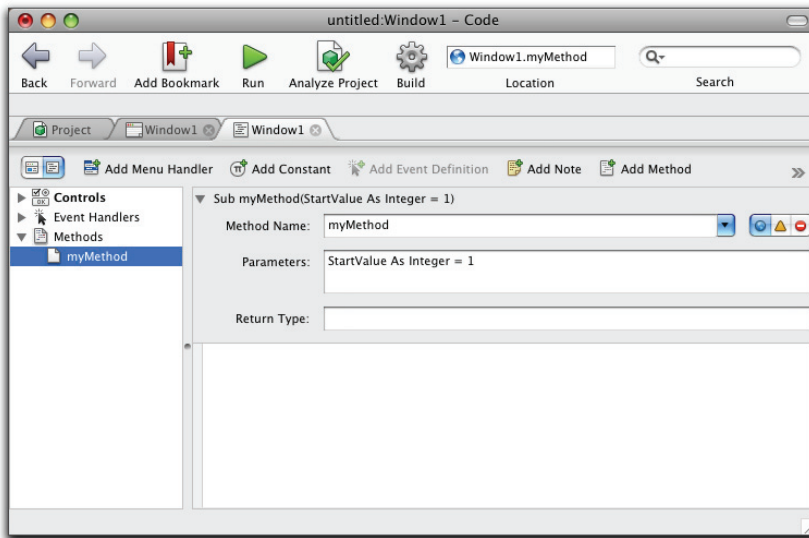
Setting Default Values for a Parameter

When you declare the parameters to a method, you can optionally provide a default value for each parameter. You do so by making the declaration an assignment statement in the Method declaration area. For example, if you want to declare the parameter “StartValue” as an Integer and give it value of 1, you would write

```
StartValue as Integer = 1
```

in the Parameters area of the Method declaration area. This is shown in Figure 285.

Figure 285. Declaring an Integer parameter.



When you call the method, any parameter that is provided with a default value in the declaration statement becomes optional.

When you don't provide a default value for the parameter, you must pass an integer value to myMethod whenever you call it. When a default value is provided, you can omit the parameter in the call. When you want to use the default value, simply omit the parameter in the call:

```
myMethod
```

In this case, the default value of 1 will be used.

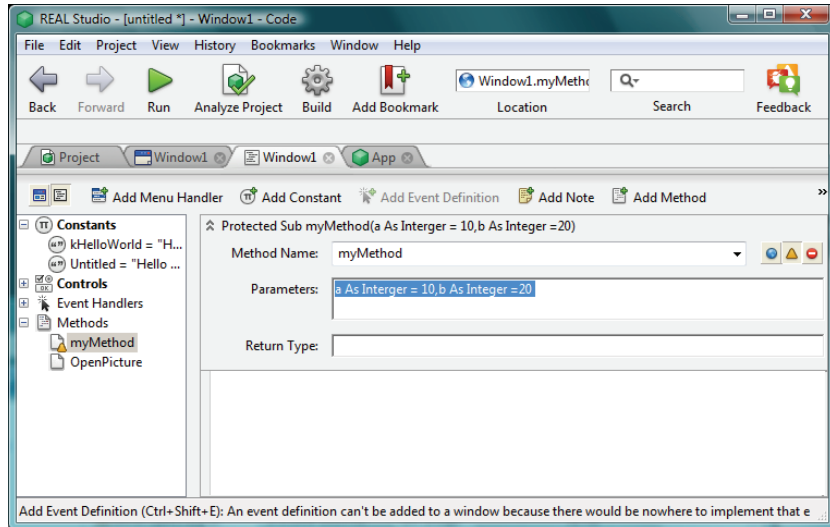
If you want to use another value, pass it as a parameter in the normal way. For example,

```
myMethod(10)
```

In this case, myMethod will use the passed value,10, and ignore the default.

You can provide default values for more than one parameter. For example, the following declaration is valid.

Figure 286. Assigning default values to two parameters.



In this case, you can call the method with the statement:

```
myMethod
```

The two default values will be used. If you want to override the default value for the first parameter, pass one value. For example:

```
myMethod(100)
```

passes the value of 100 to the parameter `a`. If you want to override only the value of the second parameter, you must pass two parameters. Just pass the default value for the first parameter.

When you assign a default value when you declare a parameter, you can use a literal value (as shown here) or a constant or an enumeration. See the section “Constants” on page 236 and the section “Adding an Enumeration to a Module” on page 397. A constant is like a variable except that it is assigned its value when it is created and retains its value for its life. An Enumeration holds a list of constant values.

For example, suppose you define a global constant in a module, `InitialValue`. You can use it as a default value for a parameter like this:

```
a As Integer = InitialValue
```

Similarly, you can use an enum as a default value. Suppose you have created an enum in a module called `SecurityLevel`, with values `None`, `Minimum`, `Maximum`, and

Forced. You can then assign the default value of a parameter using one of the enum values:

```
a As Integer = SecurityLevel.Forced
```

Making a Parameter Optional

If you need to specify that one of the parameters in the method is optional without giving it a default value, use the “Optional” keyword. This modifier precedes the parameter name. An optional parameter does not take on a default value in the called method. If the caller omits this parameter, it will receive the standard default value for its data type.

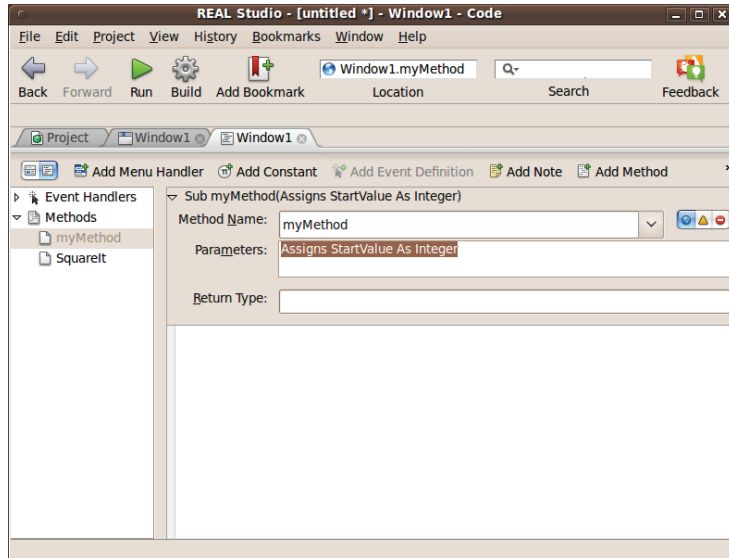
Setter Methods

When you pass a value to a method, you have the option of using the assignment operator. For example, you can pass the value of 10 to a method using the statement:

```
myMethod=10
```

However, the assignment operator can be used only if you use the “Assigns” keyword when you declare the method. In Figure 287, the “Assigns” keyword is used. This syntax must be used whenever you want to use the assignment operator when you pass a value.

Figure 287. Using the “Assigns” keyword in a Method Declaration.

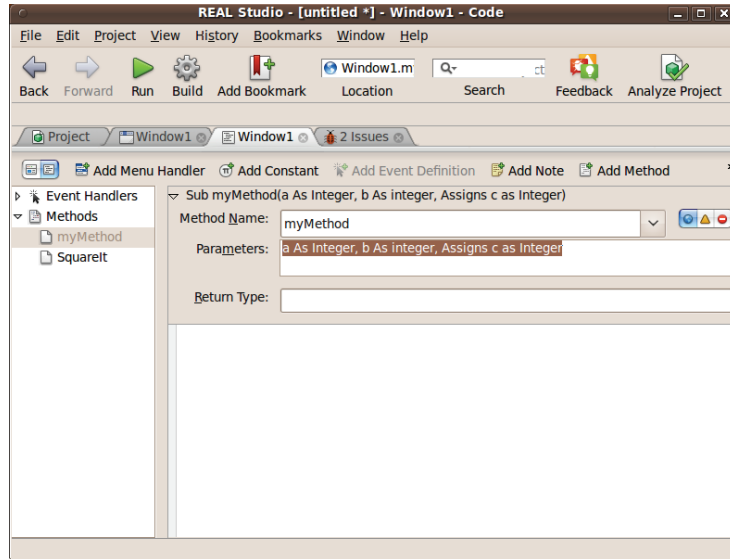


If myMethod is declared as shown in Figure 287, then you can pass a value in the following way:

```
myMethod = 10
```

You can use the “Assigns” keyword with methods that take more than one parameter. When you do so, you must use the Assigns keyword with the last parameter. You can use “Assigns” for only one parameter per method. For example, the following method declaration is valid:

Figure 288. Using “Assigns” with more than one parameter.



To call this method, use the following syntax:

```
myMethod(5,4)=10
```

When you use the Assigns keyword as shown in Figure 288, you cannot use the ‘normal’ syntax shown below:

```
myMethod(5,4,10) //doesn't work
```

Constructors and Destructors

A *constructor* is a method that runs automatically when the object that owns it is first created. You usually use a constructor to do some type of initialization of the object. A *destructor* is a method that runs when the object is destroyed or goes out of scope.

For information on constructors and destructors, see the section “Constructors and Destructors” on page 569.

Attributes

Attributes are compile-time properties that can be added to both Project and Code Editor items. An attribute consists of an identifier as the Name and an optional value. For Code Editor items, attributes are created via the Attributes Editor within the Code Editor.

The names “Deprecated” and “Hidden” are reserved for internal use and should not be used as an attribute’s Name.

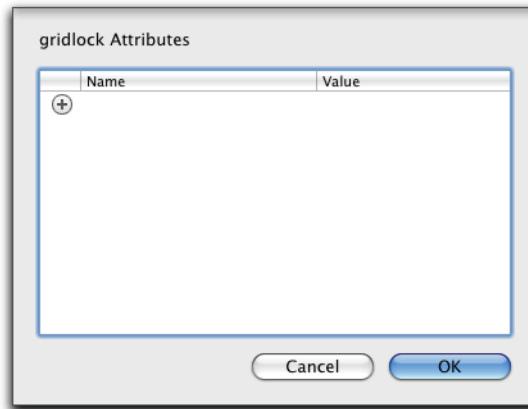
Each Code Editor item has a contextual menu, from which you can add, modify, or delete the item's attributes.

To add an attribute, do this:

- 1 **Right+Click (Command-Click on Macintosh) on the Code Editor item and choose Attributes... from the contextual menu.**

The Attributes Editor appears.

Figure 289. The Attributes Editor



- 2 **To add an attribute, click on the Plus sign.**

An entry area for the attribute appears. The first column is for the (mandatory) attribute name; the second column is for the optional attribute value.

- 3 **Enter the name and, if desired, its value.**

- 4 **Repeat this process for each additional attribute.**

- 5 **Click OK to put away the Attributes Editor.**

To modify or delete an attribute, do this:

- **To modify an existing attribute, click twice in its name field to get an insertion point and replace the text as desired.**
- **To delete an existing attribute, click on its minus sign.**

Accessing an Attribute

Attributes can be accessed at runtime via the Introspection system. Use the `AttributeInfo` class. See the example for the `AttributeInfo` class in the *Language Reference*.

Accessing Items of Other Windows

Items in other windows can be accessed using the window name followed by a dot and the control, method, or property name, i.e., *windowName.itemName*.

Of course, this is true only if the Scope of the item you want to access is Public. If the item's Scope has been declared Protected or Private, you cannot access it from another window.

All controls have a Scope property that can be set to either Public or Private. A control's Scope must be set in the IDE, not in code. If a control's Scope is set to Public, then it can be accessed from outside its parent window in the manner described in this section. Setting a control's Scope property to Private makes it impossible to access the control outside its own window. You would do this to prevent code outside the window from inadvertently changing one of a control's properties or calling one of its methods. If the window and its controls is designed to operate independently of the rest of the application, then you can make its controls Private.

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

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

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

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

In the case of controls, the control name can then be followed by one of its property names. For example, suppose a button in Window1 will, when clicked, place the text "Hello World" in the Text property of a Public control called StaticText1 in another window, Window2. The syntax is:

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

That is, the syntax is *windowname.controlname.propertyname*. If you don't want code that is outside Window2 to change StaticText1's Text property, you should set its Scope to Private in the IDE.

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

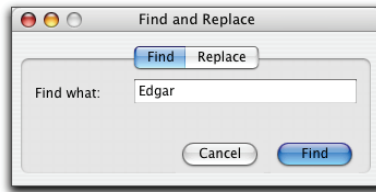
If there can be more than one instance of the target window open, you need to store a reference to that window somewhere so your code will know which instance of the window you are referring to. Where you store this reference depends on how your application works. Suppose you have many instances of a window named "DocWindow" open that displays the contents of a text document. A button in this window opens a Find window that lets the user enter a value he wishes to search for in that instance of DocWindow. Since there can be many DocWindows open, you will need to store a reference to the specific instance of the DocWindow that opens

the Find window in a property of the Find window. You do this by adding a property (let's call it "Target") to the Find window of type DocWindow. When the Find button in an instance of the DocWindow opens the Find window, it can store a reference to the DocWindow in that property. Assuming your application only allows one Find window to be open at a time (perhaps by making the Find window modal), the syntax looks like this:

```
FindWindow.target=Self
```

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

Figure 290. An Example Find window.



In Figure 290 the FindWindow has a TextField named "FindValue" where the user types what he wishes to find. Let's also assume that the DocWindow has a method called "Find" that, when passed a value, locates that value (if it exists) in a TextField in the DocWindow and highlights the value found. When the user clicks the Find button in the FindWindow, the Find button's Action event handler calls the Find method of the instance of the DocWindow that opened the FindWindow. It does this using the FindWindow's target property and the following syntax:

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

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

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

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

In this example, the Target property of the FindWindow is defined as being of type DocWindow. However, if the FindWindow needs to reference more than one

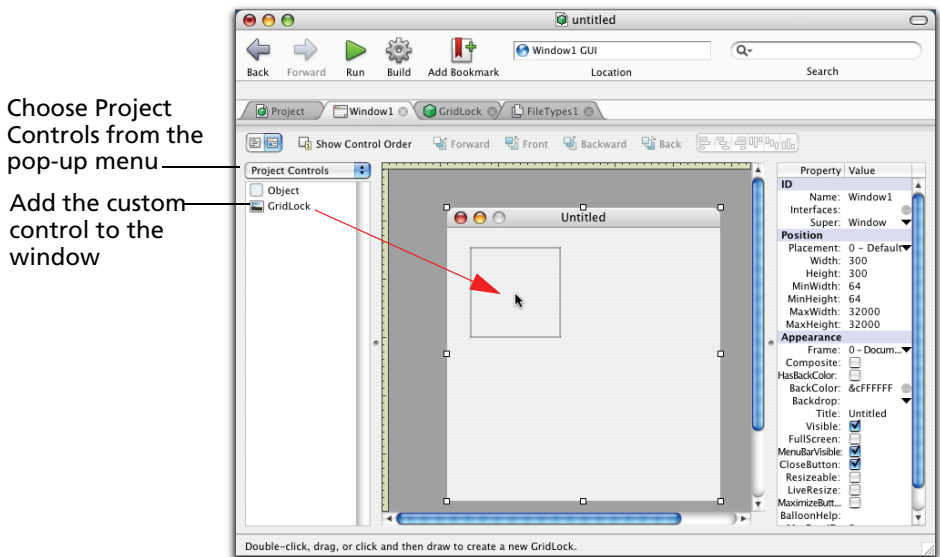
window class, you would define the Target property as type Window to be more generic. This allows the Target property to store a reference to an instance of any kind of window rather than just an instance of DocWindow. However, it also makes the code less readable because it is not clear which windows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

Controls

Controls that appear inside a window can have their own code to respond to events directed to them. You cannot add methods or properties to the Built-in controls. However, you can create controls that have custom properties, methods, event definitions, and even menu handlers by creating new classes based on Built-in controls.

To do this, you first create a new class in the Project Editor that is based on one of the controls. You declare its Super class to be a control and then add custom methods, properties, and/or event definitions to the new class. To add an instance of the custom class to a window, use the Controls pop-up menu in the Window Editor to switch to the Project controls and then add the custom control to the window in the usual way.

Figure 291. Adding a custom control based on the Canvas class to a window.



This action adds an instance of the custom control class to a window. See the section, “Understanding Subclasses” on page 533 for information on custom classes based on controls.

Events

Controls, like windows, receive events and have event handlers to respond to the events they receive. For every event a control receives that you can respond to, there is a corresponding event handler. The *Language Reference* contains the complete list of events that windows and controls can respond to. When reading the *Language*

Reference, keep in mind the fact that each control inherits properties and events from its parent. For example, most controls are subclassed from the RectControl class, and therefore have all the RectControl class’s events and properties.

Creating New Instances of Controls On The Fly

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

Suppose you need to clone a PushButton that is already in the window, named PushButton1.

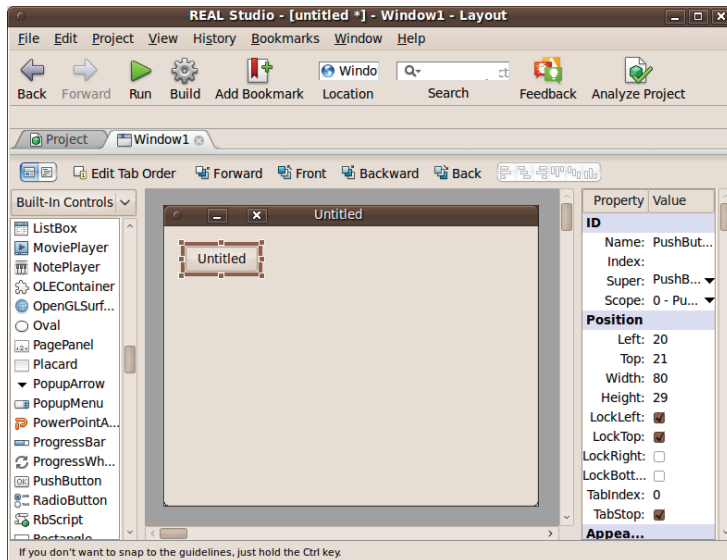


To create a new PushButton control on the fly via code, do this:

- 1 In the Properties pane for the PushButton, set its Index property to zero.**

When the new controls are created while the application is running, they will, be elements of a control array.

Figure 292. The template control and its properties.



In this example, a new PushButton will be created when the user clicks on the original PushButton.

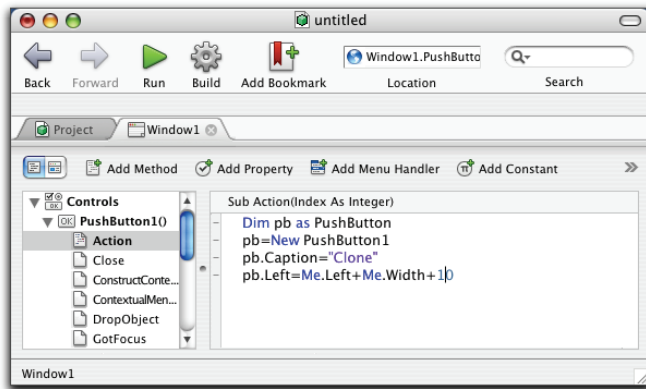
- 2 In the Action event of PushButton1, dimension a variable of type PushButton.
- 3 Assign the variable a reference to a new control using the New operator and pass it the name of the template control.

This example shows a new PushButton being created using an existing PushButton named PushButton1 as a template. When the new control is created, it is moved to the right of the template control:

```
Dim pb as PushButton
pb= New PushButton1 //clone of PushButton1
pb.Caption="Clone" //change caption just to be clear about this
pb.Left=me.Left+me.Width+10 //move it to the right
```

The Code Editor for PushButton1 should look like this:

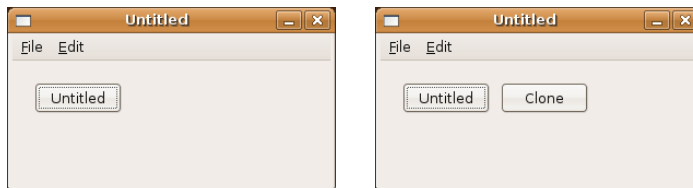
Figure 293. Action event that clones PushButton1.



- 4 Click the Run button.
- 5 When the application launches, click the “Original” button.

REAL Studio creates a new PushButton to the right of Original. You can see it’s the variable “pb” from its Caption property. Your screen should show the sequence shown in Figure 294.

Figure 294. The test application before and after creating the cloned PushButton.



If you click “Clone,” you will create another clone to the right of the first two, and so on.

Since any new control you create shares the same code as the template control, you may need to be able to differentiate between them in your code. You use the index property of the control to identify which control was clicked. For more information on using the Index parameter, see the following section, “Sharing Code Among An Array of Controls” on page 348.

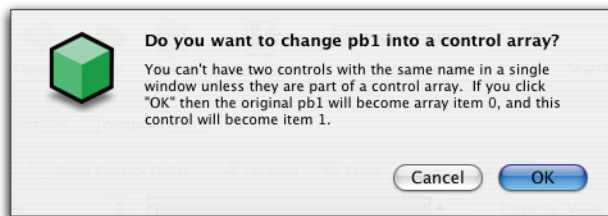
If your code needs to create different kinds of controls and store the reference to the new control in one variable, you can dimension the variable as being of the type of object that all the possible controls you might be creating have in common. For example, if a variable can contain a reference to a new `RadioButton` or a new `CheckBox`, the variable can be dimensioned as a `RectControl` because both `RadioButtons` and `CheckBoxes` are derived from the `RectControl` class. Keep in mind, however, since the variable is a `RectControl`, the properties specific to a `RadioButton` or `CheckBox` will not be accessible. If you need to see which classes of control are common to different controls, see the sections on each control in the *Language Reference*.

Sharing Code Among An Array of Controls

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

The first time you give a control the same name as another control (that’s not already part of a control array), REAL Studio will ask you if you wish to create a control array. For example, if you create a `PushButton` called `pb1` and then add another `PushButton` to the window that you also name `pb1`, REAL Studio will present the dialog box shown in Figure 295.

Figure 295. Creating an array of controls with a dialog box.



If you click OK, REAL Studio will assign the first control’s Index property the value 0. The control you are renaming will then have its Index property set to 1.

Figure 296. The ID properties of the first two controls in a control array.

Notice that the names of the two controls described in Figure 296 have the same name and are distinguished only by the value of their Index property. After that, any controls in the same window with *the same name* will be assigned the next Index number in the sequence automatically.

The other way to create an array of controls is to enter zero as the value of the Index property of the first control and then assign the first control's name to the second control. REAL Studio will then automatically assign 1 to the Index property of the second control, and so on. This process bypasses the dialog box shown in Figure 295 but achieves the same effect.

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

In the following example, the three RadioButton controls actually consist of a control array. All three controls are named "rb1" and are differentiated by their Index property. In the Code Editor, the value of the Index property is automatically passed to each method. The Action event handler, for example is where you determine which RadioButton was clicked.

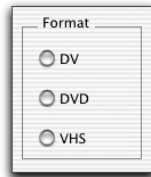
In Figure 297, notice that the Index parameter is automatically added to the method declaration line. Instead of just saying "Sub Action", it says "Sub Action (Index as Integer)". The parameter Index is the index value shown in the Properties pane (shown in Figure 296).

Here is a simple Select Case statement that tests the value of the Index parameter and determines which RadioButton was clicked. This Select Case statement looks at a control array with three RadioButtons.

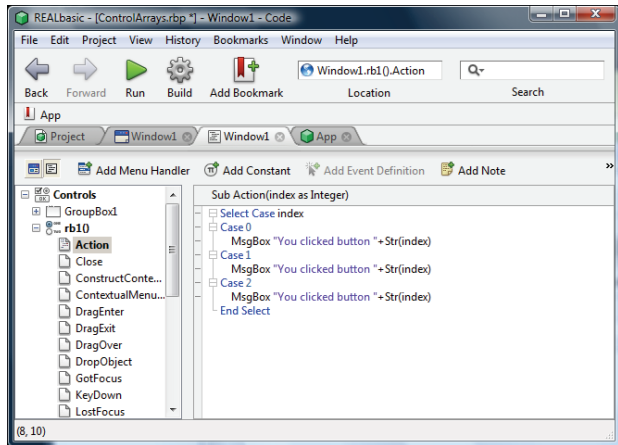
```
Sub Action(Index as Integer)
  Select Case Index
    Case 0
      MsgBox "you selected radio button "+Str(index)
    Case 1
      MsgBox "you selected radio button "+Str(index)
    Case 2
      MsgBox "you selected radio button "+Str(index)
  End select
```

Within each Case, simply insert the code for handling that selection.

Figure 297. A control array and its Action event handler.



When the user clicks a radio button, it sets the value of the Index parameter.



Drag and Drop

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

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

A DragItem object can actually contain more than one item and the items don't necessarily have to be of the same type. When you allow the user to drag multiple items, you need to create additional items within the DragItem object and, when the DragItem is dropped, cycle through all items in the DragItem.

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

Dragging Text From TextFields

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

The text in a `TextArea` can be dragged automatically without any coding necessary. A `DragItem` object is automatically created and the text the user is dragging is placed in the `Text` property of the `DragItem`.

Dragging a Row From a ListBox

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

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

Dragging from an ImageWell

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

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

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

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

`Self` is a reference to the parent window, `X` and `Y` are coordinates where the user held the mouse button down, and `Me.Width`, and `Me.Height` are the width and height of the `ImageWell`.

Dragging from a Canvas Control

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

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

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

Dropping

In order for the user to be able to drop a DragItem on a control or window in your application, the control or window must have previously indicated that it will accept the kind of data the user wishes to drop on it. There are four methods that any control can call to indicate the type or types of data that can be dropped on that control. You can indicate that the control or window can accept more than one data type by using as many methods in Table 11 as appropriate.

Table 11: Methods that control the type of data that can be dropped on a control.

Name	Description
AcceptTextDrop	Indicates that the control or window will accept text being dropped on it.
AcceptPictureDrop	Indicates that the control or window will accept a picture being dropped on it.
AcceptFileDrop (<i>Type</i>)	Indicates that the control or window will accept files (of the type or types passed) being dropped on it. The file types must be defined as file types for this project in the File Type Set or via the FileType class. See the section "Using The File Types Editor" on page 480 for information on defining file types.
AcceptRawDataDrop (<i>Type</i>)	Indicates that the control or window will accept the data type specified by the four-character Type code, e.g., AcceptRawDataDrop ("mytp"). Use this to drag and drop data in special formats or to control where text strings can be dropped.

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

In most cases, when something acceptable is dropped on a control or window, the target's DropObject event handler is executed. This event handler is passed a DragItem object that represents the item being dropped. If the target has indicated that only

some kinds of data are acceptable, your code can get the data from the appropriate property of the `DragItem`. The data types are shown in Table 12:

Table 12: DragItem properties that indicate the type of object dropped on a control.

Name	Description
FolderItem	Represents an application, folder, or document that has been dropped.
Picture	The picture, if any, that has been dropped.
Text	The text, if any, that has been dropped.
RawData (<i>Type</i>)	Data of the Type indicated by the four-character type code. Data of the format indicated by the Type code is returned as a string. Supported only on Macintosh and within the REAL Studio application on Windows.
PrivateRawData (<i>Type</i>)	The data (of the Type specified) being dragged. Type is a four-character resource type or programmer-defined four-character code. This data cannot be dragged to another application.

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

Table 13: DragItem functions indicating the type of data that has been dropped.

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

Dropping Items On TextAreas

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

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

```
Sub DropObject(Obj as DragItem, action as Integer)
  If Obj.FolderItemAvailable then
    Call Me.Open(obj.FolderItem)
  End if
```

The `Open` method of the `TextArea` opens the `FolderItem` passed to it and adds its contents to the `TextArea`.

If more than one file can be dropped at a time, you need to place the code in a `Do` loop that will read all the files. In this way you can determine when it has processed the last file to be dropped.

The easiest way to this is to use the `TextInputStream` class. `TextInputStreams` have methods that allow to read from a file, check to see if you are at the end of the file, and close the file when you are done reading from it. They are created by calling the `Open` shared method of the `TextInputStream` class. If you are working with a file with an encoding that is not UTF-8, you should set the value of the `Encoding` property.

The following example allows the user to drag several text files from the desktop to a `TextArea`. The example places the contents of all the text files in the `TextArea`, appending each file's contents to the `Text` property of the `TextArea`.

To support several files, the `NextItem` function is used to determine whether there are any more files remaining to be dropped.

```
Dim textStream as TextInputStream
If obj.folderItemAvailable then
  Do
    textStream=TextInputStream.Open(obj.FolderItem)
    Me.AppendText(textStream.ReadAll)
  loop until Not obj.NextItem
End if
```

The `TextArea` has the line:

```
Me.AcceptFileDrop(FileTypes1.Text)
```

in its `Open` event handler, where `FileTypes1` has a common file type of `Text`.

Dropping Items on ListBoxes

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

```
Me.AcceptTextDrop
```


Next, you need to tell the ListBox what to do when dragged text is coming its way. You do that in its DropObject event handler:

The following DropObject event handler determines whether the dragged item has text; if it does, it creates a new row and assigns the dragged item's text property to the new row.

```
Sub DropObject (Obj as DragItem, action as Integer)
  If Obj.TextAvailable then
    Me.AddRow(obj.text)
  end if
```

This will work, for example, on text clippings dragged from the desktop on a Macintosh. If you want to drag text files, then you need to modify this code to support dragged files. In the Open event, change the code to:

```
Me.AcceptFileDrop(FileTypes1.Text)
```

The DragItem event handler needs to be modified to accept files instead of text clippings:

```
Dim textStream as TextInputStream
If obj.folderItemAvailable then
  Do
    textStream=TextInputStream.Open(obj.FolderItem)
    Me.Addrow(textStream.ReadAll)
  loop until Not obj.NextItem
End if
```

The first row of text from each file will appear in a new row in the ListBox.

Dropping Items on ImageWells and Canvas controls

To allow the user to drop a picture or a PICT document that is being dragged from the desktop, add the following statements to another ImageWell or Canvas control's Open event handler:

```
Me.AcceptPictureDrop
Me.AcceptFileDrop("image/x-pict")
```

The second statement assumes that the PICT file type has previously been added in the File Types Set Editor or via the FileType class via the language (see "Using The File Types Editor" on page 480 for more information).

Next, you need to tell the ImageWell or Canvas what to do when the user drops either type of DragItem. You do this in the DropObject event handler. The following works for an ImageWell.

```
Sub DropObject (Obj as DragItem, action as Integer)
  If Obj.PictureAvailable then
    ImageWell1.Image=Obj.Picture
  ElseIf Obj.FolderItemAvailable then
    me.image=Obj.FolderItem.OpenAsPicture
  End if
```

It tests whether the DragItem is a picture or a file (FolderItem). If it's a picture, it assigns the Picture property of the DragItem to the Image property of the ImageWell; if it's a FolderItem, it opens the document as a PICT file and assigns that to the ImageWell's Image property.

If you want to assign the dropped object to a Canvas control's BackDrop property, the DropObject event handler is practically identical:

```
If Obj.PictureAvailable then
  Canvas1.Backdrop=Obj.Picture
Elseif Obj.FolderItemAvailable then
  Canvas1.Backdrop=Obj.FolderItem.OpenAsPicture
End if
```

You can also drag a text item to a Canvas control and use the DrawString method in the Graphics class to draw the text. To allow the drag, use the line:

```
me.AcceptTextDrop
```

in an event handler that runs prior to the user's drop. Next, test for text in the DropObject event handler. The following code accepts dragged text and writes it at a specified location:

```
If Obj.TextAvailable then
  Canvas1.Graphics.DrawString(obj.text,20,20,50)
End if
```

You will need to update the data using the Paint event handler if you want the data to persist.

RawData and PrivateRawData Properties

The RawData and PrivateRawData properties allow you to drag and drop other data types. Each property takes a four-character Type that corresponds to the four-character resource code of the format you wish to drag. For example, if you want to support dragging sound clippings, you would use "snd " as the four-character code.

Regardless of format, the data is stored in the DragItem as a string and the DropObject event handler would pass the data to a property that stores a string. The

REAL Studio control or window that receives the `DragItem` must be capable of working with the format. In most cases, that means that the control is a custom control, uses toolbox calls, and/or uses a plug-in that manages the data.

For example, to allow the user to drop a ‘snd’ resource on a control, you would write

```
acceptRawDataDrop("snd ")
```

in the control’s Open event handler. The `DropObject` event handler would pass the data to a property that stores a string. To actually play the sound, the control would have to make toolbox calls or pass the data to a plug-in since REAL Studio doesn’t provide a way to pass sound data into a `Sound` class.

You can also use `RawData` or `PrivateRawData` to manage internal drag and drop. If you make up a format type that is different than the name of any resource type, you can use it to control which objects are dropped on a control. For example, the `DragRow` event handler of a `ListBox`:

```
Function DragRow(drag as DragItem, row as Integer) as Boolean  
Drag.RawData("mytp")=me.List(row)  
Return True
```

uses the Type “mytp” to define the format. The `DragItem` is assigned the row in a `ListBox` that is being dragged. Although the row is an ordinary string, this `DragItem` cannot be dropped on a control unless it accepts dragged data in the “mytp” format. In this manner, for example, you can create a control that will only accept a `DragItem` from a specific control and reject dragged items from the desktop, other applications, and other controls.

The `ListBox` receiving the data would use the statement:

```
me.AcceptRawDataDrop("mytp")
```

to permit the data to be dropped. The `DropObject` event handler assigns the data to a new row:

```
If obj.RawDataAvailable("mytp") then  
me.AddRow(obj.RawData("mytp"))  
End if
```

The `PrivateRawData` property works the same way, except that the `DragItem` cannot be dragged to the desktop or to any other application.

Menus and Menu Items

Menus and menu items are handled in a way similar to controls and are just as object-oriented. This means that you can handle menu selections at the application, window, or even control level. When you create a new Desktop Application project, REAL Studio automatically includes one menubar object in the project, named

MenuBar1. This is the default global menubar for the entire application. By default, this is the only menubar that is used for the application.

You can also add additional menubars to your project and assign menubars to windows. On Windows and Linux, the menubar “belongs” to the window and is always used when the window is visible. On Macintosh, the window’s menubar replaces the current menubar when the window becomes active. For information on creating menus, see “Adding Menus and Menu Items” on page 189 of chapter 3.

When the user selects a menu item or presses the menu item’s command key equivalent, an event occurs much in the same way that an event occurs when the user clicks on a PushButton. In this case, the event is called a *menu event* and the event handlers are instead called *menu handlers*.

A menu event message is sent to objects in the following sequence:

- If a control has the focus, that control receives the menu event,
- The frontmost non-modal window receives the event, if it exists,
- The “blessed” App class in the Project Window receives the event¹,
- If the menu item is an instance of a MenuItem subclass, it receives the event, provided a modal window does not put the application in a modal state.

Except for the last item in the sequence, the menu event has the same name as the menu item and, if it is handled, it is handled by a menu handler that you must add. The menu handler matches the name of the menu event. You add the menu handler via the procedure described below, in the section “Adding Code To a Menu Handler” on page 359.

In the last case, the menu event message goes to the menu item’s Action event handler. This event handler is present by default and you only need to add your code.

Menu handlers can be added to a project at four levels. These options give you the flexibility to handle most any situation, but normally you will add only one menu handler per menu item.

- Controls that receive the focus can manage menu items. You can add a menu handler to such a control and it will receive a menu event only when the control gets the focus. For example, you might want to define menu items and menu handlers for managing the cells in a Listbox that work only when the ListBox has the focus or work differently when the ListBox has the focus.
- Any window except a modal or floating window. They receive the menu events when the window is frontmost.

1. The “blessed” App class is the one that has the properties of the application in its Properties pane. Usually there is only one App class in a project, but others can be added.

- The “blessed” App class. It can manage menu events for the whole application, except when the application is in a modal state because of a modal window.
- If the menu item is subclassed from the MenuItem class, it has its own menu handler in the form of its Action event.

When a menu event occurs (that is, when the user chooses a menu item), the menu event message is sent to the relevant objects in the application in the order shown above. If you have created more than one menu handler for the same menu item at several levels, they will receive the menu event message in the above order.

By default, the menu handlers for the menu item will execute sequentially. Any menu handler in the sequence can prevent the message from proceeding to the next level by returning True. For example, if a menu item should have a different menu handler when a ListBox has the focus than other controls in a window, you should put menu handlers in both the window and the ListBox, but return True from the ListBox’s version of the menu handler to prevent the window’s version from executing.

Adding Code To a Menu Handler

To add a menu handler to a window or class, do this:

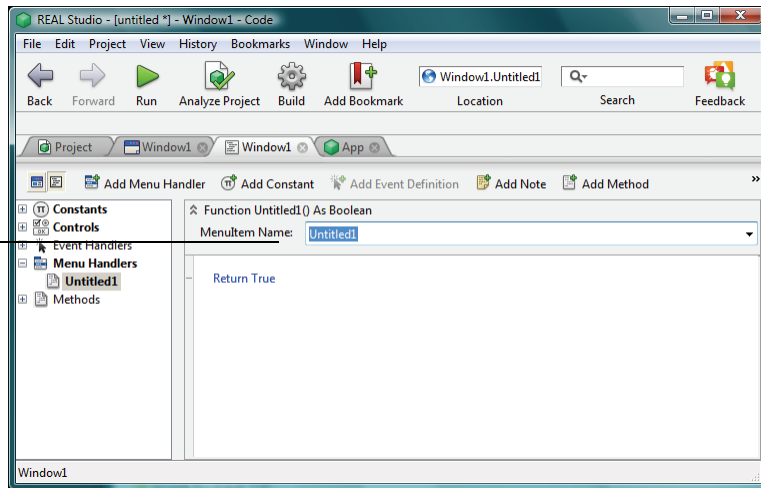


- 1 **Open the Code Editor for the window or class.**
- 2 **Click the Add Menu Handler button in the Code Editor toolbar or choose Project ► Add ► Menu Handler.**

The Menu Handler declaration area appears in the Code Editor.

Figure 298. The Menu Handler declaration area.

Menu Item pop-up menu populated with existing menu items



- 3 **Choose a menu item from the Menu Item pop-up menu.**

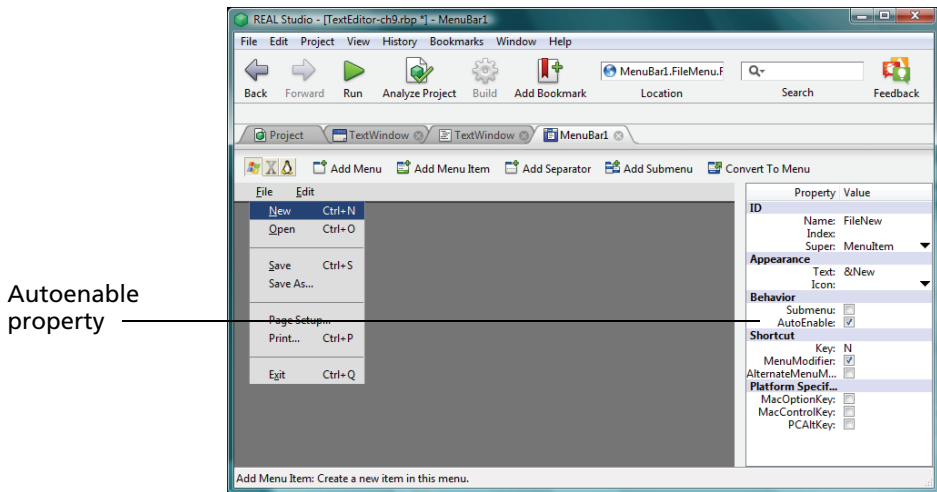
4 Click in the Code Editor area and enter the code that will execute when the user chooses the menu item.

Each menu handler optionally returns a Boolean value. The default is False. Return True from a menu handler to prevent subsequent menu handlers in the chain from handling the menu event. Comment this line out to return False.

Enabling Menu Items

Each menu item object has a boolean AutoEnable property. When the AutoEnable property is set to True, the menu item is enabled unless you explicitly disable it. You set the AutoEnable property in the Behavior group in the menu item's Properties pane.

Figure 299. The AutoEnable property in a menu item's Properties pane.



Unless you turn the AutoEnable property off, a menu item that has a menu handler will be enabled automatically. You should take advantage of the AutoEnable property of menu items that should be enabled all the time, such as a menu command that creates a new document or opens an existing document.

If you turn off AutoEnable, the menu item is disabled by default. You have the responsibility of enabling it whenever it is appropriate for it to be enabled. You do so with the EnableMenuItems event handler. You will want to turn AutoEnable off and use the EnableMenuItems event when a menu item should be enabled only under certain conditions. For example, if you want a “Save” menu item to be enabled only when the user has made changes to a document since the last Save, you would use the EnableMenuItems event to determine when to enable and disable the menu item.

When the user clicks on a menu to select a menu item or presses a keyboard equivalent, an EnableMenuItems event occurs. The purpose of this event is to give you the opportunity to determine whether the menu item being selected should be enabled or disabled based on conditions at the time. REAL Studio first checks to see

if the control that has the focus is capable of handling menus. If it is, it is sent an EnableMenuItems event. Then, assuming a window is open, the frontmost window is sent the EnableMenuItems event. Finally, the application object is sent the EnableMenuItems event.

Menu items are objects just like controls. Consequently they have an Enabled property that determines if the menu item is enabled or disabled. You can enable a menu item by setting this property to True or by calling the menu item's Enable method from within an EnableMenuItems event.

For example, this EnableMenuItems event handler is checking a property called Changed to determine if the Save menu item should be enabled:

```
Sub EnableMenuItems()  
  If Me.Changed Then  
    FileSave.Enable  
  End If
```

Handling Menu Items From Individual Controls

If the control that has the focus is capable of handling menus, its EnableMenuItems event handler will be executed. If the menu item selected is then enabled and the user selects it, the control's menu handler for the selected menu item (if it has one) will be executed. In order for a control to be able to handle menu items, it must be able to receive the focus and it must be based on a class you have added to your project rather than created by dragging a control from the Controls list. See Chapter 10, "Creating Reusable Objects with Classes" on page 531 for more information on handling menu items from control classes.

Handling Menu Items When a Window Is Open

You already know that when the user attempts to select a menu item, the frontmost window's EnableMenuItems event handler is executed followed by the application object's EnableMenuItems event handler. This gives you the opportunity to determine if conditions in the current window are right to permit the user to select various menu items. When the user selects the menu item, REAL Studio executes the frontmost window's menu handler for the selected menu item (assuming one exists) followed by the application object's menu handler.

Handling Menu Items When No Windows Are Open

When there are no windows open, the EnableMenuItems event is sent to the Application object. This condition can occur on Macintosh. Assuming the Application object enables the menu item and the user selects the menu item, the Application object's menu handler for the selected menu item (if one exists) is executed.

When you create a new project, a class based on the Application class is included in your Project Editor. This is the App class.

If the user closes all windows in your application and then decides to use the menu item to create a new window, you must enable such a menu item in the Application object, since no windows are available to enable the menu item and handle the

menu selection (assuming you are not using the menu item's `AutoEnable` property). Open the App class's Code Editor from the Project Editor, expand the Events item, and select the `EnableMenuItems` event. Add code that enables the menu item under the correct circumstances.

Creating New Menu Items On The Fly

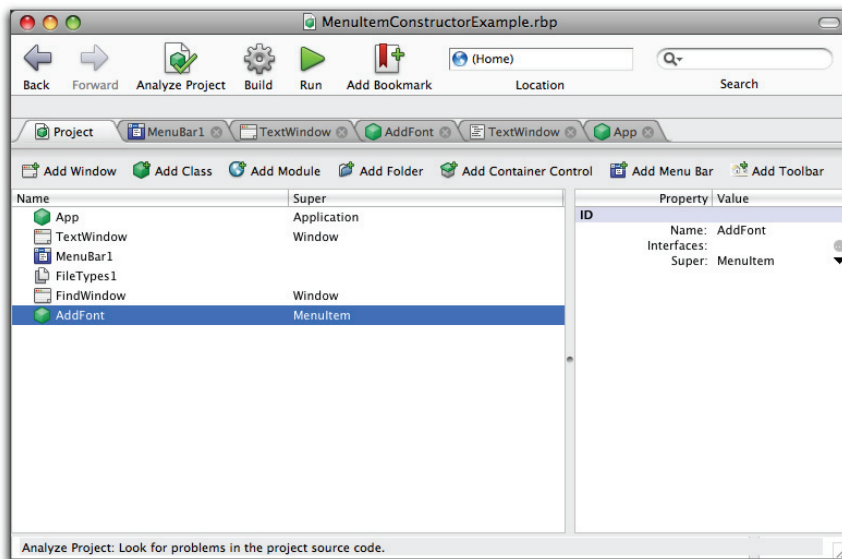
This approach uses a subclass of `MenuItem` rather than the `Index` property approach that was covered in the section “Creating MenuItems on the Fly” on page 205. This approach doesn't use the Menu Editor.

You begin by creating an instance of `MenuItem` in the Project Window and then instantiate an instance of the subclass to create each menuitem. This class does most of the work. Instead of using a menu handler, you use the subclass's `Action` event to handle the menu selection.

Suppose you want to create the dynamic Fonts menu that was used in the example in Chapter 3. To create the Font menu, you can use the `App.Open` event.

First, add a subclass of `MenuItem` to the Project Editor. In this example, it is called “AddFont” (For more information on classes, see the section “Creating Classes” on page 539).

Figure 300. The MenuItem subclass in the Project Editor.



Double-click the new subclass to add code in its Code Editor. The subclass has two Events: `Action` and `Enable Menu`. `EnableMenu` is not used in this example because `AutoEnable` is `True` by default and there are no conditions in which any of the menuitems should be disabled.

The Action event specifies what will happen when a user chooses a font from the Font menu. We want it to set the selected text in a TextArea to the chosen font. Therefore, the Action event is:

```
TextWindow.TextField.SetTextFont=Text
Return True
```

TextWindow is the name of the window that contains the TextArea and TextField is the TextArea. Text is the name of the selected font.

To create the Fonts menu and its menuitems, you can use either the Open event of the App class (for the global menu) or the open event of a Window, in case you want the menu to be local to a window. If you use the App class, the App class's menubar will be used for all the windows that have menus.

In this case, we'll use the App class. In its Open event, the following code will create the Fonts menu:

```
Dim m,mNew as MenuItem
m=self.MenuBar
mNew=New MenuItem
mNew.text="Fonts"
mNew.name="FontsMenu"
m.append mNew
if mNew = Nil then
    MsgBox "parent is nil!"
    Return
End if
```

It is simply appended to the main menubar. Next we add the code that dynamically adds the menu items:

```
Dim child as MenuItem
Dim nFonts as Integer
nFonts=FontCount-1

//build the font menu
For i as Integer=0 to nFonts
    child=New AddFont(Font(i))
    mNew.Append(child)
Next i
```

This code identifies the parent menuitem (FontsMenu) and then instantiates AddFont for each font on the user's system. Each instantiation uses AddFont's constructor, which takes the menuitem's text as the only parameter. A call to Append appends the new menuitem to FontsMenu.

Displaying a Contextual Menu

In addition to the menus that you add to your application via the Menu Editor, you can also add menus that display only as contextual menus. There are several ways to display a contextual menu in REAL Studio.

The preferred way is to use the `ConstructContextualMenu` and `ContextualMenuAction` events of the `Window` class (for windows) and the `RectControl` class (for all visible controls).

The `ConstructContextualMenu` event handler “figures out” when the user has requested a contextual menu, regardless of how he did it. It could be a right+click (Windows and Linux), Ctrl-click (Macintosh), the user has pressed the contextual menu button on a Windows keyboard, or the user has pressed Shift+F10. It also takes care of cross-platform differences in when the contextual menu is displayed. On Windows and Linux, it is displayed in the `MouseDown` event but on Macintosh, it is displayed in the `MouseUp` event.

The `ConstructContextualMenu` event is passed a “base” menu item, which serves as the “menu bar” for the contextual menu. You add your contextual menu items to the base. It is also passed the X and Y coordinates of the mouse click, so you know where to draw the contextual menu.

You can build the contextual menu in the `ConstructContextualMenu` event. To display the contextual menu, simply return `True`.

The following is a simple `ConstructContextualMenu` event handler. It constructs a contextual menu with two items and displays it.

```
base.append(New MenuItem("Import"))
base.append(New MenuItem("Export"))

Return True //display the contextual menu
```

To handle the menu selection, you can use a `Select Case` statement in the `ContextualMenuAction` event. The following `Select` statement in the `ContextualMenuAction` event handler inspects the selected menu item, which is passed in as the `HitItem as MenuItem` parameter.

```
Select Case HitItem.text
case "Import"
    MsgBox "You chose Import"
case "Export"
    MsgBox "You chose export"
End select

Return True
```

Displaying a Menu as a Contextual Menu

In some applications, you will want to give the user the option of choosing a menu item from one of the application's menus from a contextual menu. For example, you may want to let users choose the Cut, Copy, and Paste menu items from the "regular" Edit menu.

You use the `PopupMenu` method of the `MenuItem` class to do exactly this. If you want to display the contextual menu when the user right-clicks an object, use the `IsContextualClick` function to test whether the user has right-clicked (or Control-clicked on Macintosh) in the correct area. Then call the `PopupMenu` method.

By default, the contextual menu will appear where the mouse button was depressed. If you want it to appear elsewhere, pass the `PopupMenu` method the `X` and `Y` coordinates of the desired point on the screen. The top-left corner of the monitor is the 0,0 point.

The following example is in the `MouseDown` event handler of a `TextArea`. If the user right-clicks, the Edit menu is displayed as a contextual menu. The selected menu item is returned in the `MenuItem`, `m`. Before the `MenuItem` is returned, the selected `MenuItem`'s Action event occurs, so you can handle the menu selection there, just as if the user chose the menu item from the menu in the menu bar or window title bar

```
if IsContextualClick then
    Dim m as New MenuItem
    m=EditMenu.Popup
End if
```

Classes

Classes can be used to create custom controls that can also respond to the user. For more information on using classes to create custom controls, see Chapter 10, "Creating Reusable Objects with Classes" on page 531.

Adding Global Functionality with Modules

Object-oriented programming can be very efficient but you may find occasions when you need to add items that are not associated with any one object. For example, you might need to add some custom financial functions that will be called from many different places within your application. You may need to store values that are associated with those functions. In most cases, when you need to add an item that isn't associated with any particular object and needs to be accessible globally, a module is the perfect place to add it.

In addition to storing methods, properties, and constants, a module can store classes and class interfaces. A class in a module can have its own properties, computed properties, methods, event definitions, and constants. Each item can have its own Scope settings.

Modules can also contain other modules. The relationship between the modules is hierarchical; a contained module is nested in the higher-level modules. The nesting affects what the items in a module can “see.” Nested modules can also contain classes, class interfaces, and other modules.

The ability of modules to contain classes, class interfaces, and other modules represents the module *namespace* system in REAL Studio. The classes, interfaces, and modules “live” in the module's namespace. A *module namespace* is in contrast to the *root namespace*, which is represented by the project itself. Classes, class interfaces, and modules that are listed in the Project Editor are technically at the “root” namespace of the project.

In this chapter, you will learn what modules are, when to use them, and how to add items to them.

Contents

- Understanding Modules
- Adding Items to Modules
- Importing and Exporting Modules
- Encrypting Modules

Understanding Modules

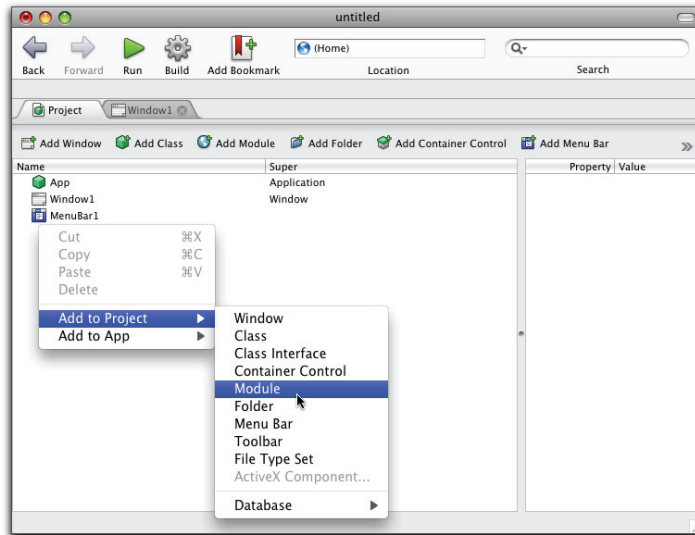
Modules are not objects. A module is not a class and does not have a Super Class. You don't instantiate modules with the New command in order to access them. Once you add a module to your project and then add items to it, global and public items are immediately accessible outside the module.

Although a module is not an object, it can (optionally) contain objects. A class in a module can contain properties, methods, and constants just like a Project class.

In order to understand classes in modules, it is best to first read Chapter 10, "Creating Reusable Objects with Classes" on page 531. Chapter 10 covers Project classes, class interfaces, and event definitions.

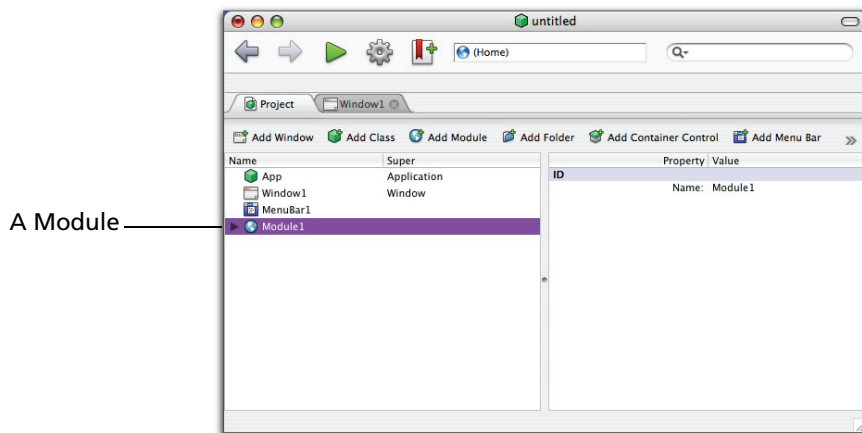
Adding A New Module

You can add a new module to your project by clicking the Add Module button in the Project Editor Toolbar or by choosing Project ► Add ► Module. You can also use the Project Editor's contextual menu to add an item to the project. Right+click (Control-click on Macintosh) in the Project Editor and choose Add to Project ► Module from the contextual menu.

Figure 301. Adding a module to the project with the contextual menu.

The new module appears in the Project Editor with a default name (the first module you add will be named “Module1,” for example). You can use the Properties pane to rename the module to something more appropriate. For example, if the module will contain your financial functions, you might name it “Financial.”

You create and modify modules with the Code Editor. To access the Code Editor for a module that is not already open, simply double-click the module’s name in the Project Editor. If it is open, click on its tab in the Tab bar. Modules can be identified by their special icon in the Project Editor.

Figure 302. A module in the Project Editor

A module in the Project Editor has a disclosure widget (a plus/minus sign on Windows and a triangle on Macintosh and Linux). It looks like the disclosure

widget for a folder. However, a folder is only an organizational convenience that enables you to group similar items and helps you organize a complex project. On the other hand, placing a item in a module affects how the application is compiled. Items in a module belong to the module and are contained in its namespace.

Scope of a Module's Items

When you add an item to a module, you need to set its Scope attribute. The Scope of an item determines which other items in the project can access it. There are three possible values:

- **Global:** A Global item is available to code throughout the application. The Global scope is available only for items in modules. For example, you can use a global property to store a piece of information that needs to be available to several different windows and to the application as a whole even if no window is open (Macintosh only). When your code needs to access a global item you simply reference it by name from anywhere in the application. The Global scope is not available for items in nested modules.
- **Public:** A Public item is also available to code throughout the application. When you need to access a public item outside of the module, you use the “dot” notation and precede its name with the module’s name. For example, if you declare a Public property, `myPublicProperty`, in `Module1`, you call it with the syntax “`Module1.myPublicProperty`” outside `Module1`. If you create a Public property in a nested module, you need to include the full path to the property when referring to it outside its module. For example, if you declare a Public property, `myPublicProperty`, in `Module2` which is nested in `Module1`, you refer to it outside its module as `Module1.Module2.myPublicProperty`.
- **Private:** A Private item is available only within the module. It is “invisible” to the rest of the application. When you need to access the Private item inside the module in which it was created, you simply reference it by name. If you create a Private item in a nested module, it cannot be accessed by higher-level modules. However, Private items in higher-level modules can be accessed from nested modules.

Adding Methods to Modules

Adding methods to modules is done in the same way you add methods to a window. If you have created classes, a method can be either a method of the module or a method of one of the module’s classes. In the first case, the method is referred to as *ModuleName.MethodName*. In the second case, it is referred to as *ModuleName.Class-Name.MethodName*. For more information, see the discussion about creating methods in the section, “Adding Methods to Windows” on page 329.

If you have not created any classes, then a new method is contained within the module itself. If you have created module classes then you have the ability to add a method to one of the classes.



To add a method to a module, do this:

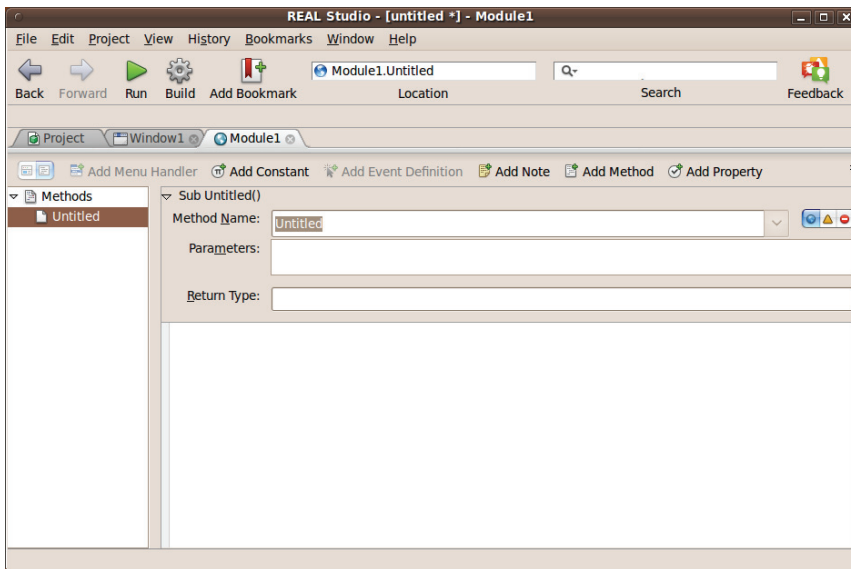
- 1 If the Code Editor for the module is not already open, double-click the module's name in the Project Editor.**

The module's Code Editor appears.

- 2 Click the Add Method button in the Code Editor toolbar or choose Project ► Add ► Method.**

The Method Declaration area appears above the Code Editor area (Figure 303).

Figure 303. The Method declaration area.



- 3 Enter the method name and parameters.**

When you enter the parameters, indicate the data type of each parameter. For example, if you are going to pass the value of a real number that you'll refer to as "X" in the method, write "X as Double" in the Parameters area. If you want to pass several parameters, separate each parameter declaration by a comma. If you want to pass an array, write empty parentheses after the name of the array.

- 4 If the method is going to be a function, choose the data type of the value the function will return in the Return Type field.**

The value that a function returns can be an array or just a single value. If you want to return an array, write empty parentheses after the name of the data type in the Return Type area. For example, if you want to return an array of integers, write "Integer ()" as the Return Type.

There are several advanced options available in the parameter declarations area. For more information, see the sections "Passing a Parameter by Value or Reference" on page 336, "Setting Default Values for a Parameter" on page 338, "Setter Methods"

on page 340, “Accessing Items of Other Windows” on page 342, and “Constructors and Destructors” on page 341.

5 Choose the Scope of the method by clicking one of the three Scope buttons.

Your choices, from left to right, are Global, Public, and Private. If the method is in a nested module, its Scope cannot be Global.

Note that elsewhere in REAL Studio, the icons stand for Public, Protected, and Private.

Figure 304. The Scope buttons.



For more information, see the previous section “Scope of a Module’s Items” on page 370.”

To add a method to a module class, do this:

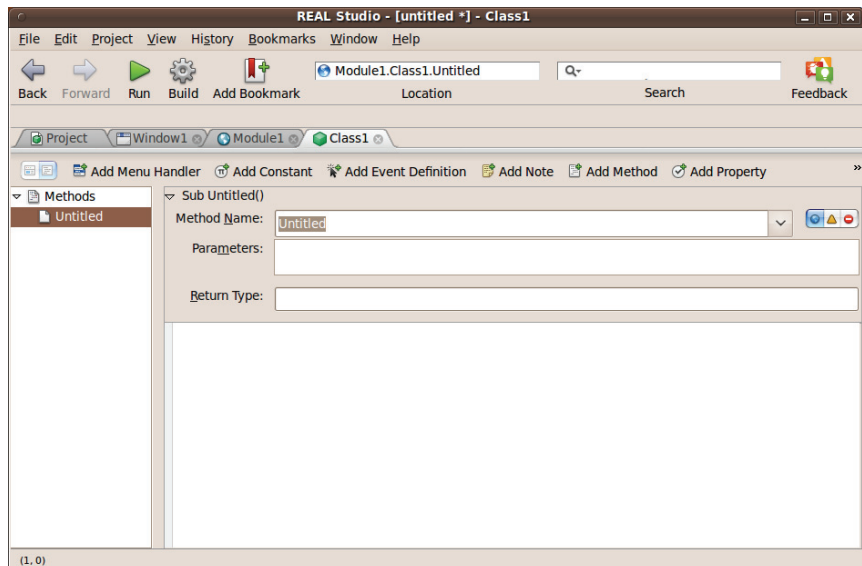
1 In the module’s Code Editor, double-click on the name of the class in the browser area.

The Code Editor for the module class appears. If there are no items belonging to the class, its browser area is blank.

2 Click the Add Method button in the Code Editor toolbar or choose Project ► Add ► Method.

A new, untitled method is added to the Module class’s Code Editor. If it is the first method, a “Methods” group is added and the new method is its first member.

Figure 305. A new method belonging to a module class.



Notice that the Location area indicates that the method will be accessed via the syntax *ModuleName.ClassName.MethodName*.

3 Follow steps 3 to 5 in the prior section to specify the name, parameters, return type and scope of the class method.

If you are creating a method in a nested module, the Global scope is not available.

Adding Properties to Modules

Adding properties to modules is done in the same way you add properties to a window. You can add properties to modules or to module classes. You set the Scope of a property to determine whether the property will be available to the whole application or only to code within the module.

To add a property to a module, do this:

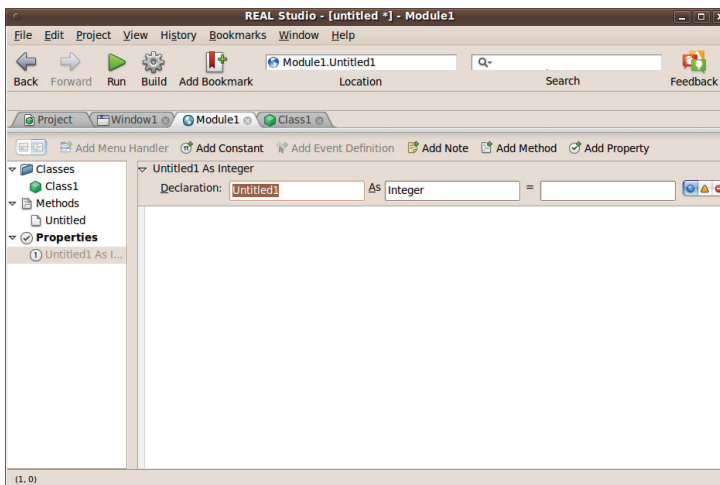
1 If the Code Editor for the module is not already open, double-click the module's name in the Project Editor to open it.

The Module's Code Editor appears.

2 Click the Add Property button in the Code Editor toolbar or choose Project ► Add ► Property.

The Property declaration area appears above the Code Editor area. A “placeholder” property declaration is entered by default.

Figure 306. The Property declaration area.



The Property Declaration area has three fields. They are for the name of the property, its data type, and its default value. The first two are required. If you do not provide a default value, the new property will take the default value for the data type that you choose. Strings have a default value of a empty string, numbers have a default value of zero, booleans have a default value of False, colors have a default value of black, and objects have a default value of Nil.

3 Fill in the Name and Data Type fields and, if desired, provide a default value.

A property can be an array. For example, if you want to declare a four-element integer array of properties called `myProperties`, you would write **myProperties(3)** in the Name field and **Integer** in the Data Type field.

If the data type of the property is a Module class, use the syntax *ModuleName.ClassName* in the Data Type field.

You can also declare a property as an array with no elements and add elements later. In that case, you would declare `myProperties` using empty parentheses, **myProperties()**.

4 Choose a Scope for the property by clicking one of the three Scope buttons.

Your choices, from left to right, are Global, Public, and Private. Note that elsewhere in REAL Studio, the icons stand for Public, Protected, and Private.

Figure 307. The Scope buttons.



For information on Scope, see the section “Scope of a Module’s Items” on page 370. If the property is in a nested module, its Scope cannot be Global.

If a property is Public or Private, a Scope icon appears with a badge in the Code Editor browser.

5 (Optional) In the Code Editor area, add notes and comments about the property.

The text entered into the Code Editor for a property is automatically non-executable, even if you write valid REAL Studio code. Add any comments you wish, including code samples.

If you are creating a module for the sole purpose of adding properties to your application that will be global (accessible from everywhere in the application), consider placing them in the App class that was included in your project by default. Declare the Scope of these properties as Public. This enables you to access them throughout the application using the syntax `App.PropertyName`.

They will still be global and this approach is more object-oriented since the properties are associated with the application directly rather than with a module that happens to be part of the application. See the section “The Application Class” on page 580 for more information on the App class.

To add a property to a module class, do this:

1 If the Code Editor for the module class that the property will belong to is not open, double-click it in the Project Window or in the module’s browser.

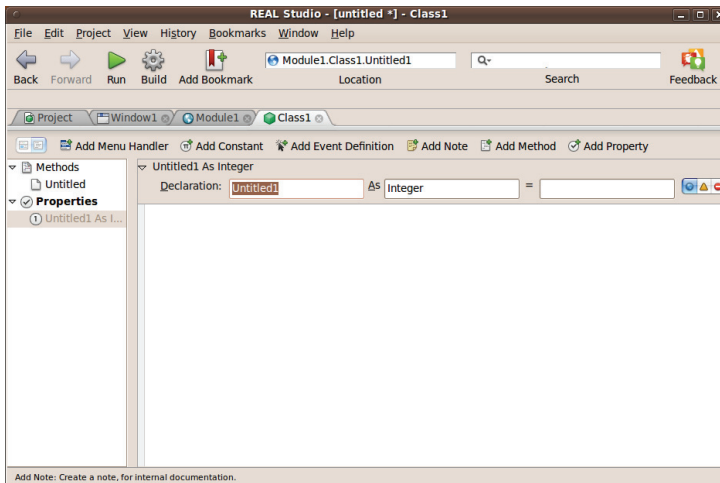
The Code Editor for the Module class appears.



2 Click the Add Property button in the Code Editor toolbar or choose Project ► Add Property.

A new property appears in the Module class's Code Editor with a “placeholder” declaration.

Figure 308. A new property in a Module class.



Notice that the Location area indicates that the property will be accessed via the syntax *ModuleName.ClassName.PropertyName*.

3 Follow steps 3 to 5 in the previous set of instructions to specify the name, data type, initial value, and scope of the property.

If the data type of the property is a Module class, use the syntax *ModuleName.ClassName* in the Data Type field.

The Scope of a property in a nested module cannot be Global.

Adding Constants to Modules

A constant acts like a variable but it holds a fixed value for its entire “life.” When you create a constant, you give it its value. You can read the constant’s value in your code, but you cannot use an assignment statement to change the value of a constant.

You can create constants in REAL Studio for windows, modules, and classes that are added to the Project Editor. You can also create a local constant inside any method you write.

Each constant has a Scope. The Scope determines which parts of your application can “see” the constant and read its value. A constant added to a module can be Global, Public, or Private in Scope. For information about Scope, see “Scope of a Module’s Items” on page 370.

Global constants cannot be assigned non-printing characters such as Return, Tab, Space, and so on. One way to create a global object that returns a non-printing



character is to add a function to a module that returns the desired character. For example, to create an object that returns a Carriage Return character (ASCII 13), add the following function to a module:

```
Function CR as String  
Return Chr(13)
```

Adding a Constant to a Module

To add a constant to a module, do this:

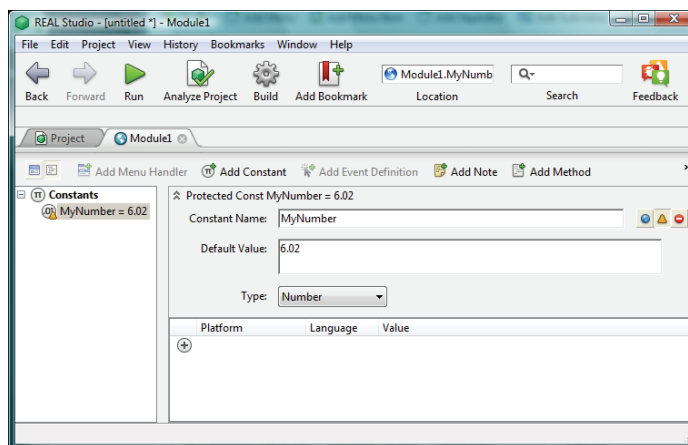
- 1 If the Code Editor for the module is not already open, double-click the module's name in the Project Editor to open it.

The Code Editor for the module appears.

- 2 Click the Add Constant button or choose Project ► Add ► Constant.

The Add Constant declaration area appears above the Code Editor area.

Figure 309. The Add Constant declaration area.



- 3 Enter the name of the constant, its value, and its data type.

When you enter a value, REAL Studio guesses the data type and sets the Type drop-down list accordingly. Any number sets the data type to Number, a string other than “True” or “False” sets it to String, and a hex value that starts with “&c” sets it to “Color.” Entering “True” or “False” sets the Type to Boolean.

If its guess is incorrect, set its data type by selecting a data type from the Type drop-down list, Number, String, Boolean, or Color. The data type of the constant will be indicated by the small icon to the left of the constant's name in the browser area.

If you chose Color, a color patch appears to the right of the Type drop-down list with the default color of black. Click it to display the Color Picker to choose the

color constant. When you choose a color, its value in hexadecimal is added to the Default Value area.

If you chose string, a “Dynamic” checkbox appears to the right of the Type drop-down list. Dynamic constants are used to facilitate localization. For more information about Dynamic constants, see the section “Dynamic Constants” on page 379.

4 Enter the value for the constant in the Default Value area.

5 Choose a Scope for the constant by clicking one of the three Scope buttons.

Your choices, from left to right, are Global, Public, and Private. If the constant is in a nested module, its Scope cannot be Global.

Note that elsewhere in REAL Studio, the three icons stand for Public, Protected, and Private.

Figure 310. The Scope buttons.



For information on Scope, see the section “Scope of a Module’s Items” on page 370.

NOTE: The New Constant pane supports standard Cut, Copy, and Paste operations.

To add a constant to a module class, do this:

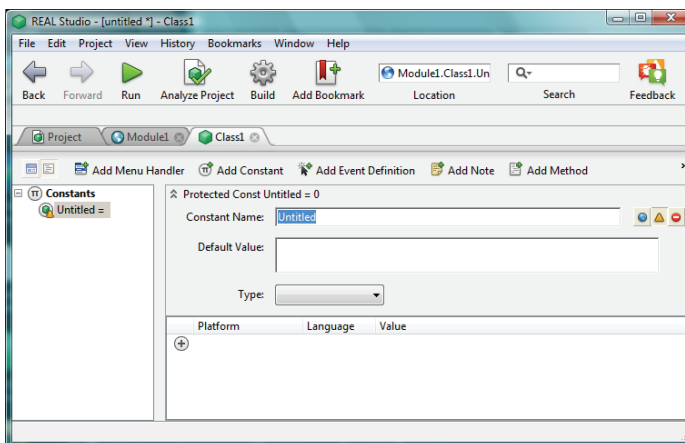
1 If the Code Editor for the module class that the constant will belong to is not open, double-click it in the Project Window or in the module’s browser.

The Code Editor for the Module class appears.

2 Click the Add Constant button in the Code Editor toolbar or choose Project ► Add Constant.

A new constant appears in the Module class’s Code Editor.

Figure 311. A new constant in a Module class.



Notice that the Location area indicates that the constant will be accessed via the syntax *ModuleName.ClassName.ConstantName*.

3 Follow steps 3 to 5 in the previous set of instructions to specify the name, data type, value, and scope.

If the constant is in a nested module, its Scope cannot be Global.

Color constants

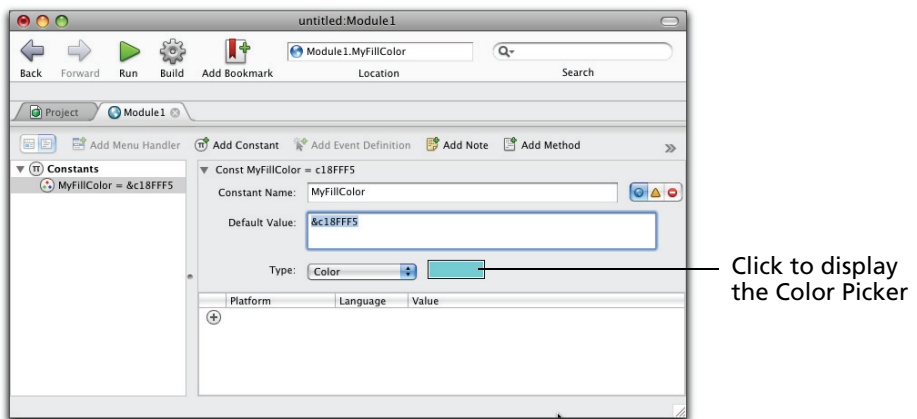
You can create constants of type color. Using color constants is a great way to ensure that the colors your application uses are consistent. When you specify the value of a color constant, you use the RGB (Red, Green, Blue) model. You specify the amounts of each primary color using the following format:

&cRRGGBB

where *RR* is the value of Red in hexadecimal, *GG* is the value of Green in hexadecimal, and *BB* is the value of Blue in hexadecimal. If you type the character string “&c” into the Default Value area, the Type pop-up menu will switch to “Color” and display the color patch to its right, shown in Figure 312. Or, you can choose “Color” from the pop-up instead of entering any value.

Click on the color patch to the right of the value area to display the Color Picker and select the color. When you close the Color Picker, REAL Studio inserts the RGB values for the selected color in the Value area. A color constant is shown in Figure 312.

Figure 312. A constant of type Color.



Using Constants to Localize your Application

Global constants also provide a very convenient way to localize your application. If you use global constants for all the text that appears in your application's interface, you can instantly localize the application simply by changing the Default Language setting in the Build Settings dialog box when you are ready to create a standalone application. For more information, see the section “Building Your Application” on page 695.

Public constants are equally suitable for localization; the only advantage of Global scope is that you can refer to them by name only, without needing to precede the name with the name of the module, class, or window in which they were declared. If you like, you can put your localization constants in the App class or even in a window, as long as you make them Public. The App class in the default Desktop Application project has constants that are used in the IDE to customize menu items on Macintosh, Windows, and Linux.

The Localization table at the bottom of the Add Constant declaration area lets you assign different values to the constant depending on platform and language. When you change the Default Language in Project Settings or the Build Application dialog box, the corresponding values for each constant take effect automatically. All you need to do is define a value for each combination of platform and language that you build.

Your choices for platform are:

- Windows (Windows 2000, XP, Vista),
- Macintosh Universal Binary (Mac OS X 10.2 or above on either a PowerPC or Intel Mac; REAL Studio no longer builds in the Macintosh PEF format or for Mac OS “classic.”)
- Linux (GTK 2.8 or higher).

This set of features allows you to create different definitions of the constant for each type of operating environment.

Dynamic Constants

String constants that have values for multiple languages can be set to be dynamically localizable. That means that the built application will load the proper values for the language on which the application is running. On Mac OS X, it will generate .lproj folders inside of a package that will contain a single Localizable.strings file with the values of the strings. On Linux and Windows, it will generate .mo files according to the gettext format.

To enable the dynamically localizable feature, click the Dynamic checkbox that appears to the right of the blocks of data types. The Dynamic checkbox appears only if you have selected String as the constant’s data type.

REAL Software provides a free localization utility, Lingua, that enables you to localize strings outside of the REAL Studio IDE. To utilize Lingua, you first set up all of your string constants to be localized as “Dynamic.”

An Example of a Localized String



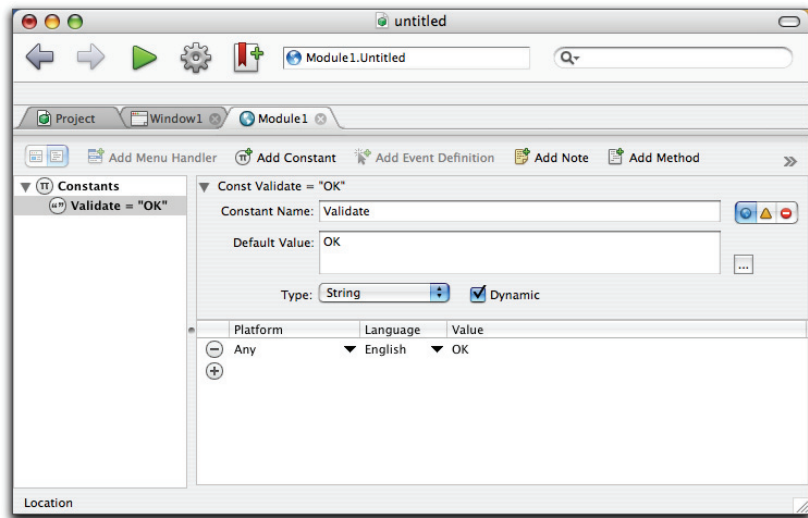
The following illustrates how to set up a constant that will be used as the caption for a button control that is used generically as the ‘accept’ button (i.e., “Save”, “OK”, and so forth). This illustrates how to localize within REAL Studio; for information on how to localize externally, see the section “Using Lingua to Localize your Application” on page 383.

To define several values for a constant, do this:

- 1 **Using the New Constant declaration area, add a new constant.**
- 2 **Enter a value for the constant for the default language and set the type to String.**
- 3 **Choose the Scope for the constant.**
Ordinarily you will make the scope of the constant Global so that it can be used in menus, menu items, and in windows and their controls. You can also use Public.
- 4 **Click the Plus sign in the Localization table.**

REAL Studio adds a new blank row to the Localization table. This is shown in Figure 313.

Figure 313. A new row added to the Localization table.



The Platform column enables you to choose whether the value will be for any platform, any Windows version, Linux, and Mac OS X (Note: REAL Studio 2007 Release 4 and above builds only for Mac OS X 10.2 and above for the PowerPC, Intel architecture, or Universal Binary). The Language column offers choices of Default (the default language of the host OS) or virtually any specific language.

- 5 **Use the pop-up menus to choose a Platform and Language and enter the value for that platform/language combination in the Value area.**

If you are entering a string constant that will be used as an item's Text property and want to assign a keyboard accelerator, precede the letter with the ampersand (&) character. On Windows and Linux, the key will be underlined.

- 6 **Repeat these steps to add additional platform/language combinations, as needed.**



It is mandatory that a value be provided for each Platform/Language combination that you build. If you omit a value, REAL Studio will have no idea what to use when you build the application for that platform and language.

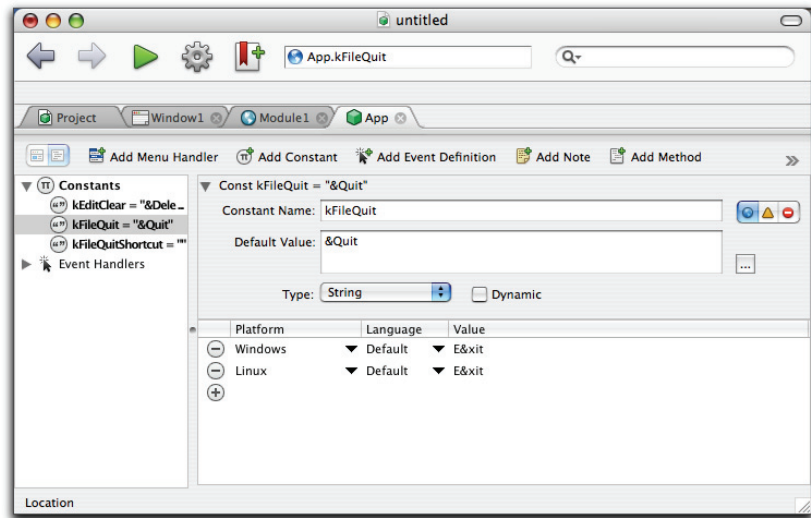
You can localize menus and menu items in exactly the same way. Create a global constant for each text string that will be used as a menu and menu item. Then use a constant's name as the menu's Text property, preceded by the number sign (“#”). If you want to specify a keyboard accelerator, place an ampersand prior to the accelerator character.

A Localization table example

The App class that is included in Desktop applications by default contains constants that are used to manage the File ► Exit and the Edit ► Delete menu items. On Macintosh, these menu items are Quit and Clear respectively. MenuBar1 contains references to these constants rather than literal text. Since the constant uses different values by platform, the localization table is filled in for platform only, not language.

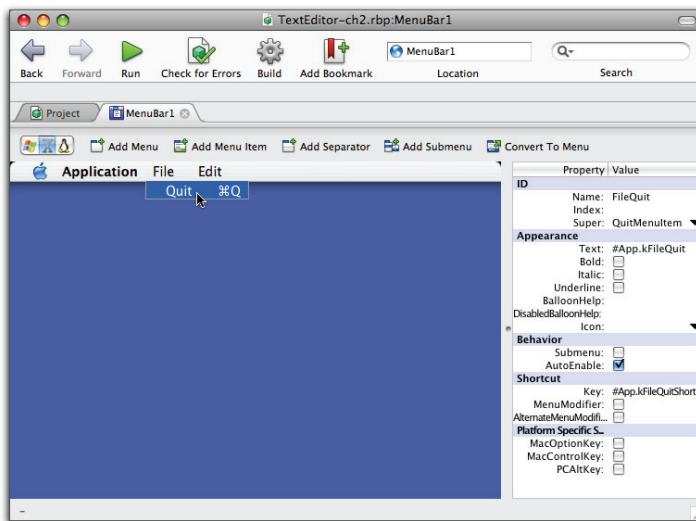
For example, the constant for the File ► Exit menu item specifies the value of “Quit” which is used on Macintosh, and has entries in the Localization table for Windows and Linux. They change the Value to “Exit” and specify a keyboard accelerator.

Figure 314. The specifications for the kFileQuit constant.



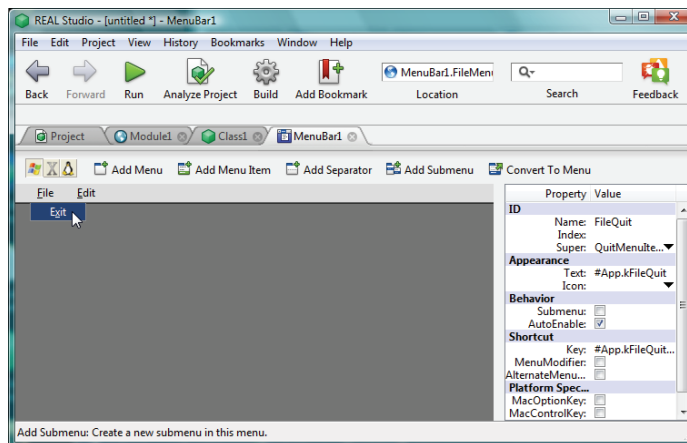
The kFileQuit constant is referred to in the Text property of the Quit menu item in the Menu Editor for MenuBar1. Note the use of the number sign in the Text property. Since the constant's Scope is Public, not Global, it is necessary to include the name of the object in which it is declared, thus it's “#App.kFileQuit”. Notice that the Menu preview area shows that it has picked up the value of “Quit”.

Figure 315. The properties of the Quit menu item (Macintosh).



On the Windows version of the same application template, the Menu Editor shows that the value of “Exit” is used.

Figure 316. The Exit menu item in the Windows IDE.



On Windows, the keyboard accelerator is indicated by the underscore in the Localization table.

This technique also works for all static text that appears in windows: PushButtons, bevel button menus, contextual menus, tab panel labels, etc. You can reference the constant simply by entering its name in the Properties pane, preceded by the number sign, as shown in Figure 316. If it is a public constant, you need to include the name of the object that contains the constant; if it is a global constant that was cre-

ated in a module, then you can omit the name of the module that contains the constant.

You can also use constants in enterable fields in App class's Properties pane where you could also enter static text. For more information, see Chapter 15, "Building Stand-Alone Applications" on page 693.

Using Lingua to Localize your Application

REAL Software provides a free utility called Lingua that you can use to localize REAL Studio applications. It does the localization outside the REAL Studio IDE.

Lingua utilizes the dynamic constants option that you can select when you create a constant. The first step is to create dynamic constants for all strings that need to be localized. A recommended approach is to create a separate module for your localizable constants, but you can put these constants anywhere.

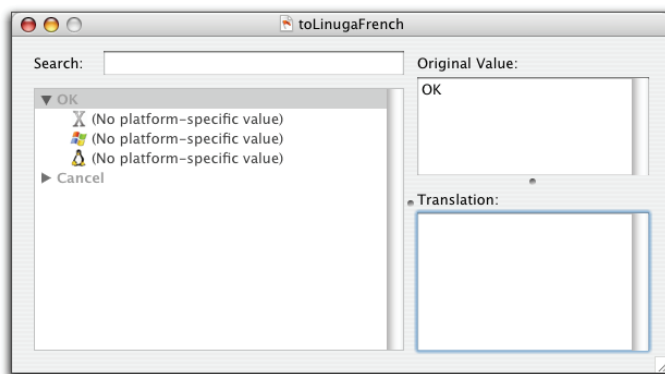
When all of your strings have been defined as dynamic constants, choose File ► Export Localizable Values. A dialog box will appear, asking you to select the language you want to localize to. Select the language you are localizing to and click Export.

REAL Studio will then write out a file that can be opened by Lingua on Windows, Linux, and Mac OS X.

Launch Lingua and then open the file you exported. The main Lingua window opens, showing a list of all the dynamic strings in your application. The values are grayed out when there is no localized version. If there are any different values specific to Windows, Linux, or Mac OS X, there will be an icon to the far right of the string in the list. Like the strings, the icons will be grayed out if the value is not localized.

To localize a string, select it in the list. The original value will be displayed in its entirety in the upper right pane, and in you can type the translated text in the lower right panel.

Figure 317. The Lingua main screen.



To add a value specific to a platform, expand the string in the list (as shown for the OK button, above), and select the individual platform to edit it.

To test the strings, choose File ► Export to Application. Lingua presents an open-file dialog box. Select the target application and click Open. When the import is complete, switch back to the REAL Studio application and debug the application normally.

When finished localizing, you can save the file from within Lingua and then import the strings file back into REAL Studio by dragging it into a project or choosing File ► Import.

Adding Classes to Modules

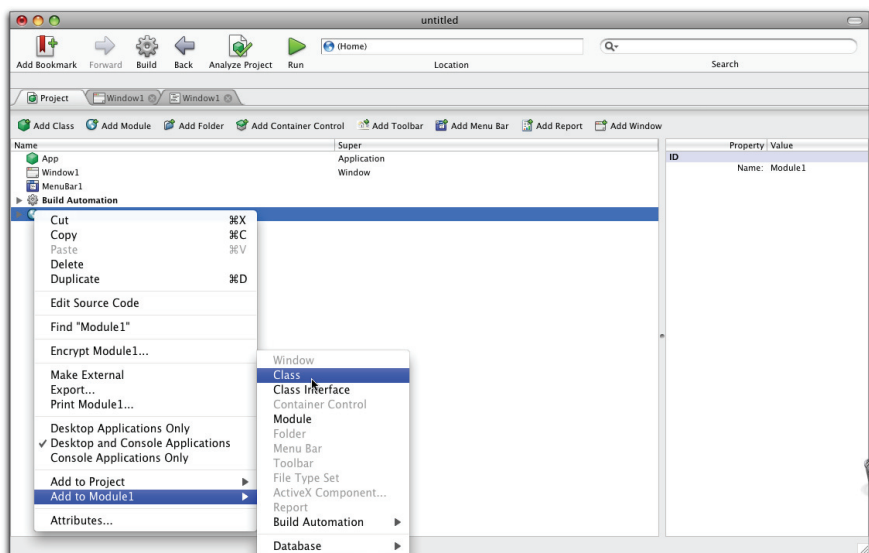
A module can contain classes. Unlike classes that are added to the project, a class in a module “lives” inside the module. Outside the module, a module class is referenced by dot notation, e.g., “*Module1.Class1*”. Items in a module class are also referenced by dot notation, e.g., “*Module1.Class1.Method1*”.

To add a class to a module, do this:

- 1 **Highlight the module in the Project Editor and right+click (Control-click on Macintosh) and choose Add to *ModuleName* ► Class from the contextual menu.**

This is illustrated Figure 318 on page 384.

Figure 318. Adding a class to a module.

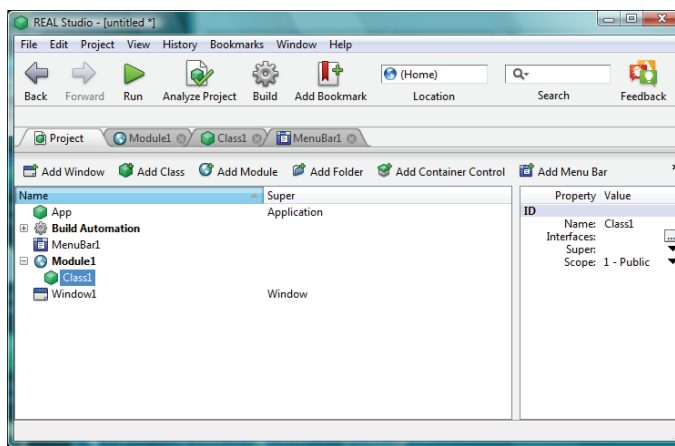


A new class is added to the module’s namespace. It appears as a nested item when you expand the module in the Project Editor. This is shown in Figure 319 on page 385.

Notice that the Properties pane for a module class includes an item for setting its Scope. This setting is unavailable for a project class. A project class is available throughout the application and this cannot be changed.

By default, the Scope of a module class is Public. This means that you need to refer to it outside the module using the format *ModuleName.ClassName*. If the class is Public, the it has no badge in the Project Editor. If it is global, it gets the “globe” badge and if it is private, it gets the standard “private” badge.

Figure 319. A new Public module class in the Project Editor.



- 2 Use the Properties pane to set the Super for the class, its name, and Scope.
- 3 To add items to the class, double-click on the class to display its Code Editor.

For information about adding items to classes, see the section “Modifying Classes” on page 543.

- 4 Add items to the class using the same procedures that you would use for adding items to a project class.

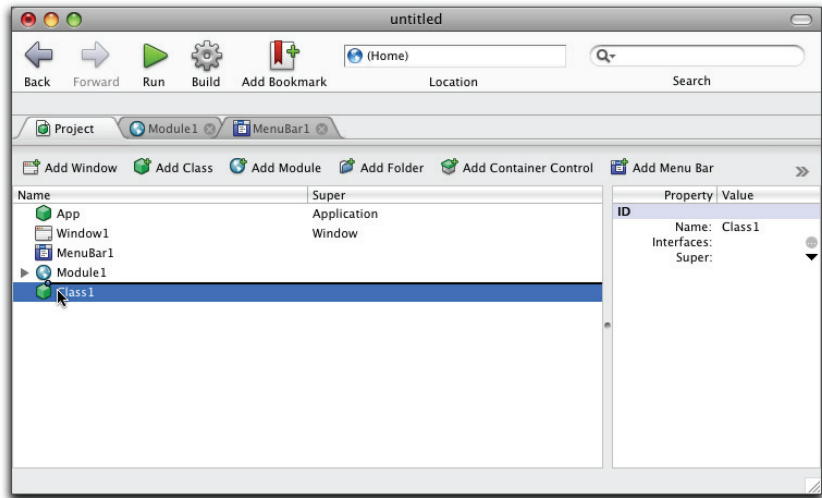
See the section referred to above for information on adding items to a class. The crucial difference is that items added to a module class “live” inside the module. For example, if you add a method to Class1 inside Module1, it is referred to as *Module1.Class1.Method1*.

Converting a Project Class to a Module Class

You can also change a project class to a module class. Project classes appear in the Project Editor and are available globally to the project.

To convert a Project class to a module class in the Project Editor, drag the class diagonally to the right and over the name of the module (If you do not drag toward the right, the class will remain at the project level, but change position).

Figure 320. Dragging a global class to a module.



Notice that the horizontal line that indicates the destination position of the drag is indented.

When the drag is successful, the class appears indented (as shown in Figure 319) in the module.

Adding Class Interfaces to Modules

A class interface is a construct that you can use to tie together classes that do not share a super class but have something in common in your application. Class interfaces are used to specify what an object does without specifying how it does it.

In order to understand class interfaces better, it's best to read Chapter 10, "Creating Reusable Objects with Classes" on page 531 in order to understand the role of classes and interfaces in project development. Class Interfaces enable you to separate the code that implements a method or function from the calling methods. If two or more classes need to do the same thing, but do it in different ways, you use an interface instead of a super class. The class interface is the "public" interface that outside methods call, while the class's methods are the "private" implementations.

This means you can use private interfaces to provide an extra level of access between classes that live in the same module, without exposing those methods to code outside the module. Simply define the methods you want to expose on the private interface, then have the class implement the interface with private methods. The code inside the module will be able to cast the object to that interface and call the methods, but code outside the module won't be able to use the interface.

Class interfaces in modules are created in the same way as project-level class interfaces, as described in Chapter 10.

The process involves three basic phases:

- Creating the class interface,
- Creating the classes that implement the class interface,
- Adding the classes to your project and calling the class interface methods in your program. Typically, that means writing generic code that tests whether a class implements a class interface and executing class interface methods where appropriate.

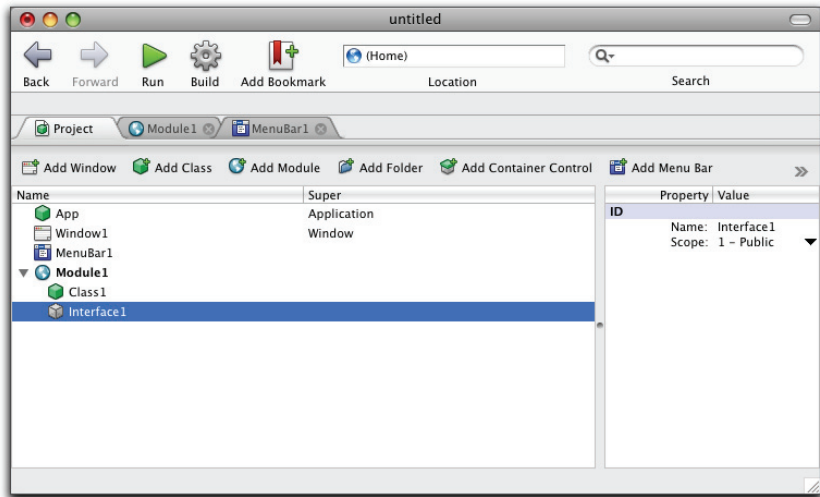


To create a class interface in a module, do this:

- 1 In the Project Editor, right+click (Control-click on Macintosh) on the module to which you want to add the class interface and choose Add to *ModuleName* ► Class Interface.**

A new class interface is added to the selected module in the Project Editor. It is indented and its icon is hollow, indicating that it doesn't actually hold code.

Figure 321. The Project Editor with a new module class interface.



The Properties pane for the interface enables you to name it and set its scope.

2 If desired, use the Properties pane to change the name of the class interface.

3 Use the Properties pane to set the Scope for the class interface.

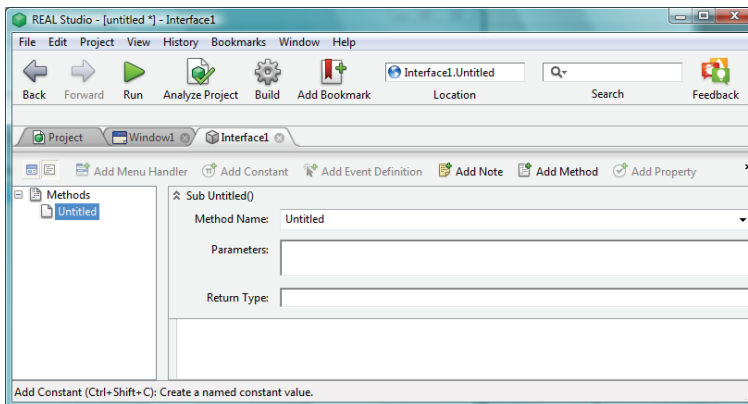
Unlike project class interfaces, module interfaces have a Scope. Global interfaces are accessible to classes outside the module.

4 Double-click the Class Interface item in the Project Editor to display the Code Editor for the class interface.

The Code Editor for a class interface has items only for methods and notes. You cannot create properties or constants for a class interface.

5 Click the Add Method button or choose Project ► Add ► Method to add a method declaration to the class interface.

The Method declaration area appears above the Code Editor area.

Figure 322. The Method declaration area for a Class Interface.

6 Enter the name of the method, its parameters, and, if it will return a value, the data type of the value being returned.

In other words, declare the method in the normal manner. You use the Method declaration only to provide the ‘spec’ for the methods that will be written (a.k.a., “implemented”) elsewhere. The method will have several implementations, one in each class that implements the class interface.

For more information on declaring a method, see the section “Adding Methods to Windows” on page 329.

7 Repeat steps 4 and 5 for each method or function declaration of the Class Interface.

After you have created your class interface, you must ‘hook it up’ to one or more custom classes. To be non-trivial, we assume it will be two or more custom classes. You add the code for the methods declared in the class interface in *each* class that implements the class interface.

To implement a class interface, do this:

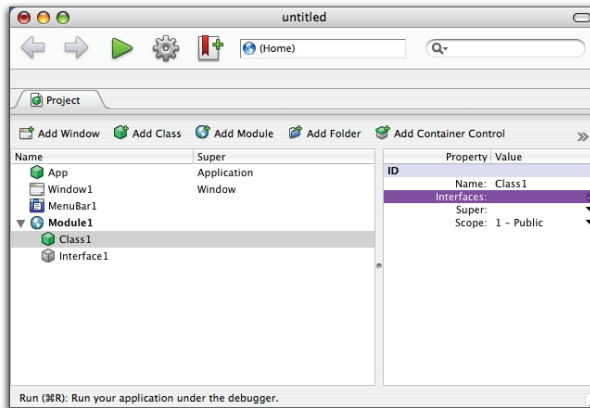
1 In the Project Editor, select the class to which you want to add the class interface.

Notice that the Properties pane for the class contains a field for specifying the class interface (or interfaces) for the custom class.

You can specify the class interface or interfaces that the class implements by entering their names in the Interfaces field in the Properties pane or via the Project pane’s contextual menu.



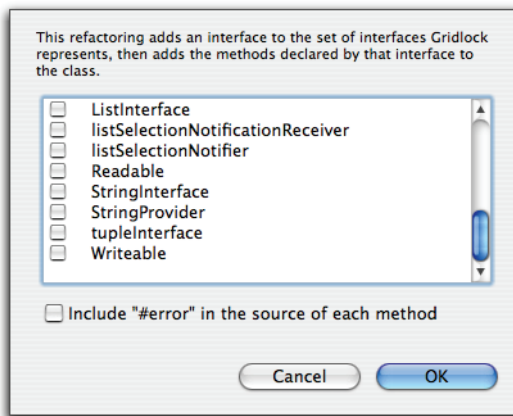
Figure 323. The Interfaces property for a module class.



- 2 In the Properties pane’s Interfaces field, click on the ellipsis (the box on the right with three dots in it) or right+click (Control-click on Macintosh) on the name of the class in the Project Editor and choose Implement Interface.**

The Implement Interface dialog box appears. It presents a list of all the currently defined class interfaces in the application.

Figure 324. The Implement Interfaces dialog box.



- 3 Click the checkboxes for the class interface or interfaces you wish to add and click OK.**

When you do so, the names of the class interfaces are added to the Interfaces field in the class’s Properties pane. REAL Studio also adds an Interfaces column to the Project Editor shows the names of the newly added interfaces.

When you choose a class interface, REAL Studio adds all the method declarations for the interface to the class’s Method Editor. The class’s Method Editor then displays the first such method, ready for you to write the method. Each method that

is generated by the Implement Interface dialog has a comment line that explains which Class Interface the method belongs to.

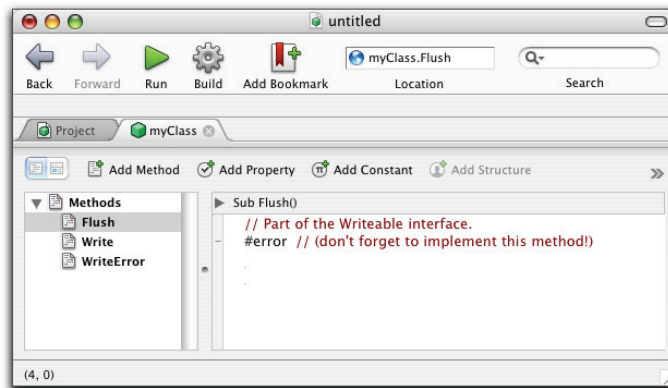
4 If desired, choose the “Include #error” option.

If you select the “Include #error in the source of each method” option in the Implement Interfaces dialog, it also includes an uncommented line with the directive “#error”. This line causes the compiler to generate a syntax error. The purpose of the line is to remind you to implement the method. If it weren’t there and you forget to implement the method, you would satisfy the technical requirement that the method exists, but it would be an empty method. The resulting compiler error would remind you to implement the method.

When you finish implementing the method, you should remove or comment out this line.

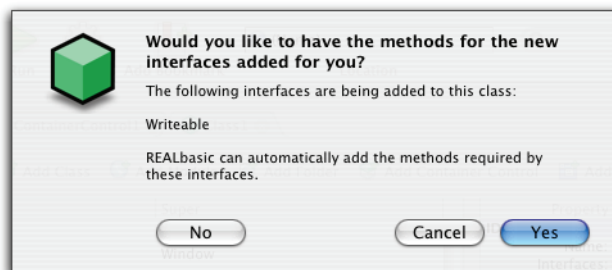
Here is an example method that was generated by the Implement Interface dialog.

Figure 325. The Flush method of the Writable class interface.



You can also enter the names of class interfaces directly into the Interfaces field in the Properties pane. Click in the text area of the Interfaces field to get an insertion point and enter the name of the interface into the Interfaces field of the Properties pane. When you enter an interface, REAL Studio displays a dialog asking you whether you’d like it to generate all the method declarations for the interface.

Figure 326. The Add Methods dialog.



If you click Yes, it displays the Implement Interface dialog where you can choose the interface you entered.

If you don't accept this choice, you must take care to implement all the methods yourself.

Adding Event Definitions to Modules

When you add code to an event handler of a module class, you cannot, by default, add more code to that event handler for an instance of the module class. Consider this example. You create a module class based on the ListBox class and you put some code in its Open event handler. Any instances of that class that appear in a window will not have their own Open event handler. The assumption is that since the event handler of the class has code for the Open event, it is handling that event.

There may be times, however, when you want the module class to have code in an event handler but you also want to be able to put code in that event handler for an instance of the module class. You want the code in the class instance to override the module class's event handler.

You solve this problem by adding an *event definition* to the module class. For information about event definitions, see the section "Adding Event Definitions" on page 558.

Adding Delegates to Modules

A Delegate data type is an object representing a specific method. Delegates decouple interface from implementation in a similar way to events or interfaces. This decoupling allows you to treat a method implementation as a variable, that is changeable based on run-time conditions. They represent methods that are callable without knowledge of the target object. You can change the function the delegate points to on the fly.

In effect, a delegate is a class with a single method, named “Invoke,” whose parameters and return value match the delegate’s parameters and return type. The Invoke method calls the method the delegate instance represents. While delegates are objects, you cannot create a subclass of a delegate type.

A Delegate can be created in modules and classes. You use the Add Delegate menu command or the Add Delegate button in the module’s Code Editor to create a Delegate.

To create a delegate, do this:

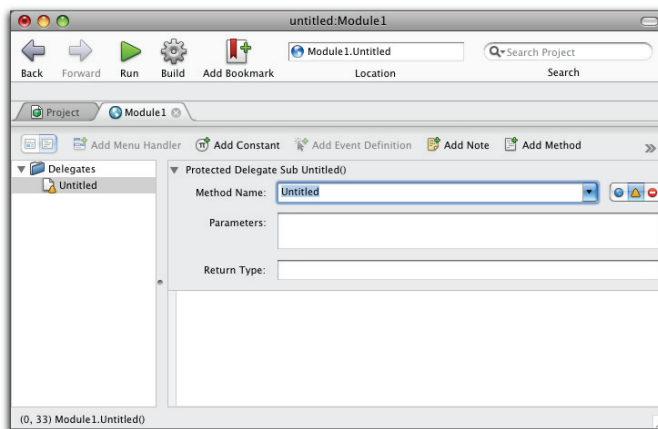
- 1 Open the module to which you want to add the delegate.**

Its Code Editor appears.

- 2 Choose Project ► Add ► Delegate.**

REAL Studio adds a Delegates folder in the module’s browser area and creates a new, untitled delegate.

Figure 327. A new delegate.



- 3 Declare the delegate by specifying its parameters and, optionally, its return type.**

This declaration creates a new object type: in effect, a class with a single method, named “Invoke.” Its parameters and return value match the delegate’s parameters and return type. The Invoke method calls the method the delegate instance repre-

sents. Although delegates are objects, you cannot create a subclass of a delegate type.

Delegate values come from the `AddressOf` operator. The `AddressOf` operator returns a delegate representing the target method. Invoking the delegate invokes the method on the same object instance the delegate originally came from.

Delegate types are considered to be compatible if their parameter lists and return types match. Casting, assignment, and the `IsA` operator work by comparing the delegate type signatures, not by comparing actual types as with classes.

The delegate type has an implicit conversion to `Ptr`, so you can continue to use the `AddressOf` function to obtain function pointers for use as external callbacks. In addition, you can create a new instance of a delegate using the `New` operator; its constructor expects a `Ptr` to an external function which the delegate will represent.

Structures

A *Structure* is a compound value type. It consists of a series of fields that are grouped together as a single block. You can control the size and order of the fields so you can declare a structure in REAL Studio to match a structure defined by an external library or as part of a binary file format or communications protocol. A structure can provide a convenient alternative to the `MemoryBlock`.

You might also use a Structure when porting a Visual Basic application to REAL Studio; it is very similar in concept and syntax to Visual Basic's "User-Defined Type" feature, also known as a "UDT". In Visual Basic .NET this is called a *structure*.

A Structure is a data type, like an integer or a color. It is not a reference type like an object or an array. When you assign an object value to an object variable, you copy a reference to the object data; when you assign a structure value to a structure variable, you copy the entire contents of the structure. When you pass a structure as a parameter `ByVal`, the whole contents of the structure is copied; when you pass a structure `ByRef`, the callee ends up modifying the caller's original structure instead.

The `New` operator does not apply to structures. When you create an array of structures, each element is an actual value (not a reference, like it would be with an array of objects). You can use the same dot syntax to access structure fields as you would use to access object properties, but when you use dot syntax with a structure, you are manipulating the structure variable itself, not a reference to data somewhere else.

Creating a Structure

Structures can be created in modules and classes. In modules, they can be given Global, Public, or Private scope. A structure contains a list of fields and/or arrays. You must declare the data type of each field or array.

Structure fields can be defined as arrays, using the usual array syntax:

fieldName(UBound) As *DataType*

Arrays in structure fields can't be manipulated in the same ways as normal arrays; they represent a fixed chunk of storage inside the structure, not a dynamic object that can be resized and manipulated. Structure field arrays cannot be resized, cannot be assigned, and do not support any of the array methods.

Strings also have a special syntax and behavior inside a structure:

fieldName As String * size

A string in a structure is a simple array of bytes. Unlike String variables, a string field has a fixed size and does not store text encoding information. If a string value contains fewer bytes than the declared size, unassigned bytes are assigned null bytes. If you use the **Len** function to get the length of the field, it will return the declared length.

Just as you convert text to a specific encoding when writing it to a file or a socket, and assign the correct encoding to it when reading it back in, you must convert strings to a specific encoding when you assign them to a structure and define them to the correct encoding when reading them back out.

Structure fields can contain any of the simple value types, but cannot contain objects. The Structure Editor in the IDE will show you the size and location of each field, so that you can match your structure up exactly with an external data format.

To create a Structure, do this:



1 Open the Code Editor for a module and choose Project ► Add ► Structure.

If you have added the Add Structure button to the Code Editor toolbar, you can click that button instead. A structure has a name and a list of fields, and can be global or private.

2 In the structure declaration area, give the structure a name.

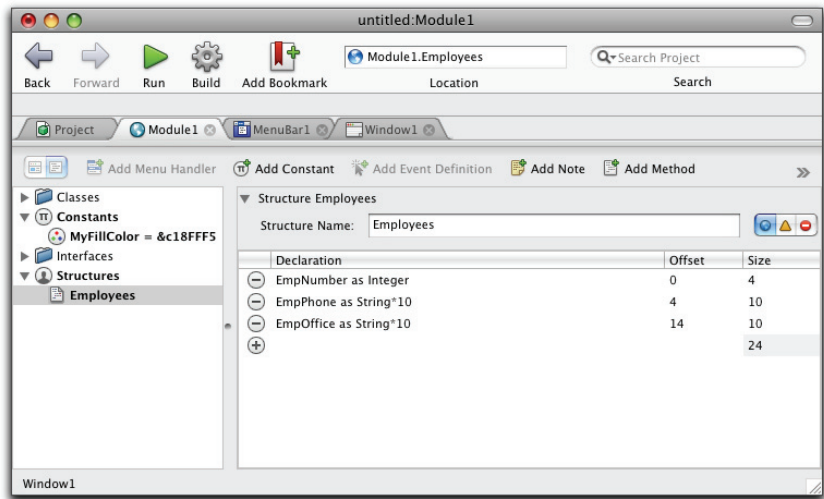
3 Click the plus button in the field list to create a new field, then type in the declaration.

The field declaration syntax is the same as for a Dim statement or a property declaration: *fieldName* As *DataType*.

Like a property or local variable, fieldnames must be simple identifiers and must be unique within the field.

A completed structure definition is shown in Figure 328 on page 396.

Figure 328. A Structure declaration.



Using Structures

Once you’ve defined a structure, you can use it in almost any context in which you would use any other data type. Use the dot syntax to access the fields. You can define an object or module property as a structure; you can declare a method parameter as a structure; you can even embed one structure as a field in another. Variants, however, cannot contain structures. Store the `StringValue` instead.

In addition to the fields you define, structures contain three built-in items:

Name	Parameters	Description
Size		This constant returns the total size of the structure in bytes.
StringValue	littleEndian as Boolean	Gets the StringValue of the structure. You must pass the desired endianness, which should match the LittleEndian property on the MemoryBlock on BinaryStream. StringValue will convert the structure’s fields to or from the appropriate endianness as necessary.
StringValue	littleEndian as Boolean	Sets the StringValue of the structure. You must pass the desired endianness, which should match the LittleEndian property on the MemoryBlock on BinaryStream. StringValue will convert the structure’s fields to or from the appropriate endianness as necessary.

The `StringValue` getter and setter methods let you treat the structure as a string. This is useful for copying structures into and out of `MemoryBlocks`, for reading and writing structures to files, and for transmitting structures through sockets.

To work with the structure, you can declare a variable or property as a structure and get and set the fields that you declared. For example,

```
Dim Employee1 as Employee
Employee1.EmpNumber=5
Employee1.EmpOffice="Tyler Hall"
Employee1.EmpPhone="555-1212"
```

Then you can get any of the values in the structure, i.e.,

```
MsgBox Employee1.EmpOffice
```

Structure Alignment

Structure alignment refers to aligning the data at a memory offset equal to some multiple of the word size. Alignment can increase the computer's performance.

Structures can be aligned via the Attributes system. You add the attribute "StructureAlignment" and use one of the legal values: 1, 2, 4, 8, 16, 32, 64, and 128.

To specify a structure alignment, right-click the structure name in the Module's Code Editor and choose Attributes... from the contextual menu. The Attributes list appears. Add an attribute to the list and specify "StructureAlignment" in the Name field and enter the desired value.

Adding an Enumeration to a Module

An *enum* or enumeration is a set of constants. It's a group of constants that are assigned values. You can assign a value to each constant or accept the default values. By default, the constants are numbered consecutively, starting with zero.

When you create an enumeration, you create a new data type. Enumerations accept only integer constants. When you want to get an enumeration, you need to cast it to an integer data type.

You can add an enum to a module or a class.

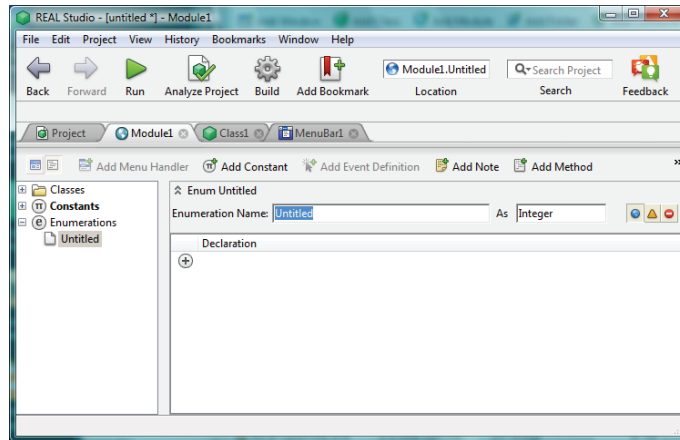
To create an Enum, do this:



1 Open a module and choose Project ► Add ► Enum.

If you have added the Add Enum button to the Code Editor toolbar, you can click that button instead. An Enum has a declaration area in which you name the Enum and set its Scope.

Figure 329. The Enum Declaration area.



- 2** In the declaration area, give the Enum a name and click one of the Scope buttons to set its scope.
- 3** Click the plus button in the list to enter the name of the first constant.
- 4** If desired, assign a value to the constant by typing an equals sign, followed by the value.

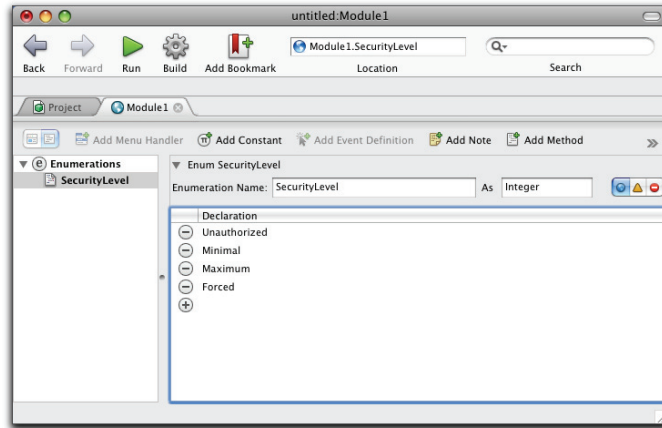
For example, if the first constant is named “Windows” and you want its value to be 13, you’d write:

```
Windows = 13
```

If you only enter the constant name, its value is its sequence number in the list, with the first item being zero.

Here is an example of a finished Enum. It defines the Enum named “SecurityLevel” and gives it four constants: Unauthorized, Minimal, Maximum, and Forced. Their values range from 0 to 3 since no values are included in the definition.

Figure 330. A global Enum defined for 'Security Level'.



You use the dot notation to get the values of the items. For example, the expression

```
SecurityLevel.Maximum
```

accesses the value of 2 because `Maximum` is the third constant in the Enum definition. To return the integer 2, you need to explicitly cast the enum using the desired integer data type. There is no implicit conversion from the enum data type to an integer data type. For example, the following code in a window returns the integer value 2 associated with this item.

```
Dim i as Integer
i=Int32(SecurityLevel.Maximum)
```

You can also declare a variable of the data type of the enum and get its values that way:

```
Dim MaxSec as SecurityLevel
Dim i as integer
MaxSec=SecurityLevel.Maximum
i=int32(MaxSec)
```

Nesting a Module in a Module

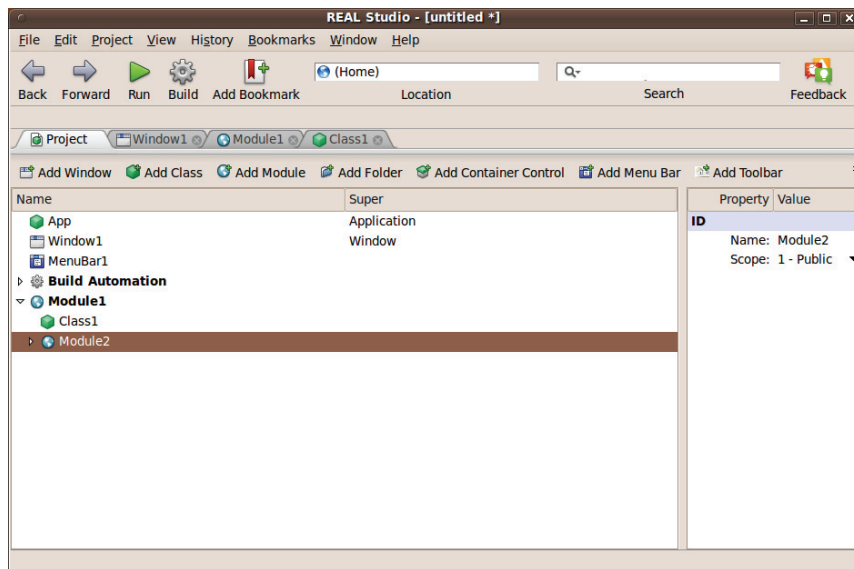
A module can contain other modules. A module nested in another module can have classes, class interfaces, methods, properties, constants, structs and enums just like the top-level module. The scope of the item determines how you refer to it outside its module. However, higher-level modules cannot “see” the items in nested modules.



To add a module to an existing module, do this:

- 1 **In the Project Editor, highlight the module to which you want to add the new module.**
 - 2 **Right+Click on the module and choose *Add to **ModuleName** ► Module*.**
- The new module appears in the Project Editor, nested inside the existing module.

Figure 331. A new module nested in an existing module.



The new module can contain other modules. The nesting of modules can continue indefinitely.

A method inside a class inside a module sees the module's members in the same way that a method directly inside the module would. The contained class has access to all the private module members, and can refer to the public module members without having to specify the module's name.

Scope of a Nested Module's Items

When you add an item to a nested module, you need to set its Scope attribute. The Scope of an item determines which other items in the project can access it. There are two possible values:

- **Public:** A Public item is also available to code throughout the application. If you create a Public item in a nested module, you need to include the full path to the item when referring to it. For example, if you declare a Public property, `myPublicProperty`, in `Module2` which is nested in `Module1`, you refer to it outside its own module as `Module1.Module2.myPublicProperty`. If you declare a Public property in `Module1`, it can be accessed in `Module2` as `Module1.myPublicProperty`.
- **Private:** A Private item is available only within the module. It is "invisible" to the rest of the application. When code inside the module needs to access a Private

method, property, or constant, you simply reference it by name. If you create a Private item in a nested module, it cannot be accessed by higher-level modules. However, Private items in higher-level modules can be accessed from nested modules.

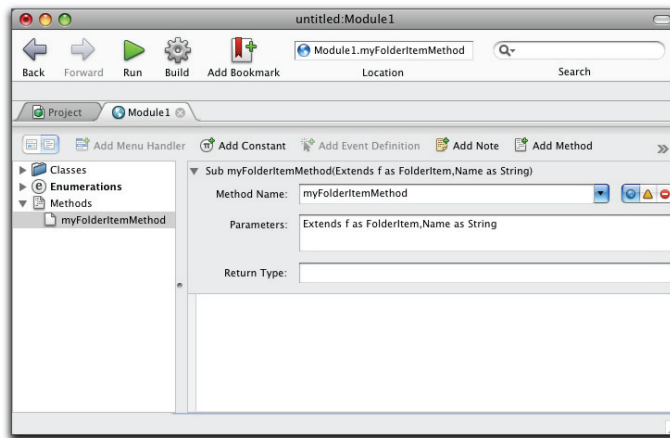
Class Extension Methods

A “class extension method” is a method that can be called using syntax that indicates that it belongs to another object. For example, you can add a method that is called from any FolderItem object that saves in a particular format. After you add the class extension method to a module, you can call it as if it were built into the FolderItem class.

To define a method as a class extension method, use the “Extends” keyword prior to the first parameter. The data type of the first parameter is the object type from which the method must be called. In other words, the use of the “Extends” keyword indicates that the parameter is to be used on the left side of the dot (“.”) operator in a calling statement.

For example the following example creates a method that will be called as a method of the FolderItem class.

Figure 332. Declaring a class extension method of the FolderItem class.



The Extends keyword can be used only for methods that reside in modules. As long as the module is in the project, the class extension method can be called from anywhere in the project. The Scope of this method is Global.

To use the class extension method in another project, simply import the module into that project. For information about importing modules, see the section “Importing and Exporting Modules” on page 402.

For an example of a class extension method, see the section, “Extending Classes” on page 568.

Importing and Exporting Modules

Modules can be imported from other REAL Studio projects. Modules that have been exported from other projects appear on the desktop with a cube icon.

A nice feature of modules is that they are modular. For example, you might want to create a module that contains basic trig functions that you use in animations. You can easily reuse the module in other projects.

Exporting

Modules can be exported for use in other REAL Studio projects. You can export a module using two different procedures:

- Right-click on the module in the Project Editor and choose Export from the contextual menu. The Export menu item will not be available if it is a namespace module.
- Click on the module in the Project Editor to select it and choose File ► Export Module.

Either procedure will export the module in its current state—protected or unprotected. Encrypted and unprotected modules share the same desktop icon; an encrypted module's protected status is apparent only within the Project Editor.

Figure 333. An exported module's desktop icon (Windows).



Encrypting

Modules can be encrypted prior to exporting to prevent other developers from accessing your code (see the following section, “Encrypting Modules” on page 403). When an encrypted module is imported into another project, it appears in the Project Editor with a key icon (🔑); if the developer double-clicks the module to display its Code Editor, he is prompted to enter the decryption password.

The developer can *use* the encrypted module in the project, but he/she cannot access any of the module's code. Encryption is a great way to sell standalone modules that provide enhancements to others' applications without the risk of having your work stolen.

Encryption is supported only in the Professional and Studio editions of REAL Studio. Decryption is supported in all editions.

Importing

To import a module into your project, choose File ► Import and locate the module to be imported using the open-file dialog box. If the module is encrypted, the locked version of the module's icon appears in the Project Editor.

If you need to share a module among two or more projects, you can import it as an external project item. For more information, see the section “External Project Items” on page 80.

Encrypting Modules

You can encrypt (protect) or decrypt (unprotect) a module while it is in your project. When a module is encrypted, no one can access its code (including you) without supplying the decryption password.

Encryption is supported only in the Professional and Studio editions of REAL Studio. Decryption is supported in all editions.



Namespace modules (i.e., modules that contain a class) cannot be encrypted.

When encrypting a module, you supply a password which can be used to decrypt it later.



To encrypt a module, do this:

- 1 **Right-click on the module in the Project Editor (Control-click on Macintosh) and choose Encrypt from the contextual menu or choose Edit ► Encrypt.**

You can optionally add an Encrypt button to the Project Editor toolbar. If it is available, you can encrypt a module by selecting the module in the Project Editor and clicking the Encrypt button.

The Encrypt *Module* dialog box appears, as shown in Figure 334.

Figure 334. The Encrypt *Module* Dialog box.



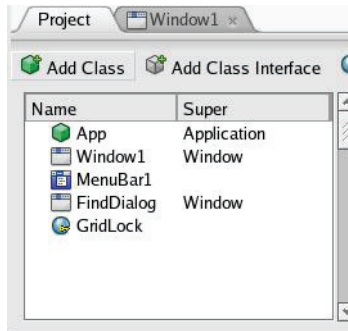
- 2 **Enter and confirm a password for encryption.**

Important Note: Don't forget your password.

- 3 **If you want the module to be accessible only to REAL Studio 2006r3 and above, check the "Use REAL Studio 2006r3 Encryption" checkbox.**

An encrypted module appears in the Project Editor with a small key in the lower right corner of the module icon. This is shown in Figure 335.

Figure 335. A project with an encrypted module.



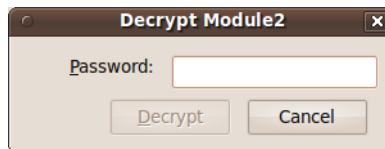
When a programmer tries to open an encrypted module, REAL Studio presents the Decrypt *Module* dialog box, shown in Figure 336.

To decrypt an encrypted module, do this:

- 1 **Right+click on the module in the Project Editor (Control-click on Macintosh) and choose Decrypt from the contextual menu or choose Edit ► Decrypt.**

The Decrypt *Module* dialog box appears.

Figure 336. The Decrypt *Module* dialog box.



- 2 **Enter the decryption password and click Decrypt.**

If the correct password was entered, the key will disappear from the module's icon, indicating that it has been successfully decrypted. If you entered an incorrect password, a message box will inform you of that fact.

Working With Text and Graphics

Almost every application manipulates text and graphics in some way. Fortunately, REAL Studio provides a rich set of functions for creating, manipulating, displaying, and printing text and graphics. Should you wish to create your own custom control, you can use the Canvas control and its graphics methods to create it.

Contents

- Working With Fonts
- Working with the Selected Text
- Handling Styled Text
- Working with Text Encodings
- Formatting Numbers, Dates, and Times
- Searching using Regular Expressions
- Understanding the Canvas Control and the Graphics Object
- Drawing Pictures
- Working with Color
- Printing Text and Graphics
- Transferring Text and Graphics with the Clipboard

Working With Fonts

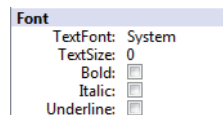
REAL Studio gives you the ability to set the font, font size, and font style of many of the objects and controls in your application. TextAreas support multiple fonts, styles, and sizes (collectively referred to as *styled text*) and ListBoxes support multiple styles. Controls that use a single font have a TextFont property that you can set by assigning it the name of the font you want used to display text for the control. TextAreas have a TextFont property but they can also display multiple fonts. For information on TextAreas, See “Handling Styled Text” on page 409.

The System and SmallSystem Fonts

The System font is the font used by the system software as its default font. It’s the font used for the menus as well.

If you want text to be displayed or printed in the user’s System font, use the name “System” as the font when you assign it. You can enter it as the TextFont property in the Properties pane. If you also enter zero as the TextSize, REAL Studio will choose the font size that works best for the platform on which the application is running. Because of differences in screen resolution, different font sizes are often required for each platform. This feature enables you to use different font sizes on different platforms without having to create separate windows for each platform. Use the Properties pane to set the TextFont to “System” and the TextSize to zero.

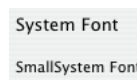
Figure 337. Using the System font with Font Size of 0.



If the system software supports both a large and small System font, you can also specify the “SmallSystem” font as your TextFont. This option selects the small system font on the user’s computer, if there is one. If there is no small system font, the System font is used.

On Mac OS X, both System font and the small System font are supported. Figure 338 illustrates the difference between the two fonts. The TextSize is zero in both cases.

Figure 338. The System and SmallSystem fonts on Mac OS X.



What Fonts Are Available?

You may want to use fonts other than the System font. In this case you will need to determine if a particular font is installed on the user’s computer. REAL Studio has two global functions, FontCount and Font, that make determining available fonts

easy. The following function, when passed a font name, will return True or False to inform you if the font passed is installed:

```
Function FontAvailable(FontName as String) As Boolean
Dim i,nFonts as Integer
nFonts=FontCount-1
For i=0 to nFonts
    If Font(i)=FontName Then
        Return True
    End If
Next
Return False
```

Building a Font Menu on the Fly

Suppose you want to create a Fonts menu that will display all the fonts on the user's computer. You don't know which fonts are installed in advance, you need to create the menuitems dynamically at startup.

To do so, you create a instance of the MenuItem class and instantiate it for each font. The Action event for the class instance handles the menu selection. For details of this technique, see the section "Creating New Menu Items On The Fly" on page 362.

Working with the Selected Text

The term "Selected Text" refers to text that is selected (or "highlighted") in TextFields and TextAreas that currently has the focus. TextFields and TextAreas have three properties that can be used to get and/or set the selected text.

Table 14: Properties for getting or setting selected text.

Name	Description
SelLength	The number of characters currently selected. You can change the selected text by changing this number. Setting this value to 0 (zero) will position the insertion point based on the value in the SelStart property rather than selecting any text.
SelStart	The number of the character just before the selected text. For example, if the fifth character in a TextField was selected, this property would be 4. Setting this value to 0 (zero) will start the selection at the beginning of the TextField.
SelText	A string containing all of the selected text. Changing this value will replace the selected text with the SelText value. If no text is selected, the SelText value will be inserted at the insertion point (the value in SelStart).

TextFields and TextAreas have one method, SelectAll, that performs the same function as the Edit ► SelectAll menu item. A call to SelectAll selects all the text in the field.

If you need to execute some code when the user moves the insertion point or highlights some characters, place your code in the `SelChange` event handler of the `TextField` or `TextArea`.

Creating a Password Field

`TextFields` have `Password` and `LimitText` properties that can be used to create password fields. When you set the `Password` property, asterisks (on Windows and Linux) or bullet characters (on Macintosh) appear instead of the characters you type. However, the characters you enter are placed in the `TextField`'s `Text` property. The `LimitText` property allows you to control the maximum number of characters the user can type in the `TextField`.

Formatting and Filtering Text Entry

The `TextField` has two properties that enable you to format text when it is entered and filter entries on a character-by-character basis. These properties are especially useful when a `TextField` is used as a data entry field in a database or has an equivalent role in applications that don't explicitly use a database engine to store the information.

For example, if a `TextField` is used to enter a US Social Security number, a valid entry must adhere to a specific format—namely three numbers followed by a dash, two more numbers and another dash, and four numbers, e.g., “578-68-7891”. Many other types of information follow a regular structure—phone numbers, credit card numbers, drivers licence codes, and so forth.

The Format Property

The `TextField`'s `Format` property is designed to allow a `TextField` to have a different display value than what was entered. For example, you could display all numbers with 2 digits of precision using this property. When the `TextField` has the focus, any formatting is cleared and the text is shown in its unaltered state. If a `TextField` doesn't have the focus and you have specified a format, the text will be shown according to the given format. The formats supported are exactly the same as those supported in the `Format` function. See the `Format` function in the *Language Reference* for definitions.

To turn off formatting, set the format property to the empty string, `""` — two quotes with nothing between them.

Example Formats

Here are some sample formats for real numbers:

```
// Always display number with 2 decimal places pad with 0 if necessary.
TextField1.format = "0.00"

// Display at most 2 decimal places but don't pad with 0.
TextField1.format = "#.##"

// Turn off formatting
TextField1.format = ""
```

The Mask Property

The TextField's Mask property is designed to permit only certain types of characters in certain positions of the field. For example, if you want users to enter a US Social Security number, you can use a mask to ensure that the user doesn't type any letters (numbers only), and that the number given is only nine digits long. Each character in the mask corresponds to a place holder for a specific type of character or a literal character. Such a mask is "###-##-####".

The input mask is completely compatible with Visual Basic with the exception of the "~" which REAL Studio reserves for future use and expansion.

If a TextField has a Mask property, there is no visible indication of that to the end user until text that is not permitted by the Mask is rejected or the user tries to enter too many characters.

See the entry for TextField in the *Language Reference* for the symbols you can use in a mask.

Example Masks Here are some simple masks that illustrate how the feature works.

```
// Allow U.S. Social Security numbers to be entered.
TextField1.Mask = "###-##-####"

// Allow dates of the form 14-Dec-1972
TextField1.Mask = "##-??-####"

// Auto-capitalize a serial number of the form AWS-1925-ASD
TextField1.Mask = ">??-####-???"

// Turn off the mask
TextField1.Mask = ""
```

Handling Styled Text

The term *styled text* refers to text that can have more than one font, font size, and/or font style. The TextArea supports styled text. In order for a TextArea to support styled text, its MultiLine property must be True (checked) and its Styled property

must also be True (checked). These are the defaults. In order to print styled text, you must use the `StyledTextPrinter` class. See the section “Printing Styled Text” on page 450 for more information.

Determining the Font, Size, and Style of Text

`TextAreas` have properties that make it easy to determine the font, font size, and font style of the selected text. The `SelTextFont` property can be used to determine the font of the selected text. If the selected text has only one font, the `SelTextFont` property contains the name of that font. If the selected text uses more than one font, the `SelTextFont` property is empty.

This function returns the names of fonts for the selected text of the `TextArea` passed:

```
Function Fonts(item as TextArea) as String
    Dim fonts, theFont as String
    Dim i, Start, Length as Integer
    If Field.SelTextFont="" Then
        Start=Field.SelStart
        Length=Field.SelLength
        For i=Start to Start+Length
            Field.SelStart=i
            Field.SelLength=1
            If InStr(fonts,Field.SelTextFont)=0 Then
                If fonts="" Then
                    fonts=Field.SelTextFont
                Else
                    fonts=fonts+ ", "+Field.SelTextFont
                End if
            End if
        Next
        Return fonts
    Else
        Return Field.SelTextFont
    End If
```

The `SelTextSize` property is used to determine the font size of the selected text and works the same way as the `SelTextFont` property. If all characters of the selected text are the same font size, the `SelTextSize` property will contain that size. If different sizes are used, the `SelTextSize` property will be 0.

There are also boolean properties for determining if all of the characters in the selected text are the same font style. Since text can have multiple styles applied to it, these properties determine if all of the characters in the selected text have a particular font style applied to them. For example, if all of the characters in the selected text are bold but some are also italic, a test for bold returns True. On the other hand, a test for italic returns False since some of the selected text is not in the italic font style. For all of these properties, you test to see if the property is True or False. If the test returns True, then all of the characters in the selected text have that font style. If it returns False, the selected text contains more than one font style. If

you want to determine which styles are in use, you can programmatically select each character in the selected text and then test the style properties. This is an operation similar to the sample `Fonts` function that determines which fonts are in use in the selected text. The properties for testing the various available font styles are:

Table 15: Properties that test for font styles.

Property	Style
<code>SelBold</code>	Bold
<code>SelItalic</code>	Italic
<code>SelUnderline</code>	Underline

In this example, if the selected text of the `TextArea` is bold, then the Bold menu item is checked:

```
StyleBold.Checked=TextArea1.SelBold
```

If all of the characters in the selected text are not bold then `TextArea1.SelBold` returns `False` which will then be assigned to the `Checked` property of the `StyleBold` menu item.

Setting the Font, Size, and Style of Text

The properties used to check the font, font size, and font styles of the selected text are also used to set these values. A `TextArea` can support multiple fonts, font sizes, and styles. A `TextField` can support one font, one font size, and the plain style. On Windows only, a `TextField` can also support the Bold, Underline, and Italic styles for all the text in the `TextField`. It cannot support a mixture of styles.

For example, to set the font of the selected text to Helvetica, you do the following:

```
TextArea1..SelTextFont="Helvetica"
```

Keep in mind when setting fonts that the font must be installed on the user's computer or the assignment will have no effect. You can use the `FontAvailable` function mentioned earlier in this chapter to determine if a particular font is installed.

You can set the `TextSize` property of a control to zero to tell REAL Studio to use the font size that looks best for the platform on which the application is running.

You can set the font size of the selected text using the `SelTextSize` property. For example, the following code sets the font size of `TextArea1` to 12 point:

```
TextArea1.SelTextSize=12
```

To apply a particular font style to the selected text, set the appropriate style property to `True`. For example, the following code applies the Bold style to the selected text in `TextArea1`:

```
TextArea1.SelBold=True
```

Table 15 on page 411 lists all the font style properties of TextAreas that can be used in this same way. They are also available for TextFields, but TextFields permit only one font for all of its text.

TextAreas also have built-in methods for toggling the font styles on and off. “Toggling” in this case means applying the style if some of the selected text doesn’t have the style already applied or removing the style from any of the selected text that already has it applied. The following code toggles the bold style of the selected text in TextArea1:

```
TextArea1.ToggleSelectionBold
```

The methods for toggling the styles of the selected text are shown in Table 16.

Table 16: Methods for toggling text styles.

Method Name	Style
ToggleSelectionBold	Bold
ToggleSelectionItalic	Italic
ToggleSelectionUnderline	Underline

Working with StyledText Objects

When you are working with styled text that is displayed in a TextArea, you can work with the properties of TextAreas that get and set style attributes (as described in the previous section) to manage styled text. However, REAL Studio also provides tools for opening, saving, and managing styled text separately from a TextArea or any other control. In fact, the styled text doesn’t even have to be displayed at all.

This set of techniques uses the properties and methods of the StyledText class. Its Text property contains the styled text that is managed by the StyledText object. It has six properties for getting and setting style attributes:

Table 17: Properties for getting or setting Style Attributes.

Name	Description
Bold	Gets or sets the Bold style to the selected text in <i>Text</i> .
Font	Gets or sets the font for the selected text in <i>Text</i> .
Italic	Gets or sets the Italic style to the selected text in <i>Text</i> .
Size	Gets or sets the font size to the selected text in <i>Text</i> .
TextColor	Gets or sets the color of the selected text in <i>Text</i> .
Underline	Gets or sets the Underline style to the selected text in <i>Text</i> .

Each method takes parameters for the starting position and length of the text for which the attribute applies. These numbers are zero-based. For example, a call to the `Bold` property would look like this:

```
Dim st as New StyledText
st.Text="How now Brown Cow."
st.Bold(0,3)=True
```

This sets the first word, “How,” in bold. Each contiguous set of characters that has the identical set of style attributes makes up a *StyleRun* object. In this example, the first three characters make up one *StyleRun*. The remaining text is the second *StyleRun*. In the language of a word processor, each *StyleRun* is an instance of a character style. The entire *Text* property is made up of a sequence of *StyleRuns*.

The *StyledText* class has six methods for managing *StyleRuns*.

Table 18: Methods of the *StyledText* class for working with *StyleRuns*.

Name	Description
<code>AppendStyleRun</code>	Appends a <i>StyleRun</i> to the end of <i>Text</i> .
<code>InsertStyleRun</code>	Inserts a <i>StyleRun</i> at a specified position.
<code>RemoveStyleRun</code>	Removes a specified <i>StyleRun</i> from <i>Text</i> .
<code>StyleRun</code>	Provides access to a particular <i>StyleRun</i> in <i>Text</i> . The <i>StyleRun</i> class has its own properties that describe the style that’s applied to all the characters in the <i>StyleRun</i> .
<code>StyleRunCount</code>	Returns the number of <i>StyleRuns</i> that make up <i>Text</i> .
<code>StyleRunRange</code>	Accesses the starting position, length, and end position of the <i>StyleRun</i> .
<code>Text</code>	The text that is managed by the <i>StyledText</i> object. Technically, <i>Text</i> is a method, but you can get and set its value as if it were a property.

The *Text* method of a *StyledText* object can have multiple paragraphs. A paragraph is the text between two end-of-line characters. A paragraph can be defined either with the `EndOfLine` function or the end-of-line character for the platform the application is running on.

A paragraph can be made up of multiple *StyleRuns*. It has only one style property of its own, paragraph alignment (Left, Centered, or Right).

There are three methods of the *StyledText* class for working with paragraphs:

Table 19: Methods of the *StyledText* class for working with Paragraphs.

Name	Description
<code>Paragraph</code>	Provides access to a particular <i>Paragraph</i> in <i>Text</i> . The <i>Paragraph</i> class has its own properties that return the start position, length, end position, and alignment of the paragraph.

Table 19: Methods of the StyledText class for working with Paragraphs.

Name	Description
ParagraphCount	Returns the number of Paragraphs that make up <i>Text</i> .
ParagraphAlignment	Sets the alignment of the specified paragraph (Default, Left, Centered, or Right). The ParagraphAlignment method takes one parameter, the number of the paragraph to be aligned (starting at zero). You assign it a Paragraph alignment constant. The four alignment constants are: AlignDefault (0): Default alignment AlignLeft (1): Left aligned AlignCenter (2): Centered AlignRight (3): Right aligned For example, to right align the first paragraph, you would use a statement such as <code>StyledText1.ParagraphAlignment(0)=Paragraph.AlignRight</code>

Although you can work with a `StyledText` object entirely in code—without ever displaying it—the `TextArea` control is “hooked up” to the `StyledText` class in the sense that you can access all the methods and properties of the `StyledText` class via the `StyledText` property of the `TextArea`.

In order to work with a `StyledText` object in a `TextArea`, you must turn on the `MultiLine` and `Styled` properties of the `TextArea`. You can do this using the Properties pane.

Suppose the styled `TextArea` already has the text that you want to manipulate using the `StyledText` class. The following code loads the text into the `StyledText` object.

```
Dim st as New StyledText
st=TextArea1.StyledText
TextArea1.AppendText("This is the appended text.")
st.Bold(0,4)=True
```

The `StyledText` object is actually an alias to the `TextArea`’s text, not a static copy. This means that the third line of code changes the contents of the `TextArea` and the last line sets the first four characters of the `TextArea` to bold.

In the following example, the line:

```
TextArea1.StyledText.Text="Here is my styled text."+EndOfLine _
+"Aren't you really impressed?"
```

sets the `Text` property of the `StyledText` object and displays it in the `TextArea`. From there, you can go ahead and assign style properties to the text. The changes

reformat the contents of the `TextArea`. Here is a simple example that works with these two paragraphs:

```
Dim st,ln as Integer
Dim Text as String
Text="Here is my styled text."+EndOfLine+"Aren't you really impressed?"
TextArea1.StyledText.Text=Text

//assign Font and Size to entire text
TextArea1.StyledText.Font(0,Len(Text))="Arial"
TextArea1.StyledText.Size(0,Len(Text))=14

//apply character highlights to 'my' in first paragraph
TextArea1.StyledText.Bold(8,2)=True
TextArea1.StyledText.textColor(8,2)=&cFF0000 //Red

//get positions of second paragraph; the index is zero-based.
st=TextArea1.StyledText.Paragraph(1).StartPos-1
ln=TextArea1.StyledText.Paragraph(1).Length

//Second paragraph in Bold
TextArea1.StyledText.Bold(st,ln)=True
//Second paragraph Centered
TextArea1.StyledText.ParagraphAlignment(1)=Paragraph.AlignCenter
```

The result is shown in Figure 339.

Figure 339. The `StyledText` as it appears in a `TextArea`.



This example happens to work with the `StyledText` object “hooked up” to the `TextArea`, but you can also work with styled text “offline.” You declare a `StyledText` object in a `Dim` statement and operate on it without reference to any control. When you’re ready to display it, you can assign it to the `StyledText` property of a `TextArea`. You would do this with a line such as:

```
Dim st As New StyledText
//do whatever you want right here; when you're done, just write...
TextArea1.StyledText=st
```

You can also export the styled text as a series of `StyleRuns` and read them back in and reconstruct the `StyledText` object using the `AppendStyleRun` method. See the entries on `StyleRun` and `StyledText` in the *Language Reference* for more information.

Working with Text Encodings

All computers use encoding systems to store character strings as a series of bytes. The oldest and most familiar encoding scheme is the ASCII encoding. It is documented in the *Language Reference*. It defines character codes for only values 0-127. These values include only the upper and lowercase English alphabet, numbers, some symbols, and invisible control codes used in early computers. You can use the `Chr` function to get the character that corresponds to a particular ASCII code.

Many extensions to ASCII have been introduced which handle additional symbols, accented characters, non-Roman alphabets, and so forth. In particular, the Unicode encoding is designed to handle any language and a mixture of languages in the same string. REAL Studio supports two different Unicode formats, UTF-8 and UTF-16. All of your constants, string literals, and so forth are stored internally using UTF-8 encoding.

If the strings you work with are created, saved, and read within REAL Studio, you shouldn't have to worry about encoding issues because REAL Studio stores the encoding it uses along with the content of the string.

If you are creating applications that open, create, or modify text files that are created outside of REAL Studio, you need to understand how text encodings work and what changes you may need to make to your code to make sure it continues to work properly.

Text Encodings: From ASCII to Unicode

As you know, computers don't really store or understand characters. They store each character as a numeric code. For example, the Return is ASCII character number 13. When the computer industry was in its infancy, each computer maker came up with their own numbering scheme. A numbering scheme is sometimes called a *character set*. It is a mapping of letters, numbers, symbols, and invisible codes (like the carriage return or line feed) to numbers. With a character set, information can be exchanged between computers made by different manufacturers.

In 1963 the American Standards Association (which later changed its name to the American National Standards Institute) announced the American Standard Code for Information Interchange (ASCII) which was based on the character set available on an English language typewriter.

Over the years, computers became more and more popular outside of the United States and ASCII started to show its weaknesses. The ASCII character set defines only 128 characters. That covers what is available on an English-language typewriter, plus some special "control" characters that can be used on computers to control output. It doesn't include special characters that are commonly used in typeset books such as curved quotes or the curved apostrophe, bullet characters, and long dashes—like this one. Also, many languages (like French and German) use accented characters that are not defined as part of the ASCII specification.

When the Macintosh and Windows operating systems were introduced, each OS defined extensions to standard ASCII by defining codes from 128-255. This enabled

both operating systems to handle accented characters and other symbols that are not supported by the ASCII standard. However, the Macintosh and Windows extensions do not agree with one another. Cross-platform applications have to build in some way of managing text that uses characters in the 128-255 range.

The problem is even worse for users of languages that don't use the standard Roman alphabetic characters at all—like Japanese, Chinese, or Hebrew. Because there are so many characters, the character sets devised to support some of these languages use two bytes of data per character (rather than one byte per character, as in ASCII).

Apple eventually created various text encodings to make it easier to manage data. MacRoman is a text encoding for files that use ASCII. MacJapanese is a text encoding for files that store Japanese characters. There are others as well. But these encodings were Mac specific. They didn't make exchanging data with other operating systems any easier and mixing data with different encodings (typing a sentence in Japanese in the middle of an English-Language document, for example) was problematic.

In 1986, people working at Xerox and Apple Computer both had different problems to solve that required the same solution. Before long, the concept of a universal character encoding that contained all the characters for all languages, became the obvious solution. The universal encoding was dubbed "Unicode" by one of the people at Xerox that helped to create it. Unicode solves all of these problems. Any character you need from any language is supported and will be the same character on any computer that supports Unicode. And as a bonus, you can mix characters from different languages together in one document since all are defined in Unicode.

Unicode support began appearing on the Macintosh with System 7.6 and on Windows with Windows 95. You could translate files between other text encodings and Unicode but Unicode was still the exception and not the rule. It wasn't until Mac OS X and Windows 2000 that Unicode became the standard.

Computer users are now in a transition. There are some using older systems where Unicode is not the standard. All new systems that are running Mac OS X, Windows, or Linux use Unicode as the standard encoding. As a result, you may have to deal with text files of different encodings for a while. That means you may need to modify your code to handle this. At some point in the future, it may be so rare that you can assume all files are in Unicode format but until then, you may need to make some modifications to your code so that your application operates properly when it encounters text with different types of encoding.

Changing Your Code To Handle Text Encodings

Unfortunately, there is no perfectly accurate way to determine the encoding of a file. You have to know what encoding the file is using. If it is coming from an English-speaking user of Windows 98, it's probably Windows ANSI.

If the encoding of a string is defined, you can use the Encoding function to get its encoding, like this:

```
TextEnc=Encoding(s)
```

where the variable `s` contains the string whose encoding is to be determined and `TextEnc` is a `TextEncoding` object. If the encoding is not defined, the `Encoding` function returns `Nil`.

Reading a Text File

This example code reads data from a text file and displays it in a `TextArea`. It makes no assumptions about encoding. If it's not a Unicode file, it may not display properly. This example has been kept simple intentionally to focus on the encoding issue:

```
Dim f As FolderItem
Dim t as TextInputStream
f = GetFolderItem("Sample.txt")
t = TextInputStream.Open(f)
TextArea1.text = t.ReadAll
t.close
```

If you know the encoding, this code can easily be changed to read the file properly. The `TextInputStream` class has an `Encoding` property that you can use to set the Encoding of the text that will be read using either the `Read` or `ReadAll` methods.

In the following example, the `Encoding` property of the `TextInputStream` is set to `MacRoman`. If the example file is in `MacRoman` format, it will display properly.

```
Dim f As FolderItem
Dim t as TextInputStream
f = GetOpenFolderItem(FileTypes1.Text)
If f<> Nil Then
    t = TextInputStream.Open(f)
    t.Encoding = Encodings.MacRoman
    TextArea1.text = t.ReadAll
    t.close
End if
```

The line that sets the value of the `Encoding` property uses the `Encodings` module. It contains functions for all the different encodings. The default encoding is UTF-8, a specific format of Unicode. When you type “`Encodings.`” in the Code Editor and press `Tab`, the autocomplete feature of the Code Editor will display all the encodings that are available.

You can set the Encoding in another way. Instead of assigning a `TextEncoding` to the `Encoding` property, you can pass it as an optional parameter to the `Read` or `ReadAll` methods. Here is an example of this:

```
Dim f As FolderItem
Dim t as TextInputStream
f = GetFolderItem("Sample.txt")
If f<> Nil Then
    t = TextInputStreamOpen(f)
    TextArea1.text = t.ReadAll( Encodings.MacRoman)
    t.close
End if
```

Writing a Text File

If your application needs to write out a file in a particular encoding, you must specify it when you write out data. This can come up when the file will be read by a different application, sent to someone in another country, or to someone using a different operating system.

You use the `ConvertEncoding` function to convert text in one encoding to another encoding. Here's a simple example that writes text from a `TextArea` to a file specifying `MacRoman` as the encoding. The `ConvertEncoding` function converts the encoding of the text to `MacRoman` before passing it to the `Write` method:

```
Dim f As FolderItem
Dim t as TextOutputStream
f = GetFolderItem("Sample.txt")
If f<> Nil Then
    t = TextOutputStream.Create(f)
    t.Write ConvertEncoding(TextField1.text, Encodings.MacRoman)
    t.Close
End if
```

If your application reads and writes its own files, you don't have to worry about this issue. UTF-8 is assumed when reading text files and is assumed when writing text to a file. If you do nothing, your files will be read as Unicode and will write out in Unicode format.

Getting an Individual Character

As was mentioned earlier, when you need to obtain an individual ASCII character, you can use the `Chr` function by passing it the ASCII code for the character you want. But if you want a non-ASCII character, you must specify the encoding as well. The `Chr` function is for the ASCII encoding only; you may not get the expected character if you pass it a number higher than 127. You should instead use the `Chr` method of the `TextEncoding` class. It requires that you specify both the encoding

and the character value. For example, the following returns the ™ symbol in the variable, s:

```
Dim s as String
s=Encodings.MacRoman.Chr(170)
```

Formatting Numbers, Dates, and Times

REAL Studio provides the ability to display and print numbers, dates, and times in many different formats.

Numbers

Numbers are stored unformatted. Fortunately, REAL Studio provides a Format function that makes providing formatting to numbers easy. To use this function, pass it a format specification and the number you wish formatted. The Format function then returns a string that represents the number with the formatting applied to it. The syntax for the Format function is:

```
result=Format(Number, FormatSpec)
```

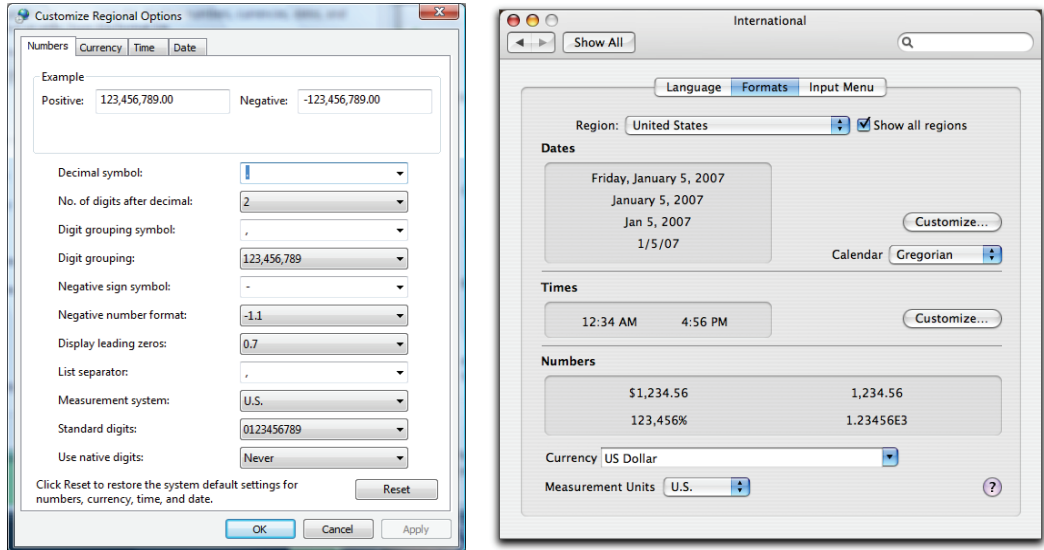
The FormatSpec is a string made up of one or more characters that control how the number will be formatted. For example, the format spec “\$###,##0.00” applies the dollars and cents formatting used in the United States.

Table 20: Characters used in FormatSpec.

Character	Description
#	Placeholder that displays the digit from the value if it’s present.
0	Placeholder that displays the digit from the value if it’s present. If no digit is present, 0 (zero) is displayed in its place.
.	Placeholder for the position of the decimal point.
,	Placeholder that indicates that the number should be formatted with thousands separators.
%	Displays the number multiplied by 100.
(Displays an open paren.
)	Displays a closing paren.
+	Displays a plus sign to the left of the number if the number is positive or a minus sign if the number is negative.
–	Displays a minus sign to the left of the number if the number is negative. There is no effect for positive numbers.
E or e	Displays the number in scientific notation.
\character	Displays the character that follows the backslash.

On Windows, the character that is used as the Decimal and Thousands separator is specified by the user in the Regional Settings Control Panel. In Mac OS X, these characters are specified on the Formats panel of the International system preference.

Figure 340. The Regional Settings and Formats Control Panels.



By default, the FormatSpec applies to all numbers. If you want to specify different FormatSpecs for positive numbers, negative numbers, and zero, simply separate the formats with semi-colons within the FormatSpec. The order in which you supply FormatSpecs is: *positive*, *negative*, *zero*. The last three examples in Table 21 on page 421 show this. It shows some examples of FormatSpecs:

Table 21: Examples of various FormatSpecs.

Format Syntax	Result
Format(1.784, "#.##")	1.78
Format(1.3, "#.0000")	1.3000
Format(5, "0000")	0005
Format(.25, "%")	25%
Format(145678.5, "#.##")	145,678.5
Format(145678.5, "#.##e+")	146e+5
Format(-3.7, "-#.##")	-3.7
Format(3.7, "+#.##")	+3.7
Format(3.7, "#.##; (#.##); \z\l\l\o")	3.7
Format(-3.7, "#.##; (#.##); \z\l\l\o")	(3.7)
Format(0, "#.##; (#.##); \z\l\l\o")	zero

Dates

Dates are objects and have properties that hold the date in various different formats. To get a date as a string formatted in a specific way, you simply access the appropri-

ate property. Table 22 on page 422 lists the properties of date objects and an example of the format the property contains:

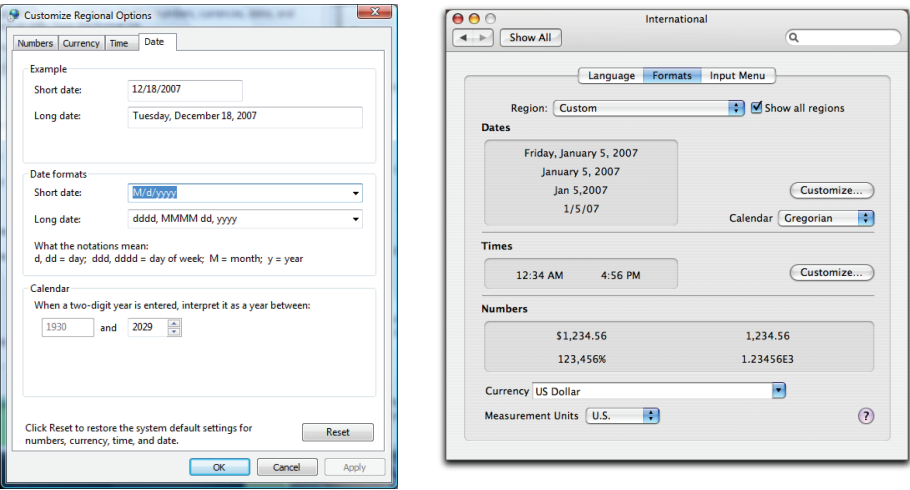
Table 22: Formatting properties of Date objects.

Property	Example (default)
ShortDate	12/31/97
LongDate	Wednesday, December 31, 1997
AbbreviatedDate	Wed, Dec 31, 1997

Date formats are controlled by the user’s Date Properties (Windows) or Date Formats (Macintosh) system settings. On Macintosh, the Date Formats dialog is accessed from the Formats panel of the International system preference (Click Customize... in the Date area in the Formats panel shown in Figure 341). On Windows, Date Properties is a screen in the Regional Options control panel. Users can choose the order of the day, month, year, as well as the separators.

These screens are shown in Figure 341.

Figure 341. The Date Formats (Windows) and International Formats (Macintosh) screens.



The Date class’s ShortDate, AbbreviatedDate, and LongDate properties use whatever format that the user has set in these system settings. Therefore, the example formats shown in Table 22 are not necessarily the ones that a particular computer will use.

To get the current date in any of these formats, simply create and instantiate a date object and then access the appropriate property. In this example, the current date formatted as a long date, is assigned to a variable:

```
Dim today as New Date
Dim theDate as String
theDate=today.LongDate
```

The TotalSeconds property of a Date object is the 'master' property that stores the date/time associated with the object. The TotalSeconds property is defined as the number of seconds since 1/1/1904.

Other property values are derived from TotalSeconds. If you change the value of the TotalSeconds property, the values of the Year, Month, Day, Hour, Minute, and Second properties change to reflect the second on which TotalSeconds occurs. Conversely, if you change any of these properties, the value of TotalSeconds changes commensurately.

Times

Time values are stored as part of a date. Date objects have two properties that store time values in two different formats. Table 23 lists the two properties and shows examples of how the time is returned.

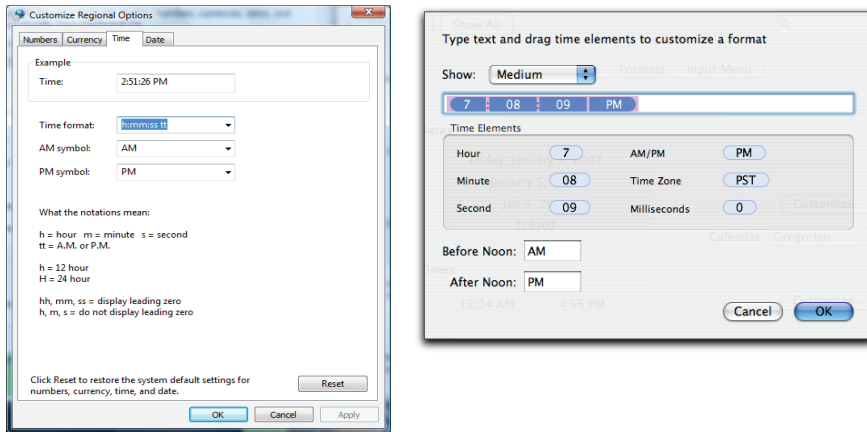
Table 23: Formatting properties of Time objects.

Property	Example
ShortTime	2:32 PM
LongTime	2:32:34 PM

To get the current time in either of these formats, create and instantiate a Date object and then access the appropriate property. In this example, the current time formatted as a LongTime, is assigned to a variable:

```
Dim today as New Date
Dim Now as String
Now=today.LongTime
```

As is the case with date formats, several aspects of the Short Time and Long Time formats are controlled by the user via the Time screen in the Regional Settings Control panel on Windows or the Time Formats panel in International preferences (Macintosh). These screens are shown in Figure 342 on page 424.

Figure 342. The Time Formats and Time Properties screens.

Searching using Regular Expressions

REAL Studio enables you to search and replace text using *regular expressions*. Regular expressions use a meta-language in which you can search for special characters, specific characters (e.g., only vowels), and search by position (e.g., at the beginning or end of a line). You use the language to define the string to search for and (optionally) the replacement string.

You use the properties of the RegEx class to define a regular expression search/replace or search operation. Table 24 shows these properties:

Table 24: Properties of the RegEx class.

Name	Description
Options	These options are various states which you can set for the Regular Expressions engine. See Table 26.
ReplacementPattern	This is the replacement string, which can include references to substrings matched previously, via the standard '\1' or '\$1' notation common in regular expressions. This pattern is used either with the Replace property or passed to the RegExMatch class when Search returns, and subsequently used with Replace if no parameters are specified.
SearchPattern	This is the pattern you are currently searching for.
SearchStartPosition	Character position at which you want to start the search if the optional TargetString parameter to <i>Replace</i> is not specified. Keep in mind if you set it, it will only be used if you don't specify a TargetString, since setting a new TargetString resets the value.

The methods of the `Regex` class, shown in Table 25, are used to do the find and replace.

Table 25: Methods of the `Regex` class.

Name	Parameters	Description
Replace	Optional: TargetString as String; SearchStartPosition as Integer	Finds <i>SearchPattern</i> in Target and replaces the contents of <i>SearchPattern</i> with <i>ReplacementPattern</i> . Returns the resulting String. Replace can take the optional parameters shown at left. Returns a String.
Search	TargetString as String SearchStartPosition as Integer	Finds <i>SearchPattern</i> in <i>TargetString</i> . If it succeeds it returns a <code>RegexMatch</code> . The <code>RegexMatch</code> will remember the <i>ReplacementPattern</i> specified at the time of the search.

The `RegexOptions` class lets you specify the options shown in Table 26:

Table 26: Regular Expression options.

Name	Description
CaseSensitive	Specifies whether case is to be considered when matching a string. The default is <code>False</code> .
DotMatchAll	Normally the period matches everything except a new line, this option allows it to match new lines.
Greedy	<p>Greedy means the search finds everything from the beginning of the first delimiter to the end of the last delimiter and everything in-between. For example, Say you want to match the following bold-tagged text in HTML:</p> <p>The <code>quick</code> brown <code>fox</code> jumped If you use this pattern:</p> <p style="text-align: center;"><code>.+</code></p> <p>You end up matching "<code>quick</code> brown <code>fox</code>", which isn't what you wanted.</p> <p>So, you can turn <i>Greedy</i> off or use this syntax:</p> <p style="text-align: center;"><code>.+?</code></p> <p>and you will match "<code>quick</code>", which is exactly what you wanted.)</p>

Table 26: Regular Expression options.

Name	Description
LineEndType	This is in effect for the current Regular Expression "session") Has no effect on SearchPatterns if TreatAsOneLine is True. Changes the way \n is expanded for ReplacementPatterns 0 = any line ending (Mac or Win32 or Unix) 1 = defaultForPlatform (if running on Mac same as 2) (if running on Win32 same as 3) 2 = Mac ASCII 13 or \r 3 = Win32 ASCII 10 or \n 4 = Unix ASCII 10 or \n
MatchEmpty	Indicates whether patterns are allowed to match the empty string.
ReplaceAllMatches	Indicates whether all occurrences of the pattern are to be replaced.
StringBeginsLineBegin	Indicates whether a string's beginning should be counted as the beginning of a line.
StringEndsLineEnd	Indicates whether a string's end should be counted as the end of a line.
TreatTargetAsOneLine	Ignores internal newlines for purposes of matching against '^' and '\$'.
UTF8	If True, treats all processed characters as UTF8 characters. If False, ASCII is assumed.

When you find a match using a regular expression search, you use the properties of the `RegexMatch` class to obtain the matching string (or strings) and optionally replace the match with a string that you provide. The properties of the `RegexMatch` class are shown below:

Table 27: Properties of the `RegexMatch` class.

Name	Description
SubExpressionCount	Number of SubExpressions that are available with the search just performed. SubExpressions allow replacement of parts of the pattern.
SubExpressionString	Returns the SubExpression as a string for the given matchNumber. 0 returns the entire MatchString (the implicit 0 th subExpression), and 1 is the first real subExpression.
SubExpressionStart	Returns the starting position of the subExpression given by the matchNumber parameter.
Replace	Substitutes the matched result in a manner specified by the given <i>ReplacementPattern</i> . If no <i>ReplacementPattern</i> is specified, it uses the ReplacementPattern which was specified in the <code>Regex</code> object at the time of the search.

See the entries in the *Language Reference* for more information about these three classes and a description of the syntax of regular expressions.

Adding Pictures and Drawing Graphics

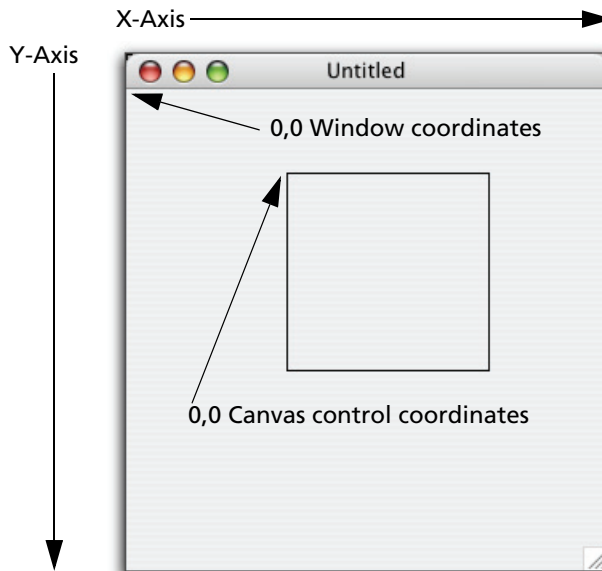
You can add pictures from documents or draw your own pictures in REAL Studio. In some cases you can add the graphics you want without writing any code. When you do need to write code, REAL Studio provides methods for creating all kinds of graphics.

Understanding the Coordinates System

Most of the graphics methods require you to indicate the location inside the window or within a Canvas control where you wish to begin drawing. This location is specified using the coordinates system. This system is a grid of invisible horizontal and vertical lines that are 1 pixel apart. If you have never done a computer drawing with a coordinates system, you might expect the origin (0,0) to be in the center of the window, but it's not. The origin is always in the upper-left corner of the area. For the entire screen, this is the upper-left corner of the screen. For a window, the origin is the upper-left corner of the window, and for a control, it's the upper-left corner of the control. The X axis (the horizontal axis) increases in value moving from left to right and the Y axis (the vertical axis) increases in value moving from top to bottom.

So, a point that is at 10, 20 (within a window) is 10 pixels from the left side of the window and 20 pixels from the top of the window. If you are working within a Canvas control, the point 10, 20 is 10 pixels from the left edge of the control and 20 pixels down from the top edge of the control.

Figure 343. The X,Y Coordinates System.



Displaying Pictures In a Window

There are different techniques you use to display pictures in a window. The technique you use depends on what you plan to do with the picture.

Using the Entire Window

If you want to use a window to display a picture, the window's Backdrop property is one way to do it. The Backdrop property is a picture that will be displayed behind any controls in the window. By default, the Backdrop is set to "None" meaning that no Backdrop picture will be displayed.

There are several ways to assign a picture to a window's BackDrop property. You can assign a picture to a window's Backdrop property by dragging a picture document into your Project Editor and then choosing it by name as the picture for the Backdrop property in the Properties pane. If you are not displaying the Project Editor, you can instead drag the picture to the Tabs bar. The IDE will accept the picture and add it to the Project Editor. If you are using the Window Editor, the Backdrop property will list the picture in its drop-down list.

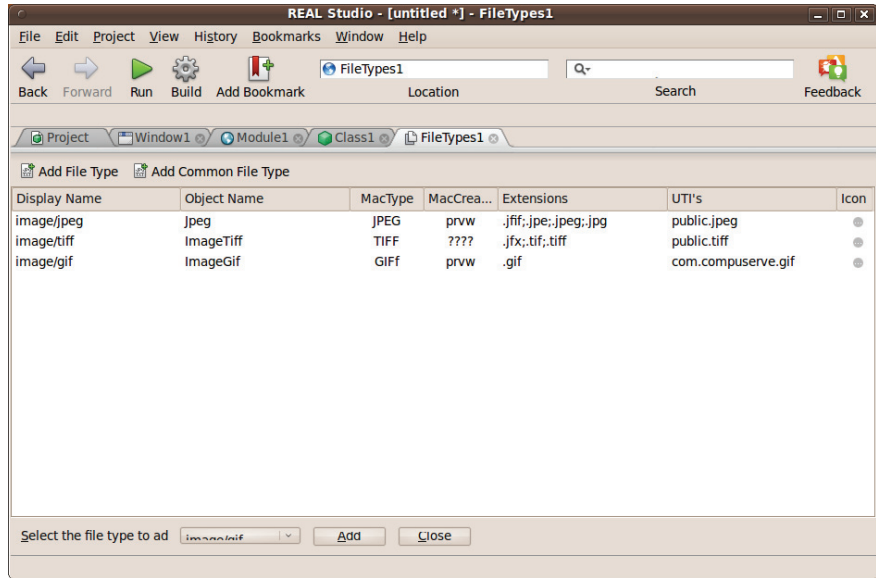
If the picture is not already added to the project, you can display the Backdrop property's drop-down list and choose Browse. The IDE displays an open-file dialog in which you can select a picture. When you select a picture, it appears as the window's backdrop and is also added to the Project Editor.

If you wish, you can also add a picture using code. For example, in a window's Open event handler, you can write:

```
Backdrop=picturename
```

where *picturename* is the name of the image, as it appears in the Project Editor.

This example presents the standard open file dialog box and lets the user choose a TIFF, JPEG, or GIF file to be used as the backdrop of the current window. The file types used as parameters in the GetFolderItem call must be defined in the File Type Sets Editor or by the FileType class. In this case, the TIFF, JPEG, and GIF file types were defined individually in File Type Sets editor as components of FileTypes1. In the File Type Sets editor, use the Add Common File Types button to add these three file types, as shown in Figure 344 on page 429.

Figure 344. JPEG, TIFF, and GIF file types defined in the File Type Sets Editor.

Thus, "FileTypes1.All" refers to all three types:

```
Dim f as FolderItem
f=GetOpenFolderItem(FileTypes1.All)
If f<> Nil Then
    Backdrop=f.OpenAsPicture
End If
```

After you have assigned a picture to the BackDrop property, you can then resize the window to the size of the picture by setting the window's width and height properties to the backdrop's width and height properties:

```
width=Backdrop.width
height=Backdrop.height
```

You don't need to worry about redrawing the Backdrop. REAL Studio will handle redrawing the Backdrop when necessary.

Using a Portion of the Window

If you only want to display the picture in an area in the window, you can use an ImageWell control. An ImageWell is similar to a Canvas control, except that it has no drawing tools: you can only display an image that has been created elsewhere.

To assign a picture to an ImageWell's Image property, simply drag it from the desktop to the Project Editor and then assign it to the ImageWell's Image property using its Properties pane. Or, you can use the Browse menu command in the ImageWell's Image pop-up menu in the Properties pane.

You can also add a picture at runtime by loading an image using code. For example, the following code displays an open-file dialog box that allows the user to choose a PICT, JPEG, or GIF file and display it in the ImageWell. It is assumed that the file types used as parameters in `GetOpenFolderItem` have been assigned in the File Type Sets Editor or with the `FileType` class via the language.

```
Dim f as FolderItem
f=GetOpenFolderItem(FileTypes1.All)
if f <> Nil then
    ImageWell1.Image=f.OpenAsPicture
End if
```

You can also allow your users to drag a picture document from the Finder to the ImageWell rather than using the open-file dialog box. In the ImageWell's Open event handler, allow a file drop using the line:

```
Me.acceptfileDrop(FileTypes1.All)
```

In the ImageWell's `DropObject` event handler, use the code:

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

Your other option is to use a Canvas control to display a picture in a portion of the window. This type of control also gives you a graphics area that can be drawn in and also receives events. You might use a Canvas control if you need to display a picture that the user will interact with. With the Canvas control's `Backdrop` property you can display an existing picture. This can be done manually in the Window Editor by clicking on the Canvas control in a window to select it and then choosing a picture from the `Backdrop` property's pop-up menu in the Properties pane. The picture must have been added to the project in the Project Editor. A picture can also be assigned to the `Backdrop` property at runtime. This example displays an open-file dialog box when the user clicks on the Canvas control and then lets the user choose a picture to be displayed in the Canvas control:

```
Dim f as FolderItem
f=GetOpenFolderItem(FileTypes1.All)
If f<> Nil Then
    Me.Backdrop=f.OpenAsPicture
End If
```

You can also support drag and drop to a Canvas control in the same manner as for ImageWells. The advantage of using a Canvas control is that you can also customize the image using the methods of the Graphics class using the Canvas control's `Paint` event handler. This feature is described in the following section.

Creating Pictures

You can create pictures programmatically using the methods of the Graphics class. A Graphics object is simply an object in memory that holds an image. For example, windows and Canvas controls have a Paint event. This event is executed any time the window or Canvas control needs to be redrawn. For example, when a window opens, its Paint event is executed because the contents of the window needs to be drawn. Any Canvas controls in a window will also execute their Paint event when the window opens because the Canvas control needs to be drawn. These Paint events are also executed when a portion of the window and/or Canvas control that was previously hidden by another window is exposed.

The Paint event is passed a Graphics object. When the Paint event is finished executing, this graphics object will be drawn in the window or Canvas control. You draw in a window or Canvas control by calling the drawing methods of this graphics object.

Displaying Pictures

You can display a picture in a Graphics object using the DrawPicture method of the Graphics class. This method is used for three common tasks:

- Displaying a picture
- Cropping a picture
- Scaling a picture

The method takes as many as nine parameters depending on your objective. If you want to display a picture full size, it needs only three parameters. It is passed a picture and the coordinates that describe where you want the picture drawn within the graphics object.

In this case, its syntax is:

```
g.DrawPicture image, x, y
```

where *Image* is the picture to be displayed and *x* and *y* are the distance in pixels from the top-left corner of the control.

This example uses the Paint event of a Canvas control to draw two pictures that have been dragged into the Project Editor (BartPict and LisaPict) side by side:

```
Sub Paint(g As Graphics)
  g.DrawPicture BartPict, 0,0
  g.DrawPicture LisaPict,BartPict.Width, 0
```

Note that the second call to DrawPicture offsets the horizontal coordinate by the width of the first image.

Copying a Portion of a Picture

The DrawPicture method of the Graphics class can be used to copy a portion of a picture to a Graphics object. This is done using the optional parameters of the DrawPicture method. The parameters allow you to specify the portion of the picture

you want to draw. You can specify the coordinates where you wish to begin copying from the picture as well as the amount (in width and height) you wish to copy. In this case, the parameters are:

```
Image X, Y, DestWidth, DestHeight, SourceX, SourceY, SourceWidth, SourceHeight
```

The following example draws a 20 pixel square portion of the source picture starting 10 pixels from the left and 10 pixels from the top of the source picture and drawing the picture 5 pixels from the left and 5 pixels from the top of the Canvas control or window background:

```
Sub Paint(g As Graphics)  
    g.DrawPicture Lisa,5,5,Lisa.width,Lisa.height,10,10,20,20
```

Scaling Pictures

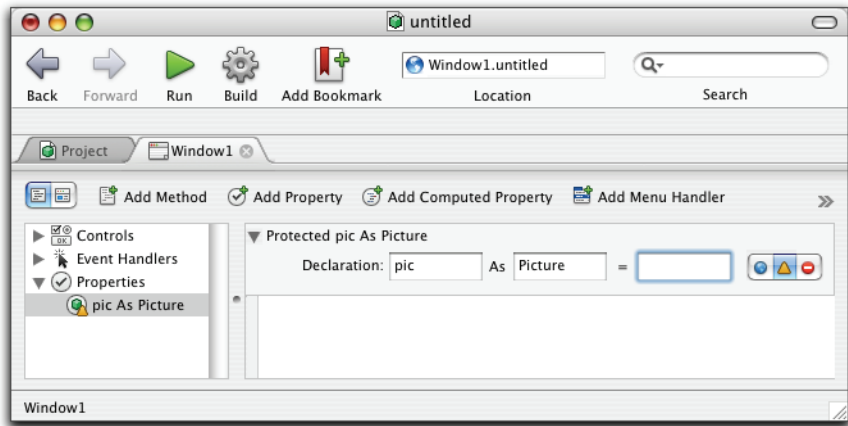
The DrawPicture method of the Graphics class can scale a picture when it is drawn. To do this, you must include all of the DrawPicture parameters. Scaling is done by specifying a destination width and/or height that is larger or smaller than the picture's original width and/or height. This example draws a picture at two times its original size:

```
Sub Paint(g As Graphics)  
    Dim w,l as integer  
    w=lisa.width  
    l=lisa.height  
    g.DrawPicture lisa, 0,0,w*2,l*2,0,0, lisa.width,lisa.height
```

Here's a very useful example that automatically scales an imported picture to fit in a Canvas control.

Assume that you have a window that has a Canvas control and a PushButton (or other control) that triggers code that does the import and scaling.

Create a property of the parent window that will hold the scaled picture. For example:



In the Action event of the PushButton, use the following code to import the picture and do the scaling:

```
Dim f As FolderItem
Dim p As Picture
Dim maxWidth, maxHeight As Integer
Dim factor As Double

maxWidth = Canvas1.width
maxHeight = Canvas1.height

f = GetOpenFolderItem(FileTypes1.ImagePict)
If f <> Nil then
    p = f.OpenAsPicture
end if

factor = Min( maxWidth / p.Width, maxHeight / p.Height )
factor = Min( factor, 1.0 ) // (don't scale it up if it's too small!)

pic = NewPicture( p.Width * factor, p.Height * factor, 32 )
pic.graphics.DrawPicture p, 0,0,pic.width,pic.height, 0,0,p.width,p.height

Canvas1.Refresh
```

This assumes you want to import a PICT file and you have defined a file type of ImagePict in the File Type Sets Editor using the Common File Type of image/pict.

The values of `maxWidth` and `maxHeight` are set in this example to the size of the Canvas control, but you can, of course, supply other values. The `DrawPicture` method of the `Graphics` class uses the following parameters:

Parameter	Type	Description
Image	Picture	The picture to be drawn at the location specified by X and Y.
X	Integer	X coordinate within the control of the left side of the image. Defaults to zero.
Y	Integer	Y coordinate within the control of the top of the image. Defaults to zero.
DestWidth	Integer	Width of Image within the control.
DestHeight	Integer	Height of Image within the control.
SourceX	Integer	X coordinate of the picture you copy from. Defaults to zero.
SourceY	Integer	Y coordinate of the picture you copy from. Defaults to zero.
SourceWidth	Integer	Width of the picture you wish to copy. Defaults to width of the picture.
SourceHeight	Integer	Height of the picture you wish to copy. Defaults to height of the picture.

The final step is to add code to the Paint event of the Canvas control to actually assign the scaled image, `pic`, to the `Backdrop` property of the Canvas.

```
If pic <> Nil then  
    g.DrawPicture pic,0,0  
End if
```

Scrolling Pictures

A picture that is drawn into a Canvas with the `DrawPicture` method can be scrolled by calling the Canvas class's `Scroll` method. It takes three parameters: the picture to be scrolled, and the amounts to be scrolled in the horizontal and vertical directions.

To use the `Scroll` method to scroll the picture in a Canvas control, you need to store the last scroll value for the axis you are scrolling so you can use this to calculate the amount to scroll. This can be done by adding properties to the window that contains the Canvas control or by creating a new class based on the Canvas control that contains properties to hold the last X scroll amount and last Y scroll amount.

The following example scrolls a picture. The picture has been added to the project. The properties `XScroll` and `YScroll` have been added to the window to hold the amounts the picture has been scrolled.

A convenient way to scroll a picture is with the four arrow keys. To do this, you place code in the `KeyDown` event handler of the active window. This event receives each keystroke. Your code can test whether any of the arrow keys have been pressed

and then take the appropriate action. For example, this code in the KeyDown event of the window scrolls the picture 8 pixels at a time:

```
Function KeyDown (Key as String) as Boolean
Select Case Asc(Key)

Case 31 'up arrow
Yscroll=YScroll-8
Canvas1.Scroll 0,-8

Case 29 'Right arrow
Xscroll=XScroll-8
Canvas1.Scroll -8,0

Case 30 'Down arrow
Yscroll=Yscroll+8
Canvas1.Scroll 0,8

Case 28 'Left arrow
Xscroll=Xscroll+8
Canvas1.Scroll 8,0

End Select
```

The Paint event of the Canvas has the line of code that draws the picture:

```
g.DrawPicture myPicture, Xscroll,Yscroll
```

Drawing Standard Dialog Icons

REAL Studio has a MsgBox function for displaying a standard message box with a note icon and an OK button and a more versatile MessageDialog box.

If you need to use these icons in a message dialog box, you can get them via the MessageDialog class. The MessageDialog class can create a dialog box with a main text message, a subordinate explanation, an icon, and one to three buttons. The user must respond to the dialog by clicking one of the buttons and your code can detect the button the user clicked and take appropriate action.

You can set the icon to be displayed simply by setting the value of the MessageDialog's Icon property. The MessageDialog object positions the icon, text, and buttons for you. For more information about the MessageDialog class, see the section "Message Dialog Boxes" on page 108 and the entry for the MessageDialog class in the *Language Reference*.

However, there may be times when you need to design your own dialog box. The Graphics class has methods that draw any of the standard system icons and, unlike the MessageDialog class, you can position the icon anywhere you want.

Figure 345. Mac OS X, Windows XP, Vista, and Linux Note, Caution, and Stop icons.



The Graphics class provides the `DrawNoteIcon`, `DrawCautionIcon`, and `DrawStopIcon` methods that make it easy to display these icons in a Canvas control or a window background. These methods make system calls that draw the correct icon for the current version of the operating system. Using these methods, you will display the appropriate icon for the user's platform. The following example draws the note icon in a Canvas control in its Paint event:

```
Sub Paint(g As Graphics)  
    g.DrawNoteIcon 0,0
```

Drawing Pixels

You can get and set the color of individual pixels in a Graphics object using the Pixel property. You use this property by passing its X and Y coordinates and then setting the color of that pixel to a color object or getting its color.

This example draws pixels at randomly selected coordinates within a Graphics object using randomly selected colors until the user presses Escape or ⌘-Period (Macintosh):

```
Sub Paint(g As Graphics)  
    Dim c as Color  
    Do  
        c=Rgb(Rnd*255,Rnd*255,Rnd*255)  
        g.Pixel(Rnd*Me.Width,Rnd*Me.Height)=c  
    Loop until UserCancelled
```

This example gets the color of the pixel the mouse is over in a Canvas control and fills another Canvas control called PixelColor with that color:

```
Sub MouseMove(X As Integer, Y As Integer)
    Dim c as Color
    c=Me.Graphics.Pixel(X,Y)
    PixelColor.Graphics.ForeColor=c
    PixelColor.Graphics.FillRect 0,0,PixelColor.Width,PixelColor.Height
```

Drawing Lines Lines are drawn using the DrawLine method of the Graphics class. The color of the line is the color stored in the ForeColor property of the Graphics object the line is being drawn in. To use the DrawLine method, you pass it starting coordinates and ending coordinates of the line.

This example uses the DrawLine method to draw a grid inside a Canvas control or window background. The size of each box in the grid is defined by the value of the boxSize variable:

```
Sub Paint(g as Graphics)
    Dim i, boxSize as Integer
    boxSize=10
    For i=boxSize to Me.Width Step boxSize
        g.DrawLine i,0,i,Me.Height
    Next
    For i=boxSize to Me.Height Step boxSize
        g.DrawLine 0,i,Me.Width,i
    Next
```

The thickness of the line is controlled by the PenHeight and PenWidth properties of the Graphics object.

Drawing Ovals

Ovals are drawn with the DrawOval and FillOval methods of the Graphics class. Both require the same parameters: the X and Y coordinates where the oval starts and the width and height of the oval. Both draw ovals using the ForeColor property of the Graphics object. Both use the PenWidth and PenHeight properties of the Graphics object to determine the line thickness. The difference between the two is that DrawOval draws only the border of the oval, leaving the interior blank. FillOval draws an oval with the interior filled with the ForeColor.

This example draws an oval in a Canvas control or Window background:

```
Sub Paint(g as Graphics)
    g.DrawOval 0,0,50,75
```

Drawing Rectangles

Rectangles are drawn using the DrawRect, FillRect, DrawRoundRect, and FillRoundRect methods of the Graphics class. All of these methods use the ForeColor property of the Graphics object and the PenWidth and PenHeight properties to

determine the line thickness. All of these methods require the X and Y coordinates of the upper-left corner of the rectangle, as well as the width and height of the rectangle. RoundRectangles are rectangles with rounded corners. Therefore, DrawRoundRect and FillRoundRect require two additional parameters: the width and height of the curve of the corners.

DrawRect and DrawRoundRect both draw empty rectangles. FillRect and FillRoundRect draw solid rectangles.

This example draws a rectangle and fills it with the color red.

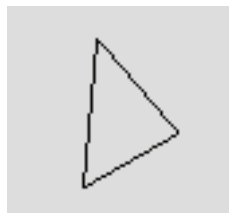
```
Sub Paint(g as Graphics)
    g.DrawRect 0,0,150,100
    g.ForeColor=&cFF0000
    g.FillRect 0,0,Canvas1.Width,Canvas1.Height
```

Drawing Polygons

Polygons are drawn using the DrawPolygon and FillPolygon methods of the Graphics class. Polygons are drawn by passing the DrawPolygon or FillPolygon method an integer array that contains each point in the polygon. This is a 1-based array where odd numbered array elements contain X values and even numbered array elements contain Y coordinates. This means that element 1 contains the X coordinate of the first point in the polygon and element 2 contains the Y coordinate of the first point in the polygon. Consider the following array values:

Element #	Value
1	10
2	5
3	40
4	40
5	5
6	60

When passed to the DrawPolygon or FillPolygon method, this array would draw a polygon by drawing a line starting at 10,5 and ending at 40,40 then drawing another line starting from 40,40 ending at 5,60 and finally a line from 5,60 back to 10,5 to complete the polygon. This polygon has only three sets of coordinates so it is a triangle.



The code in the Paint event of a Canvas control or Window to draw this polygon, looks like this:

```
Sub Paint(g As Graphics)
  Dim points(6) as Integer
  points(1)=10
  points(2)=5
  points(3)=40
  points(4)=40
  points(5)=5
  points(6)=60
  g.DrawPolygon points
```

FillPolygon draws the same polygon but with the interior filled with the ForeColor:



Another way of populating the array is with the Array function. The Array function takes a list of values separated by commas and populates the array, beginning with element zero. Since the first element that DrawPolygon uses is element 1, you can use any value in element zero:

```
Sub Paint(g As Graphics)
  Dim points() as Integer
  points=Array(0,10,5,40,40,5,60)
  g.DrawPolygon points
```

Drawing into a Region in the Graphics Object

When you are drawing a complex image that involves many calls to Graphics methods, you may want to create non-overlapping regions within the area. You then draw into each “child” area, with the assurance that each drawing will not inadvertently overlap another object.

You create a child region within the parent area with the Clip method of the Graphics class. You pass it the top-left corner of the child region and its width and height. It returns a new Graphics object that is the specified region inside the parent area. You can then draw into the child area just as with any other Graphics object. The only difference is that the drawing will be confined to the child area. The coordinates of each call are with respect to the top-left corner of the child area.

Here is an example of how this works. This code is in the Paint event of a Canvas. Two regions at the top of the Canvas are defined by calls to the Clip method. Subsequent calls to the DrawRect method show where the clippings are. Calls to the DrawOval method draw shapes within the clipped areas. Notice that the first call

attempts to draw outside the area. If you were drawing from the parent Graphics object, the first oval would bump into the second.

```
Sub Paint (g as Graphics)
    Dim myclip as Graphics = g.clip(0,0,150,15)
    Dim myclip2 as Graphics=g.clip(150,0,150,15)

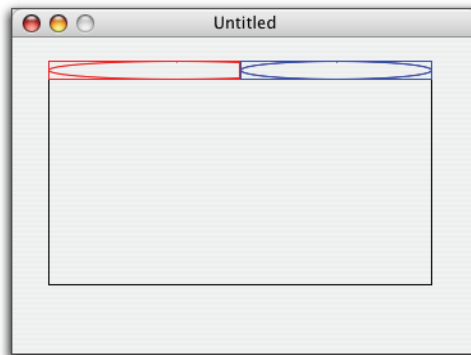
    //draw the border of the Canvas in black
    g.forecolor=&c000000
    g.drawrect(0,0,g.width,g.height)

    //draw into the first area in red
    myclip.ForeColor=&cff0000
    myclip.DrawRect(0,0,myclip.width,myclip.height) //draw the border of the area..
    myclip.DrawOval(0,0,200,15) //try to draw outside its clip..

    //draw into the second area in blue
    myclip2.ForeColor=&c0000ff
    myclip2.DrawRect(0,0,myclip2.width,myclip2.height) //draw the border
    myclip2.DrawOval(0,0,150,15)
```

Here is the Canvas that is produced from this code.

Figure 346. Two regions defined by the Clip method within a Canvas control.



Creating Custom Controls with the Canvas Control

Visible controls (controls that have a graphical interface the user can interact with directly, like PushButtons) are pictures that have code that controls how they are drawn. This means that a Canvas control can easily be used to create controls that are not built-in to REAL Studio.

Suppose you wanted to create a simple custom control like a rectangle whose fill color toggles from black to white when clicked. First you would drag a Canvas control into a window. You want the rectangle to switch colors when the user clicks the mouse, so this code goes in the MouseDown event handler of the Canvas control. The code checks to see if the rectangle is black and, if it is, fill it in white; otherwise

fill it in black. You can check the color of any particular pixel using the Pixel property of the graphics property of the Canvas control. You can determine if a pixel is a particular color by comparing it to a color value returned by the Rgb function. Passing 0 (zero) to each of the parameters of the Rgb function returns the color black. Passing 255 to each parameter of the Rgb function returns the color white. You will learn more about color later in this chapter. So, the code for the MouseDown event handler looks like this:

```
Function MouseDown(X As Integer, Y As Integer) As Boolean
  If Me.Graphics.Pixel(X,Y)=Rgb(0,0,0) Then
    Me.Graphics.ForeColor=Rgb(255,255,255)
  Else
    Me.Graphics.ForeColor=Rgb(0,0,0) //black
  End If
  Me.Graphics.FillRect Me.Left,Me.Top,Me.Width,Me.Height
```

This code checks to see if the pixel the user clicked on is black and, if it is, the ForeColor property of the graphics object of the Canvas control (generically represented here using the Me function) is set to white, otherwise it's set to black. Next, the FillRect method of the Graphics property of the Canvas control is called to fill the rectangle with the color stored in the ForeColor property.

There's one more step before our custom control is complete. If the Canvas control needs to be redrawn for some reason (such as when the window first opens or the user moves another window in front of the one with the Canvas control), REAL Studio calls the Canvas control's Paint event handler to redraw the Canvas control. If there is no code in the Paint event handler, REAL Studio won't draw the rectangle and, to the user it will seem to appear and disappear at different times, which will be confusing. To solve this problem, you need to put a slightly altered version of the code you have in the MouseDown event handler in the Paint event handler:

```
Sub Paint(g As Graphics)
  If g.Pixel(0,0)=Rgb(0,0,0) Then
    g.ForeColor=Rgb(255,255,255)
  Else
    g.ForeColor=Rgb(0,0,0)
  End If
  g.FillRect Me.Left,Me.Top,Me.Width,Me.Height
```

Since the Paint event handler is passed a reference to the Graphics object of the Canvas (the g parameter), you can make the code a bit more generic and use "g" instead of "me.graphics". Also, since the user isn't clicking anywhere, you need to choose a pixel whose color you check. In this example we chose the pixel at 0,0.

This is an example of a very simple custom control. More complex and generic controls can be created using classes. See “Creating Custom Interface Controls with Classes” on page 583 for more information.

Working with Vector Graphics

The REAL Studio language includes a group of classes that enable you to create, open, and save vector graphics. A vector graphic (as opposed to a bitmap graphic) is composed entirely of primitive objects — lines, rectangles, text, circles and ovals, and so forth — that retain their identity in the graphic. They don’t “decompose” and become part of an indistinguishable bitmap. The Object2D class in the REAL Studio language is the base class for all the classes that create primitive objects. They are shown in Table 28:

Table 28: Classes used to draw vector graphics.

Class	Description
ArcShape	Draws an arc of a circle.
CurveShape	Draws straight lines or curves using one or more “control points.”
FigureShape	Draws polygons that can (optionally) have curved sides.
OvalShape	Draws circles and ovals.
PixmapShape	Imports a bitmap picture into the image.
RectShape	Draws a square or rectangle.
RoundRectShape	Draws a square or rectangle with rounded corners (subclassed from RectShape).
StringShape	Draws text strings in a specified font, font size, and style.

Since each class in Table 28 is subclassed from Object2D, you can use the Object2D’s properties to draw the object. They are shown in Table 29.

Table 29: Properties of the Object2D class.

Name	Description
Border	Opacity of the border, from 0 (transparent) to 100 (opaque). The default is 0.
BorderColor	Color of the border.
BorderWidth	Width of the border, in points. The default is 1.
Fill	Opacity of the interior, from 0 (transparent) to 100 (opaque). The default is 100.
FillColor	Color of the interior of the shape.
Rotation	Clockwise rotation, in radians, around the X, Y point.
Scale	Scaling factor relative to the object’s original size.
X	Horizontal position of center or main anchor point.
Y	Vertical position (down from top) position of center or anchor point.

Each class in Table 28 has additional properties that pertain to the object's particular shape or characteristics. For example, the `RectShape` class has additional properties for the rectangle's height and width. The `RoundRectShape` class adds properties for specifying the width and height of the rounded corners and the number of straight line segments used to approximate the curved corners. Please refer to the entries in the *Language Reference* for information on each property.

Drawing and Displaying a Vector Object

You draw a single vector object simply by instantiating it and specifying its properties. For example, the following code draws a `RoundRectShape`:

```
Dim r as New RoundRectShape
r.width=120
r.height=120
r.border=100
r.bordercolor=RGB(0,0,0) //black border
r.fillcolor=RGB(255,102,102)
r.cornerHeight=15
r.cornerWidth=15
r.borderwidth=2.5
```

The only problem with this is that the shape doesn't appear anywhere. It's just "defined"—ready for your use. You need to add a command to draw the vector object in another object such as a window or a control that can display graphics such as a `Canvas` control.

The `Paint` event of a control or window is a good place to insert code that draws vector objects. The `Paint` event passes a `Graphics` object to the event handler, so you only need to call the `DrawObject` method of the `Graphics` class to draw the object. Access to the `Graphics` class is provided by the passed parameter, `g`. The `DrawObject` method takes three parameters, the object to be drawn and its `X` and `Y` coordinates in the `Graphics` space.

The following code in the `Paint` event of a `Window` draws the finished vector object:

```
Dim r as New RoundRectShape
r.width=120
r.height=120
r.border=100
r.bordercolor=RGB(0,0,0) //black border
r.fillcolor=RGB(255,102,102)
r.cornerHeight=15
r.cornerWidth=15
r.borderwidth=2.5
g.DrawObject r,100,100 //draw at 100,100
```

You can also create composite vector graphics objects that are made up of several individual vector graphics objects. The composite object is a `Group2D` object—it's

just a group of Object2D objects. Use the Append or Insert methods of the Group2D class to add individual vector graphic objects to the Group2D object. When you are finished, draw the object using one call to the DrawObject method.

The following code illustrates this. We've taken the code shown above and added a second RoundRectShape object and moved it 20 pixels to the right and 20 pixels down from the original RoundRectShape. The two Append statements create the composite object. The code is in the Paint event of a window, so the parameter, g, provides access to the Graphics class.

```
Dim r as New RoundRectShape
Dim s as New RoundRectShape
Dim group as New Group2D

r.width=120
r.height=120
r.border=100
r.bordercolor=RGB(0,0,0) //black border
r.fillcolor=RGB(255,102,102)
r.cornerHeight=15
r.cornerWidth=15
r.borderwidth=2.5

s.width=120
s.height=120
s.border=100
s.bordercolor=RGB(0,0,0) //black border
s.fillcolor=RGB(255,102,102)
s.cornerHeight=15
s.cornerWidth=15
s.borderWidth=2.5

s.x=r.x+20 //shift s 20 pixels to right
s.y=r.y+20 //shift s 20 pixels down

group.append r
group.append s
g.drawObject group,100,100
```

When you want to add vector graphics to an existing bitmap image, you create a Group2D object and then add each object to the Group2D using its Append

method. This example appends a StringShape to a PixmapShape. The graphic, h1, has been dragged to the Project Editor.

```
Dim px as PixmapShape
Dim s as StringShape
Dim d as New Group2D

px=New PixmapShape(h1) //h1 is a graphic in the Project Editor
d.append px

s=New StringShape
s.y=70
s.Text="This is what I call a REAL car!"
s.TextFont="Helvetica"
s.Bold=true
d.append s
graphics.drawobject d,100,100
```

Opening and Saving Vector Graphics

Two methods of the FolderItem class are relevant to vector graphics—OpenAsVectorPicture and SaveAsPicture. The OpenAsVectorPicture method opens a PICT file (Macintosh) or an .emf file (Windows) and attempts to convert the objects in the PICT to editable REAL Studio Object2D objects. The original file may contain certain elements that do not have a REAL Studio equivalent, but OpenAsVectorPicture will do its best to map these objects.

The SaveAsPicture has an optional parameter that specifies the format to use when doing the save. The GIF, TIFF, WindowsBMP, JPEG, and PNG formats are supported on all platforms. On Windows, GDI+ must be installed. If GDI+ is not installed, REAL Studio will try to use QuickTime. If neither are installed, only WMF and EMF are supported natively on Windows.

Please refer to the FolderItem entry in the *Language Reference* for information on the values of the optional parameter.

Working With Color

Color in REAL Studio is a data type. It consists of three values that define a color. A color can be specified using either the RGB, HSV, or CMY models. You use the three relevant Color properties to set the color. For example, to use the RGB model, use the RGB function and set values for the Red, Green, and Blue properties. These values range from 0 to 255. The RGB function returns a Color when passed values for the amount of red, green, and blue. Several classes have Color properties. For example, the ForeColor property of the Graphics class is a Color.

If you need to store a Color, you can create a property or variable of type Color and then use the RGB, HSV, or CMY function. In this example, a new variable of type Color is created and the values for the white are assigned using the RGB function:

```
Dim c as Color
c=Rgb(255,255,255)
```

You can also assign a color value directly, without using the RGB, HSV, or CMY functions. You use the RGB color model with the following syntax to specify a color:

```
&cRRGGBB
```

where RR is the hexadecimal value for Red, GG is the hexadecimal value for Green, and BB is the hexadecimal value for Blue. Each value goes from 00 to FF rather than 0 to 255. For example, the following is equivalent to the previous example:

```
Dim c as Color
c=&cFFFFFF
```

“FF” is hexadecimal for 255. There is an easy way to obtain the hexadecimal values. Using the Add Constant declaration area, you can define a constant of type color and use the built-in Color Picker to choose a color visually. When you select a color, REAL Studio figures out the hex values and inserts them in the Value area of the dialog box. For an example, see the section “Adding a Constant to a Module” on page 376.

In this example, the ForeColor property of a Graphics object is set to blue so the text drawn will be in that color:

```
Sub Paint(g as Graphics)
  g.ForeColor=RGB(9,13,80)
  g.DrawString "Hello World",50,50
```

Determining the RGB Values For a Color

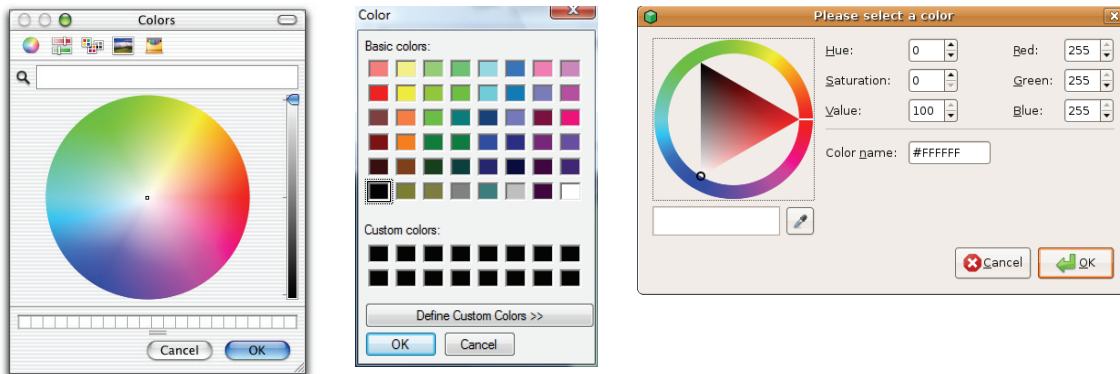
If you need to assign a color at runtime but aren’t sure which RGB values to use to get a particular color, you can use the Mac OS Color Picker. The following code displays the Color Picker:

```
Dim c as Color
Dim b as Boolean
b=SelectColor(c,"Choose a color")
if b then //user chose a color
  // do something with selected color here
end if
```

If the user cancelled out of the Color Picker dialog box, the boolean variable, `b`, is False; otherwise, the selected color is returned in the color object, `c`, and is available for assignment to a color property of an object.

You may have already used the Color Picker to assign a color to a control's property. If you haven't, the Color Picker displays a color palette and allows you to click on one to pick it (hence the name). Figure 347 on page 447 shows the Color Pickers on Mac OS X, Windows, and Linux.

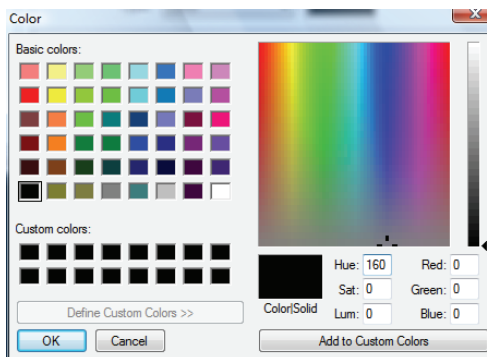
Figure 347. The Macintosh, Windows, and Linux Color Pickers.



In the Mac OS X color pickers, you click on a color and a sample of the color appears in the patch area at the top of the screen. When you click OK, REAL Studio translates your selection into RGB values.

The Windows version of the Color Picker uses only one format, shown in Figure 347. You can either select one of the predefined colors or click the Define Custom Colors button to display the “advanced” color picker, which depicts colors on a continuum and lets you specify the color using either the RGB or HSV models.

Figure 348. The Windows Custom Colors Picker.



Select a color by clicking on a point in the color spectrum or enter values in the RGB or HSV areas. Click Add to Custom Colors to add the custom color to one of the Custom Color samples on the left side of this dialog.

The Pixel Property of Graphics Objects

The Pixel property of a Graphics object lets you get and set the color of the pixel you specify. This property is an example of a property whose data type is Color. In this example, the Paint event handler is setting a pixel to black if it is white and white if it is black:

```
Sub Paint(g As Graphics)
    If g.Pixel(10,20)=Rgb(0,0,0) Then
        g.Pixel(10,20)=Rgb(255,255,255)
    Else
        g.Pixel(10,20)=Rgb(0,0,0)
    End if
```

You can see that the code to check the color of a pixel and set the color of a pixel is basically the same.

Printing Text and Graphics

REAL Studio provides a lot of flexibility when it comes to printing. You can display the Page Setup dialog box and store the settings the user chooses.

Printing is almost exactly the same as drawing text and graphics into a Canvas control or the graphics property of a Window. When you call the OpenPrinter or OpenPrinterDialog function, a Graphics object is returned. To print, you simply draw your text and graphics into this Graphics object. To cause the page to print, you call the NextPage method of the Graphics object. This method forces the Graphics object to be printed, then clears it so you can use it again to draw the next page.

Working with the Page Setup Dialog Box

The PrinterSetup class lets you create an object that can be used to display the Page Setup dialog box, get and set the individual Page Setup settings, as well as store and restore these settings. To display the Page Setup dialog box, call the PageSetupDialog method of the PrinterSetup object you have instantiated. This method returns True if the user clicks the OK button in the Page Setup dialog box and False if he clicks the Cancel button. The PrinterSetup class has properties for accessing all of the settings in the Page Setup dialog box (page orientation, scale, etc.). For a list of PrinterSetup properties, the PrinterSetup class in the *Language Reference*. However, in most cases you won't have to deal with these properties because a composite version of these settings is stored in the SetupString property. The SetupString property is read/write and is used to get all of the PrinterSetup settings as a string so you can store them and to restore that string later on. For example, in a document-based application, a string property could be added to the document window that stores the SetupString value. When the user chooses to display the Page Setup dialog box

(in most applications by choosing Page Setup from the File menu), a `PrinterSetup` object is created and its `SetupString` property is assigned the value in the window property storing these settings. Then the Page Setup dialog box is displayed showing these settings. In this example, the window property is called “Settings”:

```
Dim ps as PrinterSetup
ps=New PrinterSetup
ps.SetupString=Settings
If ps.PageSetupDialog Then
    Settings=ps.SetupString
End if
```

If the user clicks OK in the Page Setup dialog box, the window’s `Settings` property is assigned the value of the `SetupString` because settings in the Page Setup dialog box may have been changed by the user.

`PrinterSetup` class objects can be optionally passed as a parameter to the `OpenPrinter` and `OpenPrinterDialog` functions so that the Page Setup settings can be used during printing.

If you wish to store the `PrinterSetup`’s `SetupString` property with the document when the user saves the document (assuming you provide this capability), you will probably need to store it in a string resource in the resource fork of the document. See “Working With Macintosh Resources” on page 524 for more information on the resource fork.

The Page Setup dialog box is not supported in REAL Studio for Linux. Calling the `PrinterSetup` function will return `False` and no dialog will be presented to the user.

Printing With The Print Dialog Box

You use the `OpenPrinterDialog` function to display the Print dialog box and print. If the user clicks the OK button in the Print dialog box, a `Graphics` class object is returned. If the user clicks the Cancel button, the `Graphics` object returned will be `Nil`. To create the first page to be printed, you utilize the `Graphics` object returned, calling the various `Graphics` class methods such as `DrawString`, `DrawLine`, `DrawOval`, `DrawPicture`, etc. Once you have created the page, you can send the page to the printer by calling the `NextPage` method of the `Graphics` class. This method will both send the page to the printer for printing and clear the `Graphics` object so you can begin creating the next page.

This example displays the Print dialog box then prints “Hello” on the first page and “World” on the second page:

```
Dim page as Graphics
page=OpenPrinterDialog()
If page<> Nil Then
    page.DrawString "Hello", 50, 50
    page.NextPage
    page.DrawString "World", 50, 50
    page.NextPage
End if
```

If you enter a page range in the Print dialog box, it is ignored and the entire document is printed.

If you are storing the SetupString property of the PrinterSetup class object, you can optionally pass this string to the OpenPrinterDialog function if you want it to use the settings stored in the SetupString. This example assumes that the SetupString is stored in a window property called “Settings” and passes it to the OpenPrinterDialog function for consideration during printing:

```
Dim page as Graphics
Dim ps as PrinterSetup
ps=New PrinterSetup
If Settings <> "" Then
    ps.SetupString=Settings
End If
page=OpenPrinterDialog(ps)
If page <> Nil Then
    page.DrawString "Hello", 50, 50
    page.NextPage
    page.DrawString "World", 50, 50
    page.NextPage
End If
```

For more information on the OpenPrinterDialog function, see the OpenPrinterSetup function in the *Language Reference*.

Printing Without The Print Dialog Box

To print without displaying the Print dialog box, call the OpenPrinter function. This function is identical to the OpenPrintDialog function except that it doesn't display the Print dialog box before printing. For information on printing, see the section “Printing With The Print Dialog Box” on page 449. For more information on the OpenPrinter function, see the OpenPrinter function in the *Language Reference*.

Printing Styled Text

Because TextAreas are capable of displaying styled text and multiple font sizes, you will usually want to retain the styled text in your reports. The StyledTextPrinter class supports this capability. It uses the DrawBlock method (rather than the Draw-

String method) to accomplish this. Here is a simple example that prints the contents of a `TextArea` as styled text.

```
dim stp as StyledTextPrinter
dim g as graphics
g=openPrinterDialog()
if g <> Nil then
    stp=TextArea1.StyledTextPrinter(g,72*7.5)
    stp.drawBlock 0,0,72*9
end if
```

The parameters of `DrawBlock` are the top-left x, y coordinates on the page and the height of the block. This example starts at the top-left corner. See the description of the `StyledTextPrinter` class in the *Language Reference* for more information.

Transferring Text and Graphics with the Clipboard

The Clipboard is a class of object in REAL Studio with properties and methods. The properties and methods let you determine what kind of data is available on the Clipboard, get data from the Clipboard, and send data to the Clipboard. The Clipboard class supports three kinds of data: text, picture, and binary. Binary data is represented in string form and is marked with a type you specify so you can tell what the binary data represents.



NOTE: For `TextFields` and `TextAreas`, REAL Studio handles the Cut, Copy, and Paste operations of the Edit menu automatically. However, for other controls that contain data such as `Canvas` and `ListBox` controls, this is not the case.

To access the Clipboard for any reason, you must first create a new object of type `Clipboard`:

```
Dim c as Clipboard
c=New Clipboard
```

In the event handler that opened the Clipboard, you must call the `Clipboard` object's `Close` method or an error may occur.

Testing The Clipboard For Specific Data Types

You can test the Clipboard using the following methods and properties all of which return `True` or `False`: `TextAvailable`, `PictureAvailable`, and `RawDataAvailable`. `RawDataAvailable` is used to determine if a specific kind of binary data (usually data put there by your application) is available. To use the `RawDataAvailable` method, you must pass it the `MacType` string that represents the type of data. This string was passed when the binary data was passed when the data was put on the Clipboard.

Getting Data From The Clipboard

Once you know what kind of data is available on the Clipboard, you can get the data using the Text, Picture, and RawData properties. In this example, if text is available, the text is placed in a variable called "ClipText."

```
Dim c as Clipboard
Dim ClipText as String
c=New Clipboard
If c.TextAvailable Then
    ClipText=c.Text
End If
C.Close
```

If a picture is available, the picture is placed in a variable called "ClipPict."

```
Dim c as Clipboard
Dim ClipPict as Picture
c=New Clipboard
If c.PictureAvailable Then
    ClipPict=c.Picture
End If
C.Close
```

In this example, rows from a ListBox that have been copied to the Clipboard are added to a ListBox:

```
dim theRows as string
dim c as clipboard
c=New Clipboard
If c.RawDataAvailable("rows") Then
    theRows=c.RawData("rows")
    Do
        ListBox1.AddRow Left(theRows,InStr(theRows,EndOfLine)-1)
        theRows=Mid(theRows,InStr(theRows,EndOfLine)+1)
    Loop until theRows=""
End If
c.Close
```



Putting Data On The Clipboard

NOTE: Remember, you must call the Clipboard object's Close method in the event handler that opened the Clipboard or an error may occur.

You can put text, picture, or binary data (in the form of a string) on the Clipboard. To do this, you create a new Clipboard object then use the appropriate method or property based on the type of data you wish to put on the Clipboard.

Data Type	Method or Property
Text	SetText method
Picture	Picture property

Data Type	Method or Property
Binary Data	AddRawData method

In this example, text is added to the Clipboard:

```
Dim c as Clipboard
c=New Clipboard
c.SetText "Hello World"
c.Close
```

In this example, a picture from Canvas1 is copied to the Clipboard:

```
Dim c as Clipboard
c=New Clipboard
c.Picture=Canvas1.Backdrop
c.Close
```

In this example, rows from a ListBox are copied to the Clipboard. They are copied using the AddRawData method so they don't appear as text on the Clipboard:

```
Dim i as Integer
Dim c as Clipboard
Dim rows as String
c=New Clipboard
For i=0 to ListCount
    If ListBox1.Selected(i) Then
        rows=rows+ListBox1.List(i)+EndofLine
    End If
Next
c.AddRawData rows,"rows"
c.Close
```



Remember, you must call the Clipboard object's Close method in the event handler that opened the Clipboard or an error may occur.

Creating Reports

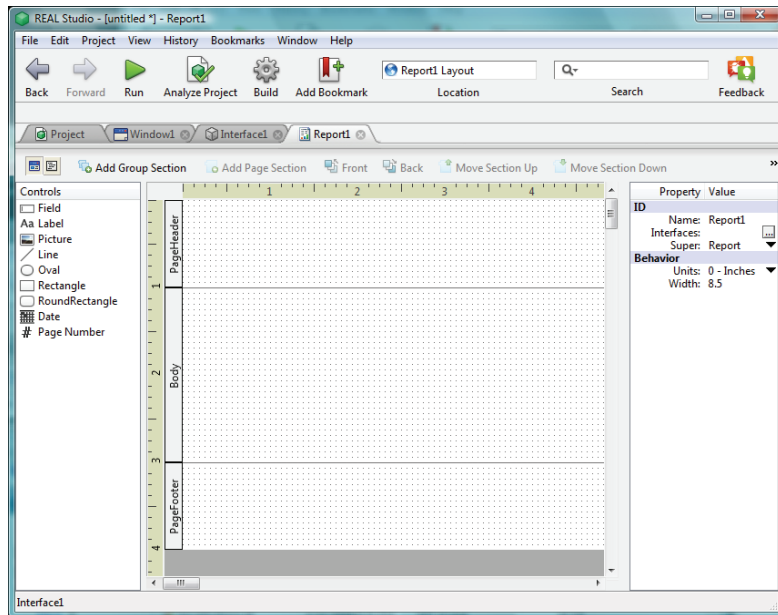
REAL Studio's integrated Report Editor allows you to design and print reports using a visual Report Layout Editor. You design report forms using the same familiar tools that you use to design your application's user interface.

The Report Layout Editor does not attempt to show a WYSWYG representation of a report. Instead, it allows you to design the report symbolically. Items added to the body of the report expand dynamically to show all the records in the selection. You can preview the report on screen to see how the report will look or print the report to a printer.

The Report Layout Editor

The Report Layout Editor resembles the Window and ContainerControl Layout Editors, with special tools for creating dynamic reports.

Figure 349. The Report Layout Editor.



The left panel contains the list of controls that can be added to a report. The right panel shows the properties for the selected object. The Report Editor has its own set of controls; you do not use any of the controls that you use to create a window or a container control.

Report Editor Controls

The Report Editor uses the following controls:

- **Field:** Use the field control to display the contents of a field. A field can be from a database field (for example, from a REAL SQL Server database) or from a text file. See the section “Report Editor Examples” on page 464 for more information on how to use data sources with reports.
- **Label:** Use the Label control to add static text to the report. Use it as you would a StaticText control in a Window Layout Editor.
- **Picture:** Use the Picture control to display either a static picture or an image from a data source. If it is a static image, it will display the picture assigned to the Image property. If it is from a data source, then it will display the field assigned to the DataField property. The Report Editor supports pictures stored as Binary fields in

the REAL SQL Database. For an example, see the List of Products report in the Database Example project and the section “A Picture Field Example” on page 463.

- **Line:** Use the Line control to add a line to a report. You can set its X and Y coordinates, color, and style (Solid, Transparent, Dot, or Dash).
- **Oval:** Use the Oval control to add an Oval to a report. You can set its position and size properties, the width and color of its border, and the fill color.
- **Rectangle:** Use the Rectangle control to add a rectangle to the report. You can set its position and size properties, the width and color of its border, the fill color, and the width and height of its corners.
- **Round Rectangle:** Use the Round Rectangle to add a round rectangle to the report.

Report Editor Toolbar

The Report Layout Editor Toolbar contains tools that are very similar to their counterparts in the Window Layout editor. A key difference is that the editing area is divided into sections that serve special purposes in a printed report. Different sections are used for different purposes. You can set its position and size properties, the width and color of its border, and the fill color. With the tools in the Report Editor toolbar, you can:

- **Add Group Section:** A break group consists of a group header and footer sections above and below the body section. The break section is controlled by a GroupByField. The section header, footer, and body repeat for every value of the GroupByField. For example, the Gas Report uses the Year field as the Group By field. For more information, see “Using a Text File as a Data Source” on page 470.
- **Add a Page section:** Adds a new page header and footer section to the report.
- **Move Section Up/Down:** If you have more than one group section, you can select a section and move it either up or down in the order in which the break sections execute.
- **Front and Back buttons:** Adjust the layers on which overlapping objects are on: The Front and Back buttons move object to the front or back layers. For example, the Rectangle control in the Gas Report should be moved to the back so that the fields and labels are visible.

Report Editor Areas

By default, the Report Layout editor is divided into three areas:

- **Page Header Area:** This area contains items that print once per page, at the top of the page.
- **Body Area:** This area prints rows of data from the data source. It represents the body of the report. One instance of the Body area is printed for each row of data in the data source.

- **Page Footer Area:** This area contains items that print once per page, at the bottom of the page and below the Body area.

Using the Add Group Section button, you can add group header and footer sections for a GroupBy field. Click the Add Group Section button to add the header and footer sections, then add a GroupByField to the group header area. Optionally, you can also add subtotal fields in the group footer section. Add additional Group Sections for each GroupBy field.

The sections can be resized by dragging the dividers vertically. The Body area should be designed so that it presents only one row of data. When the report runs, REAL Studio will populate the report with all of the selected records. You should adjust the dividers so that there is a suitable amount of white space above and below the rows in the printed report.

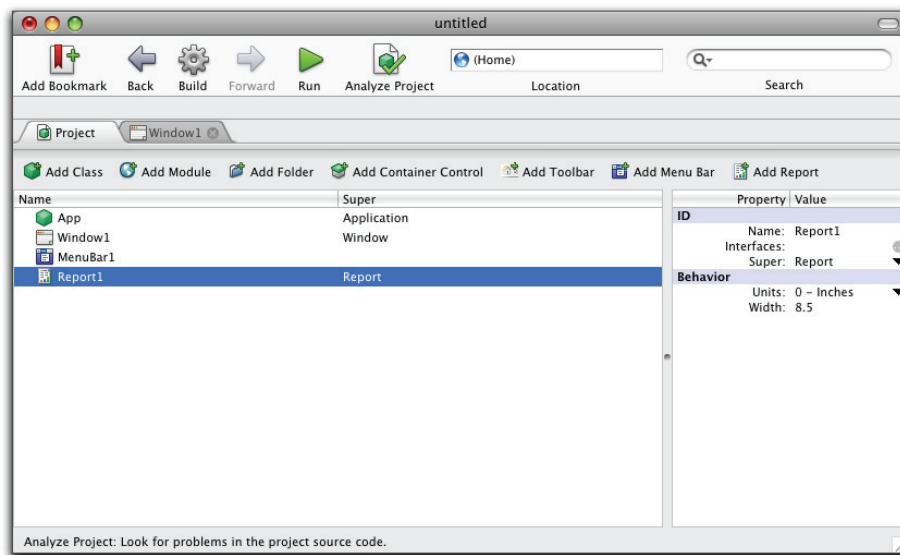
Adding a Report to a Project

To add a Report, do this:

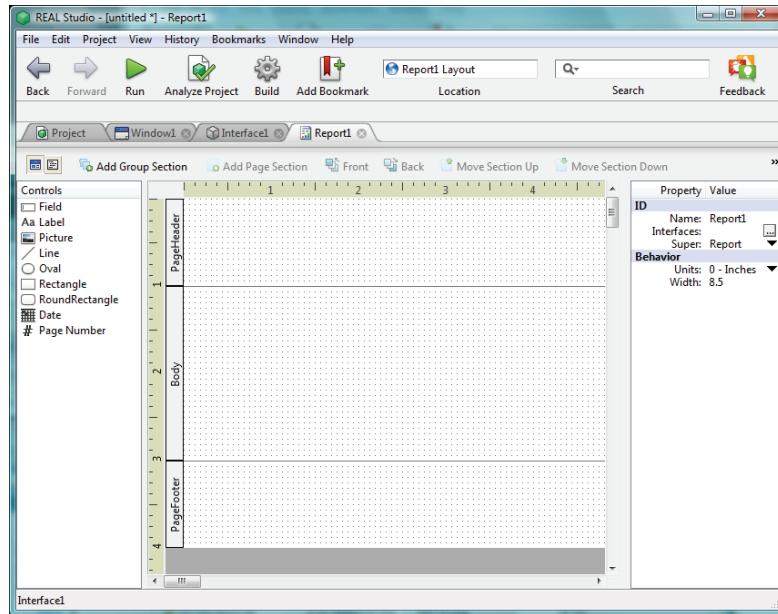
- 1 **Choose Project ► Add ► Report or click the Add Report button in the Project Editor Toolbar.**

An instance of the Report class appears.

Figure 350. The Report Editor added to a Project.

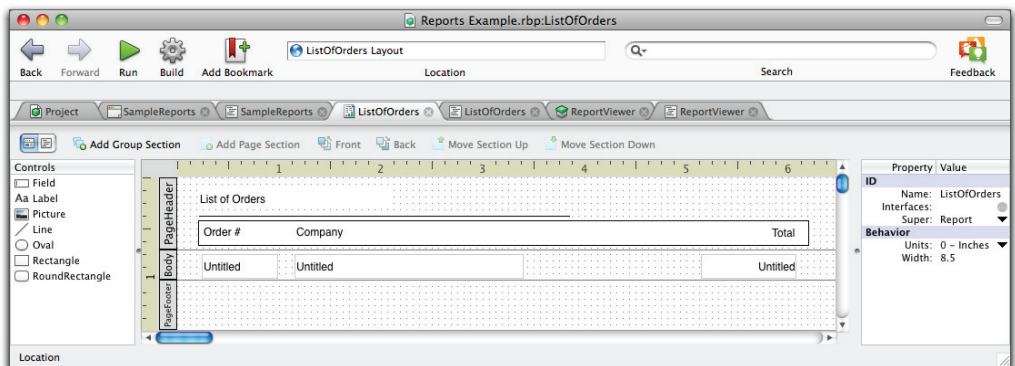


- 2 **Double-click the Report item in the Project Editor to display the Report Layout Editor.**

Figure 351. The Report Layout Editor.

In its default state, the Editing area is divided into the header, body, and footer areas. The content at the top of the page prints at the top of each printed page. The content in the body contain the rows of data in the report. The content in the page footer appears at the bottom of the printed page. In this configuration, all the records are printed in one group, in the Body section.

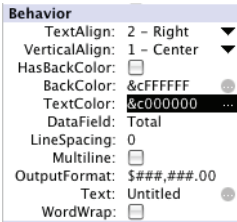
The following finished Report prints a simple listing. The Body and Page Header areas have been resized so that the report does not use excessive white space around each record.

Figure 352. A finished simple listing report.

The heading “List of Orders” is in the Page Header area and prints only at the top of the page. It is an instance of the Label control.

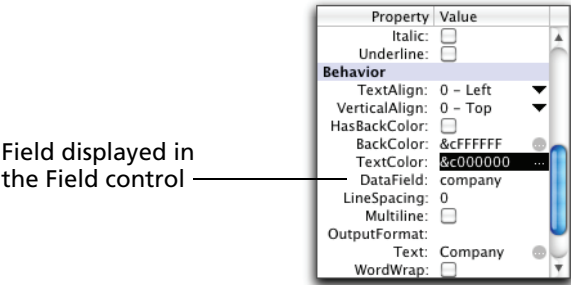
The Body area defines a row of data and the it has been resized to avoid excessive white space between rows. The Total field uses the Format property to display it in a Dollar format.

Figure 353. The Behavior properties of the Total field.



Each field contains a label that indicates which field is displayed. Each field has its Text property set to the label text so that it is clear which field is which. In this case, the data source is a REAL SQL Database. The data source is the company field in the database.

Figure 354. The Behavior properties of the Company field.

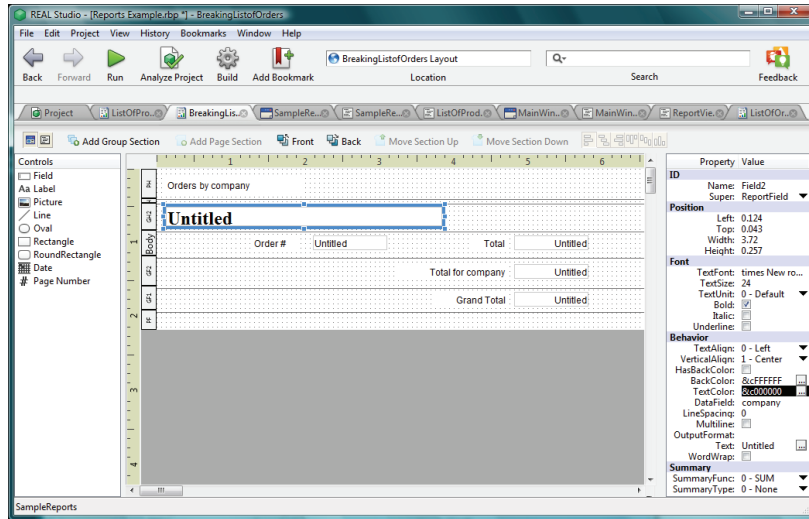


Adding a Grouping Section

If you need to break up the report into sections, add a *Group Section* by clicking the Add Group Section in the Report Editor toolbar. This enables you to divide up the report according to the values of one variable. You will get one section of the report per value of the grouping variable.

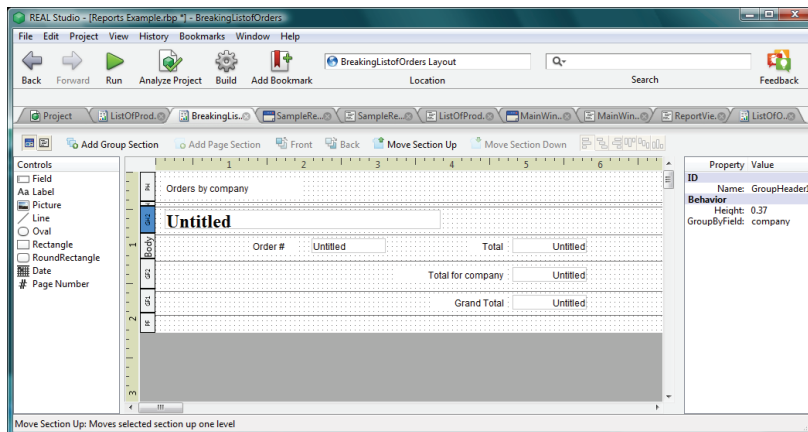
In the following report, a listing of prices is grouped by Company. Notice that the Company field that appeared in the Body area in the List of Orders report has ben moved into the Section header area. This section, together will all the records that belong to that company, is repeated for every company in the dataset.

Figure 355. The Section Header area for the Orders report.



If you click on the Section Break in the margin, you will see its properties in the Properties pane. Notice that the Company field is the GroupByField for that Break Section. Each Break Section must have a Group By field and you will get as many groups as there are values of the GroupByField. When you click on the section identifier in the margin, then the properties of the section are shown in the Properties pane.

Figure 356. The GroupByField in the BreakingListofOrders report.



With Grouping sections, you can use the Group Footer area to put summary statistics for each group. This area will repeat, once per group.

If you need to group your data on more than one field, you add a second grouping section. For example, if the records in the report are identified by both month and year, you can have a level 1 section for year and level 2 by month.

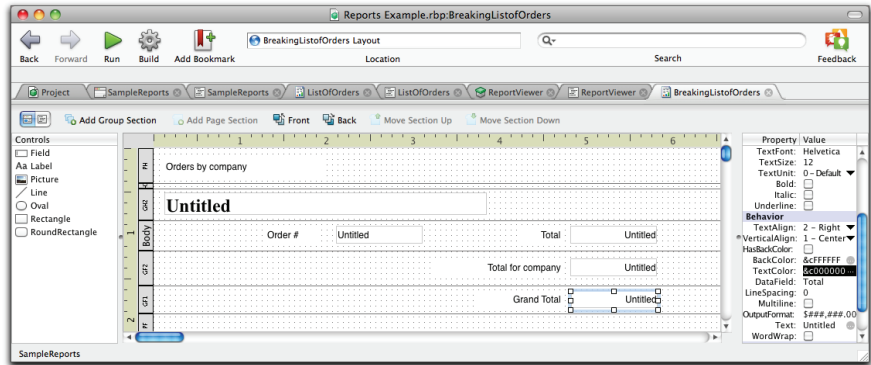
The Section Footer

When you add a Group Section, you also get a group footer section. In this case, the group footer contains the subtotal for each company.

In this example, the version of the report with a break section organizes the listing by company. This report design produces a separate listing for each company and a separate *subtotal* for each company.

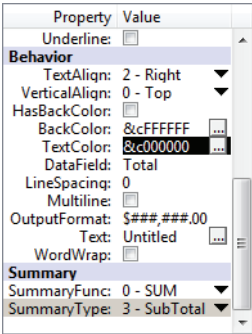
In this report, the section footer area has a new variable that computes the subtotal on the Total field that is in the Body area.

Figure 357. The Section footer field in the Orders report.



The field in the footer area is a calculated field. In its Properties pane, the DataField is set to the same field as in the Body field, but the Summary area specifies the SUM summary function and the SummaryType is SubTotal.

Figure 358. The Summary area of a subtotal field.



Adding a Section Break to a Report

To add a Section break to a report, do this:

- 1 Click the Add Group Section button in the toolbar.

Group Header 1 and Group Footer 1 sections appear above and below the Body section.

2 Add fields, labels, and any other objects to the group header and footer areas.

The section header area must have a GroupByField. The footer area does not need to be populated or have a subtotal field. If you wish to add a subtotal field, set up the field as shown in Figure 358.

Using a Picture Field

The Report Layout Editor includes built-in support for pictures that are stored in the REAL SQL Database. The database stores images as Binary fields but the reporting engine internally converts Binary fields to a form that can be viewed and printed. You do not need to do anything special to use a Picture field in a report.

To display a picture in the database, use the ReportPicture field on the layout. Set the DataField property to the Picture field in the database. You can also assign a picture to the Image property. If there is no picture in the database for a particular record, then the Image property is used instead.

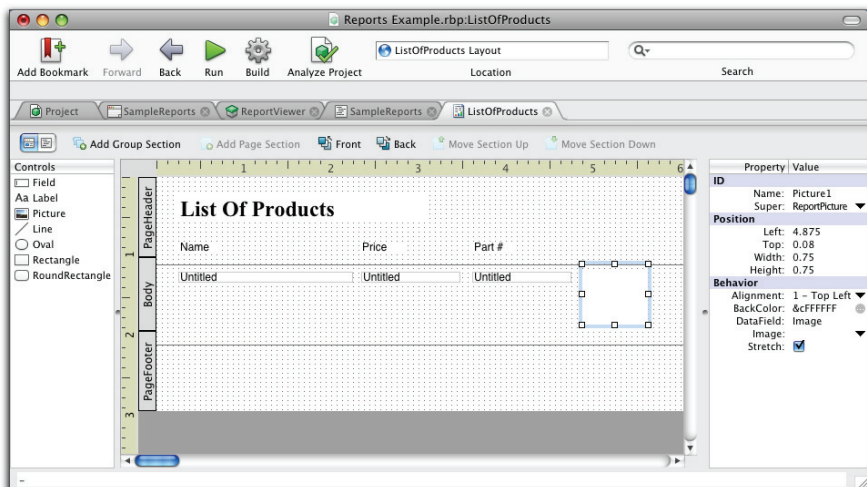
In addition, there are two other properties that control the appearance of the picture inside the ReportPicture control:

- **Stretch:** If you select Stretch, the picture will be stretched or scaled to fit the dimensions of the ReportPicture field.
- **Alignment:** You can align the picture inside the ReportPicture. Your choices are: Center, Top Left, Top Right, Bottom Left, and Bottom Right. If Stretch is selected, then the image fills the area of the ReportPicture regardless of Alignment.

A Picture Field Example

The Database Example includes a simple listing report of the contents of the Products table, including the product image. The layout is shown in Figure 359.

Figure 359. A report that includes a Picture field.



As noted, the Properties Pane for the of the ReportPicture field set the DataField to “Image.” The Stretch property is checked so that each image fits into the ReportPicture field. The ReportPicture’s properties are shown in Figure 360.

Figure 360. The ReportPicture’s properties.

Property	Value
ID	
Name:	Picture1
Super:	ReportPicture ▼
Position	
Left:	4.875
Top:	0.08
Width:	0.75
Height:	0.75
Behavior	
Alignment:	1 – Top Left ▼
BackColor:	&cFFFFFF
DataField:	Image
Image:	▼
Stretch:	<input checked="" type="checkbox"/>

This report is otherwise the same as a simple listing report and uses the same code.

Report Editor Examples

The REAL Studio standard install includes two example projects that illustrate all the main aspects of the new reporting engine. One project uses a REAL SQL Database as the data source and the other uses a comma-delimited text file.

Using a Database as a Data Source

The main purpose of the Database Example project is to illustrate how to create a relational REAL SQL Server application with REAL Studio. It contains its own notes that lead you through the design of a database project. It has been augmented with three reports. Two reports illustrate the simple and breaking list reports. A third report illustrates how to use a picture field in a report. It is also a simple listing report.

The project includes a new menu item in which a report can be requested and displayed. A new window has been added to the project for the display of the outputs of the reports. Since the reporting engine returns its output as pictures, the new window uses a Canvas control to display the output.

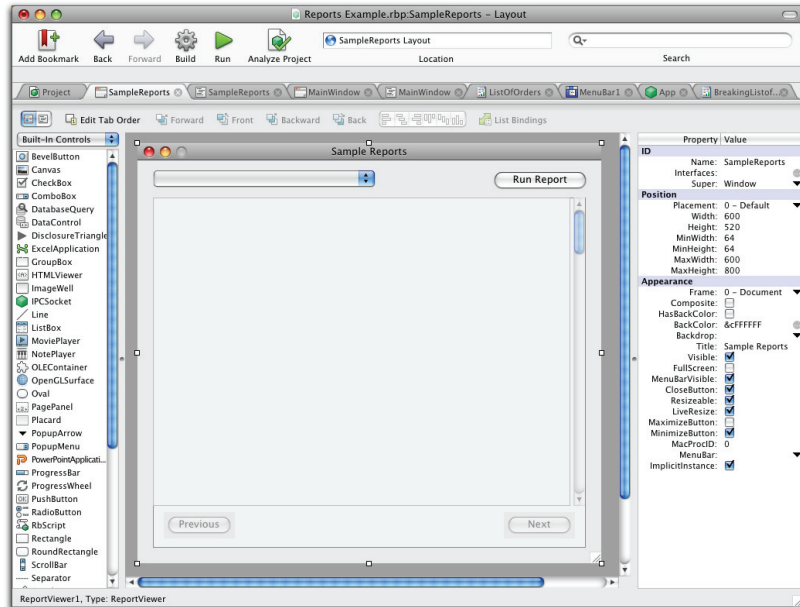
Showing the Sample Reports Window

The File ► Sample Reports... menu item shows the SampleReports window. Its menu handler is in the Application object. It simply displays the new window.

```
//FileSampleReports menu handler..  
SampleReports.Show  
  
Return True
```

The Sample Reports window is shown Figure 361 on page 465.

Figure 361. The Sample Reports window.



The Run Report button is enabled when the user chooses a report. Its Action event performs several actions that are necessary to produce a report.

SQL SELECT

The Action event handler begins with code that executes the SQL SELECT statement that corresponds to the report that the user chose. After obtaining a RecordSet, it needs to create a RecordSetQuery that will be passed to the reporting

engine. You must convert the RecordSet to a RecordSetQuery. This is the type of object that is passed to the Report.Run method.

```
Dim ps As New PrinterSetup
Dim sql as string
Dim rpt As Report
//execute the SQL SELECT that corresponds to the user's selection
select case ListOfReportsPopup.RowTag(ListOfReportsPopup.ListIndex)

case "ListofOrders"
    // Build the SQL statement that will be used to select the records
    sql = "SELECT O.OrderNumber, C.ID, C.Company, C.LastName,"+_
        "O.DateOrdered, O.Total"+ _
        " FROM Orders O, Customers C WHERE O.CustomerID = C.ID"
    rpt = New ListofOrders

case "BreakingListofOrders"
    // Build the SQL statement that will be used to select the records
    sql = "SELECT O.OrderNumber, C.ID, C.Company, C.LastName,"+_
        " O.DateOrdered, O.Total"+ _
        " FROM Orders O, Customers C WHERE O.CustomerID = C.ID order by"+_
        " c.company, o.ordernumber"
    rpt = New BreakingListofOrders

case "ListOfProducts" //listing of Product images
    //build the -SQL Statement that will be used to select the records
    sql= "SELECT * from Products"
    rpt = New ListOfProducts

end select
```

The next section of code creates a RecordSet using the SQL SELECT:

```
dim rs as recordSet
rs = app.ordersDB.sqlSelect( sql )
```


The next section a RecordSetQuery from the RecordSet. This puts the data in the form that the reporting engine requires:

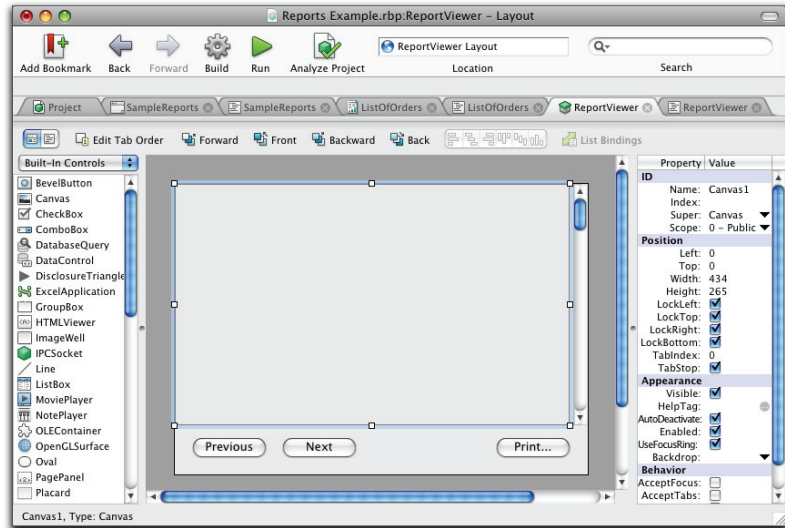
```
If rs = nil then
    beep
    MsgBox "No records found to print."
else
    //Create the RecordSetQuery for the reporting engine
    dim rsq as new Reports.RecordSetQuery(rs)
    // Now we select the records from the database
    // pass the RecordSetQuery to the reporting engine
    If rpt.Run( rsq, ps ) Then
        //copy the report to the ReportViewer for preview
        if rpt.Document <> Nil Then ReportViewer1.SetDocument( rpt.Document )
    End If
End if
```

The SetDocument method has been added to the ReportViewer ContainerControl. It assigns the Reports.Document object to the mDocument property of the ContainerControl. The Document object contains the report. The variable “doc” is the Reports.Document object that is created by running the report.

```
//SetDocument(doc as Reports.Document)
mDocument = doc
mCurrentPage = 1
If doc.PageCount > 0 Then SetCurrentPage( mCurrentPage )
```

The large area in the window is a ContainerControl, ReportViewer, that has the design shown in Figure 362.

Figure 362. The ReportViewer ContainerControl.



The large area in the ContainerControl is the Canvas. The reporting engine works by printing each page as a picture. The report can consist of multiple pages. Each picture can be referenced by the Page property of the Reports.Document object. The Canvas is the object that images a page. The Next and Previous buttons allow you to image successive (or prior) pages.

Navigation

The next and previous buttons increment or decrement the current page by calls to SetCurrentPage.

```
//Next button
//PushButton2.Action()
If mCurrentPage < mDocument.PageCount Then SetCurrentPage_
    ( mCurrentPage + 1 )
```

and

```
//Previous button
//PushButton1.Action()
If mCurrentPage > 1 Then SetCurrentPage( mCurrentPage - 1 )
```

Printing

The code behind the Print button is similar to the code for running the report. The major difference is that the Document.Print method is called just following the call to RecordSetQuery.

The code begins by declaring the objects that it needs:

```
Dim ps As New PrinterSetup
Dim sql as String
Dim rpt As Report
```

The next block of code issues the correct SQL statement depending on which report the user chose in the window:

```
Select Case mCurrentReportType
    Case "ListofOrders"
        // Build the SQL statement that will be used to select the records
        sql = "SELECT O.OrderNumber, C.ID, C.Company, C.LastName, "+_
            " O.DateOrdered, O.Total" + _
            " FROM Orders O, Customers C WHERE O.CustomerID = C.ID"
        //set the report project item we are going to use to print the report
        rpt = New ListofOrders

    Case "BreakingListofOrders"
        // Build the SQL statement that will be used to select the records
        sql = "SELECT O.OrderNumber, C.ID, C.Company, C.LastName, "+_
            "O.DateOrdered, O.Total" + _
            " FROM Orders O, Customers C WHERE O.CustomerID = C.ID order by" + _
            "c.company, o.ordernumber"
        //set the report project item we are going to use to print the report
        rpt = New BreakingListofOrders

    Case "ListOfProducts"
        sql="SELECT * from Products"
        //set the report project item we are going to use to print the report
        rpt = New ListOfOrders

End select
```

It then executes the correct SQL SELECT statement:

```
// Now we select the records from the database and add them to the list.
dim rs as RecordSet
rs = app.ordersDB.sqlSelect( sql ) 'select the records using the sqlquery

//Next, it sets the resolution of the printer after verifying that it has a
//RecordSet object to print.
ps.MaxHorizontalResolution=300
ps.MaxVerticalResolution=300
If ps.PageSetupDialog Then
    dim g as graphics
    g = OpenPrinterDialog(ps, nil)
```

It then declares a `RecordSetQuery`. This is the form the reporting engine is expecting (rather than a `RecordSet`).

```
//Put the records found into a Reports.DataSet
Dim rsq as New Reports.RecordSetQuery(rs)
```

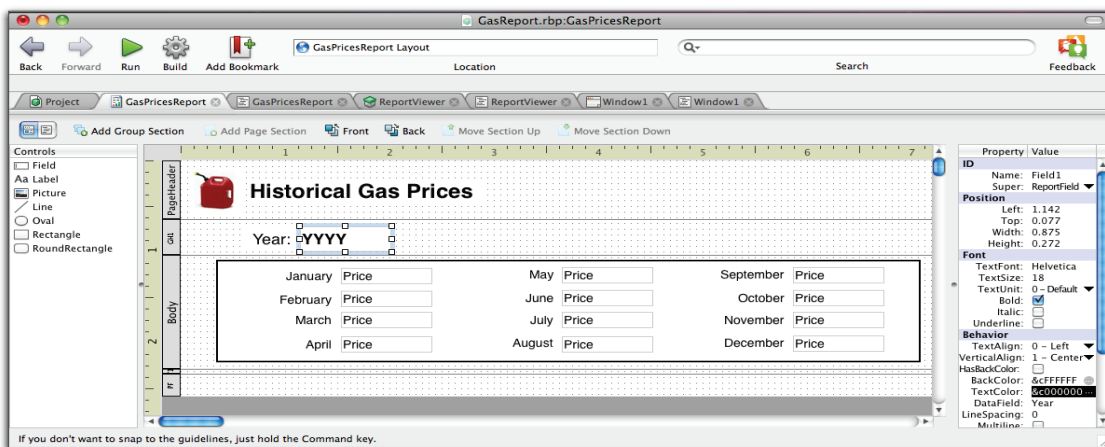
It's the `RecordSetQuery` object that is passed to the `Document.Run` command.

```
If rpt.Run( rsq, ps ) Then //if the report runs successfully
    rpt.Document.Print(g //print the report)
End If
```

Using a Text File as a Data Source

The Gas Report project contains a report that differs from the Example Database only in that the data source is a text file instead of a database. The report is similar to the break level report from the Example Database but lays out a row of data in a different manner. Each row represents a year of gas prices, labelled by month. The Year field is the `GroupByField`. In the report, the Section Header and Body repeat for every value of the `GroupByField`.

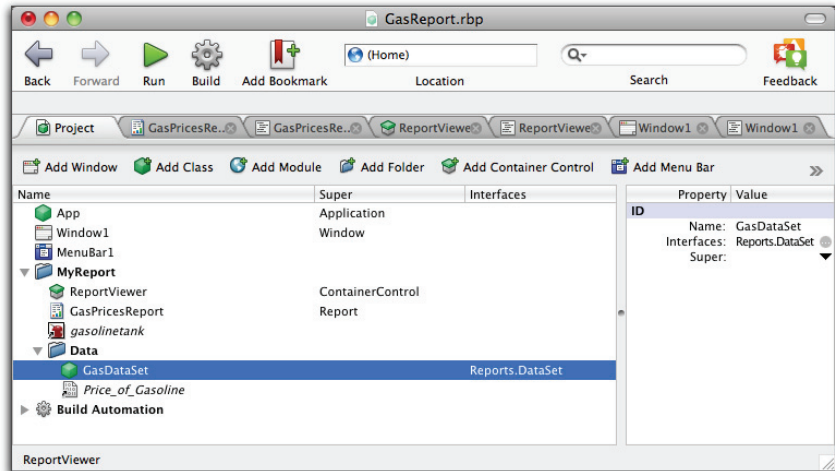
Figure 363. The Gas Report Layout editor.



The project organizes the data for the report into a folder. The text file is included and the `GasDataSet` class provides the interface between the text file and the report engine.

The Data folder in the Project Editor contains the data source for the report in the form of a comma-delimited text file and a class that serves as the interface for the data.

Figure 364. The Gas Report project.



The first field in each row of the text file is the year and the subsequent fields are the months of the year.

Figure 365. The Gas Report data.

Year	1	2	3	4	5	6	7	8	9	10	11	12
1976	0.605	0.6	0.594	0.592	0.6	0.616	0.623	0.628	0.63	0.629	0.629	0.626
1977	0.627	0.637	0.643	0.651	0.659	0.665	0.667	0.667	0.666	0.665	0.664	0.665
1978	0.648	0.647	0.647	0.649	0.655	0.663	0.674	0.682	0.688	0.69	0.695	0.705
1979	0.716	0.73	0.755	0.802	0.844	0.901	0.949	0.988	1.02	1.028	1.041	1.065
1980	1.131	1.207	1.262	1.264	1.266	1.269	1.271	1.267	1.267	1.25	1.25	1.258
1981	1.299	1.382	1.417	1.412	1.4	1.391	1.382	1.376	1.376	1.371	1.369	1.365
1982	1.358	1.334	1.284	1.225	1.237	1.309	1.331	1.323	1.307	1.295	1.283	1.26
1983	1.23	1.187	1.152	1.215	1.259	1.277	1.288	1.285	1.274	1.255	1.241	1.231
1984	1.216	1.209	1.21	1.227	1.236	1.229	1.212	1.196	1.203	1.209	1.207	1.193
1985	1.148	1.131	1.159	1.205	1.231	1.241	1.242	1.229	1.216	1.204	1.207	1.208
1986	1.194	1.12	0.981	0.888	0.923	0.955	0.89	0.843	0.86	0.831	0.821	0.823
1987	0.862	0.905	0.912	0.934	0.941	0.958	0.971	0.995	0.99	0.976	0.976	0.961
1988	0.933	0.915	0.904	0.93	0.955	0.955	0.967	0.967	0.974	0.967	0.949	0.93
1989	0.919	0.926	0.94	1.065	1.119	1.114	1.092	1.057	1.029	1.027	0.999	0.98
1990	1.042	1.037	1.023	1.044	1.061	1.088	1.084	1.19	1.294	1.378	1.377	1.354
1991	1.247	1.143	1.082	1.104	1.156	1.16	1.127	1.14	1.143	1.122	1.134	1.123
1992	1.073	1.054	1.058	1.079	1.136	1.179	1.174	1.158	1.158	1.154	1.159	1.136
1993	1.117	1.108	1.098	1.112	1.129	1.13	1.109	1.097	1.085	1.127	1.113	1.07
1994	1.043	1.051	1.045	1.064	1.08	1.106	1.136	1.162	1.177	1.162	1.163	1.143
1995	1.129	1.12	1.115	1.14	1.2	1.226	1.195	1.164	1.148	1.127	1.101	1.101
1996	1.129	1.124	1.162	1.251	1.323	1.299	1.272	1.24	1.234	1.227	1.25	1.26
1997	1.261	1.255	1.235	1.231	1.236	1.229	1.205	1.253	1.277	1.242	1.213	1.177
1998	1.131	1.082	1.041	1.052	1.092	1.094	1.079	1.052	1.033	1.043	1.038	0.986
1999	0.972	0.955	0.991	1.177	1.178	1.148	1.169	1.255	1.28	1.274	1.264	1.298
2000	1.301	1.369	1.541	1.506	1.498	1.617	1.593	1.51	1.582	1.559	1.555	1.489
2001	1.472	1.484	1.447	1.564	1.739	1.64	1.482	1.427	1.531	1.362	1.263	1.131
2002	1.139	1.13	1.241	1.407	1.421	1.458	1.412	1.423	1.422	1.449	1.448	1.394
2003	1.473	1.641	1.748	1.659	1.542	1.514	1.524	1.628	1.728	1.603	1.535	1.494

The GasDataSet class implements Reports.DataSet. Reports.DataSet is a class interface. This interface allows you to use any data as the data source for the reporting engine. It has the following methods:

- **Run:** This is the first method that executes when the report is run. It obtains the data from the text file:

```
// Part of the Reports.DataSet interface.  
//Run()  
mData = SplitB( Price_of_Gasoline, ChrB(13) )  
mCurrentRecord = 0
```

That is, it splits the data into rows. The property mData is declared as a String array and mCurrentRecord is an Integer.

- **Field:** Field is overloaded. The one that is passed as String is implemented. Field obtains the fields in each row of mData:

```
// Part of the Reports.DataSet interface.  
//Field(name as String) as Variant  
Static months() As String = Array( "Jan", "Feb", "Mar", "Apr", "May", _  
    "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" )  
  
Dim data() As String = SplitB( mData(mCurrentRecord), "," )  
  
If name = "Year" Then //first field  
    Return data(0)  
Else //month fields  
    Dim idx As Integer = months.IndexOf( name )  
    If idx <> -1 Then Return data( idx + 1 )  
End If  
  
Return Nil
```

- **NextRecord:** NextRecord increments the mCurrentRecord property that was initialized by the Run method:

```
// Part of the Reports.DataSet interface.  
//NextRecord() as Boolean  
  
mCurrentRecord = mCurrentRecord + 1
```

- **EOF:** EOF returns True when you've reached the end of the file.

```
// Part of the Reports.DataSet interface.  
//EOF as Boolean  
If mCurrentRecord > mData.Ubound Then Return True  
  
Return False
```

- **Type:** Type returns the columntype for all the columns in the text file. The types are the SQL field types that are listed in the Database class entry in the *Language Reference*.

```
//Part of the Reports.DataSet interface
Function Type(fieldname as String) as Integer
if fieldname = "Year"
    Return 5 //Text
Else
    Return 7 //Double
End if
```

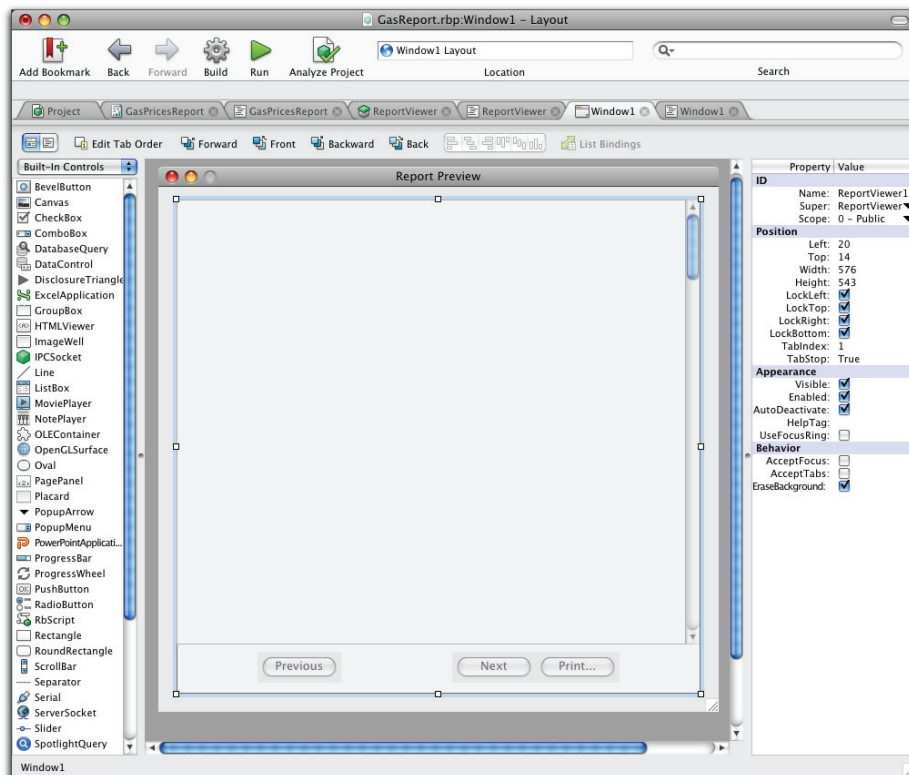
Declaring the columntype helps the reporting engine determine whether a column is numeric and a format can be applied. In this case, the year column should be interpreted as text but the gas prices in the body of the table are doubles.

The Report Layout

The Gas Report, shown in Figure 363 on page 470, is a listing report with one break level. The Body area uses a Rectangle control to enclose the pairs of labels and fields. You use the Front and Back buttons in the Report Editor toolbar to adjust the levels of the objects. In order for the labels and fields to be visible, the rectangle should be sent to the back using the Back button. One group section is used to label the data by year and there is no group footer (aka, subtotal) section.

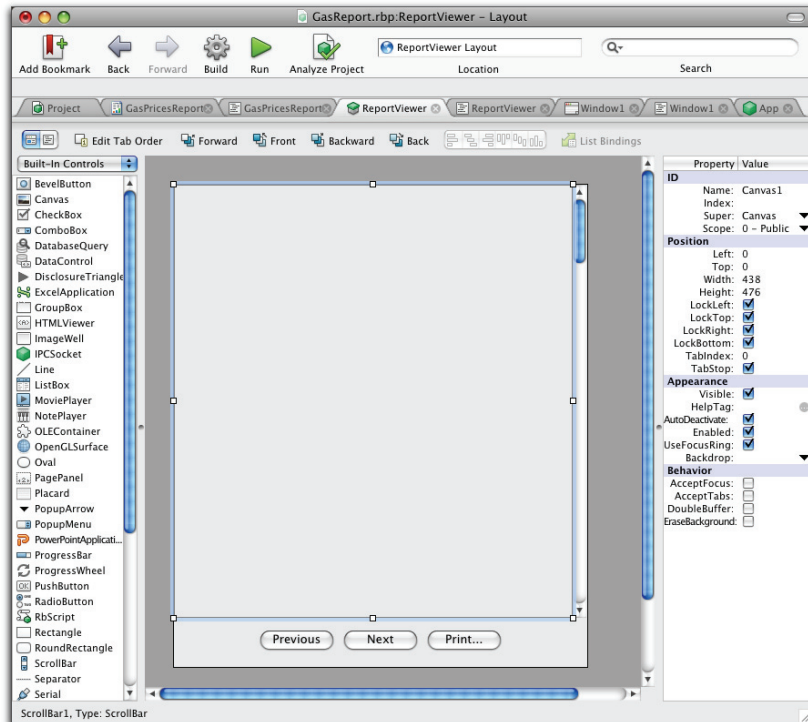
The main window for the project, Window1, contains the ContainerControl, ReportViwer, and the PushButton that triggers the report. The ContainerControl is seen in Figure 366.

Figure 366. The main project window.



The ContainerControl includes a large Canvas that is used to image the pages of the report and Next, Previous, and Print buttons. The Next and Previous buttons enable the user to navigate from page to page and Print displays a standard Print dialog for printing.

Figure 367. The ReportViewer ContainerControl.



The report runs when the application is launched. The code is in the Open event of Window1. It is

```
Dim ds As New GasDataSet
Dim ps As New PrinterSetup
Dim rpt As New GasPricesReport //This is the Report editor object

If rpt.Run( ds, ps ) Then
    //copy the report to the ReportViewer for preview
    If rpt.Document <> Nil Then ReportViewer1.SetDocument(rpt.Document )
End If
```

If the call to rpt.Run is succesful, it returns the document in rpt.Document.

The call to SetDocument (above) sets Reports.Document to the mDocument property of the Report and calls SetCurrentPage. SetDocument is shown below.

```
//SetDocument method
mDocument = doc
mCurrentPage = 1
If doc.PageCount > 0 Then SetCurrentPage( mCurrentPage )
```

The SetCurrentPage method defines the current picture, mCurrentPicture, which will be imaged by the Canvas.

```
//SetCurrentPage(pageNum as Integer)
mCurrentPage = pageNum
mCurrentPicture = mDocument.Page(mCurrentPage)
ScrollBar1.Maximum = mCurrentPicture.Height - Canvas1.Height
ScrollBar1.Value = 0
Canvas1.Refresh( False )
```

The Canvas's Paint event uses a call to DrawPicture to image the current page:

```
//Canvas1.Paint(g as Graphics)
If mCurrentPicture <> Nil Then
    g.DrawPicture( mCurrentPicture, 0, 0, Me.Width, Me.Height, 0,_
        Scrollbar1.Value, Me.Width, Me.Height )
Else
    g.DrawRect( 0, 0, Me.Width, Me.Height )
End If
```

The next and previous buttons increment or decrement the current page by calls to SetCurrentPage.

```
//Next button
//PushButton2.Action()
If mCurrentPage < mDocument.PageCount Then SetCurrentPage_
    ( mCurrentPage + 1 )
```

and

```
//Previous button
//PushButton1.Action()
If mCurrentPage > 1 Then SetCurrentPage( mCurrentPage - 1 )
```

In the ContainerControl, the Print PushButton calls Document.Print method to image each page of the report using the Graphics object returned by

OpenPrinterDialog. The end user will see the Printer dialog for the selected printer followed by a progress dialog that shows the status of the print job.

```
Dim ds as New GasDataSet //the Report.DataSet object
Dim ps as New PrinterSetup
Dim rpt as New GasPricesReport //the Reports object
Dim g as Graphics

//set up the printer resolution to 300 dpi
ps.MaxHorizontalResolution=300
ps.MaxVerticalResolution=300

if ps.PageSetupDialog then
  g = OpenPrinterDialog(ps,nil)
  If g <> Nil then
    if rpt.Run(ds,ps) then //if the report ran successfully, print it
      rpt.Document.Print(g)
    end if
  end if
end if
```

Many applications read from and/or write to files. Some create files that have their own special formats. Often this process starts with the user's selecting a file with the Open File dialog box or saving a file with the Save As dialog box. REAL Studio makes it easy to use the Open and Save dialog boxes, as well as to read from and write to many different types of files.

Contents

- Understanding File Types
- Understanding FolderItems
- Accessing files
- Working with text and binary Files
- Working with picture, sound, and video files
- Reading and writing to the resource fork
- Handling files double-clicked at the Desktop

Understanding File Types

There are many different file types. The type of a file defines a unique type of data stored in that file. For example, a text file stores text while a PICT file stores pictures. Files have a file extension (or suffix) that defines the file type. For example, a Text file has the extension “.txt”. A file named “myNotes.txt” is recognized as a Text file.

The file type makes it easy for an application to know if it is prepared to deal with a particular file. For example, any application that can open text files expects the file type of any text file it shall open is “TEXT”. This file type tells the application that this is a standard text file. PICT files are so named because “PICT” is the file type of a PICT file. Applications are also files but all applications have a file type of “APPL” that tells the Mac OS that this file is executable and not just data.

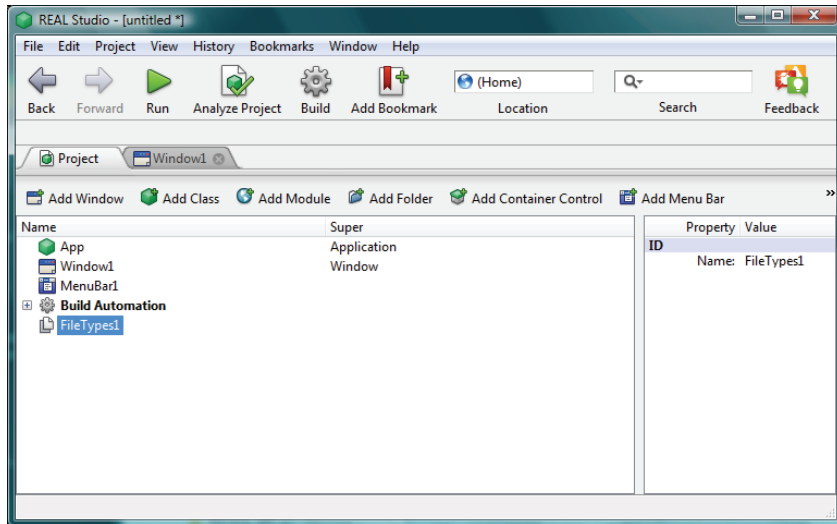
Rather than writing code that deals directly with all of these file types, creator codes, and file suffixes, REAL Studio abstracts you and your code from them with *file types*. A file type in REAL Studio is an item stored with your project that represents a specific file type, creator, and one or more extensions. Each file type has a name that is used in your code when opening and creating files. This allows you to work with names you can choose and easily remember instead of cryptic codes. It also abstracts your code from the operating system, making it easier for you to create versions of your application for other operating systems.

REAL Studio file types can be created in the IDE with the File Type Sets Editor or via the language, via the FileType class. The File Type Sets Editor is described in the following section and the FileType class is described in the *Language Reference*. The attributes that you give to file types in the editor map directly to properties of FileType class objects.

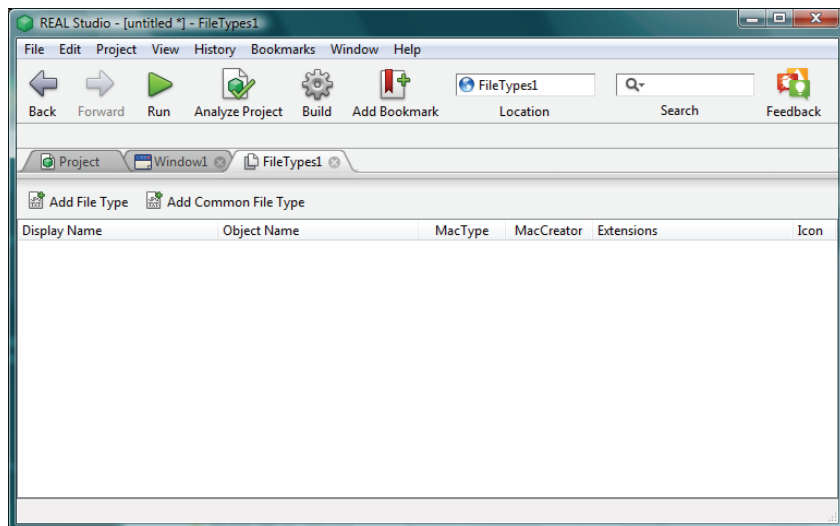
Using The File Types Editor

You use the File Type Sets Editor to create the items that will represent the different kinds of files you want your application to be able to open or create. To add file types to your project, you first add a File Type set to the IDE to hold your file types. The screen is called a File Type Set and it appears as an item in your Project Editor.

To add a File Types Set to your project, choose Project ► Add ► File Type Set. REAL Studio adds a blank File Types Set to the project.

Figure 368. A File Type Set added to the Project Editor.

Double-click the FileTypes item to add file types to the File Types Set. The File Types Editor appears.

Figure 369. The File Type Set Editor.

The File Type Set toolbar has two items: Add File Type and Add Common File Type. Normally you will use the Add Common File Type button. It enables you to choose a file type from a drop-down list. Most file types are listed.

In contrast, the Add File Type Set button creates a blank file type entry in the editor and enables you to define the file type yourself. This is best suited for defining custom file types that are unique to your application.

The body of the editor contains columns for the following file type attributes:

- **Display Name:** The name shown to the users in open-file dialog boxes. You can use either a string literal or a constant for the Display Name.

When using dynamic constants, the names are automatically localized on Mac OS X. On other platforms, using *FileTypeSet.MyFileType.Name* will return the dynamic constant value, which allows you to register/update your file type with the system using the localized name. For more information on using constants to localize the application, see the section “Using Constants to Localize your Application” on page 378.

- **Object Name:** The name used in your REAL Studio code to identify the file type. This is the string that you can pass to a method to tell the method to use that file type.
- **MacType and MacCreator:** The Macintosh Type and Creator codes used by the original Macintosh operating system to identify files. If you are going to assign custom icons to the file type, be sure that the Creator code matches the Creator code that you give your application.
- **Extensions:** The file extensions used on Mac OS X, Windows, and Linux to identify file types. You can specify more than one extension per file type. Separate multiple extensions with semicolons.
- **Icon:** The document icon for that file type. When the user double-clicks such an icon, the standalone REAL Studio application will start and open the file.

In the language, a File Type set has the string property “All” that returns all of the file types in the set. You can use All in your code when you want to refer to all of the file types in the set. Suppose you create a File Type Set called ImageTypes in which you specify all of the valid image types that your application can open. You can specify the entire list of image types with a line such as:

```
f=GetOpenFolderItem(ImageTypes.All)
```

If you need to add or remove image file types, you can simply modify the File Type Set and this line of code will automatically refer to the new members of the set.

Adding a Common File Type

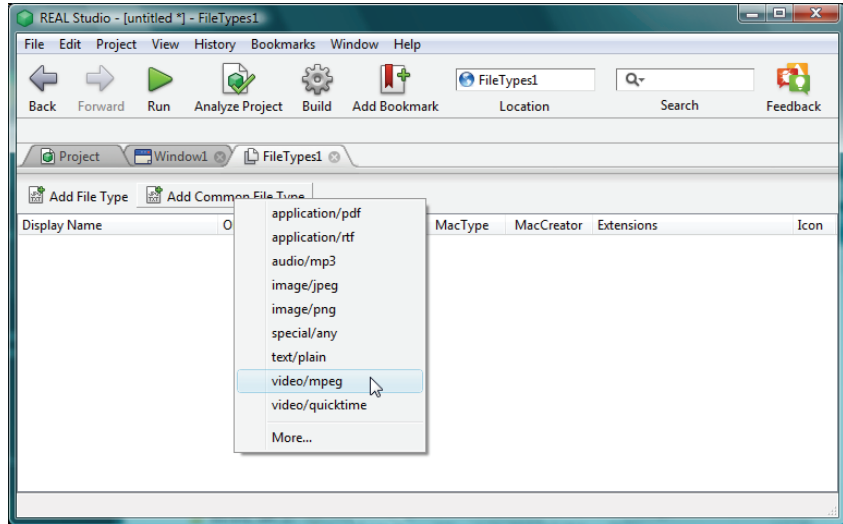
To add a common file type, do this:



- 1 **Double-click a File Types Set item in the Project Editor.**
A File Types Set Editor appears (Figure 369).
- 2 **Click on the Add Common File Type button.**

A drop-down list of common file types appears.

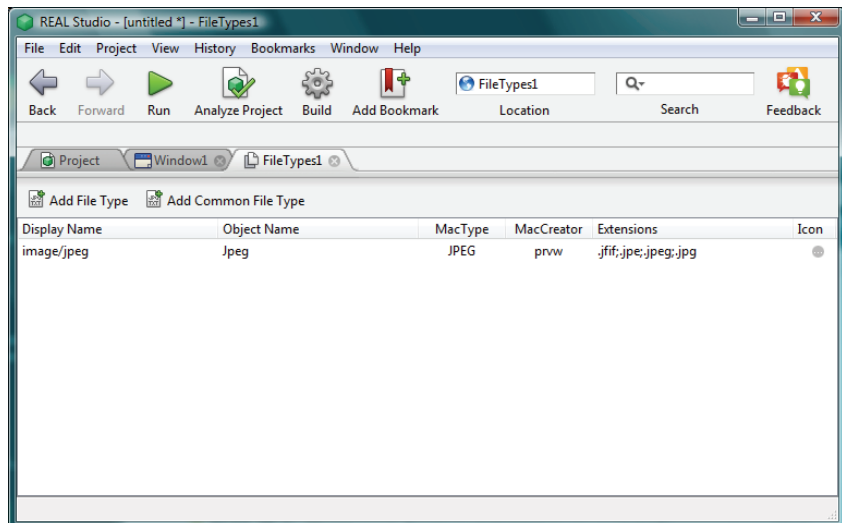
Figure 370. The Common File Types list.



- 3 Choose a file type or, if you don't see your choice, choose the "More..." item from the drop-down list.**

If you chose a file type, a new row is added to the File Type Set list editor, populated with values for the file type you chose. An example is shown in Figure 371.

Figure 371. The prebuilt file type for "image/jpeg."

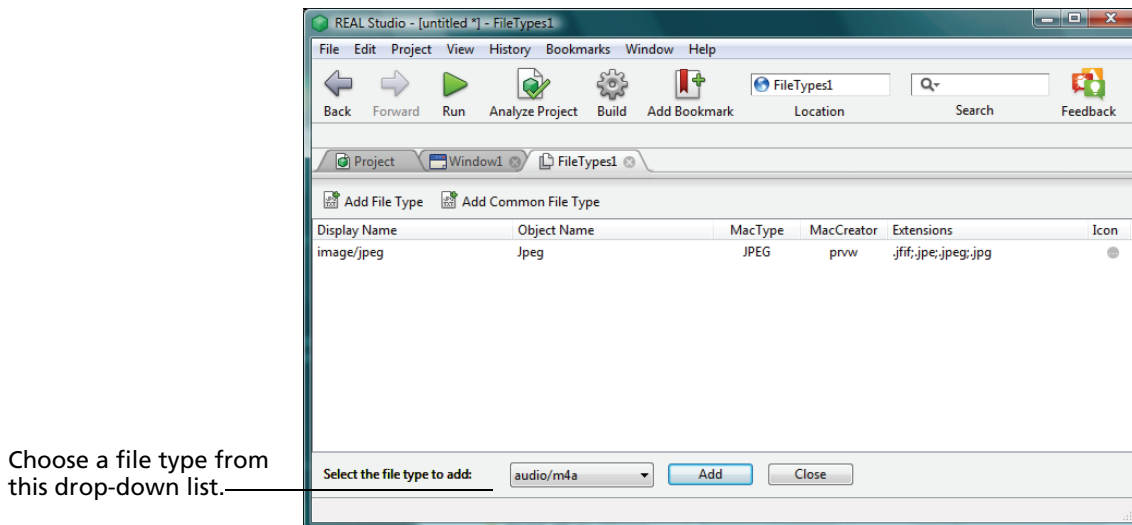


The row is editable. If you wish to change any of its values, click twice in the cell to change it to editable. The value will be selected. Delete or edit the text as you wish.

A new row is added to the File Types table. Click twice in a cell to get an insertion point and press the Tab key to move from field to field.

- 4 If you clicked “More...”, a secondary drop-down list appears at the bottom of the File Type Set editor.

Figure 372. The “More...” secondary drop-down list.



- 5 Display the drop-down menu at the bottom of the screen and choose a file type.

The file type you selected is added to the File Type Sets editor.

Adding a Custom File Type

If you are defining a file type of your own or do not see the desired file type in the Common File Types list, you can manually add a file type.

To add a custom file type, do this:

- 1 Double-click a File Types Set item in the Project Editor.

A File Types Set Editor appears (Figure 369).

- 2 Click on the Add File Type button.

A new blank row is added to the File Type Sets editor.

- 3 Enter the Display and Object names for the file type, the Mac Type and Creator codes, and the extensions.

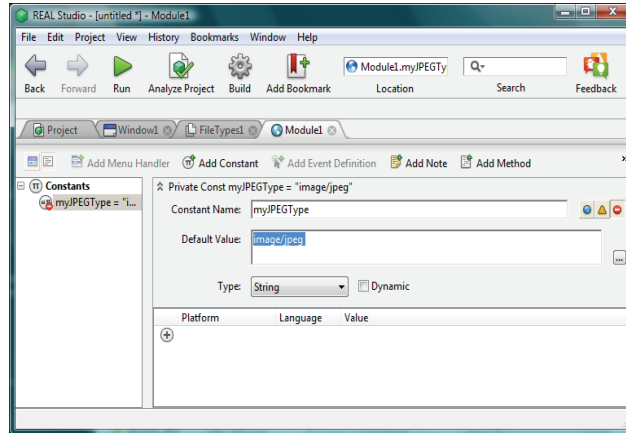
To enter more than one extension, separate them by semicolons. If you used a constant for the Display Name, you precede its name with a number sign.

You can type a Display Name or use a constant as the Display Name. Figure 373 shows a constant defined for a Display Name in a module and its use in the File Types Sets editor.

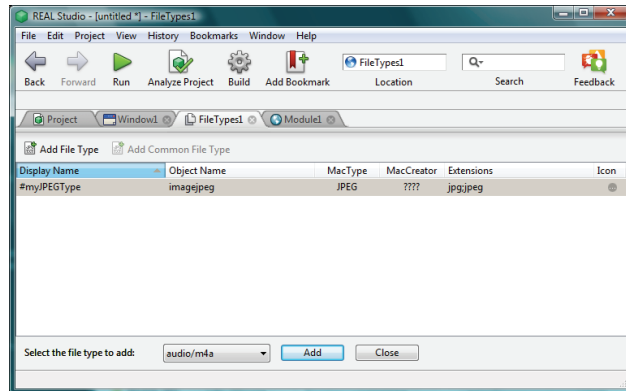


Figure 373. Using a constant to specify a Display Name.

In a module, the dynamic constant 'myJPEGType' is defined.



In a File Types Set, the constant is referred to as "#myJPEGType".



- 4 If desired, repeat the process to add additional File Types to the set.
- 5 When you are finished with the File Types Set, click another tab or use the toolbar to move to another area of your project.

Adding a Document Icon

You can define a custom document icon for the each file type that the application can open and save. REAL Studio does not include an icon editor; you must design your icons in an icon editor program or image creation application and import or paste them into the Edit Icon dialog in REAL Studio.

You need to provide at least the following items:

- For Mac OSX, a 128 x 128 image, and for other platforms smaller sized icon images. Even for Mac OS X it is best to provide smaller sized image files at 32 x 32 and 16 x 16.
- A mask that defines the icon's edges so that the operating systems can determine which regions in the square image are clickable.

The document icons will be used when the application saves documents in that file type and double-clicking the document icon will start the application.

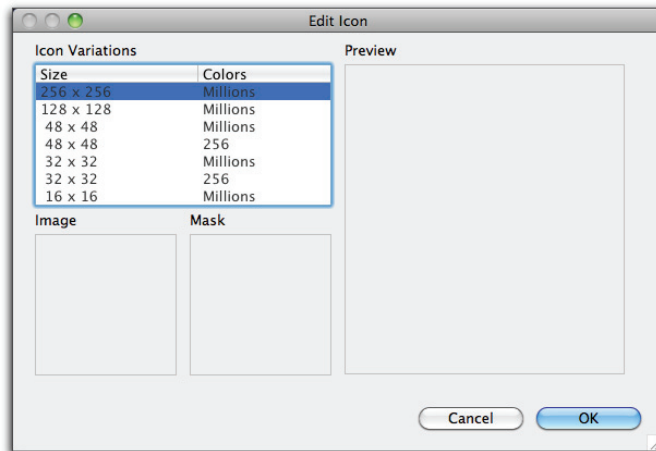
To copy and paste a document icon for the file type, do this:



- 1 **Place the icon you wish to paste into the Icon editor on the Clipboard.**
- 2 **Click the blank document icon in the Icon column for the file type.**

A Document icon dialog box appears, with variations for all sizes of document icons.

Figure 374. The Edit Icon dialog box.



The Edit Icon dialog can be enlarged and the Preview area will grow as the dialog is resized.

- 3 **In the Icon Variations list, select the icon size that matches the icon on the Clipboard and select the “Image” ImageWell.**
- 4 **Paste the icon on the Clipboard into the Image area.**
- 5 **Repeat this process for the icon’s mask and, if needed, repeat the process for other icon sizes.**

For Mac OS X you need to specify the 128 x 128 size icon and the OS will scale it as required.

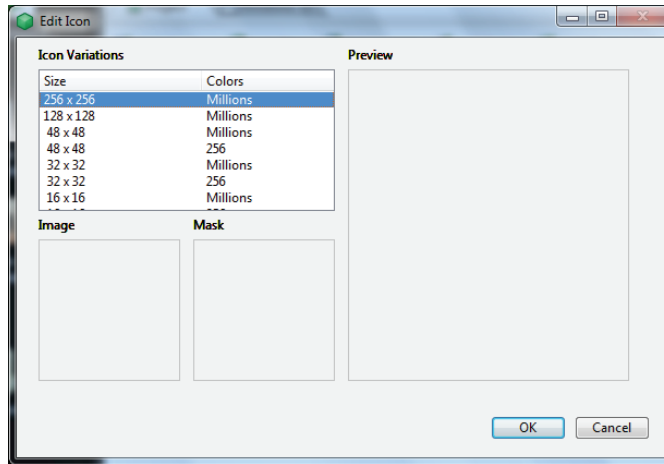
To import a document icon for the file type, do this:



- 1 **Click the blank document icon in the Icon column for the file type.**

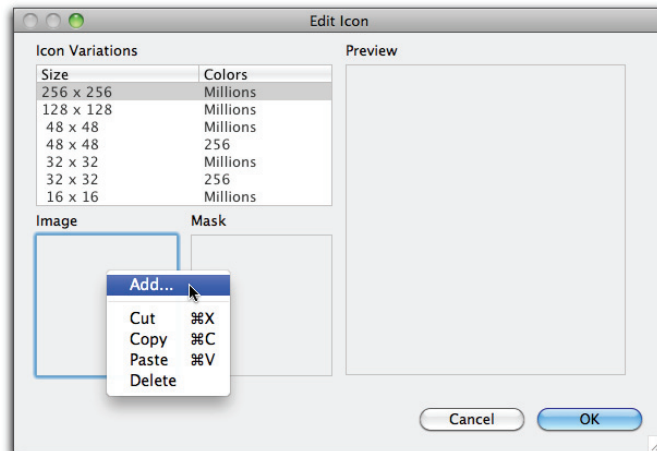
A Document icon dialog box appears, with entries for all sizes of document icons.

Figure 375. The Edit Icon dialog box.



- 2 In the Icon Variations list, select the icon size that you want to use and select the “Image” ImageWell.
- 3 Right+click (Ctrl+click on Macintosh) on the Image area to display its contextual menu and choose Add...

Figure 376. The Add contextual menu.



An open-file dialog box appears.

- 4 Choose the file that matches the selected icon size and click Open.
On Windows, you can choose either a document in a picture format (e.g., bmp, jpg, gif) or a .ico file.
- 5 Repeat this process for the icon’s mask and, if needed, repeat the process for other icon sizes.

For Mac OS X you need to specify at least the 128 x 128 size icon and the OS will scale it as required. Mac OS X 10.5 (and above) and Windows Vista (and above) can use the 256 x 256 size icon and scale it as needed.

Editing a File Type

Making changes to file types is easy.

To edit a file type, do this:



- 1 In the Project Editor, click the File Type Set you want to edit or click on its tab if it is already open.**
- 2 Click on the file type in the File Type Set Editor you wish to edit to select it.**
- 3 Click click twice in a cell in the row to get an insertion point.**
- 4 Make any changes you wish and use the Tab key to move the insertion point to the next cell or click twice in another cell.**
- 5 When you are finished, click another tab or use the toolbar to move to another area of your project.**

If you change the name of the file type, make sure you update any code that references the name of the file type. You can replace any occurrences of the old file type name with the new one easily using the Find/Change dialog box.

Deleting a File Type

Deleting a file type is simple.

To delete a file type, do this:



- 1 In the Project Editor, click a File Type Set tab.**
- 2 Click on the File Type you wish to edit to select it to select the whole row.**
- 3 Press the Delete button on your keyboard.**
- 4 When you are finished, click another tab or use the toolbar to move to another area of your project.**

If you delete a file type, make sure you update any code that uses this file type.

Using File Types

You pass one or more file types to commands to indicate that only the passed file types are appropriate. For example, the following statement in a control's Open event handler specifies that it can accept TEXT files that have been dragged from the Desktop:

```
me.AcceptFileDrop("text")
```

The statement

```
f=GetOpenFolderItem("video/quicktime")
```

displays an open-file dialog box that allows the user to view and open only QuickTime movies.

The FileType class has a built-in string conversion operator. This enables you to refer to the file type by its Object Name and it will return the appropriate string. For example:

```
f=GetOpenFolderItem(ImageTypes.JPEG)
```

would refer to the file type with the ObjectName of “JPEG” in the “ImageTypes” File Type Set.

You can concatenate two file type using the “+” operator to specify two file types. For example:

```
f=GetOpenFolderItem(ImageTypes.JPEG + ImageTypes.MacPICT)
```

refers to both the “JPEG” and “MacPICT” file types in the ImageTypes set.

The easiest way to specify several file types is to put all the file types that you want to refer to in one File Type Set and then use the All method of the File Type Set class. It automatically returns concatenates all the file types. For example, the following returns all the file types in the ImageTypes File Type Set.

```
f=GetOpenFolderItem(ImageTypes.All)
```

You can also specify more than one acceptable file type using their Object Names only. Separate the file type names by semicolons. For example, the following line allows the user to see and open PICT, GIF, and JPEG files:

```
f=GetOpenFolderItem("image/x-pict;image/jpeg;image/gif")
```

Consult the *Language Reference* for additional methods that take file types as arguments.

Creating Custom File Types for Your Application

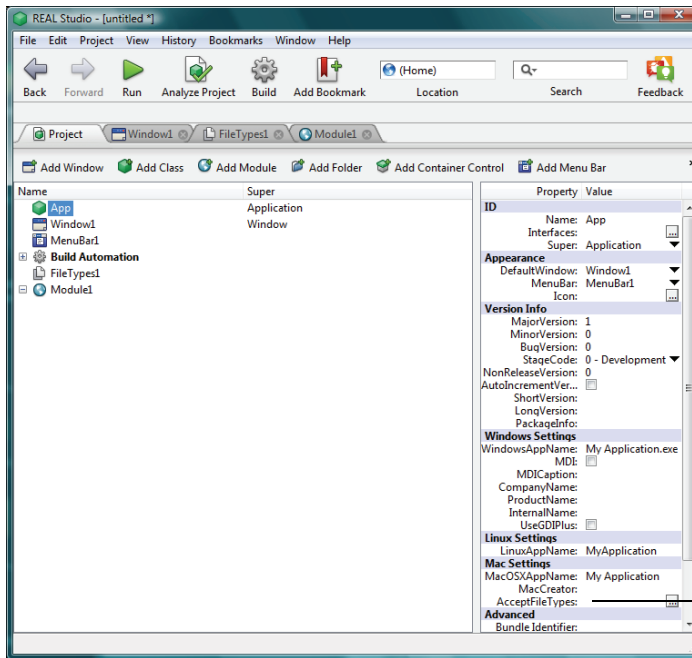


Most applications create files and assign custom icons to them. These icons usually look similar to the application’s custom icon. This makes it easier for the user to recognize that the file goes with the application that produced it. Any custom icons you add will appear only if you have assigned a Creator code to your project and built a stand-alone application.

The Personal version of REAL Studio enables you to build standalone applications for the platform on which REAL Studio is running and demo versions of applications for all other platforms that REAL Studio supports. The Professional and Studio versions of REAL Studio enable you to build fully-functional applications on all platforms.

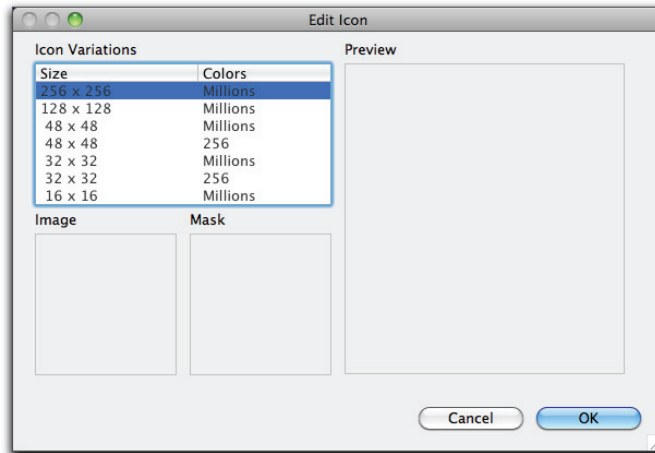
To add custom icons to any of the file types for your project, first assign a unique Creator code to your application. You set the creator code in the Mac Settings group in the App class's Properties pane. It is shown in Figure 377.

Figure 377. The Mac Settings group in the App class's Properties pane.



Enter your Creator code into the Mac Creator area. Creator codes are case sensitive and must be unique. You can register a unique creator code for your application with Apple, Inc. at their web site.

You assign an application icon via the Icon property in the Appearance group. You need to design your icons in an external image or icon editing program. Click the “...” icon to display the Edit Icon dialog box and add your icons using the procedure described in the section “Adding a Document Icon” on page 485.

Figure 378. The Edit Icon dialog box.

Understanding FolderItems

To REAL Studio, volumes, folders, applications, and documents are all considered to be *FolderItems*. A *FolderItem* is anything that can appear on the desktop. This doesn't mean that only items on the desktop are *FolderItems*. It means that if the item could be placed on the desktop, it's a *FolderItem*. For example, the Recycle Bin is a *FolderItem* because it appears on the desktop. See the section "Getting The Selected File From An Open File Dialog Box" on page 501 for more information on getting *FolderItems* for items that are created by the operating system.

The *FolderItem* class is your first point of contact with any item on a disk you want to read from or write to. To read from a file, for example, you get a *FolderItem* that represents the file, then use various methods to read from the file via the *FolderItem*. There are many different ways to get a *FolderItem* object that represents a particular volume, folder, application, or document. You can present the user with an Open File or Save As dialog box, you can get the *FolderItem* at a specific path, or you can even get a *FolderItem* from another *FolderItem*.

FolderItems have properties that store the path to the item, the name of the item, the size of the item, its type, etc. *FolderItems* also have methods you can use to create files, open files, delete files, copy files, etc.

For detailed information on the properties and methods of the *FolderItem* class, see the *FolderItem* class in the *Language Reference*.

How Are Shortcuts and Aliases Handled?

Shortcuts (aliases in Macintosh terminology) are files that actually represent a volume, application, folder, or file stored in another location and possibly under another name. REAL Studio contains commands that allow you to either resolve the

shortcut and work with the actual object or work with the object directly. The `GetFolderItem` function automatically resolves an shortcut when it encounters it, while the `GetTrueFolderItem` function works with the shortcut itself.

Getting a File at a Specific Location

If you know the full path to a file and you wish to access the file, you can do so by specifying the path to the file.

For example, suppose you have a document called “Schedule” stored in the same folder as the REAL Studio application. The relative path starts with the directory REAL Studio (or the REAL Studio application) is in. If the file is in the same directory as REAL Studio, then the relative path consists only of the filename itself.

The following code creates a `FolderItem` object in the local variable “f” that represents the document mentioned above:

```
Dim f as FolderItem
f=GetFolderItem("Schedule")
```

The `GetFolderItem` function should be used to either specify a file or directory in the same directory as the application or the empty string (“”). If you specify the latter, `GetFolderItem` will return the `FolderItem` for the directory in which your application is located.

```
Dim f as FolderItem
f=GetFolderItem("") //returns the directory containing your application
```

The full path (sometimes called the absolute path) to a volume, directory, application, or document starts with the volume name followed by the path delimiter character (a backslash on Windows, a colon on the Macintosh, and a forward slash on Linux), the names of any folders in the path (each separated by the path delimiter) and ending with the name of the item.

To create an absolute path to a file or folder, you should use the `Volume` global function to build a full path to the item, starting with the hard disk it is on. You then use the `Child` method of the `FolderItem` class to navigate to the item. `Volume` returns a `FolderItem` for one of your mounted volumes. You specify the volume by passing an integer, indicating the volume. Volume 0 is the volume that contains the operating system — the “boot” volume.

```
Dim f as FolderItem
f=Volume(0) //returns a folderitem for the boot volume
```

`Parent` returns the `FolderItem` for the next item up in the absolute path for the current `FolderItem`. It does not work if you try to get the parent of a volume. The `Child` method lets you access any items one level below the current `FolderItem`.

You can build a full path starting from a volume with the Child method. For example, if you want to get a FolderItem for the file “Schedule” in the directory “Stuff” on the boot volume, the statement would be:

```
Dim f as FolderItem
f=Volume(0).Child("Stuff").Child("Schedule")
```

The following example works with a relative path. It uses the Parent property to get the FolderItem for the directory that contains the directory in which the application is located. Passing the empty string to GetFolderItem gets the current directory, so the parent of that directory is one level up in the hierarchy.

```
Dim f as FolderItem
f=GetFolderItem("").Parent
```

Once you have a FolderItem, you can (depending on what type of item it is) copy it, delete it, rename it, read from it or write to it, etc. You will learn how to read and write to files using FolderItems later in this chapter.

The GetFolderItem has an optional parameter that allows you to pass an absolute path, a shell path, or a URL path. It uses the following class constants from the FolderItem class: PathTypeShell, PathTypeURL, and PathTypeAbsolute. The default is an absolute path.

You specify the type of path by passing one of the class constants as the second parameter in a call to GetFolderItem. For example, the following uses a shell path on Linux. It returns a FolderItem for the “Documents” folder in the home directory for the user “Joe.”

```
Dim f as FolderItem
f=GetFolderItem("/home/Joe/Documents",_
    FolderItem.PathTypeShell)

If f.exists then
    TextField1.text=f.AbsolutePath
else
    MsgBox "The folderitem does not exist."
End if
```

A URL path must begin with “file:///” The following example uses the URL path to the user’s “Documents” folder on Windows:

```
Dim f as FolderItem
f=GetFolderItem("file:///C:/Documents%20and%20Settings/" _
    + "Joe%20User/My%20Documents/", FolderItem.PathTypeURL)

If f.exists then
    MsgBox f.AbsolutePath
Else
    MsgBox "The folderitem does not exist."
End if
```

The FolderItem class’s properties AbsolutePath, URLPath, and ShellPath contain the types of paths.

Accessing Specific System Folders

REAL Studio provides a module that returns FolderItems representing various folders that are part of the System software or the desktop. When you need to access one of these folders, you should use the SpecialFolder object to obtain the FolderItem. These functions will still work properly even if the directory’s name changes. They are designed to return the proper FolderItem on all platforms. They are also language independent.

You obtain the desired FolderItem using the syntax:

```
result=SpecialFolder.FolderName
```

where *result* is the FolderItem you want to obtain and *FolderName* is the name of the SpecialFolder function that returns that FolderItem. For example, the following gets a FolderItem for the Application Support folder on Mac OS X and the Application Data directory on Windows.

```
Dim f as FolderItem
f=SpecialFolder.ApplicationSupport
```

The entry for SpecialFolder in the *Language Reference* has the complete list of supported functions and the FolderItems that they return on Mac OS X, Windows, and Linux. Not all functions return FolderItems on all platforms. If a FolderItem is not defined on all platforms, you should use an alternative function that returns a FolderItem on every platform. Check that the result is not Nil before using the FolderItem. For example, SpecialFolder.Documents returns the current user’s

Documents folder on Macintosh and Windows but returns Nil on Linux. On Linux, you should call SpecialFolder.Home instead. For example:

```
Dim f as FolderItem
#If Not TargetLinux
    f=SpecialFolder.Documents
#Else
    f=SpecialFolder.Home
#EndIf
If f <> Nil then
    If f.exists then
        //use the folderitem
    End if
Else
    MsgBox "FolderItem is Nil!"
End if
```

Verifying that you have accessed the Item

When you try to get a FolderItem, either of two things can go wrong. First, the path may be invalid. An invalid path contains a volume reference and/or a directory name that doesn't even exist. For example, if you specify Volume (1) when the user has only one volume, the Volume function returns a Nil value in the FolderItem instance, f. If you try to use any of the FolderItem class's properties or methods on a Nil FolderItem, a NilObjectException error will occur. If the exception is not handled in some way, the application will shut down.

Second, the path may be valid, but the file you are trying to access may not exist. The following shell code checks for these two situations:

```
Dim f as FolderItem
f=SpecialFolder.Documents.Child("Schedule")
If f <> Nil then
    If f.Exists then
        MsgBox f.AbsolutePath
    Else
        MsgBox "The document does not exist!"
    End if
Else
    MsgBox "You supplied an invalid path!"
End if
```

If the path is valid, the code checks the Exists property of the FolderItem to be sure that the file already exists; if the file doesn't exist or the path is invalid, a warning message is displayed.

You can also handle an invalid path using an Exception Block. They are discussed in the section “Runtime Exception Errors” on page 652.

Creating a New FolderItem

You can create a FolderItem for an existing item by passing it the pathname. When you create a FolderItem with the New command, you can pass the path to the new FolderItem as an optional parameter. For example:

```
Dim f as FolderItem
f=New FolderItem("myDoc.txt")
```

specifies the name of the new FolderItem and it is located in the same folder as the REAL Studio application (if you're running in the IDE) or the same folder as the built application.

If you pass a FolderItem instead of a path, the New method will create a copy of the passed FolderItem. In this example, the FolderItem "f2" refers to a copy of the original FolderItem, not a reference to it.

```
Dim f, f2 as FolderItem
f=SpecialFolder.Documents.Child("Schedule")
if f <> Nil then
    f2 = New FolderItem(f)
End If
```

Getting Information About a FolderItem

If GetFolderItem returns a valid FolderItem to an existing item, you can get information about the FolderItem using the local variable "f". For example, you can get the modification date of the FolderItem. This example displays the modification date of the FolderItem above:

```
Dim f as FolderItem
f=SpecialFolder.Documents.Child("Schedule")
if f <> Nil then
    if f.Exists then
        MsgBox f.ModificationDate.ShortDate
    End if
End if
```

Deleting a FolderItem

Once you have a FolderItem that represents an item that can be deleted, you can call the FolderItem's Delete method. The following example deletes the file represented by the FolderItem returned:

```
Dim f as FolderItem
f=SpecialFolder.Documents.Child("Schedule")
If f <> Nil Then
    if f.Exists then
        f.Delete
    End if
End if
```



If the `FolderItem` is locked, an error will occur. You can check to see if the `FolderItem` is locked by checking the `FolderItem`'s `Locked` property.

Deleting a `FolderItem` does not simply move the `FolderItem` to the trash. The `FolderItem` is deleted permanently from the volume.

Getting and Setting Ownership

On Unix-based operating systems, permissions to read, write to, and run a file (or view the contents of a directory) are regulated by a system of permissions. Central to permissions system is file/directory ownership. Each file or directory has an Owner and the Owner is in a Group.

The `FolderItem`'s Owner and Group properties let you read and set this information. For example, if we use the previous example, we can get the name of the Owner of the document in the following way:

```
Dim f as FolderItem
f=SpecialFolder.Documents.Child("Schedule")
if f <> Nil then
    if f.Exists then
        MsgBox "Schedule owner is "+f.Owner
    End if
End if
```

To get the name of the owning Group, read the value of `f.Group` instead of `f.Owner`. You can assign a new owner or group simply with assignment statements, such as:

```
f.Owner="Matt"
f.Group="Marketing"
```

Getting and Setting Permissions

Unix-based operating systems (Mac OS X and Linux), use a system of permissions to regulate access to files and directories. Permissions is granted to the Owner of the item, members of the owning Group of users, and all other users not in the owning Group.

Three levels of permissions may be granted:

- **Read:** A user may only read the file or view the name of a directory.
- **Write:** A user may change the file or directory.
- **Execute:** A user may run the file (application) or view the contents of a directory and examine the files.

Under Unix, permissions is indicated in a rather cryptic way. It uses a three-digit octal number in which each digit indicates the permissions for the Owner, Group, and Others, in that order from left to right.

The code for each digit is computed using the following values:

Table 30: Values for Read, Write, and Execute Permissions

Value	Permission Type	Description
4	Read permissions	User or group can open and read the FolderItem.
2	Write permissions	User or group can open and write to the FolderItem.
1	Execute permissions	User or group can execute the file or read the directory.

For each digit, the permissions are expressed by adding up the values. Each digit can take on eight possible values. Since each digit can only take on one of 8 values, its an *octal* system, or Base 8.

Table 31: All possible values for a permissions integer.

Value	Description
0	No permissions
1	Execute permissions
2	Write permissions
3	Write and Execute permissions
4	Read permissions
5	Read and Execute permissions
6	Read and Write permissions
7	Read, Write, and Execute permissions

For example, a permissions of 664 means that the Owner and Group (who each have a value of 6 in this example) have Read and Write permissions, but Others have only Read permissions.

The FolderItem's Permissions property lets you get and set permissions by via this octal value. You can read the value or just set it to a new value by assigning this property the three-digit octal value that sets the desired permissions.

You can also get or set permissions for a FolderItem via the Permissions class. This class lets you avoid having to deal with octal numbers and their meaning and, instead indicates permissions by a group of Boolean properties. Each property is True if the user has the specified permissions.

The Boolean properties are as follows.

Table 32: The Permissions class's properties for setting access permissions.

Name	Description
OwnerExecute	If True, the owner of the FolderItem can execute the FolderItem.
OwnerRead	If True, the owner of the FolderItem can read the FolderItem.
OwnerWrite	If True, the owner of the FolderItem can write to the FolderItem.

Table 32: The Permissions class’s properties for setting access permissions.

Name	Description
GroupExecute	If True, a member of the Group that owns the FolderItem can execute the FolderItem.
GroupRead	If True, a member of the Group that owns the FolderItem can read the FolderItem.
GroupWrite	If True, a member of the Group that owns the FolderItem can write to the FolderItem.
OtherExecute	If True, a user outside the Group that owns the FolderItem can execute the FolderItem.
OtherRead	If True, a user outside the Group that owns the FolderItem can read the FolderItem.
OtherWrite	If True, a user outside the Group that owns the FolderItem can write to the FolderItem.

In addition, the Permissions class gives you access to the Set User Bit and Set Group Bit. These Boolean flags grant the current user the permissions of the Owner and Group, respectively, without actually changing the FolderItem’s permissions. These properties are shown in Table 33.

Table 33: The Set User Bit and Set Group Bit properties.

Name	Description
GIIDBit	Set Group ID bit. If True, the current user has the permissions of the owning Group without altering the Read/Write/Execute permissions for the FolderItem.
UIDBit	Set User ID bit. If True, the current user has the permissions of the owner without altering the Read/Write/Execute permissions for the FolderItem.

You can also use the Permissions class to get or set the so-called “Sticky” bit of a FolderItem. If the Sticky bit is set, the operating system will not allow someone to remove or rename the file or files in the directory that he does not own, even he has Read/Write permissions. This can be used for directories in which different users need to be able to create files but should not touch others’ files.

To use the Permissions class, you create an instance of the class by passing it the value of the Permissions property of the FolderItem. For example,

```
Dim f as FolderItem
f=GetOpenFolderItem("")
if f <> Nil then
    if f.Exists then
        Dim perms as New Permissions(f.Permissions)
        //get or set permissions class bits here...
        //with the properties in Table 32 and Table 33
    End if
End if
```

Getting The Path To Your Application's Folder

Passing a null string (two quotes with no characters in between them) to the GetFolderItem function returns a FolderItem representing the folder your application or project is in. You can then use the FolderItem's Item method to access all the items in the folder your application is in. The Item method returns an array of FolderItems in the directory. The array is one-based. You get a FolderItem for an item by passing the Item method the index of the item.

For example, the following method gets a FolderItem for the directory and populates a ListBox with the absolute paths to each item in the directory. It uses the Count property to get the number of items and the Item method to get a FolderItem for each item.

```
Dim f as FolderItem
Dim i as integer
f=GetFolderItem("")
For i=1 to f.Count
    ListBox1.AddRow Str(i)
    ListBox1.Cell(ListBox1.LastIndex,1)=f.Item(i).AbsolutePath
Next
```

Getting Specific Items In the Application's Folder

If the first item in the path is not a volume, the GetFolderItem function assumes that the first item in the path is in the same folder as the application. If you are running your project in REAL Studio, GetFolderItem looks for the item in the folder your project is in. If you haven't saved your project yet, GetFolderItem will look in the folder that REAL Studio is in.

The following example returns a FolderItem that represents a file called "My Template" in a folder called "Templates" that is located in the same folder as the application:

```
Dim f as FolderItem
f=GetFolderItem("Templates").Child("My Template")
```

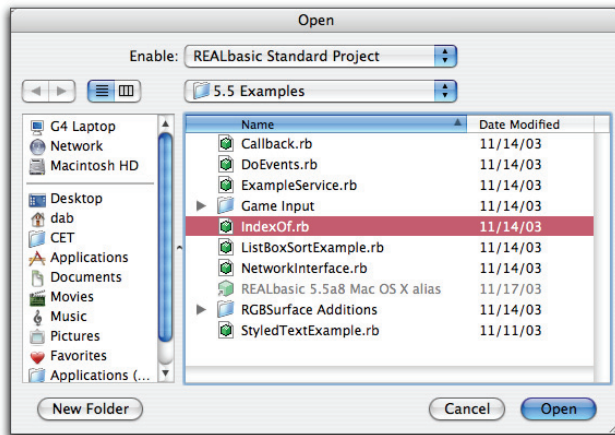
If the file is in the same folder as the application, then you can just pass `GetFolderItem` the name of the file. The following works:

```
Dim f as FolderItem
f=GetFolderItem("My Template")
```

Getting The Selected File From An Open File Dialog Box

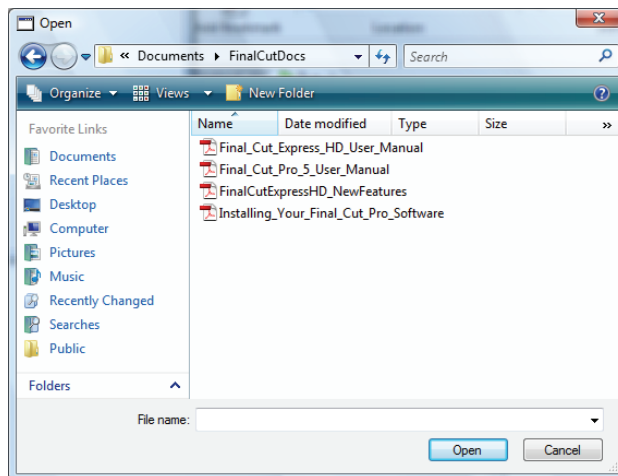
The Open File dialog box lets the user navigate to a particular location on any mounted volume and select a file to open. If the user is running Mac OS X 10.3 or above, the following dialog box appears:

Figure 379. The Mac OS X file browser (10.3 and above).

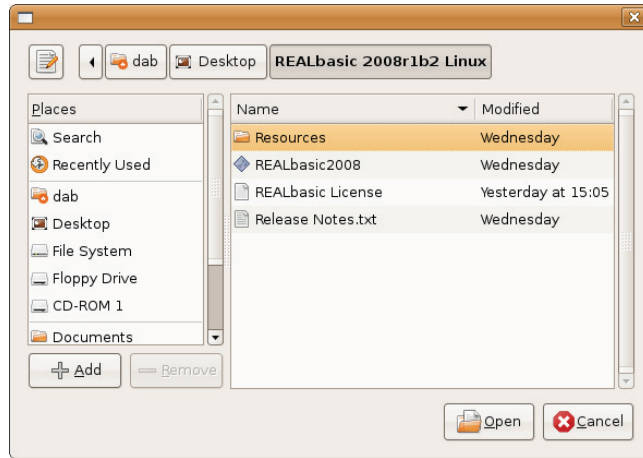


The Windows open file dialog box is shown in Figure 380 on page 501.

Figure 380. The Windows Open FolderItem dialog box.



A Linux version of the dialog box is shown below.

Figure 381. The Linux Open FolderItem dialog box.

To present the user with an Open File dialog box, you can call either the `GetOpenFolderItem` function or use the `OpenDialog` class. The former is a standard function that presents a standard open-file dialog box. The latter allows you to create a customizable open-file dialog box in which you can specify the following aspects of the dialog:

- Position (Left and Top properties)
- Default directory (Initial Directory property)
- Valid file types to show (Filter property)
- Text of Validate and Cancel buttons (ActionButtonCaption and CancelButtonCaption properties)
- Text that appears above the file browser (Title property)
- Text that appears below the file browser (PromptText property)

The `GetOpenFolderItem` function displays the Open File dialog box and returns a `FolderItem` object that represents the file the user selected. One or more file types (that have been defined in the File Type Sets Editor or with the `FileType` class via the language.) must be passed to the `GetOpenFolderItem` function. It presents only those file types to the user in its browser. In this way, the user can only open files of the appropriate type. To pass more than one file type, separate them with semicolons.

The following example displays the Open File dialog box, allowing the user to select only jpeg files, and then displays the selected file's modification date:

```
Dim f as FolderItem
f=GetOpenFolderItem(FileTypes1.jpeg)
MsgBox f.ModificationDate.ShortDate
```

If the user clicks the Cancel button rather than the Open button in the Open File dialog box, `GetOpenFolderItem` returns `Nil`. You will need to make sure the value returned is not `Nil` before using it. If you don't, a `NilObjectException` error will be generated. The following example shows how the code from the previous example should be written to check for a `Nil` object:

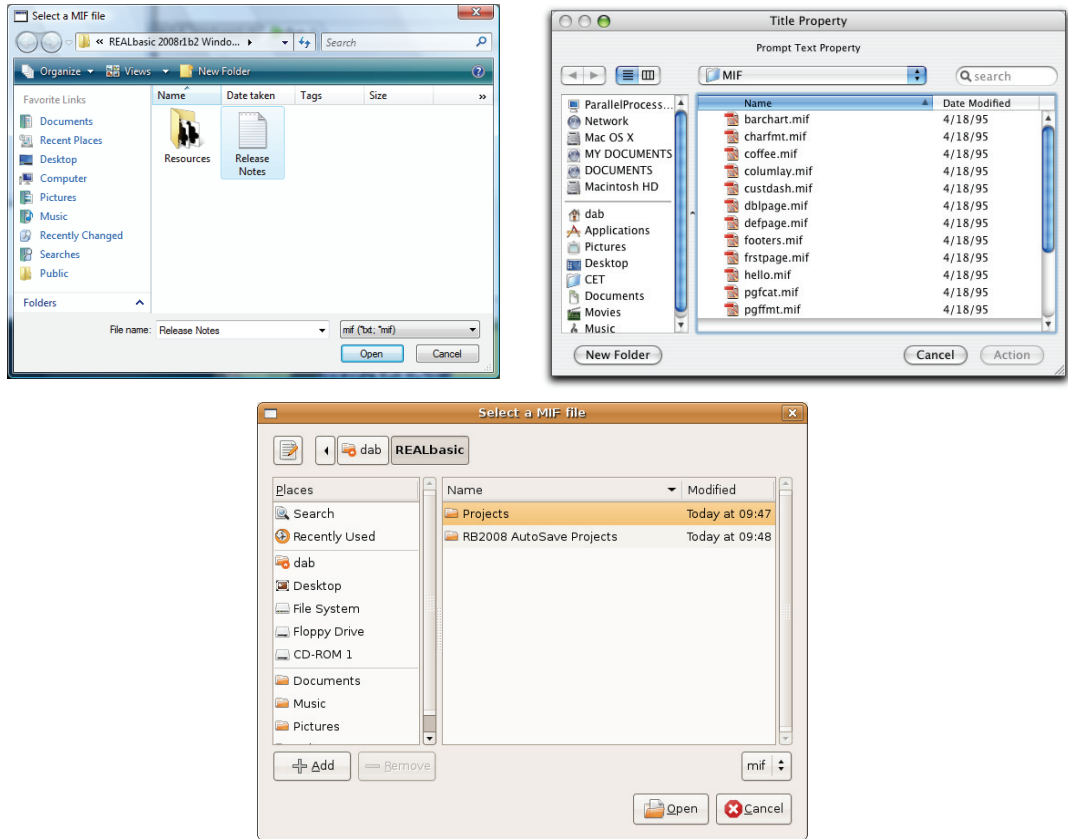
```
Dim f as FolderItem
f=GetOpenFolderItem(FileTypes1.jpeg)
If f <> Nil Then
    MsgBox f.ModificationDate.ShortDate
End if
```

When you use the `OpenDialog` class, you create a new object based on this class and assign values to its properties to customize its appearance. The following example uses a custom prompt and displays only one file type:

```
Dim dlg as OpenDialog
Dim f as FolderItem
dlg= New OpenDialog
dlg.InitialDirectory= SpecialFolder.Documents
dlg.Title="Select a MIF file"
dlg.Filter=FileTypes1.pdf
f=dlg.ShowModal()
If f <> Nil then
    //proceed normally
Else
    //User Cancelled
End if
```

Customized open-file dialog boxes are shown in Figure 382.

Figure 382. OpenDialog boxes on Windows Vista, Mac OS X, and Linux.



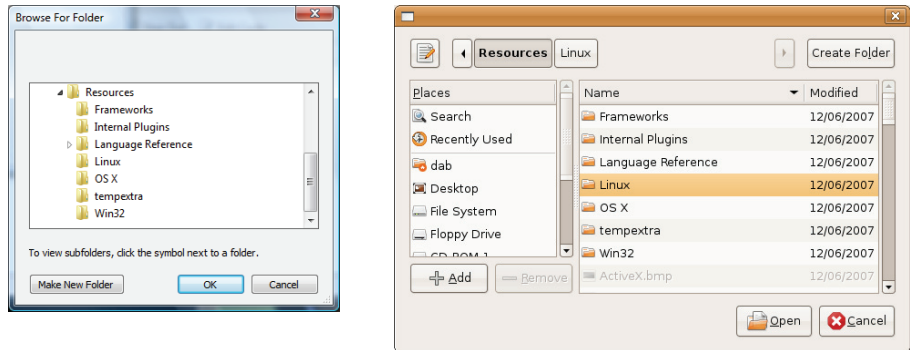
For more information, see the `GetOpenFolderItem` class and the `OpenDialog` class in the *Language Reference*.

For more information on file types, See “Understanding File Types” on page 480.

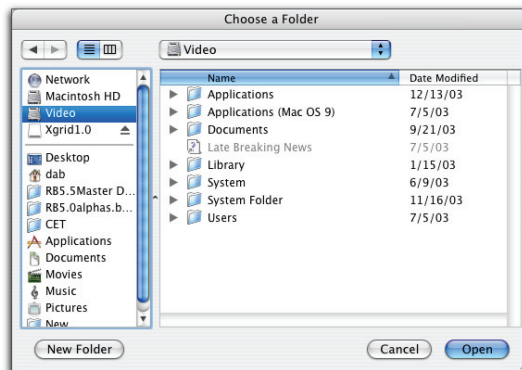
Getting The Selected Folder From An Open Folder Dialog Box

The Open File dialog box doesn’t allow the user to select a folder. Fortunately, REAL Studio’s `SelectFolder` function displays an Open Folder dialog box that lets the user choose a folder rather than a file. The `SelectFolderDialog` class performs the same function, but allows you to customize the appearance of the dialog.

On Windows and Linux, the `SelectFolder` function displays the following dialog box:

Figure 383. The Windows and Linux SelectFolder dialog boxes.

If the user is running Mac OS X (10.3 or above), the browser shown in Figure 384 appears.

Figure 384. The Mac OS X (10.3 or above) Select Folder dialog.

The SelectFolder function returns a FolderItem that represents the folder the user selects when he clicks the Open button at the bottom of the dialog box (or the Choose button in the Navigation Services Open Folder dialog box). If the user clicks the Cancel button rather than the Select/Choose button, SelectFolder returns Nil. You need to check for it before using the returned value.

The following example displays the number of items in the folder selected by the user:

```
Dim f as FolderItem
f=SelectFolder
If f <> Nil Then
    MsgBox Str(f.Count)
End if
```

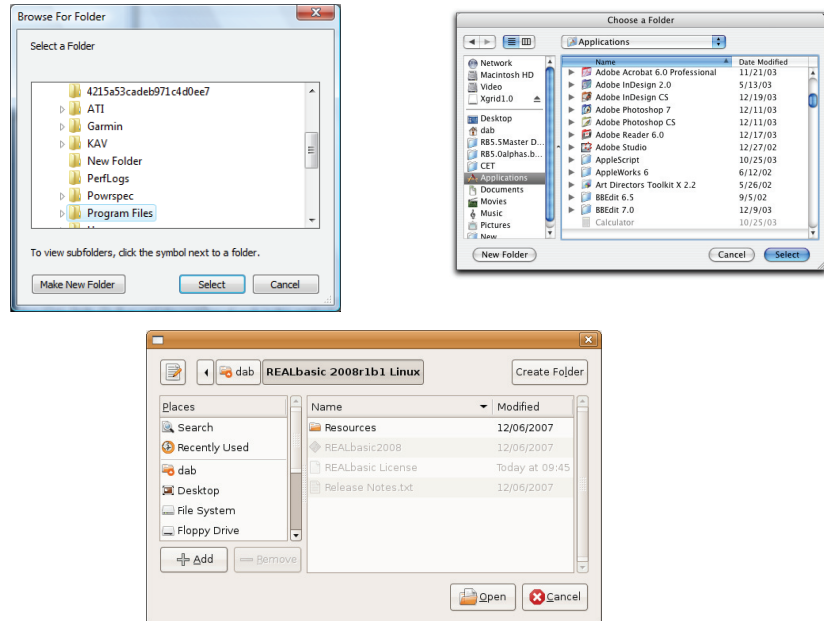
When you use the `SelectFolderDialog` class, you create an object based on this class and assign values to its properties to customize its appearance. You can customize the following properties:

- Position (Left and Top properties)
- Default directory (Initial Directory property)
- Valid file types to show (Filter property)
- Text of Validate and Cancel buttons (ActionButtonCaption and CancelButtonCaption properties).
- Text that appears above the file browser (Title property)
- Text that appears below the file browser (PromptText property)

The following example opens a select folder dialog box and presents the contents of the Applications folder on the user's startup volume in the browser:

```
Dim dlg as SelectFolderDialog
Dim f as FolderItem
dlg=New SelectFolderDialog
dlg.PromptText="Select a folder"
dlg.ActionButtonCaption="Select"
dlg.InitialDirectory=SpecialFolder.Applications
f=dlg.ShowModal()
if f <> Nil then
    //use the folderitem here
else
    //user cancelled
end if
```

SelectFolder dialog boxes created with the `SelectFolderDialog` class are shown in Figure 385.

Figure 385. Select folder dialogs on Windows, Mac OS X, and Linux.

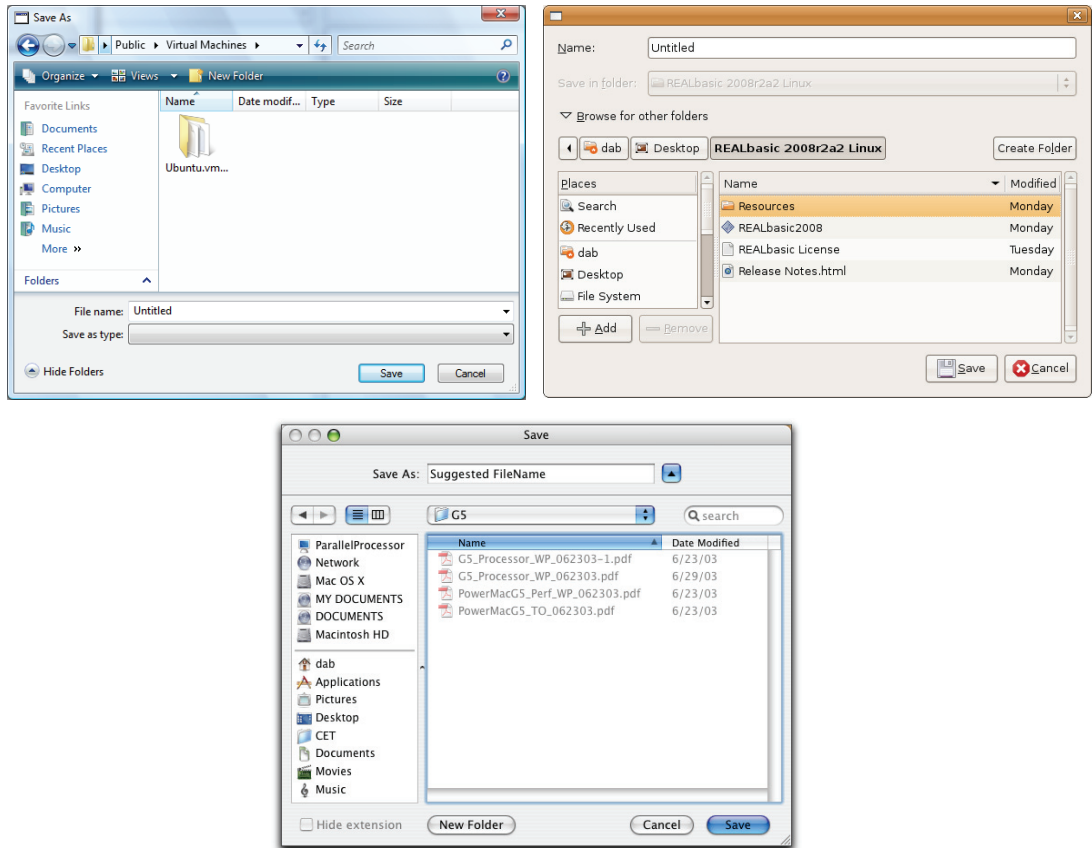
For more information, see the `SelectFolder` and `SelectFolderDialog` classes in the *Language Reference*.

Using the Save As Dialog Box

The Save As dialog box is used to let the user choose a location in which to save a file and give the file to be saved a name.

The Windows, Linux, and Mac OS X versions of this dialog box are shown in Figure 386.

Figure 386. The Windows, Linux, and Mac OS X Save As dialog boxes.



REAL Studio's `GetSaveFolderItem` function presents the Save As dialog box. The `SaveAsDialog` class allows you to create a customized version of this dialog. Both objects return a `FolderItem` that represents the file the user wishes to save. This is an important distinction because the file doesn't exist yet. You must provide additional code that will create the file and write the data to the file. You will learn about creating files and writing data later in this chapter.

When you call the `GetSaveFolderItem` function, you define the type of file and the default name for the file (that will appear in the Name field in the Save As dialog box). The file type (which is the first parameter of the function) is the name of any file type defined for the project in the File Types dialog box.

Like the other functions that return `FolderItems`, you should make sure the `FolderItem` returned by `GetSaveFolderItem` is not `Nil` before using it (The `FolderItem` will be `Nil` if the user clicked Cancel).

The following example presents the Save As dialog box. The dialog presents a default file name of "Untitled". It also returns a `FolderItem` whose Type and Creator match the "jpeg" file type as defined for the project in the File Types dialog box, in

a File Types Set called “FileTypes1”. If the user clicks the Save button, the name the user chose for the file is displayed:

```
Dim f as FolderItem
f=GetSaveFolderItem(FileTypes1.jpeg,"Untitled")
If f <> Nil and f.Exists Then
    MsgBox f.name
End if
```



NOTE: If you are going to create a text file with the FolderItem returned, you can pass an empty string as the first parameter of the GetSaveFolderItem function. The method that creates a text file (CreateTextFile) will assign the file type and creator automatically.

When you use the SaveAsDialog class, you create a new object based on this class and customize the dialog by assigning values to its properties. You can customize the following aspects of the dialog:

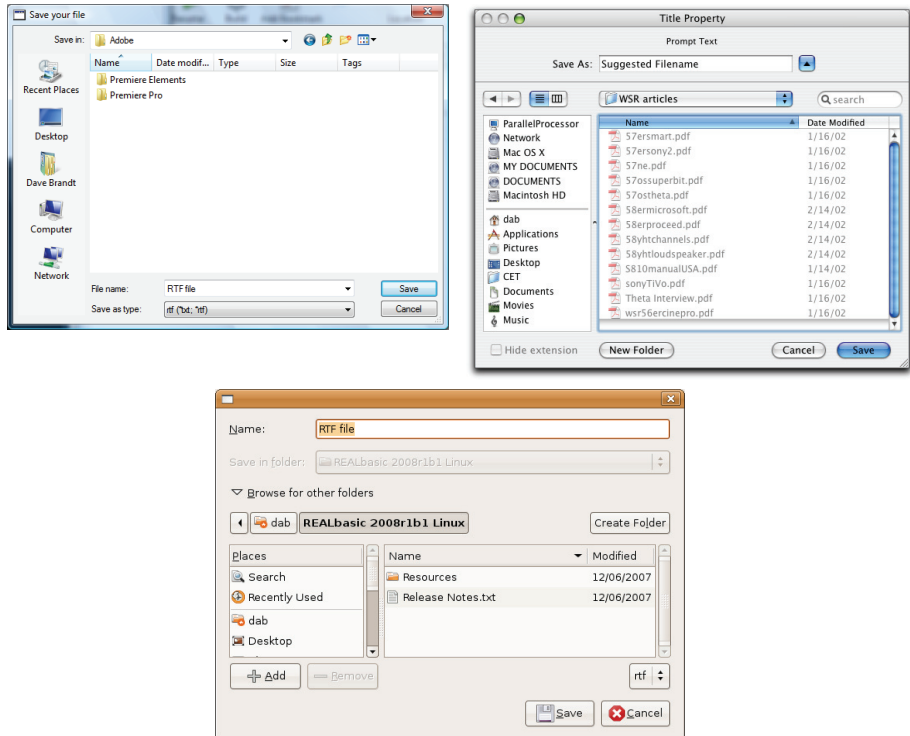
- Position (Left and Top properties)
- Default directory (Initial Directory property)
- Valid file types to show (Filter property)
- Default filename (SuggestedFileName property)
- Text of the Validate and Cancel buttons (ActionButtonCaption and CancelButtonCaption properties)
- Text that appears above the file browser (Title property)
- Text that appears below the file browser (PromptText property)

The following example opens a customized save-file dialog box and displays the contents of the Documents directory in the browser area.

```
Dim dlg as OpenFileDialog
Dim f as FolderItem
dlg= New OpenFileDialog
dlg.InitialDirectory= SpecialFolder.Documents
dlg.Title="Select a pdf file"
dlg.Filter=FileTypes1.Pdf
f=dlg.ShowModal()
If f <> Nil then
    //proceed normally
Else
    //User Cancelled
End if
```

Customized save-as dialog boxes created using the SaveAsDialog class are shown in Figure 387.

Figure 387. Save as dialogs on Windows, Mac OS X, and Linux.



For more information on file types, See “Understanding File Types” on page 480.

For more information, see the `GetSaveFolderItem` function and the `SaveAsDialog` class in the Language Reference.

Working With Text Files

Text files can be read by text editors (like SimpleText, NotePad, gedit, and BBEdit) and word processors (like Microsoft Word and Pages). Text files can easily be created, read from, or written to with REAL Studio. Text files are convenient since they can be read by many other applications.

Whether you are going to read from a text file or write to a text file, you must first have a `FolderItem` that represents the file you are going to read from or write to.

Reading From a Text File

Once you have a `FolderItem` that represents an existing text file you wish to open, you open the file using the `Open` shared method of the `TextInputStream` class. This method is a function that returns a “stream” that carries the text from the text file to your application. The stream is called a *TextInputStream*. This is a special class of object designed specifically for reading text from text files. You then use `ReadAll` or `ReadLine` methods of the `TextInputStream` to get the text from the text file. The `TextInputStream` keeps track of the last position in the file you read from.

The `ReadAll` method returns all the text from the file (via the `TextInputStream`) as a string. The `ReadLine` method returns the next line of text (the text after the last character read but before the next end of line character). As you read text, you can determine if you have reached the end of the file by checking the `TextInputStream`'s `EOF` (end of file) property. This property will be `True` when the end of the file has been reached. When you are finished reading text from the file, call the `TextInputStream`'s `Close` method to close the stream to the file, making the file available to be opened again.

This example lets the user choose a text file using the Open-file dialog box and displays the text in a `TextArea`. It assumes that the valid text file types have been defined in a File Type Set called `TextTypes`:

```
Dim f as FolderItem
Dim stream as TextInputStream
f=GetOpenFolderItem(TextTypes.All) //all the file types in this set
If f <> Nil Then
    stream=TextInputStream.Open(f)
    TextArea1.text=stream.ReadAll()
    stream.Close
End if
```



NOTE: Because `ReadAll` reads all of the text in the file, the resulting string will be as large as the file. Keep this in mind because reading a large file could require more memory than the user has available for the application.

This example reads the lines of text from a file stored in the Preferences folder in the System folder into a `ListBox`.

```
Dim f as FolderItem
Dim stream as TextInputStream
f = SpecialFolder.Preferences.child("My Apps Prefs")
If f <> Nil and f.Exists then
    stream = TextInputStream.Open(f)
    While Not stream.EOF
        ListBox1.addrow stream.ReadLine
    Wend
    stream.Close
End if
```

Specifying an Encoding

If you are reading and writing text files with only your REAL Studio application, this code will work. However, if the files are coming from other applications or platforms, in other languages or a mixture of languages, then you may need to specify the encoding of the text. This is because the character codes above ACSII 127 may differ from what your application expects. When you read text, you can set the `Encoding` property of the `TextInputStream` to the encoding of the text file.

Here is the first example, amended to specify the text encoding of the incoming text stream. The code assumes that there is a File Type Set in the application named TextTypes.

```
Dim f as FolderItem
Dim stream as TextInputStream
f=GetOpenFolderItem(TextTypes.All)
If f<> Nil Then
    stream=TextInputStream.Open(f)
    Stream.Encoding=Encodings.Windows.ANSI //specify the ANSI encoding
    TextArea1.text=stream.ReadAll()
    stream.Close
End if
```

The Encodings object provides access to all encodings. Use it whenever you need to specify an encoding. You can also specify the text encoding by passing the encoding as an optional parameter to Read or ReadAll.

For more information about text encodings, see the section “Working with Text Encodings” on page 416.

Writing to a Text File

Once you have a FolderItem that represents the text file you wish to open and write to, you open the file using the Append shared method of the TextOutputStream class. If you are creating a new text file or overwriting an existing text file, use the Create shared method of the TextOutputStream class. These methods are functions that return a “stream” that carries the text from your application to the text file. The stream is called a *TextOutputStream*. This is a special class of object designed specifically for writing text to text files. You then use the WriteLine method of the TextOutputStream class to write the text to the text file.

The WriteLine method, by default, adds a carriage return to the end of each line. This is controlled by the TextOutputStream’s Delimiter property which can be changed to any other character.

When you are finished writing text to the file, call the TextOutputStream’s Close method to close the stream to the file making the file available to be opened again.

This example displays the Save As dialog box then writes the contents of three TextFields to the text file and closes the stream. It assumes that there is a file type

called “Text” in the TextTypes File Type Set This is one of the common file types that are built into the File Type sets Editor.

```
Dim f As FolderItem
Dim fileStream As TextOutputStream
file=GetOpenFolderItem(TextTypes.Text)
if f <> Nil then
    fileStream=TextOutputStream.Append(f)
    fileStream.WriteLine namefield.Text
    fileStream.WriteLine addressfield.Text
    fileStream.WriteLine phonefield.Text
    fileStream.Close
End if
```

If you want to create a new text file, then call TextOutputStream.Create instead. This example passes a default filename for the new text file:

```
Dim t as TextOutputStream
Dim f as FolderItem

f=GetSaveFolderItem(FileTypes1.Text,"CreateExample.txt")

if f <> Nil then
    t=TextOutputStream.Create(f)
    t.WriteLine namefield.text
    t.WriteLine addressfield.text
    t.WriteLine phonefield.text
    t..Close
End if
```

Specifying an Encoding

As is the case with reading text files, you may need to specify an encoding when you write out a text file. If the application that will read the file is expecting that the text is in a specific encoding, you should convert the text to that encoding before exporting it.

Before writing out a line or the entire block of text (with the Write method) use the ConvertEncoding function to convert the encoding of the text. Here is a revised example. It converts the text to the MacRoman encoding.

```
Dim file As FolderItem
Dim fileStream As TextOutputStream
file=GetSaveFolderItem(TextTypes.Text,"My Info")
fileStream=TextOutputStream.Create(file)
fileStream.Write ConvertEncoding(namefield.Text,Encodings.MacRoman)
fileStream.Close
```

Limitations of Text Files

Text files can only be accessed sequentially. This means that to read some text that is in the middle of the file, you must read all of the text that comes before it. It also means that to write some text to the middle of a text file, you have to write all of the text that comes before the text you wish to insert, then write the text you wish to insert, then the text that follows the text you wish to insert. You can not read text from a text file and write to the same text file at the same time. If these limitations are going to be a problem for your project, consider using a binary file instead. For more information on binary files, See “Working With Binary Files” on page 521.

Working With Styled Text Files

REAL Studio makes it easy to read from and write to text files that support styled text. TextEdit is an example of an application that supports styled text.

Loading Styled Text Into a TextArea

Once you have a FolderItem that represents the styled text file you wish to read text from, you can read the styled text using the Open method of the TextArea class. To use this method, pass it the FolderItem whose text you wish to display. This TextArea must have its Styled property set to True. This is the default.

This example displays an Open File dialog box. It then reads the styled text from the file chosen and displays it in a TextArea. It assumes that all the text file types that you want to read are defined in a File Type Set called “TextTypes”.

```
Dim f as FolderItem
f=GetOpenFolderItem(TextTypes.All)
If f <> Nil Then
  If Not TextArea1.Open(f) then
    MsgBox "Open Failed"
  End if
End if
```

If the Open method returns True, then the open succeeded; if it failed, it returns False. This is handled by placing the call in the If statement.

Writing Styled Text From a TextArea to a File

Once you have a TextArea that contains the styled text that you wish to save, you can write the styled text using the Save method of the TextArea class. To use this method, pass it the FolderItem will hold the styled text. This TextArea must have its Styled and MultiLine properties set to True. This is the default.

This example displays the Save As dialog box. It then writes the styled text from TextArea1 to the new file. It uses the Text file type as defined in the File Type Set “TextTypes”.

```
Dim f as FolderItem
f=GetSaveFolderItem(TextTypes.Text,"Untitled")
If Not TextArea1.Save(f,True) then
    MsgBox "Save Failed"
End if
End if
```

If the Save was successful, then the Save method returns True.

Working with StyledText Objects

The other way to save and load styled text is to use the StyledText class. It enables you to work with styled text that is not connected to a TextArea. You can manipulate the styles and paragraph alignments within the StyledText object and save it to disk for later use.

You work with StyledText as a series of concatenated StyleRun objects. The StyleRun entry in the *Language Reference* shows how to save a StyledText object to disk using a BinaryStream. The approach is to save the sequence of StyleRuns into a MemoryBlock object and then write the MemoryBlock to disk by calling the Write method of the BinaryStream class. See the entry for StyleRun for the sample code that does this.

The RTFData property of the StyledText class stores the styled text in RTF format. With it, you can parse RTF generated elsewhere. The parser supports only the styled text features supported by the StyledText class itself. It also enables you to save your styled text to disk so that it can be read by any other application that understands RTF on any platform.

The following code saves the styled text in a TextArea to disk in RTF format. It assumes that the File Type Set, “TextTypes” has one item, ApplicationRTF, that defines the RTF file type. This file type can be added to a FileType set by clicking the Common File Type button then selecting the “application/RTF” type from the pop-up menu at the bottom of the File Type Set editor.

```
Dim s as TextOutputStream
Dim f as FolderItem
f=GetSaveFolderItem(TextTypes.ApplicationRtf,"TestSaveRTF")
s=TextOutputStream.Create(f)
s.Write TextArea1.StyledText.RTFData
s.Close
```

Working With Picture Files

REAL Studio has built-in support for opening and saving both bitmap and vector picture files. On Macintosh, it supports vector and raster PICT files and on Windows, it supports the .bmp (bitmap) and .emf (Extended Metafile Format) formats.

Beyond that, it utilizes QuickTime and/or GDI+ to support other file formats if they are available on the end-user's computer.

Before opening or saving a picture file, you must have a `FolderItem` that represents the picture file you wish to work with. From there, you can open picture files with the `FolderItem`'s `OpenAsPicture` method and save a picture to the file with the `SaveAsPicture` or `SaveAsJPEG` methods. If you are working with a vector graphics file, you can use the `FolderItem`'s `OpenAsVectorPicture` method. REAL Studio will try to convert the objects in the file to editable `Object2D` objects for you.

Saving Pictures

To save a picture to a file, you need a `FolderItem` that represents a new picture file or an existing file. Next you call the `FolderItem`'s `SaveAsPicture` method, passing it the picture you wish to save. This example saves the backdrop of a `Canvas` control to a jpg file, the name of which is specified by the user in a `Save As` dialog box. The jpg file type is defined in the File Type Set named "ImageTypes".

```
Dim f as FolderItem
f=GetSaveFolderItem(ImageTypes.jpg,"Untitled")
If f <> Nil Then
    f.SaveAsPicture Canvas1.backdrop
End If
```

The `SaveAsPicture` method has an optional second parameter that enables you to specify the format that you want to use for the saved file. This is how you can specify either a vector or bitmap format. For example, you can specify a meta-format that maps to various concrete formats based on the target, the data being saved, or other criteria. The BMP, PNG, and JPEG formats are supported on all platforms.

The following table gives the codes for meta-formats.

Table 34: Codes, class constants, and descriptions of meta-formats used by the `SaveAsPicture` method.

Value	Class Constant	Description
0	<code>SaveAsMostCompatible</code>	Most widely-used format for the platform Mac = PICT Win32 = BMP)
1	<code>SaveAsMostComplete</code>	Format most likely to retain all vector info Mac = PICT Win32 = EMF)

Table 34: Codes, class constants, and descriptions of meta-formats used by the SaveAsPicture method. (Continued)

Value	Class Constant	Description
2	SaveAsDefault	DefaultVector or DefaultRaster, depending on picture data Mac = PICT Win32 vector=EMF Win32 raster= BMP
3	SaveAsDefaultVector	Platform's standard vector format Mac = PICT Win32 = EMF)
4	SaveAsDefaultRaster	Platform's standard raster format Mac = Raster PICT Win32 = BMP)

**Macintosh-only
Formats**

Value	Class Constant	Description
100	SaveAsMacintoshPICT	Macintosh PICTs are "type 2" PICTs that are saved with the full resolution of the image. Includes simple vector data.
250	SaveAsMacintoshRasterPICT	Flattens all vector data to pixels.

Windows Formats

Value	Class Constant	Description
150	SaveAsPNG	Portable Network Graphics. Supported on Windows only if GDI+ is installed.
151	SaveAsJPEG	Joint Photographics Expert Group. Supported on Windows only if GDI+ is installed.
300	SaveAsWindowsWMF	Windows Metafile format (old vector format).
301	SaveAsWindowsEMF	Extended Metafile format (newer vector format).
350	SaveAsWindowsBMP	Windows bitmap format.
402	SaveAsGIF	Graphics Interchange Format. Supported on Windows only if GDI+ is installed.
403	SaveAsTIFF	Tag Image File Format. Supported on Windows only if GDI+ is installed.

In your code, you can use a class constant as the second parameter of `SaveAsPicture`. For example:

```
Dim f as FolderItem
f=GetSaveFolderItem(ImageTypes.ImagesJPEG,"Untitled")
If f <> Nil Then
    f.SaveAsPicture(DSC_0001,FolderItem.SaveAsJPEG)

    If f.LastErrorCode > 0 then
        MsgBox Str(f.LastErrorCode)
    Else
        MsgBox "Picture saved!"
    End if
End if
```

Saving the image drawn into the graphics property of a Canvas control (perhaps by its Paint event handler) is a bit trickier. That's because the graphics property isn't a picture. The way to solve this is to add a picture property to the window. Any drawing you do in the Canvas control's graphics property should also be drawn into the picture property. The picture can then be saved using the `SaveAsPicture` method. The picture property you add to the window must be filled with a reference to a new picture before you attempt to write to it. This is accomplished using the `NewPicture` function in the window's Open event handler. In this example, the picture property (called "p") is set to a new picture:

```
p=newpicture(Canvas1.width,Canvas1.height,32)
```

In this example, the `MouseDown` event handler of the `Canvas1` control draws a black pixel when the user clicks on the `Canvas1` control. The drawing is also done to the window's `p` (picture) property:

```
Me.Graphics.Pixel(x,y)=Rgb(0,0,0)
p.Graphics.Pixel(x,y)=Rgb(0,0,0)
```

Finally, the picture property "p" can be saved to a picture file:

```
Dim f as FolderItem
f=GetSaveFolderItem(FileTypes1.jpg,"Untitled")
If f <> Nil Then
    f.SaveAsPicture p
End If
```

Opening Pictures

To open a picture, you need a `FolderItem` that represents the image file you wish to open. If you are working in a cross-platform situation, consider using BMP files because REAL Studio can read this format without QuickTime being installed on the user's computer.

To open the picture, call the `FolderItem`'s `OpenAsPicture` method which returns the picture. If QuickTime is installed, `OpenAsPicture` will open any kind of graphics file QuickTime will open (JPEG, GIF, etc.). QuickTime is not required for opening JPEG, GIF, and BMP files on Windows. `OpenAsPicture` does not open JPEG images on Linux.

This example displays the Open File dialog box and lets the user choose a JPEG file that is then assigned to the `Backdrop` property of a `Canvas` control.

```
Dim f as FolderItem
Dim p as Picture
f=GetOpenFolderItem(FileTypes1.Jpeg)

if f <> Nil then
  If f.exists then
    p=f.OpenAsPicture
    Canvas1.Backdrop=p
  End if
End if
```

If the file consists of vector graphics, you can call `OpenAsVectorPicture` instead of `OpenAsPicture` to ask REAL Studio to try to map the objects in the file into REAL Studio vector graphic objects. This option is available for vector PICT files on Macintosh and emf files on Windows.

The original file may have objects for which there is no equivalent in REAL Studio. REAL Studio will do its best to map these objects, but there may be some loss of information, depending on the characteristics of the original file. PICTs support unrotated Rectangles, Lines, Ellipses, RoundRects, Polygons, Text, Pixmaps, and Arcs.

Extended Metafile files (.emf) support unrotated Rectangles, Lines, Ellipses, RoundRects, Polygons, Text, Pixmaps, and Arcs.

Files of type .emf are displayed as actual size. For the most part this is huge; you probably will want to scale them down before viewing (`pic1.objects.scale = scalingFactor`). We find that a scale factor of 0.045 is a good value.

Working With Sound Files

REAL Studio supports opening Macintosh and WAV sound files but not saving them. Specifically, Macintosh sound files are those files whose “Kind” field in the file’s Get Info dialog box is listed as “Sound.” On Linux, xine is used to play sounds. If it is not installed `libsndfile` is used. It supports only WAV and AIFF.

To open a sound file, you must first have a `FolderItem` that represents the sound file you wish to open. Next, you can open the sound file and place its contents into a

Sound object with the FolderItem's OpenAsSound method. This example opens a sound file and plays it:

```
Dim f as FolderItem
Dim s as Sound
f=GetFolderItem("Doh!")
If f<> Nil Then
    s=f.OpenAsSound
    s.Play
End if
```

You can also get sounds stored in a snd resource inside your application. For more information, See “Supported Resource Types” on page 526.

Working With Movie Files

To open a movie file, you must first have a FolderItem that represents the file you wish to open. REAL Studio supports the QuickTime player on Macintosh and Windows Media Player on Windows. On Linux, REAL Studio uses GStreamer by default (it requires version 0,10+) and uses Xine if GStreamer is not available.

Next, you can open the file and place its contents into a Movie object with the FolderItem's OpenAsMovie method. This example opens a QuickTime file, assigns its movie to the Movie property of a MoviePlayer control, and plays the movie:

```
Dim f as FolderItem
Dim m as Movie
f=GetOpenFolderItem(VideoTypes.QuickTime)
If f<> Nil Then
    m=f.OpenAsMovie
    moviePlayer1.Movie=m
    moviePlayer1.Play
End if
```

NOTE: If your application needs a specific movie, you can drag it into the Project Editor rather than use GetOpenFolderItem or GetFolderItem.



If you want to edit the movie within REAL Studio, you need to open it as an EditableMovie. You would then display the movie in a MoviePlayer by assigning it to the Movie property of a MoviePlayer control.

The following example opens an existing movie as an `EditableMovie` and displays it in a `MoviePlayer` control named `ThePlayer`.

```
Dim f As FolderItem
Dim theEMovie as EditableMovie
f=GetOpenFolderItem(VideoTypes.QuickTime)
If f<>Nil and f.exists then
    theEMovie=f.OpenEditableMovie
    If theEMovie<>Nil then
        ThePlayer.movie=theEMovie
    End if
End if
```

The `EditableMovie` class includes methods that allow you to:

- Create a new `QuickTime` video track,
- Define a segment in the `EditableMovie`,
- Cut or copy the segment to the Clipboard,
- Paste a segment into the current movie,
- Append another movie segment to the current movie,
- Insert a segment into the current `EditableMovie` at a specified position,
- Create new sound and/or video tracks,
- Scale the video track.

A `QuickTime` video track has properties and methods that allow you to:

- Set the frame rate for the movie, either in frames per second or, more accurately, as a timescale and frame duration,
- Set the codec used for compression,
- Set the compression quality,
- Set the bit depth.

You can save your changes to the `EditableMovie` automatically when you are finished or at any time by calling the `CommitChanges` method. Please see the `EditableMovie` and the `QTVideoTrack` entries in the *Language Reference* for more information on the methods used to work with `QuickTime` movies.

Working With Binary Files

Binary files are simply files that store values in their binary format rather than as text. For example, the number 30000 stored as text requires 5 characters of text (or bytes) to store in a text file. In a binary file, this number can be written as a short integer (or just “short”). A short requires only 2 bytes.

Binary files also have the added benefit that you can read and write to a file without having to close the file in-between. For example, you can open a binary file, read some data, then write some data, and close it. You can also read and write anywhere in the file without having to read through all the data preceding the data you want.

Most applications store data in a binary format. The format is simply the arrangement of data within the file. In order to read a binary file, you must know how the data is arranged. If your own application created the file, you will know this, but if the file was created by an application you didn't write, you may not know it. Some formats are made public. For example, the PICT format is public. Other formats are not. Many software vendors do not publish the binary formats that their applications use to create documents.

BinaryStreams Data read from or written to a binary file travels through a *BinaryStream*. A *BinaryStream* is a class of object in REAL Studio that represents the flow of information between the *FolderItem* and the file it represents. Unlike the *TextInputStream* class (which can only be used to read from a text file) and the *TextOutputStream* class (which can only be used to write data to a text file), *BinaryStreams* can be used for both reading data and writing data. You can even indicate to the *BinaryStream* that you will only be reading data from the file so that the file can continue to be available to other applications for writing.

BinaryStreams can read and write specific types of data, such as strings, short integers, long integers, currency, and single bytes. They can also be used to read and write raw unformatted binary data.

Reading From a Binary File Once you have a *FolderItem* that represents the file you wish to open, you open the file using the *Open* method of the *BinaryStream* class. It returns a *BinaryStream*. You then use the methods of the *BinaryStream* class to read data from the stream. The *BinaryStream* class includes separate methods for reading each data type that REAL Studio supports.

The *BinaryStream* keeps track of the last position in the file you read from in its *Position* property. However, you can change this property's value to move the position to any location in the file.

This example presents the Open File dialog box, reads a file made up of strings, and displays those strings in a *TextArea*. Notice that since the code is only reading data and not writing, *False* is passed to the *Open* method to indicate the file should be opened in "read-only" mode. Also, reading continues in a loop until the stream's

EOF (end of file) property is True. REAL Studio will set the EOF property to True automatically once the end of the file is reached.

```
Dim f as FolderItem
Dim stream as BinaryStream
f=GetOpenFolderItem(FileTypes1.Text)
If f<> Nil Then
    stream=BinaryStream.open(f,False)
    Do
        TextArea1.AppendText stream.Read(255)
    Loop Until stream.EOF
    stream.Close
End if
```

When you read a BinaryStream, you may need to take the encoding of the characters into account. To do so, you can pass an optional parameter to the Read and ReadPString methods that specifies the encoding. Use the Encodings object to get any encoding and pass it to Read or ReadPString. For example, the following line specifies the ANSI encoding:

```
TextArea1.AppendText stream.Read(255,Encodings.WindowsANSI)
```

For more information, see the section “Working with Text Encodings” on page 416.

Writing to a Binary File

Once you have a FolderItem that represents the file you wish to open and write to, you can open the file using the Open method of the BinaryStream class. If you are creating a new file, use the Create method of the BinaryStream class. This method returns a BinaryStream. You then use the appropriate method for writing data to the stream. The BinaryStream class includes separate methods for each data type that REAL Studio supports.

The BinaryStream keeps track of the last position in the file you wrote to in its Position property. However, you can change this property’s value to move the position to any location in the file.

When you are finished writing data to the file, call the BinaryStream’s Close method to close the stream to the file making the file available to be opened again.

This example displays the Save As dialog box and writes the contents of the TextArea1 to a text file.

```
Dim f as FolderItem
Dim stream as BinaryStream
f=GetSaveFolderItem(FileTypes1.Text,"Untitled.txt")
If f<> Nil Then
    stream=BinaryStream.Create(f,True)
    stream.Write(TextArea1.text)
    stream.Close
End if
```

Working With Macintosh Resources

All Macintosh files (including applications, which are really just files) can have two sections called “forks.” The “data” fork holds data that is in whatever format the application that created the file chose to put it in. The resource fork can contain formatted information such as icons, sounds, menu bars, pictures, string lists, etc.

REAL Studio provides support for reading from and writing to the resource fork of a file. This is done using a FolderItem object that represents the file whose resource fork you wish to access or create.

REAL Studio supports operations on the resource fork only on the Macintosh platform. The discussion and examples in this section assume that the application is running on a Macintosh. If your application manages resources, you can use the TargetMacOS constant to verify that the application is running on Macintosh before doing any resource fork operations. There is one exception to this: it is possible to install custom cursors in your application using the ‘CURS’ resource and access them from a built Windows application. This is described in the section “Custom Cursors in Windows Applications” on page 527.

Opening a File’s Resource Fork

Once you have a FolderItem, you can open the resource fork for the file the FolderItem represents. This is done using the OpenResourceFork method of the FolderItem. This method returns a ResourceFork class object which can then be used to access the resource fork of the file. If the file has no resource fork, the OpenResourceFork method returns Nil.

This example displays the Open File dialog box, allowing the user to choose a file. It then reports if the file has no resource fork or tells the user how many different types of resources are in the file's resource fork:

```
Dim f as FolderItem
Dim rf as ResourceFork
f=GetOpenFolderItem("any")
If f <> Nil Then
    rf=f.OpenResourceFork
    If rf=Nil Then
        Beep
        MsgBox "This file has no resource fork."
    Else
        MsgBox "This file has "+Str(rf.TypeCount)+" resource types."
    End if
End if
```



NOTE: The “any” file type passed to GetOpenFolderItem was defined in the File Type Sets Editor or with the FileType class via the language and uses the string “????” as both Type and Creator. The “?” is a wildcard character, matching any type or creator code.

Adding a Resource Fork to a File

Before you can write to the resource fork of a file, it must have one first. You can use the FolderItem's OpenResourceFork method to determine if the file has a resource fork. If it doesn't, you can use the FolderItem's CreateResourceFork method to add a resource fork to the FolderItem. Once the file has a resource fork, you can begin writing to it.

This example displays an Open File dialog box and adds a resource fork to the file (if the file doesn't already have one). This example assumes that a file type for any application has been added to the File Type set “FileTypes1.”

```
Dim f as FolderItem
Dim rf as ResourceFork
f=GetOpenFolderItem(FileTypes1.ApplicationAny)
If f <> Nil Then
    rf=f.OpenResourceFork
    If rf=Nil Then
        rf=f.CreateResourceFork("any")
    End if
End if
```

Adding a Resource Fork to a Project

You can add a resource fork to a REAL Studio project by dragging a resource file into the Project Editor. REAL Studio recognizes the file type of “rsrc” as a resources file. You can have as many resource files in your project as you wish. The resources from all your resource files will be copied into the built Macintosh application. In the case of a conflict, later resource files overwrite earlier ones, where the files are written in the order in which they appear in the Project Editor.

The following example opens the application's resource fork:

```
Dim rf as ResourceFork  
rf=App.ResourceFork
```

Supported Resource Types

REAL Studio provides high level support for PICT, CICN, CURS, and snd resources. You can use the AddPicture method to add PICT resources to the resource fork and use GetPicture or GetNamedPicture to get PICT resources from the resource fork. You can use GetCicn to get a cicn (color icon) resource. Sounds can be read from snd resources using the GetSound method of the ResourceFork class. However, you can access any type of resource. REAL Studio provides method for getting and setting raw data from any type of resource in a resource fork. However, you must know the format of the resource data to be able to successfully read from it or write to it.

Reading Resources

The ResourceFork class has methods for reading data from four different types of resources. You can read PICT resources using the GetPicture and GetNamedPicture methods of the ResourceFork class. You can get a color icon as a picture by calling the GetCicn method. You can get a large (32 x 32) icon using the GetIcl method and a small (16 x 16) icon with the GetIcs method. For example, the following line of code displays a picture resource in an ImageWell:

```
Me.Image=App.ResourceFork.GetPicture(128)
```

Use GetCicn, GetIcl, and GetIcs in the same way.

You can load sounds from 'snd' resources using the GetSound method of the ResourceFork class.

Reading Custom Cursors

REAL Studio lets you add a custom cursor as a resource. There are separate MouseCursor properties at the application, window, and control levels. You can assign a custom cursor using these properties.

MouseCursor Constructor

The most convenient way of creating a custom cursor, however, is to create it using the MouseCursor class's constructor. You need only to pass it the picture of the custom cursor and the x,y coordinates of the hotspot. The hotspot is the point in the cursor that the operating system uses as the position of the cursor. Here is a template for the constructor:

```
Dim pic as New Picture(32,32,32) //height,width, and depth  
pic=pointerImage //pointerimage was added to the Project Editor  
Me.MouseCursor(pic,10,10) //picture, x-hotspot, y-hotspot
```

The custom cursor image was created outside of REAL Studio and added to the Project Editor.

This is the most direct way to create a custom cursor. The `MouseCursor` constructor is supported on Windows and Linux and will be supported for Macintosh Cocoa. Macintosh Carbon builds are currently not supported.

Cursor Resources

Use `GetCursor` to assign a custom cursor to the `MouseCursor` property of the application, a window, or a control. For example, the following line in a window's `MouseEnter` event handler assigns a custom cursor to the `MouseCursor` property of the window.

```
self.MouseCursor=App.ResourceFork.GetCursor(128)
```

This line causes REAL Studio to display the custom cursor whenever the mouse enters the window's region. If you want to change the cursor when the mouse is over a control in the window, you can use a line such as this in the control's `MouseEnter` event handler:

```
self.MouseCursor=App.ResourceFork.GetCursor(129)
```

and then restore the window's custom cursor with the following line in the control's `MouseExit` event handler:

```
self.MouseCursor=App.ResourceFork.GetCursor(128)
```

Notice that these lines do not assign a cursor to the control's own `MouseCursor` property; they only change the assignment to the control's parent window.

Assign a custom cursor to a control's `MouseCursor` property only when you don't use either the Window's or the App class's `MouseCursor` properties. If either the parent window or the application has a custom `MouseCursor` property, then the control's `MouseCursor` property is ignored.

To assign a custom cursor to the application as a whole, create a new class called "App" and make its Super Class "Application." Then add a line such as this to the App class's Open event handler.

```
MouseCursor=App.ResourceFork.GetCursor(128)
```

This causes the REAL Studio application to display this custom cursor all the time. That is, the `MouseCursor` properties of any of the application's windows or controls will be ignored.

Custom Cursors in Windows Applications

You can use CURS resources to assign custom cursors in built Windows applications. This, at present, is the only case in which resources are supported on Windows builds. The relationships among the `MouseCursor` properties of the Application, Window, and Control are the same as for Macintosh applications, but you must create the resource files in a special way. This technique is also recommended for Macintosh builds.

The key is that you must create a separate resource file for each custom cursor that you add to the project. Each resource file must have only a CURS resource that contains one cursor. Typically, you will assign the custom cursor to ID 128.

You then drag all of these resource files to your project. These special resource files will appear in the Project Editor with a cursor icon.

You can then access the custom cursors by referencing their names. For example, the following statement in a control's MouseEnter event handler changes the pointer to the Sponge cursor when the mouse enters the region of the control:

```
Self.MouseCursor=Sponge
```

(You would then restore the cursor with a statement in the control's MouseExit event handler.)

Reading Other Resources

To read data from other resources, you must know the format of the resource. For example, to read the STR# resource, you can use the GetResource method of a ResourceFork class. This will return the bytes that make up the resource ID you specify. To then do anything useful with the data, you will need to know that the first two bytes are the number of strings in the resource, followed by the strings themselves. The strings are Pascal strings so their first byte is the length of the string.

When you build your application, you can enter version information about the application in the App class's Properties pane. That information is stored in a 'vers' resource that becomes part of your application. An application can access the 'vers' resource using the GetResource method of the ResourceFork class. The information written to the 'vers' resource is described in the section "Version Information" on page 702.

Writing To Resources

REAL Studio provides methods via the ResourceFork class that can be used to write to resources. You can use the AddPicture method to write REAL Studio pictures into a PICT resource. For all other types of resources, you can use the AddResource method to create new resources and the RemoveResource method to delete specific resources. To modify a resource other than PICT resources, you read the data of the resource using the GetResource method, then write the data back by deleting the resource with the RemoveResource method and then recreating the resource using the AddResource method.

More Information on the ResourceFork

For more information, see the ResourceFork class in the *Language Reference*.

Files Opened From the Desktop

If your application is designed to read from and/or write to files, you may need to consider how your application will react when the user accesses files stored on his/her computer.

Files Opened by Double-Clicking

If the user double-clicks on a file whose creator code matches your stand-alone application's creator code, the user will be expecting your application to open the file automatically. If your application is prepared to open a file and take some action, then you should also support the user's double-clicking on the file. This is done by using the App class that was added to your project by default. This class represents your application as a whole and will receive information when the user double-clicks on a document whose creator code matches your application's creator code. It has an OpenDocument event handler that is executed when the user double-clicks on a file. This event handler is passed a FolderItem as a parameter. This FolderItem represents the file the user double-clicked on.



To take action when the user double-clicks on a file from the desktop, do this:

- 1 Double-click the App object in the Project Editor.**

A project created from the Desktop Application project template has an App class based on the Application class. If you don't have an App class, create a new class and set its Super Class to Application in its Properties pane.

- 2 Expand the Events list in the Code Editor browser.**

- 3 Click on the OpenDocument event to select it.**

- 4 Enter the code that should execute when the user double-clicks on a file.**

You can access the file using the Item parameter passed to the OpenDocument event handler.

Files Dropped On Your Application's Icon

REAL Studio treats a file dropped on your application's icon at the desktop the same way it treats the user's double-clicking on a file from the desktop. For more information, See "Files Opened by Double-Clicking" on page 529.

Creating New Files

When the user launches your application without opening a file, REAL Studio assumes that the user will probably want to create a document (assuming you application is document/file based). If you have created a class based on the Application class, that class's NewDocument event handler will execute. On Macintosh, this event handler also executes when your application receives an Open Application AppleEvent (oapp) or when a user uses AppleScript to tell the Finder to open your application.

You can call the NewDocument event handler by entering NewDocument in your code. This allows you to have a single location to put the code for your application

that creates new documents. Using this event handler, your application will respond to all the appropriate calls to create a new document.

Creating Reusable Objects with Classes

Classes act as templates for objects much in the same way that the windows listed in the Project Editor act as templates for the windows you open in your application. This chapter will introduce you to the benefits of classes, explain how to create and modify them, and how you can create custom interface controls using classes.

Contents

- The benefits of classes
- Understanding subclasses
- Modifying classes
- Managing menus within classes
- Using classes in your projects
- The Application class
- Creating custom controls with classes
- Virtual methods
- Class Interfaces
- Interface inheritance

The Benefits of Classes

Classes offer lots of benefits. They are:

Reusable Code

When you add code to a PushButton control to customize its behavior, you can only use that code with that PushButton. If you want to use the same code with another PushButton, you need to copy the code and then make changes to the code in case it refers to the original PushButton (since the new PushButton will have a different name than the original).

Classes store the code once and refer to the object (like the PushButton) generically so that the same code can be reused any number of times without modification. If you create a class based on the PushButton control and then add your code to that class, any instances of that custom class will have that code.

Smaller Projects and Applications

Because classes allow you to store code once and use it over and over in a project, your project and the resulting application is smaller in size and may require less memory.

Easier Code Maintenance

Less code means less maintenance. If you have basically the same code in several places in your application, you have to keep that in mind when you make changes or fix bugs. By storing one copy of the code, you will spend less time tracking down all those places in your project where you are using the same code. Making a change to the code in a class automatically updates any places where the class is used.

Easier Debugging

The less code you have, the less code there is to debug.

More Control

Classes give you more control than you can get by adding code to the event handlers of a control in a window. In fact, some classes can even manage menus. You can also use classes to create custom controls. And with classes, you have the option to create versions that don't allow access to the source code of the class, allowing you to create classes you can share or sell to other REAL Studio users.

As you can see, there are many benefits to creating classes. Overall, classes make your programming effort more efficient.

Understanding Instances

REAL Studio has many classes built-in to it. The `PushButton`, `StaticText`, `TextField`, and `ListBox` are examples of some of the built-in control classes. Control classes are templates for objects that you use in your application's interface. As templates, the classes are abstract in the sense that you do not use any of the templates themselves in applications. Instead, each template serves as an inexhaustible supply of instances of classes. You use these instances.

For example, when you drag a `TextArea` from the Controls list to a window, you create a usable *instance* of that class. The new instance has all the properties and methods that were built into the `TextArea` template. You get all that for free—styled text, multiple lines, scroll bars, and all the rest of it. You customize the particular instance of the `TextArea` by modifying the values of the instance's properties.

Understanding Subclasses

You may find situations where you would like to have an object that is a slightly altered version of one of the built-in classes. For example, you might want a version of the `TextArea` control that disables the Cut and Copy items on the Edit menu, preventing the user from putting sensitive data on the Clipboard. You might want to create a `ListBox` that, by default, has the months of the year in it. You can create your own versions of these built-in classes by creating *subclasses* that you add to your Project Editor.

There's an important difference between adding an instance of `TextField` to a window versus adding a subclass based on `TextField` to your project. In the latter case, you can customize the subclass based on `TextField` itself and use instances of the customized subclass in several places in the application. You can also save the customized subclass so that you can reuse it in other projects.

What is a Subclass?

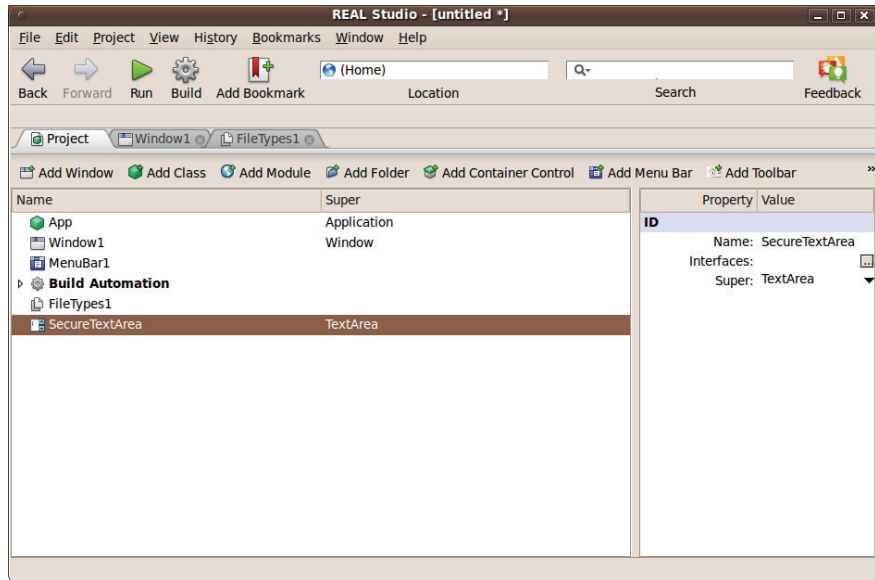
A *subclass* is simply a class that has a super class. A super class is a class the subclass is based on. The super class is also sometimes called the “parent” class. Subclasses inherit all of their super's properties, methods, constants, and events¹. The subclass can then modify them. In fact, a subclass is identical to its super class until you start modifying it. After that, it's different from its super class only in the ways you make it different by adding properties, modifying events, and adding or modifying methods.

Examples of Subclasses

For example, to create a `TextArea` that prevents the user from copying data to the Clipboard (Let's call it a `SecureTextArea`), you create a new class and choose `TextArea` as its super class. The new subclass is shown in Figure 388.

1. There is an exception to this rule. If you set the Scope of a method, property, or constant to Private and then use this class as a super class, its subclasses will not inherit the Private methods, properties, or constants.

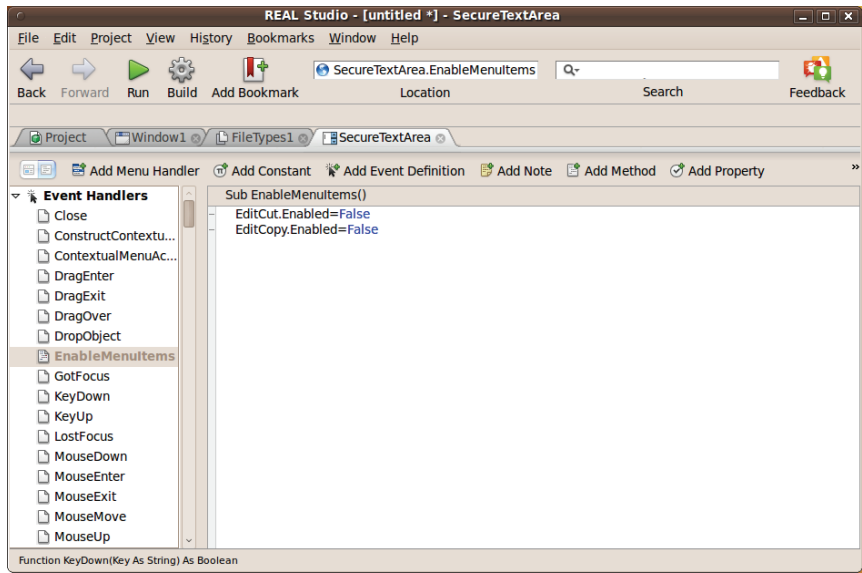
Figure 388. SecureTextArea based on the TextArea class.



REAL Studio automatically enables the Cut and Copy menu items on the Edit menu when characters are selected in a TextArea. Since SecureTextArea is based on TextArea, it inherits these features. Because TextAreas can get the focus, any subclass of the TextArea control has an EnableMenuItems event handler. This allows SecureTextArea to control the menus when it has the focus. To prevent the user from using the Cut and Copy menu items, you set the Enabled property of these menu items to False in your SecureTextArea's EnableMenuItems event handler.

In Figure 389, the user has opened the Code Editor for the SecureTextArea class and added code to disable the Cut and Copy menu items.

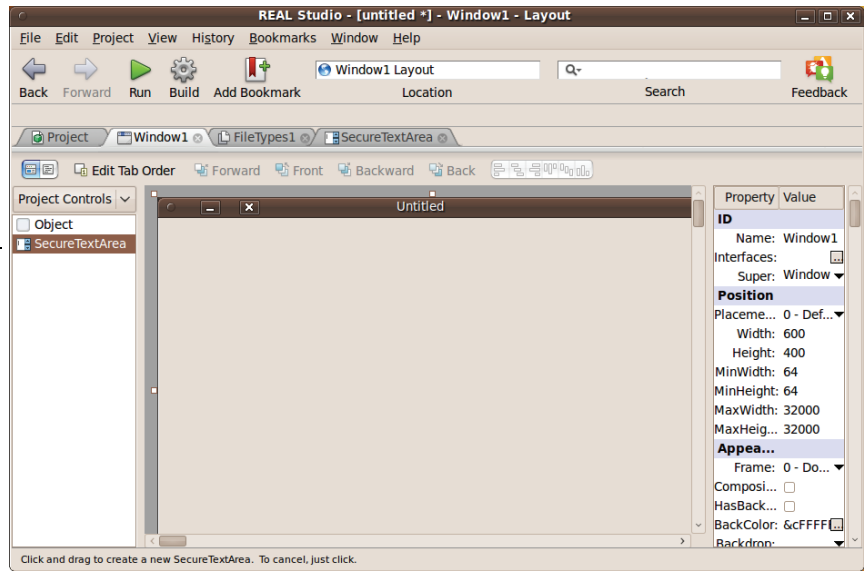
Figure 389. Disabling the Copy and Cut menu items in SecureTextArea.



To use an instance of SecureTextArea, switch to the Window Editor and use the drop-down menu above the Controls list to display the Project controls. This is the list of custom controls that have been added to the Project Editor.

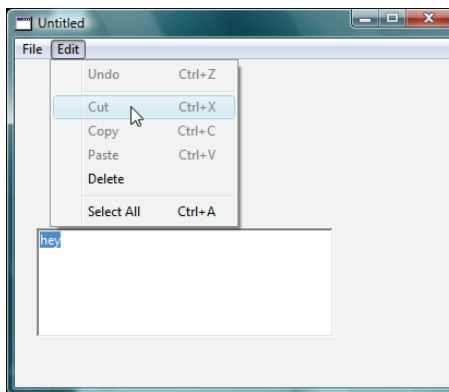
Figure 390. The Project Controls list.

Project Controls is selected from the drop-down list.



Add the SecureTextArea item to the window, just as you would a built-in control. It behaves exactly like a 'regular' TextArea, except that the Cut and Copy menu items are disabled when the user has selected text.

Figure 391. Selected text cannot be copied from an instance of SecureTextArea.

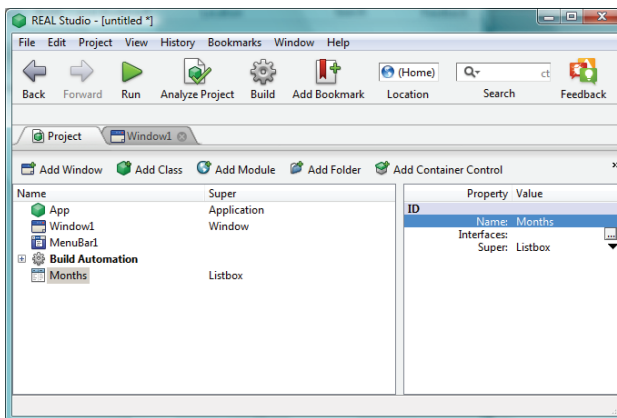


Since SecureTextArea is in your Project Editor, you can create instances of it anywhere you like and maintain its code in one central place. You can export it and import it into other projects.

Here is another example. Suppose you want to create a ListBox that, by default, displays the names of the months of the year, with the current month selected.

You create a new class and choose ListBox as its super class.

Figure 392. The Months class added to the Project Editor.



Open the Code Editor for the Months class. In the Open event handler of your new subclass add the month names, the heading, and code that selects the appropriate month in the list.

In this case, the Open event handler is:

```
Me.HasHeading=True
Me.InitialValue="Months"

Me.AddRow "January"
Me.AddRow "February"
Me.AddRow "March"
Me.AddRow "April"
Me.AddRow "May"
Me.AddRow "June"
Me.AddRow "July"
Me.AddRow "August"
Me.AddRow "September"
Me.AddRow "October"
Me.AddRow "November"
Me.AddRow "December"

Dim d as New Date
Me.Selected(d.Month-1)=True
```

To add an instance of the Months class to a window, switch to the window's Window Editor and then use the drop-down list above the Controls list to switch to Project Controls. The Months class will be listed there. Drag it to the window. When you run the application, the Open event handler will run and populate the Listbox with the months of the year. The last two lines of the Open event handler will highlight the current month.

You might want to create a TextField that only allows the user to enter numbers. Let's call it "NumbersOnlyTextField." To do this, you create a subclass of the TextField control and put code in the KeyDown event handler that allows only numbers and rejects all other characters. Once created, you can use your new subclass in many different places in your project, but the code exists only in one place.

Once created, custom control classes can be exported as self-contained objects that can be used in other projects. Just right+click (Control-click on Macintosh) on the custom control class in the Project Editor and choose Export... from the contextual menu.

Subclasses are classes. They are called subclasses to differentiate them from built-in classes and emphasize the point that they inherit the properties, events, and methods of their parent class. Because subclasses are classes, they can be the super class to other subclasses. For example, suppose you had already created the NumbersOnlyTextField subclass mentioned earlier. Now, you need a TextField that allows only numbers within a certain range. You could duplicate the NumbersOnlyTextField subclass and then modify its code. However, this would make your project larger and more difficult to maintain. If you found a bug in the code of the NumbersOnlyTextField, you would have to remember that you used

that code in other places as well, track them down, and fix them. A more efficient way is to create a new subclass and choose the `NumbersOnlyTextField` as its super class. The new subclass (let's call it "`NumberRangeTextField`") would utilize all of the properties, events, and methods of its super class. However, you can add code to the `TextChanged` event handler that allows only numbers within a specific range.

Referring to a Class's Properties and Methods From Within the Class

When you add control such as a `PushButton` to a window and then add code to that control, you are adding code to one instance of the `PushButton` class. REAL Studio gives the instance the default name of "`PushButton1`". The code in the Action event of `PushButton1` is actually a method of `Window1` and has access to the public, protected, and private items of `Window1` as any other method of that class.

Suppose you add a `PushButton` named "`PushButton1`" to a window that should be disabled after the user clicks it. The code in the `PushButton`'s Action event handler could be:

```
PushButton1.Enabled=False
```

However, it is preferable to refer to the `PushButton` using the "`Me`" keyword rather than the `PushButton`'s name. The code would be:

```
Me.Enabled=False
```

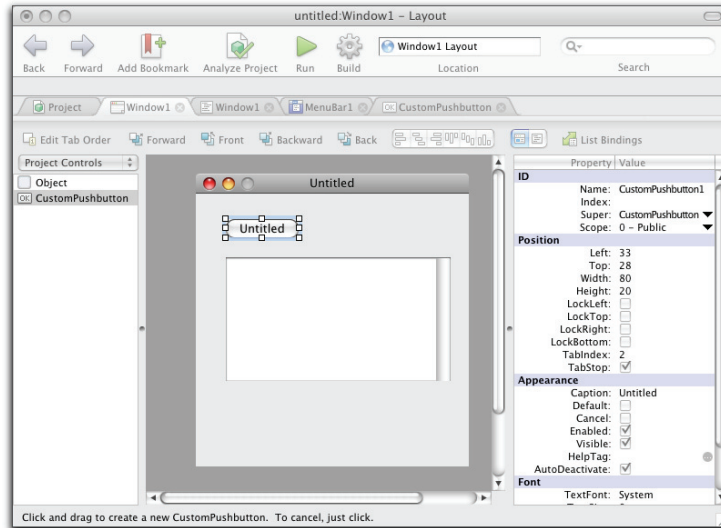
`Me` refers to the control that fired the event. It is a reference to the control that owns the event handler. In this case, it is `PushButton1`. The advantage of using `Me` instead of the control's name is that you can copy and paste the code to another control and it will work without any modification. If you used the control's name, you would need to update the code each time you copied and pasted it into another control's Action event.

A more flexible approach is to create a custom `PushButton` class that has the desired event handlers built in. With this approach, you add a custom class to the Project Editor and set its Super Class to `PushButton`. You then add the Action Event handler to the custom class.

This creates a customized `PushButton` that you can use in many places in your application. All of the code that you add to the custom class will automatically become part of all instances of that custom class. Consequently, you don't include object references to the custom class in its own code. The code would be:

```
Enabled=False
```

When you create a custom class based on a control, the custom control appears in the Window Layout editor in the Project Controls list. In the Window Layout Editor, switch from the Built-in controls to the Project controls list. Then add it to the window in the normal way.

Figure 393. A Custom PushButton in the Window Layout Editor.

When you add an instance of that custom PushButton to a window, the code added to the custom class will automatically be operating on the instance of the class. In this example, the customized PushButton is the same as a “regular” PushButton except that it has its own Action event handler. The code you added to the class becomes the Action event handler for all instances of the custom PushButton class.

Creating Classes

Adding a class to a project is easy. If you want to subclass the class from an existing class in the project, use the steps in the section “Creating a Subclass from an Existing Class” on page 540.

To add a new class, do this:

- 1 If it is not already visible, click on the Project tab to display the Project Editor.**
- 2 Click the Add Class button in the Project Editor toolbar or choose Project ► Add ► Class.**

A new class, named Class1, is added to the Project Editor. When the new class is selected, the Properties pane changes to indicate the new class’s properties.

By default, the new class is not subclassed from any other class. Often you want the new class to inherit the properties of an existing class.

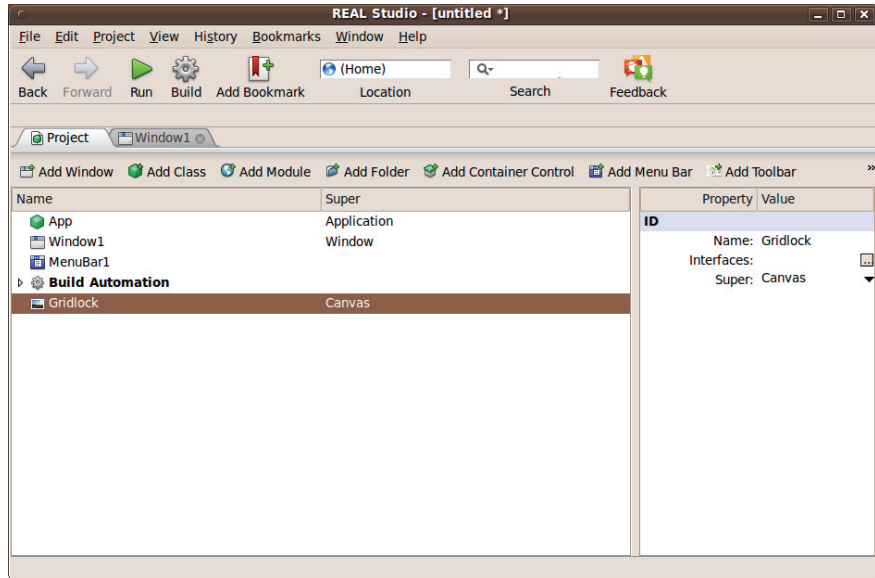
- 3 Use the Properties pane to give the new class a meaningful name and, if desired, use the Super pop-up menu to subclass it from an existing class.**

If the Super class is one of the built-in controls, the new subclass gets the icon belonging to that control in the Project Editor.



For example, in Figure 394 on page 540 a subclass of Canvas has been added to the project called Gridlock. It will be used to create a rectangular grid in a window.

Figure 394. The Gridlock class is based on the Canvas control.



The small icon for Gridlock is the same as the icon for its Super class. With the new class selected in the Project Editor, you can double-click it to open the Code Editor for the new class.

Creating a Subclass from an Existing Class

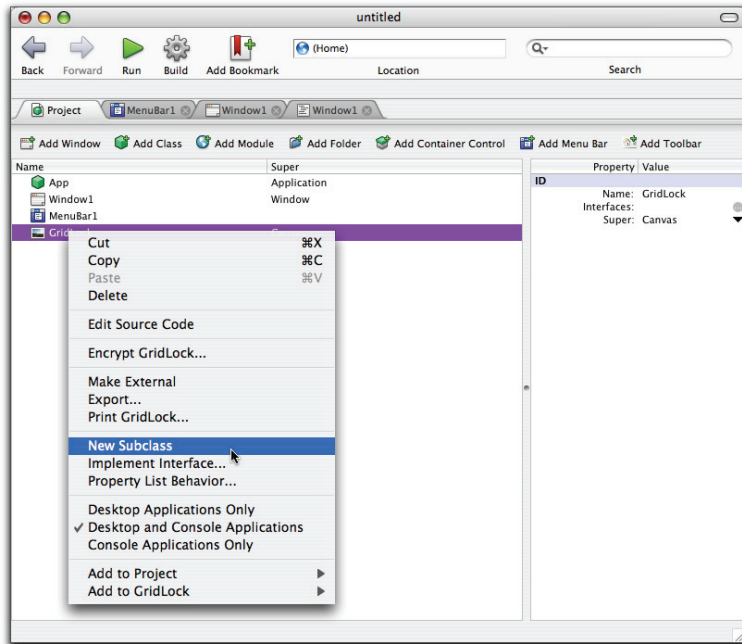
When you want to create a subclass of an existing class that you've already added to the Project Editor, you can do so using these two shortcuts.

To create a subclass of an existing class in the Project Editor, do this:

- 1 **Right+click (Control-click on Macintosh) on the class from which you want to create a subclass.**
- 2 **Choose New Subclass from the contextual menu.**

A new class is added to the project. In the Properties pane, the existing class is used as the Super class of the new class. The new class is named *CustomClassName*, where *ClassName* is the name of the parent class.

Figure 395. Using the Project Editor contextual menu to create a new class.



3 Use the Properties pane to rename the new subclass.

You can also create a subclass of an existing class by repeating the procedure for creating a new class and then manually choosing the parent class from the Super pop-up menu in the Properties pane.

Creating a Superclass from an Existing Class

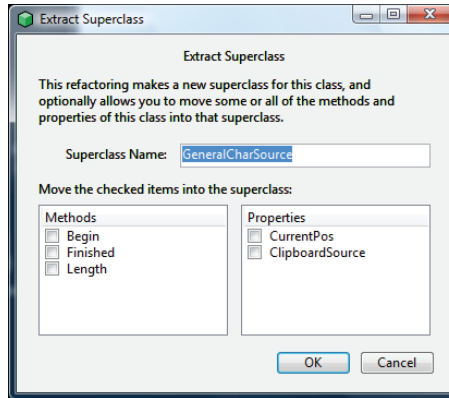
You can also create a super class from an existing class that does not have a Super class. When you do so, you can specify which properties and methods should be assigned to the super class and which should remain with the subclass.

To create a super class of an existing class in the Project Editor, do this:

- 1 **Right+click (Control-click on Macintosh) on a class that does not have a Super class.**
- 2 **Choose Extract Superclass from the contextual menu.**

The Extract Superclass dialog appears.

Figure 396. The Extract Superclass dialog for the class “CharacterSource”.



It suggests the name of the superclass, which is the name of the existing class preceded by “General”. It also lists the properties and methods of the existing class.

- 3 In the Methods and Properties lists, check the items that you want to include in the superclass.**
- 4 If desired, rename the superclass.**
- 5 When you are finished, click OK.**

REAL Studio then creates the superclass, makes the current class a subclass of this class, and moves the methods and properties that you checked to the superclass.

Saving Classes

Since classes are reusable, you will want to develop a library of classes that you can easily import into other projects. Choose **File ► Export** and save the class under a suitable name. To add the class to another project, simply drag it from the desktop into the Project Editor for the new project or choose **File ► Import** and use the open-file dialog box to choose the class to be imported. For more information, see “Exporting Classes For Use In Other Projects” on page 599.

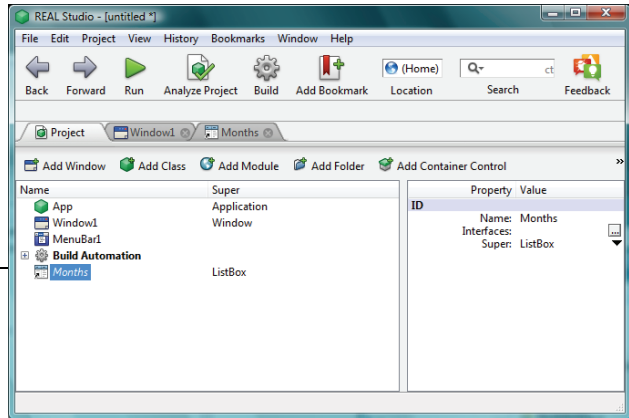
External Project Items

If you want to use the class in more than one project, you can export it as an external project item. An external project item is stored on disk and is referenced by each project that uses it. If a change to the class is made from one project, those changes are made available to all the other projects that reference the external item. Changes to the external project item are saved to disk when you save the project.

To save a class in your project as an external project item, Right+click (Windows and Linux) or Control-click (Macintosh) on the class in the Project panel and choose the **Make External** contextual menu item. An **Export File** dialog box appears in which you can save the class to disk. When you complete the save, the item will be displayed in the Project Editor with an alias badge and the name of the class in italics. This indicates that it is now referenced as an external item.

Figure 397. The Months class as an external project item.

After you import the class as an external project item, it appears as an alias.



To add an external project item to another project, hold down the Ctrl+Shift (on Windows and Linux) or the ⌘ and Option keys (on Macintosh) while you drag the item from the desktop to the Project Editor. In the Project Editor, the external project item's name will be shown in italics.

For more information, see the section, “External Project Items” on page 80.

Modifying Classes

One of the big advantages of classes is the ability to modify existing classes. You do this by adding constants, properties, adding or changing events, and adding or changing methods.

Scope of a Class's Methods, Properties, and Constants

When you add a method, property, or constant to a class, you need to set its Scope attribute. The Scope of a method, property, or constant determines which other items in the project can access it. There are three possible values:

- **Public: Accessible from Anywhere:** A Public method, property, or constant is available to code throughout the application. Public methods represent the class's interface to the rest of the application. When you need to access a Public item, you use the “dot” notation. For example, if you declare a Public property, `myPublicProperty`, in `Class1`, you call it by referring to “`Class1.myPublicProperty`” outside `Class1`. Within a method or event handler belonging to `Class1`, it is in scope, so you can access it by referring to “`myPublicProperty`”.
- **Protected: Accessible from the Current Class and its Subclasses:** A Protected method, property, or constant is available only to other code within the class and subclasses based on this class. It is “invisible” to the rest of the application. When other code in the class needs to access a Protected method, property, or constant, you simply reference it by name. If you try to access a Protected method, property,

or constant outside of the class, REAL Studio will display an informative error message indicating that the item is out of scope.

- **Private: Accessible from the Current Class only:** A Private method, property, or constant is like a Protected item except it is not accessible to classes subclassed from the current class. Public and Protected classes are accessible to classes subclassed from the current class.

Adding Properties

You can add properties to a class to store values that its super class doesn't store. For example, you might want to create a subclass of the TextField control that stores the last value the user entered. This would allow you to selectively reject the current entry and restore the last entry. You add properties to a class the same way you add properties to a window.

To add a property to a class, do this:

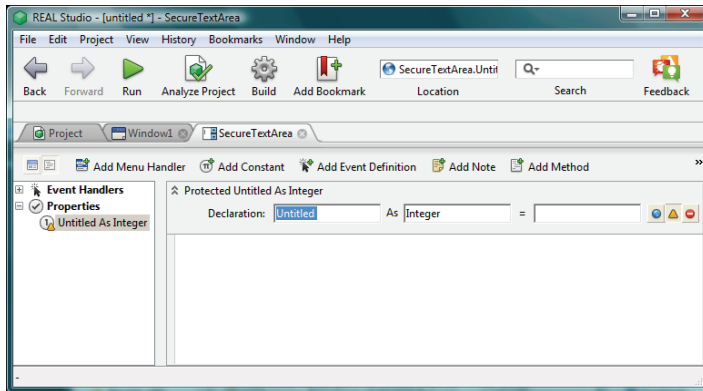
- 1 **If the class's Code Editor is not already open, double-click on the class in the Project Editor to open it.**

The Code Editor for the class appears.

- 2 **Click the Add Property button or choose Project ► Add ► Property.**

The Property declaration area appears above the Code Editor area. A “placeholder” property declaration is entered by default.

Figure 398. The Property Declaration area.



The Property Declaration area has three fields. They are for the name of the property, its data type, and its default value. The first two are required. If you do not provide a default value, the new property will be assigned the default value for the data type that you choose. Strings have a default value of an empty string, numbers have a default value of zero, booleans have a default value of False, colors have a default value of black, and objects have a default value of Nil.

You can also create classes that belong to modules. A module is a stand-alone item that serves as a container for classes, class interfaces, methods, properties, constants,

and other modules. For more information about module classes, see the section “Adding Classes to Modules” on page 384.

If you want to use a module class as the data type in the property declaration, you need to use the “dot” syntax to refer to it, i.e., *moduleName.className*. This refers to the class *className* in *moduleName*.

3 Fill in the Name and Data Type fields and, if desired, provide a default value.

The property can be an array. For example, if you want to declare a four-element String array of first and last names, addresses, and phone numbers called *aNames*, you would write:

```
aNames(3) as String
```

in the Declaration area. You can declare an array with no elements by using empty parentheses. You code can modify the number of elements. For example,

```
aNames() as String
```

4 Choose a Scope for the property by clicking one of the three Scope buttons.

Your choices, from left to right, are Public, Protected, and Private.

Figure 399. The Scope buttons.



For information about Scope, see the section, “Scope of a Class’s Methods, Properties, and Constants” on page 543.

5 (Optional) In the Code Editor area, add notes and comments about the property.

The text entered into the Code Editor for a property is automatically non-executable, even if you write valid REAL Studio code. Add any comments you wish, including code samples. For an example, see the section “Documenting Properties” on page 247.

Customizing the Properties List

You can customize the appearance and content of the Properties list for a class. You get a Properties list for a class in the Properties pane when you add an instance of the class to a window.

Figure 400. The Properties List for the Gridlock class.

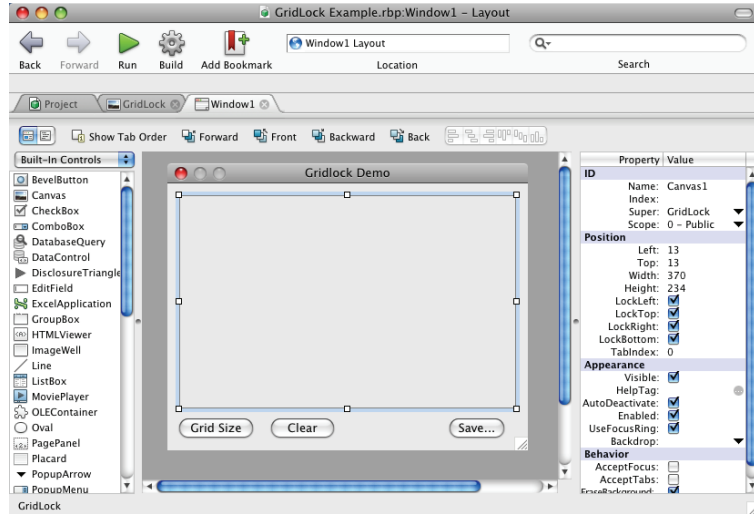


Figure 400 shows the Properties list for a custom class, Gridlock. Gridlock is based on the Canvas control and is selected in the Window Editor.

When you create a custom class based on a control you can also customize the appearance of its Properties list. You can:

- Add group headings,
- Edit the existing group headings,
- Change the order of the properties, including moving them to different groups,
- Set or modify default values,
- Change whether a property is shown or hidden,
- Add enumerations.

All of these options can be set in the Property List Behavior dialog box.

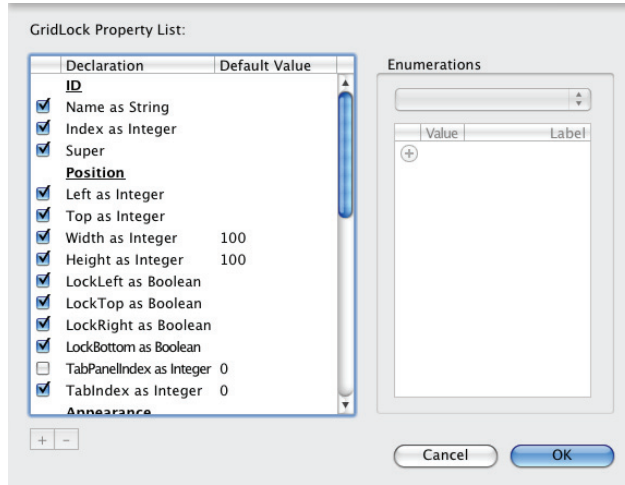
To customize the Property List, do this:



- 1 Right+click (Control-click on Macintosh) the name of any class in the Project Editor and choose Property List Behavior.**

The Property List Editor dialog box appears. The Editor lists all the properties for the class. They are grouped as they would be in the Properties list and their default values are shown.

Figure 401. The Property List Editor dialog box.



The checkmarks on the left indicate whether the property will be displayed in the Properties list. When a property appears in the Properties list, it can be assigned a value in the IDE rather than only with code. To display a property, select its checkbox; to hide it, deselect it.

- 2 To add or edit default values for a property, click twice in the Default Value field for that property to get an insertion point.**

The current default value will be selected (if there is one), and the border of field will be highlighted.

- 3 Type in the new value.**

You can reorder the properties by dragging them vertically. As you drag, a bar indicates where the property will be dropped if you release the mouse button.

Modifying Group Headings

You can edit the names of the group headings. Click twice on a header name to get an insertion point. The current text will be selected. Delete or add text as needed.

You can also add a new group heading. First select the row below which you want to add the new heading. Right+click on that row to display the contextual menu and choose Add Heading (On Macintosh you can click the plus sign below the list of properties). A new heading is added with the default text “Heading.” Edit the name and press Enter (Return on Macintosh) to save the value. To move the new heading to a different position, click on the row outside the Declaration field and then drag it vertically and drop it onto its new location.

Using Enumerations

In some cases, you would like the user to set the value of a property in the IDE by choosing the value from a pop-up menu. For example, you may have an integer property that has a limited number of acceptable values and each value has a specific meaning. For example, the Frame property of a window is an integer that can accept

the values of 0 to 10. Each value designates the type of the window. A good interface design is to present the labels for each frame type in the form of a pop-up menu instead of forcing the user enter the integers themselves.

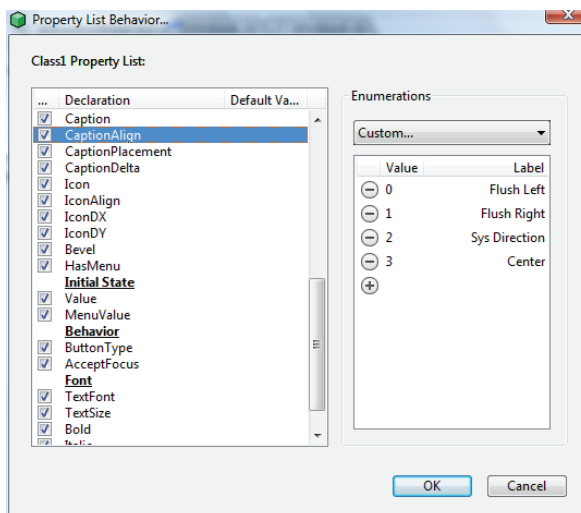
The BevelButton control has several integer properties like this. For example, the CaptionAlign property accepts the following four values:

Table 35. Values and labels for the CaptionAlign Property.

Value	Label
0	Flush Left
1	Flush Right
2	Sys Direction
3	Center

You use enumerations in the Property List behavior dialog to map the actual values of a property to their labels. Figure 402 shows the enumerations for the BevelButton's CaptionAlign property.

Figure 402. The Property List Behavior dialog for the BevelButton's CaptionAlign property.



Note that the CaptionAlign property is highlighted in the list of properties on the left, the Custom type of enumeration is selected in the drop-down list, and the values and labels for this property are entered in the Enumerations list. You follow this model when adding enumerations to properties that you create.

To add an enumeration to a property, do this:

- 1 **Right+click (Control-click on Macintosh) on the property in the Code Editor browser and choose Property List Behavior from the contextual menu.**
- 2 **Highlight the property to which you want to add enumerations.**



3 Choose Custom... from the Enumerations drop-down list.

The plus sign at the top of the Enumerations list becomes enabled.

4 Click on the plus sign to add the first value/label pair.

The Value field in the row becomes enterable.

5 Type in the first value and press Tab to select the Label field.**6 Enter the label for this value.****7 Repeat this process until you have entered all the values.****8 When you are finished with the Property List Behavior dialog, click OK to save your work.**

Adding Computed Properties

A computed property is actually made up of a pair of methods called Get and Set. It does not store a value in the property; it does the calculations that you program. This blurs the distinction between properties and methods. The Set method sets the value of a property (writes) and Get reads a value. You can implement either or both, making the computed property Read Only, Write Only, or Read/Write.

Computed properties are also referred to as getter and setter methods.

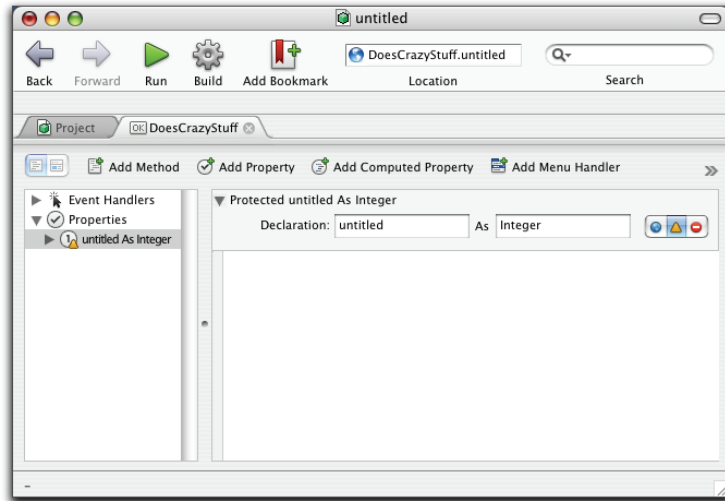
Computed properties can be declared as shared computed properties. See the following section, “Adding Shared Properties” on page 552 for information on shared properties.

To create a computed property, do this:

**1 Click the Add Computed Property button in the Code Editor toolbar or choose Project ► Add ► Computed Property.**

REAL Studio adds an untitled property to the window and displays a Property Declaration area above the Code Editor.

Figure 403. The Declaration Area for a Computed Property.



- 2 Enter the Name and Data Type for the computed property and set its scope.**

Notice that the browser area shows that you can expand the computed property.

- 3 Click the plus sign (Windows) or the disclosure triangle (Macintosh and Linux) to reveal the Get and Set methods that belong to the computed property.**

The Get method returns a value of the declared data type. The Set method is passed the value of the property. You do not have to implement both methods. You can make the computed property Read Only, Write Only, or Read/Write.

- 4 Write either the Get or the Set method or implement both methods.**

Converting a Property to a Computed Property



You can have REAL Studio convert an existing “regular” property to a computed property. It will rename the existing property and create getter and setter methods that return the property’s value and set its value to the passed value.

You cannot convert an array property to a computed property and you can’t create an array computed property. To convert an existing property to a computed property, do this:

- 1 **In the Code Editor browser area, expand the Properties item (if needed) and right-click (Control-click on Macintosh) on the property that you want to convert.**
- 2 **Choose Convert to Computed Property from the contextual menu.**

REAL Studio creates the getter and setter methods for the property and enters the code for them.

Suppose the name of the property was `myProperty`. It renames it “`mmyProperty`”. The computed property is called “`myProperty`” and its Get method is:

```
Return mmyProperty
```

Its Set method has one parameter, *value as DataType*, where *DataType* is the declared data type of `mmyProperty`. The code is:

```
mmyProperty = value
```

In other words, the computed property is all set up for the original “regular” property. It holds the value that the getter and setter methods manage.

An Example Computed Property

Here is a simple example of a computed property in a custom control class. It implements an annoying `PushButton` control that disables itself whenever the mouse pointer enters its region and enables itself when the mouse pointer exits its region. This makes it impossible to click, but it appears to be enabled until you try to use it.

First, add a class to the Project Editor and set its Super class to `PushButton`. Double-click it to open its Code Editor. Add a property to the class called `mDoesTheCrazies` and give it a data type of `Boolean`. Then select it and right+click (Control-click on Macintosh) to create a computed property.

Its getter method will be:

```
Return mmDoesTheCrazies
```

It gets the current value in the ‘regular’ property, `mmDoesTheCrazies`.

The setter method passes the current value in as the parameter, *Value*. It is:

```
mmDoesTheCrazies = Value
```

In the `MouseEnter` event for the custom `PushButton` class, add the code:

```
If mmDoesTheCrazies then me.Enabled=False
```

In the `MouseExit` event, enter the line:

```
If mmDoesTheCrazies then me.Enabled=True
```

Switch to the Window Editor for the default window and add an instance of the custom pushbutton class to the window. To initiate the process, add the following line to the `Open` event of the instance of the custom pushbutton in a window:

```
me.mmDoesTheCrazies=True
```

In other words, the “regular” property, `mmDoesTheCrazies`, stores the boolean value that is set by the computed property. The `Get` method gets the current value of `mmDoesTheCrazies`. The `MouseEnter` and `MouseExit` methods control the `Enabled` property of the custom `PushButton` depending on the computed property.

Adding Shared Properties

A shared property is like a “regular” property, except it belongs to the class, not an instance of the class. A shared property can be read or set from any instance of the class or from the class itself.

In contrast, “regular” properties are considered *instance* properties. This means that they belong to a particular instance of the class.

The key advantage of shared properties is that you can share them among all the instances of the class. For example, if you are using an instance of a class to keep track of items (e.g., persons, merchandise, sales transactions, and so forth) you can use a shared property as a counter. Each time you create or destroy an instance of the class, you can increment the value of the shared property in its constructor and decrement it in its destructor. (For information about constructors and destructors, see the section “Constructors and Destructors” on page 569.) When you access it, it will give you the current number of instances of the class.

For example, consider the example in the section “Using Classes in Your Projects” on page 577. The local variable “person” stores a reference to the instance of the custom class “Programmer”.

```
Dim person as Programmer  
person=New Programmer  
person.name="Jason"
```

Suppose the `Programmer` class contains a shared integer property, `Total`, that gets incremented each time a `Programmer` instance is created. For example, give the `Programmer` class a Constructor of:

```
Programmer.Total=Programmer.Total+1
```

The Destructor is:

```
Programmer.Total=Programmer.Total-1
```

Each time a Programmer instance is created or destroyed, the value of the Total shared property is changed to reflect the current count. The value of Total can be accessed from any Programmer instance or from the Programmer class itself.

To create a shared property, do this:

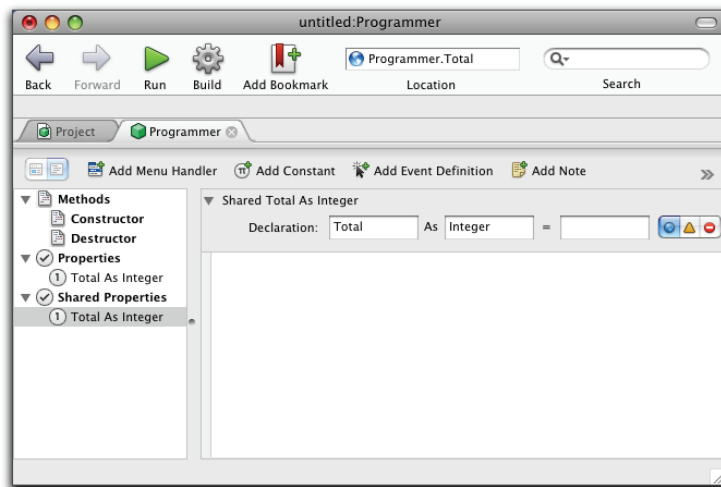
- 1 **Choose Project ► Add ► Shared Property or, if it is available, click the Add Shared Property button in the Code Editor toolbar.**

REAL Studio displays the declaration area for a shared property. If the Shared Properties item does not already exist, it is added to the Code Editor browser area. (The Add Shared Property button can be added to the Code Editor toolbar by choosing Customize from its contextual menu.)

- 2 **Declare the property and set its scope the normal way. If desired, set a default value.**

Figure 404 illustrates a shared property in the Code Editor.

Figure 404. A Shared property in the Code Editor.



Adding Constants

You can add constants to classes to store fixed values that the class or the application needs to use. A constant acts like a property but it holds a fixed value for its entire “life.” When you create a constant, you give it its value. You can read the constant’s value in your code, but you cannot use an assignment statement to change the value of a constant.

Constants that you add to classes have the choices for Scope described in “Scope of a Class’s Methods, Properties, and Constants” on page 543.



To add a constant to a class, do this:

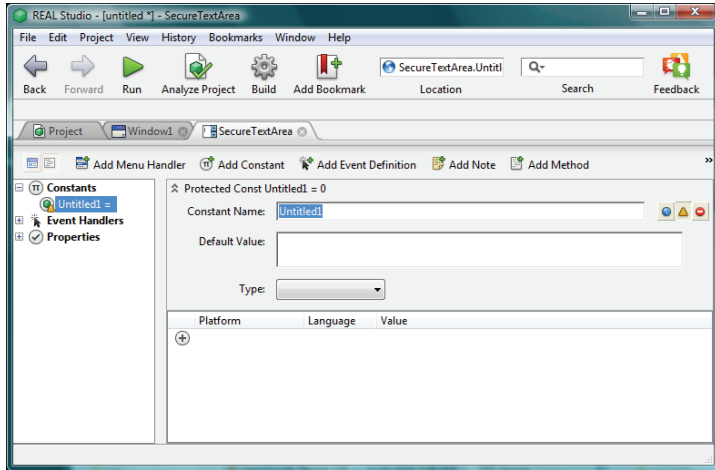
- 1 **If the Code Editor for the class is not already open, double-click the class's name in the Project Editor or click its tab.**

The Code Editor for the class appears.

- 2 **Click the Add Constant button or choose Project ► Add ► Constant.**

The Add Constant declaration area appears above the Code Editor area (Figure 405).

Figure 405. The Add Constant declaration area.



- 3 **Enter the name of the constant, its value and data type.**

When you enter a value, REAL Studio guesses the data type and sets the Type drop-down list accordingly. Any number sets the data type to Number, a string other than “True” or “False” sets it to String, and a hex value that starts with “&c” sets it to “Color.” Entering “True” or “False” sets the data type to Boolean.

If its guess is incorrect, set its data type by selecting a data type from the Type drop-down list, Number, String, Boolean, or Color. The data type of the constant will be indicated by the small icon to the left of the constant's name in the browser area.

If you chose Color, a color patch appears to the right of the Type drop-down list with the default color of black. Click it to display the Color Picker to choose the color constant. When you choose a color, its value in hexadecimal is added to the Default Value area.

If you chose string, a “Dynamic” checkbox appears to the right of the Type drop-down list. Dynamic constants are used to facilitate localization. For more information about Dynamic constants, see the section “Dynamic Constants” on page 379.

- 4 **Enter the value of the constant in the Default Value area.**
- 5 **Set the Scope of the constant by clicking one of the three Scope buttons.**

Your choices, from left to right, are Public, Protected, and Private.

Figure 406. The Scope buttons.

For information about Scope, see the section, “Scope of a Class’s Methods, Properties, and Constants” on page 543.

6 (Optional) Use the Localization table at the bottom of the dialog to define different values for the constant for different platforms and language combinations.

See the section “Using Constants to Localize your Application” on page 378 for details on the Localization table.

When you are finished, the browser area of the Code Editor shows the new constant’s name and value, with an icon that indicates its data type.

Adding Methods

You can add methods to classes to provide functionality that the class previously didn’t have. For example, in the Gridlock example, the DefineGrid method draws a row by column grid based on the number of rows and columns that is passed to it. The number of rows and columns are the *parameters* that are passed to the method when it is called. The parameters contain values that the method needs to do its job.

A method may also return a value. If it does, you must declare the data type of the return value when you declare the method.

For detailed information about the options available when you create methods, see the section, “Adding Methods to Windows” on page 329.

To add a method to a class, do this:

- 1 If the class’s Code Editor is not already open, double-click on it in the Project Editor to open it or click on its tab.**

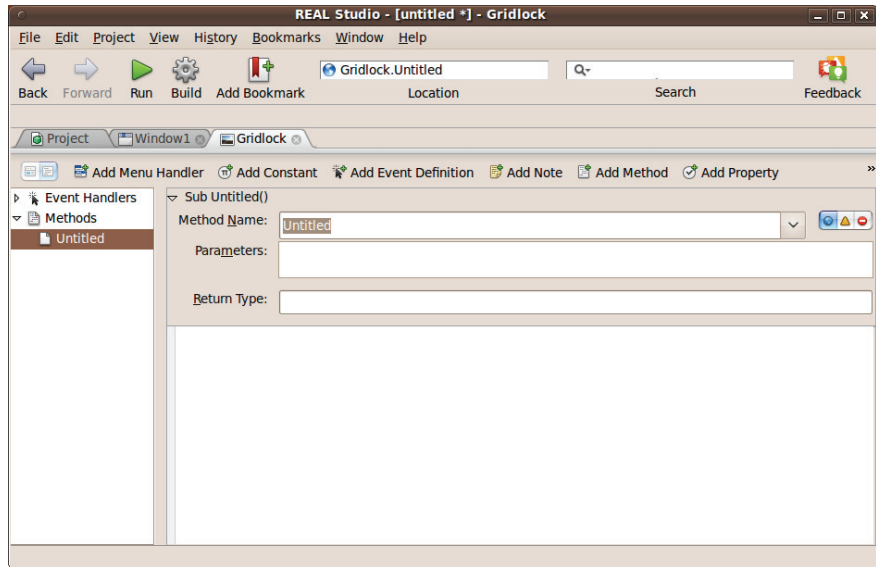
The Code Editor for the class appears.

- 2 Click the Add Method button or choose Project ► Add ► Method.**

The Method declaration area appears above the Code Editor area.



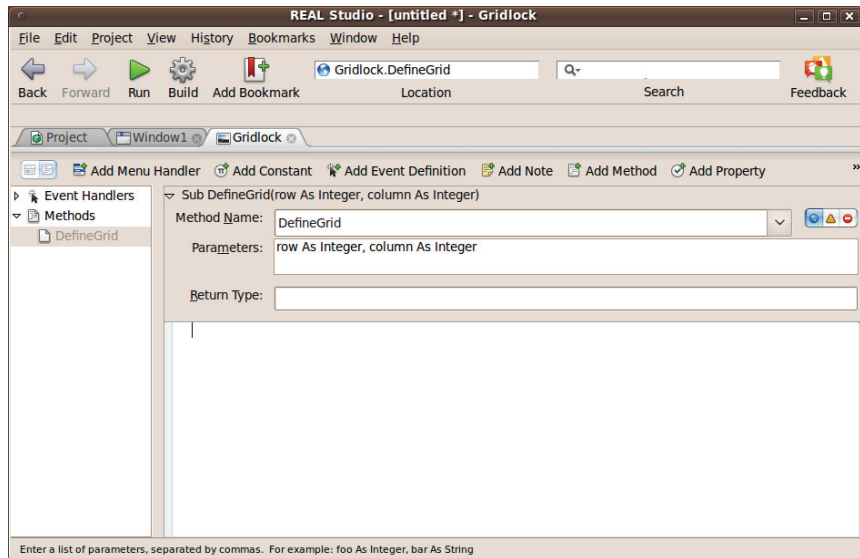
Figure 407. The Method declaration area.



- 3 Enter a name for the method.
- 4 Enter the parameters if any, separating multiple parameters with commas, as shown in Figure 408.

A method does not need to have parameters.

Figure 408. A completed Method Declaration.



If a parameter's data type is a class that was created in a module, you need to use the syntax *ModuleName.ClassName* to refer to the class. For more information on module classes, see the section “Adding Classes to Modules” on page 384.

There are several advanced features available when you declare the parameters of the method. For more information, see the sections “Passing a Parameter by Value or Reference” on page 336, “Setting Default Values for a Parameter” on page 338, “Setter Methods” on page 340, and “Constructors and Destructors” on page 341.

5 (Optional) If the method will be a function, enter the data type of the value to be returned.

If the value to be returned by the function is an array, place empty parentheses after the data type of the array. For example, to indicate that you will return an array of integers, enter “Integer ()” in the Return Type field.

6 Choose a Scope for the method or function by clicking a Scope icon.

Your choices, from left to right, are Public, Protected, and Private.

Figure 409. The Scope buttons.



For more information on Scope, see the section “Scope of a Class's Methods, Properties, and Constants” on page 543.

When you are finished, the new method is listed in the browser area in the Methods group. If the Scope of the method is Protected or Private a badge appears in the browser and the word “Protected” or “Private” is added to the Sub or Function statement in the declaration area.

Adding Shared Methods

Just as properties can be declared as either instance or shared properties, a method can be declared as either an instance method or a shared method. As is the case for shared properties, a shared method belongs to the class, not an instance of the class. A shared method can be called without instantiating an instance of the class and is also accessible from all instances of the class.

The Self keyword is not available in a shared method or shared computed property and you cannot access instance methods or instance properties inside a shared method unless you are doing so via an instance.

To create a shared method, do this:

1 Choose Project ► Add ► Shared Method or, if it is available, click the Add Shared Method button in the class's Code Editor toolbar.

(The Add Shared Method button can be added to the Code Editor toolbar by choosing Customize from its contextual menu.)

REAL Studio displays the declaration area for the shared method. If the Shared Method category does not already exist, it is added to the Code Editor browser area.

2 Declare the shared method and set its scope the normal way.



3 Write the shared method in the Code Editor.

Here is an example that illustrates the difference between instance and shared methods. Create a class, myClass, in the Project Window and create the following simple method of myClass:

```
Sub WelcomeMe(a As String)
    MsgBox a
```

If you create WelcomeMe as an instance method, you can only call it from an instance of myClass, e.g.

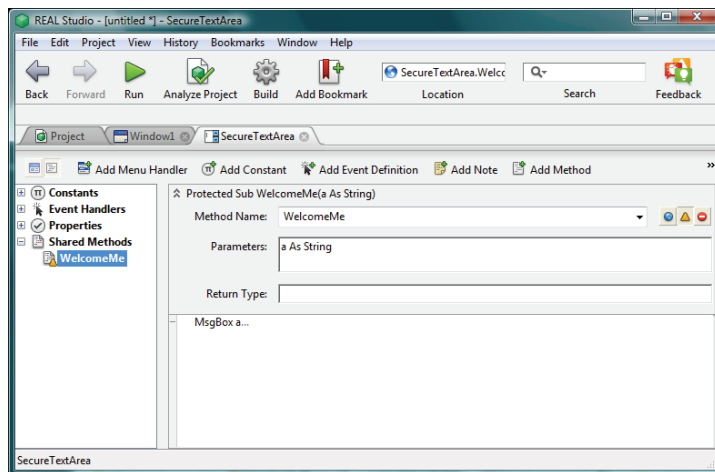
```
Dim greetMe as myClass
greetMe = New myClass
greetMe.WelcomeMe "Hello World"
```

If you create WelcomeMe as a shared method, you can call WelcomeMe with the line:

```
myClass.WelcomeMe "Hello World"
```

Figure 404 illustrates a this shared method in the Code Editor.

Figure 410. A Shared method in the Code Editor.



Adding Event Definitions

When you add code to an event handler of a class, you cannot, by default, add more code to that event handler for an instance of the class. Consider this example. You create a class based on the ListBox class and you put some code in its Open event handler. Any instances of that class that appear in a window will not have their own Open event handler. The assumption is that since the event handler of the class has code for the Open event, it is handling that event.

There may be times, however, when you want the class to have code in an event handler but you also want to be able to put code in that event handler for an instance of the class. You want the code in the class instance to override the class's event handler.

An example of this is when you set up default values. In the Open event handler, you might set the default values of the class. For example, in a class that displays the names of the months in a ListBox, you might want to select the current month name by default. However, when you use this class in a window, you might want to be able to override the default action and choose a different month instead. The instance of the month's ListBox won't have an Open event handler because its class is handling the Open event.

Adding new event definitions solves this problem. You add a new Open event definition to the class and then call it from the class's Open event handler, as if it were a method. New event definitions are available only to the instances of the class. When you add a new Open event definition to the class, you are adding that event to any instance of the class. When will this new Open event occur? Since you are calling it in the class's Open event handler, it will occur when the window opens — just like a regular Open event handler.

Here is a simple example. In the ListBox example that displays the Months (Figure 392 on page 536), modify the Open event handler of the Months class to this:

```
Me.HasHeading=True
Me.InitialValue="Months"

Me.AddRow "January"
Me.AddRow "February"
Me.AddRow "March"
Me.AddRow "April"
Me.AddRow "May"
Me.AddRow "June"
Me.AddRow "July"
Me.AddRow "August"
Me.AddRow "September"
Me.AddRow "October"
Me.AddRow "November"
Me.AddRow "December"

//call the event handler for the instance..
open
```

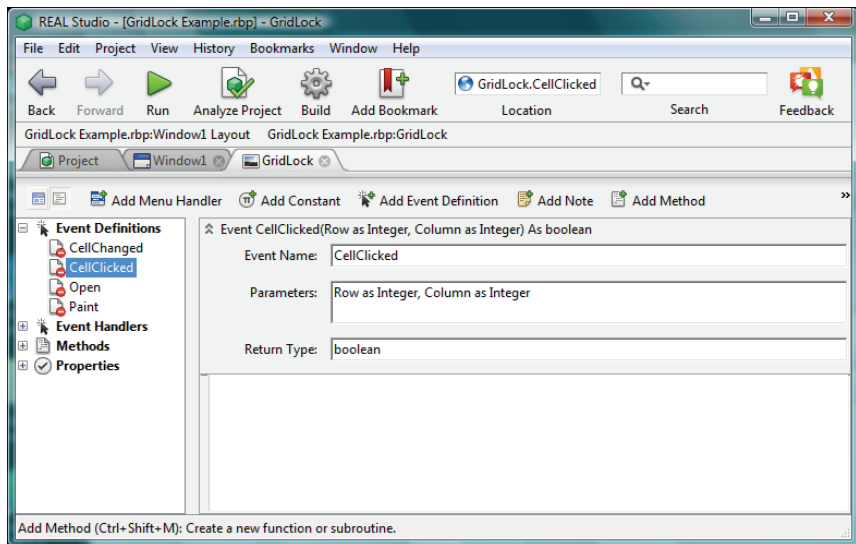
An instance of this class will have an event handler for all event definitions that have been added to the class. In this case, it will have an Open event. This Open event can set the highlighted month — or anything else, for that matter. The main point is that each instance of this class can do something different after the ListBox has

been populated with the list of months. You can see how this works by placing a breakpoint in the code at the point where the Open event handler of the instance is called and stepping into the code (For more information on breakpoints, see the section “The Debugger” on page 636).

Let’s look at another example in which you would want to add new events. Suppose you are creating a custom class that will display a grid. The grid allows the user to click on individual cells to turn them on and off. You might want to add an event that occurs when the user clicks on a cell in the grid. Let’s call this event “CellClicked.” You also want the event to be passed the row and column numbers where the click occurred. In any particular instance of the class, you could then use the CellClicked event as a place to take action when the user clicks in a cell.

So how do you go about adding the CellClicked event? First, add a new Event Definition called CellClicked to the class. You want to pass the row and column numbers to this event, so include them as parameters for the event. Figure 411 shows what the New Event declaration area might look like when you are adding the CellClicked event.

Figure 411. The Event Definition declaration area.



The next step is to determine when this event will occur. Since the user clicks the mouse to select a cell, it makes sense that this event is triggered when he clicks the mouse. In each instance of the GridLock class, the CellClicked event is available and is listed in the control’s events.

For the Canvas control (the class the grid class would be based on), this means calling this event in the MouseDown and MouseDrag event handlers of the class. To do this, call the CellClicked event as if it were a method. You do the necessary calculations to determine the row and column numbers and pass these to the CellClicked event.

When the user clicks on a cell, the `MouseDown` event handler of the class is executed. Since `CellClicked` would be called in this event, this causes the `CellClicked` event to be called and passed the row and column numbers. This causes the `CellClicked` event to occur for the instance of the class the user clicked on in the window. The class is basically calling a subroutine of the instance of the class. And, because the `CellClicked` event could be designed to return a value, the instance of the class can return data back to the super class. This could be beneficial in this particular example if you wanted to filter the click. You could code the class to only continue with handling the click should the `CellClicked` event return `False` (use `False` since this is the default value returned by a function). This would allow any instance of the class to determine which cells are valid for clicking and which cells are not.

In the `GridLock` project, `CellClicked` is set up to prohibit clicking in cell 1,1. This is done in the event handler for the instance of the `GridLock` class in a window:

```
If row=1 and column=1 then //don't allow selection
  Beep
  Return True
end if
```

When `CellClicked` is called in the `MouseDown` event of the `GridLock` class, it changes the color of the cell only if `CellClicked` returns `False`. See the “Gridlock” project in the `Graphics` folder in your `Examples` folder for an example of this kind of event definition. `CellClicked` is called as a method in the `TrackClick` method in the `Gridlock` class.

The `REAL` Studio language also has a command that you can use to unambiguously call an event when the event has the same name as a method. Use the keyword “`RaiseEvent`”, followed by the name of the event and any parameters it requires.

Adding Structures

A *Structure* is a compound value type. It consists of a series of fields that are grouped together as a single block. You can control the size and order of the fields so you can declare a structure in `REAL` Studio to match a structure defined by an external library or as part of a binary file format or communications protocol. A structure can provide a convenient alternative to the `MemoryBlock`.

You might also use a `Structure` when porting a Visual Basic application to `REAL` Studio; it is very similar in concept and syntax to Visual Basic’s “User-Defined Type” feature, also known as a “UDT”. In Visual Basic .NET this is called a *structure*.

A `Structure` is a data type, like an integer or a color. It is not a reference type like an object or an array. When you assign an object value to an object variable, you copy a reference to the object data; when you assign a structure value to a structure variable, you copy the entire contents of the structure. When you pass a structure as a parameter `ByVal`, the whole contents of the structure is copied; when you pass a structure `ByRef`, the callee ends up modifying the caller’s original structure instead.

The New operator does not apply to structures. When you create an array of structures, each element is an actual value (not a reference, like it would be with an array of objects). You can use the same dot syntax to access structure fields as you would use to access object properties, but when you use dot syntax with a structure, you are manipulating the structure variable itself, not a reference to data somewhere else.

Creating a Structure

Structures can be created in classes or in modules. A Structure in a class can be given Public, Protected, or Private scope. A structure contains a list of fields and/or arrays. You must declare the data type of each field or array.

Structure fields can be defined as arrays, using the usual array syntax:

fieldName(UBound) As *DataType*

Arrays in structure fields can't be manipulated in the same ways as normal arrays; they represent a fixed chunk of storage inside the structure, not a dynamic object that can be resized and manipulated. Structure field arrays cannot be resized, cannot be assigned, and do not support any of the array methods.

Strings also have a special syntax and behavior inside a structure:

fieldName As String * size

A string in a structure is a simple array of bytes. Unlike String variables, a string field has a fixed size and does not store text encoding information. If a string value contains fewer bytes than the declared size, unassigned bytes are assigned null bytes. If you use the **Len** function to get the length of the field, it will return the declared length.

Just as you convert text to a specific encoding when writing it to a file or a socket, and assign the correct encoding to it when reading it back in, you must convert strings to a specific encoding when you assign them to a structure and define them to the correct encoding when reading them back out.

Structure fields can contain any of the simple value types, but cannot contain objects. The Structure Editor in the IDE will show you the size and location of each field, so that you can match your structure up exactly with an external data format.

To create a Structure, do this:



- 1 Open the Code Editor for a class and choose Project ► Add ► Structure.**

If you have added the Add Structure button to the Code Editor toolbar, you can click that button instead. A structure has a name and a list of fields, and can be global or private.

- 2 In the structure declaration area, give the structure a name and assign its Scope.**

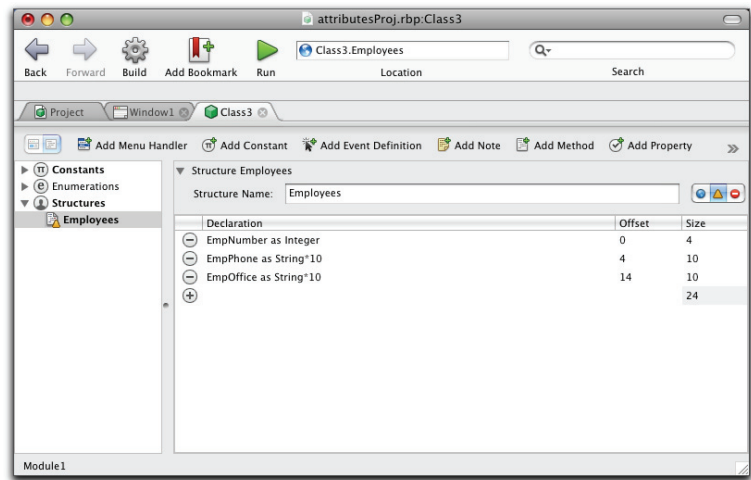
- 3 Click the plus button in the field list to create a new field, then type in the declaration.**

The field declaration syntax is the same as for a Dim statement or a property declaration: *fieldName As DataType*.

Like a property or local variable, fieldnames must be simple identifiers and must be unique within the field.

A completed structure definition is shown in Figure 412 on page 563.

Figure 412. A Structure declaration.



Using Structures

Once you’ve defined a structure, you can use it in almost any context in which you would use any other data type. Use the dot syntax to access the fields. You can define an object or module property as a structure; you can declare a method parameter as a structure; you can even embed one structure as a field in another. Variants, however, cannot contain structures. Store the `StringValue` instead.

In addition to the fields you define, structures contain three built-in items:

Name	Parameters	Description
Size		This constant returns the total size of the structure in bytes.
StringValue	littleEndian as Boolean	Gets the StringValue of the structure. You must pass the desired endianness, which should match the LittleEndian property on the MemoryBlock on BinaryStream. StringValue will convert the structure’s fields to or from the appropriate endianness as necessary.
StringValue	littleEndian as Boolean	Sets the StringValue of the structure. You must pass the desired endianness, which should match the LittleEndian property on the MemoryBlock on BinaryStream. StringValue will convert the structure’s fields to or from the appropriate endianness as necessary.

The `StringValue` getter and setter methods let you treat the structure as a string. This is useful for copying structures into and out of `MemoryBlocks`, for reading and writing structures to files, and for transmitting structures through sockets.

To work with the structure, you can declare a variable or property as a structure and get and set the fields that you declared.

To work with the structure, you can declare a variable or property as a structure and get and set the fields that you declared. For example, suppose the class “myClass” has the structure “Employee,” you can declare an instance and assign values like this:

```
Dim Employee1 as myClass.Employee
Employee1.EmpNumber=5
Employee1.EmpOffice="Tyler Hall"
Employee1.EmpPhone="555-1212"
```

Then you can get any of the values in the structure, i.e.,

```
MsgBox Employee1.EmpOffice
```

Structure Alignment

Structure alignment refers to aligning the data at a memory offset equal to some multiple of the word size. Alignment can increase the computer’s performance.

Structures can be aligned via the `Attributes` system. You add the attribute “`StructureAlignment`” and use one of the legal values: 1, 2, 4, 8, 16, 32, 64, and 128.

To specify a structure alignment, right-click the structure name in the class’s `Code Editor` and choose `Attributes...` from the contextual menu. The `Attributes` list appears. Add an attribute to the list and specify “`StructureAlignment`” in the `Name` field and enter the desired value.

Adding Enumerations

An *enum* or enumeration is a set of constants. It’s a group of constants that are assigned values. You can assign a value to each constant or accept the default values. By default, the constants are numbered consecutively, starting with zero.

When you create an enumeration, you create a new data type. Enumerations accept only integer constants. When you want to get an enumeration, you need to cast it to an integer data type.

An Enum can be created in a class or in a module

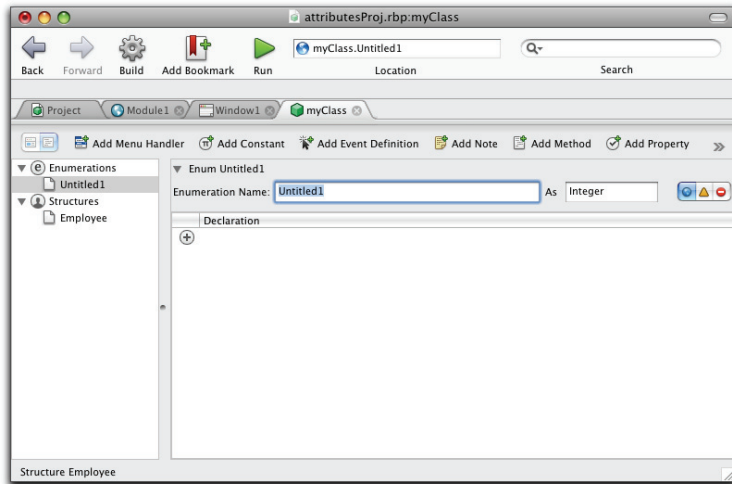
To create an Enum in a class, do this:

1 Open a class and choose **Project ► Add ► Enum**.

If you have added the `Add Enum` button to the `Code Editor` toolbar, you can click that button instead. An Enum has a declaration area in which you name the Enum and set its `Scope`.



Figure 413. The Enum Declaration area.



- 2 In the declaration area, give the Enum a name and click one of the Scope buttons to set its scope.
- 3 Click the plus button in the list to enter the name of the first constant.
- 4 If desired, assign a value to the constant by typing an equals sign, followed by the value.

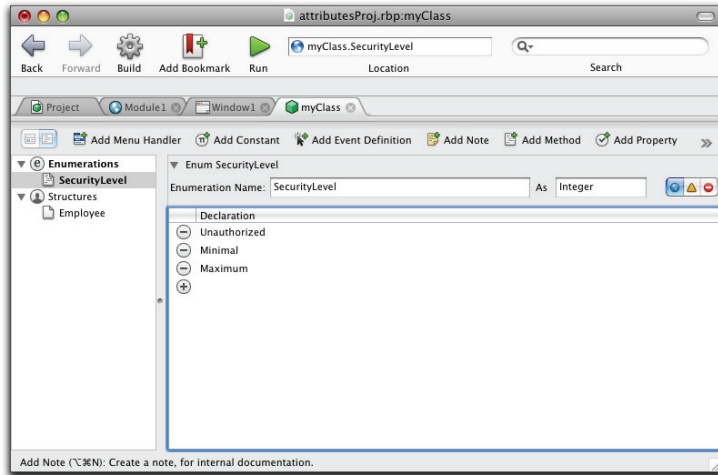
For example, if the first constant is named “Windows” and you want its value to be 13, you’d write:

```
Windows = 13
```

If you only enter the constant name, its value is its sequence number in the list, with the first item being zero.

Here is an example of a finished Enum. It defines the Enum named “SecurityLevel” and gives it four constants: Unauthorized, Minimal, Maximum, and Forced. Their values range from 0 to 3 since no values are included in the definition.

Figure 414. A global Enum defined for 'Security Level'.



You use the dot notation to get the values of the items. For example, the expression

```
myClass.SecurityLevel.Maximum
```

accesses the value of 2 because Maximum is the third constant in the Enum definition. To return the integer 2, you need to explicitly cast the enum using the desired integer data type. There is no implicit conversion from the enum data type to an integer data type. For example, the following code in a window returns the integer value 2 associated with this item.

```
Dim i as Integer  
i=Int32(myClass.SecurityLevel.Maximum)
```

You can also declare a variable of the data type of the enum and get its values that way:

```
Dim MaxSec as myClass.SecurityLevel  
Dim i as integer  
MaxSec=myClass.SecurityLevel.Maximum  
i=int32(MaxSec)
```

Adding Delegates

A Delegate data type is an object representing a specific method. Delegates decouple interface from implementation in a similar way to events or interfaces. This decoupling allows you to treat a method implementation as a variable, that is changeable based on run-time conditions. They represent methods that are callable without knowledge of the target object. You can change the function the delegate points to on the fly.

In effect, a delegate is a class with a single method, named "Invoke," whose parameters and return value match the delegate's parameters and return type. The

Invoke method calls the method the delegate instance represents. While delegates are objects, you cannot create a subclass of a delegate type.

Delegates can be created in classes and modules. You use the Project ► Add ► Delegate menu command or the optional Add Delegate button in the class's Code Editor toolbar to create a Delegate.



To create a class delegate, do this:

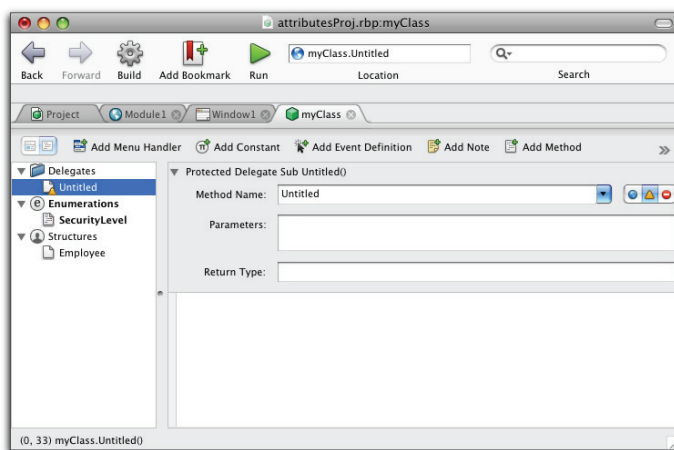
1 Open the class to which you want to add the delegate.

Its Code Editor appears.

2 Choose Project ► Add ► Delegate.

REAL Studio adds a Delegates folder in the class's browser area and creates a new, untitled delegate.

Figure 415. A new delegate.



3 Declare the delegate by specifying its parameters and, optionally, its return type.

This declaration creates a new object type: in effect, a class with a single method, named “Invoke.” Its parameters and return value match the delegate’s parameters and return type. The Invoke method calls the method the delegate instance represents. Although delegates are objects, you cannot create a subclass of a delegate type.

Delegate values come from the AddressOf operator. The AddressOf operator returns a delegate representing the target method. Invoking the delegate invokes the method on the same object instance the delegate originally came from.

Delegate types are considered to be compatible if their parameter lists and return types match. Casting, assignment, and the IsA operator work by comparing the delegate type signatures, not by comparing actual types as with classes.

The delegate type has an implicit conversion to Ptr, so you can continue to use the AddressOf function to obtain function pointers for use as external callbacks. In

addition, you can create a new instance of a delegate using the New operator; its constructor expects a Ptr to an external function which the delegate will represent.

Virtual Methods

Virtual methods provide a way for a subclass to have its own version of a method that its super class has. Ordinarily, a subclass inherits the methods belonging to its parent. Virtual methods also deals with the concept of a subclass having a different version of a behavior that is defined in the super class.

When a subclass has a method that has the same name as its parent, the subclass's version is called unless you use the syntax:

```
parentclassname.methodname
```



To create a virtual method, do this:

- 1 Create a class.**
- 2 Add a method to the class.**
- 3 Create a subclass of the first class.**
- 4 Add a method to the subclass with the same name as the method you added in step 2.**

When the subclass calls the method, it will call its own version and not its parent class's version.

Extending Classes

You can also add methods to existing REAL Studio classes. You do this by adding a *class extension method* to a module. Once added, the new method can be called as if it were built into REAL Studio. Class extension methods are added only to modules, but they are called from classes.

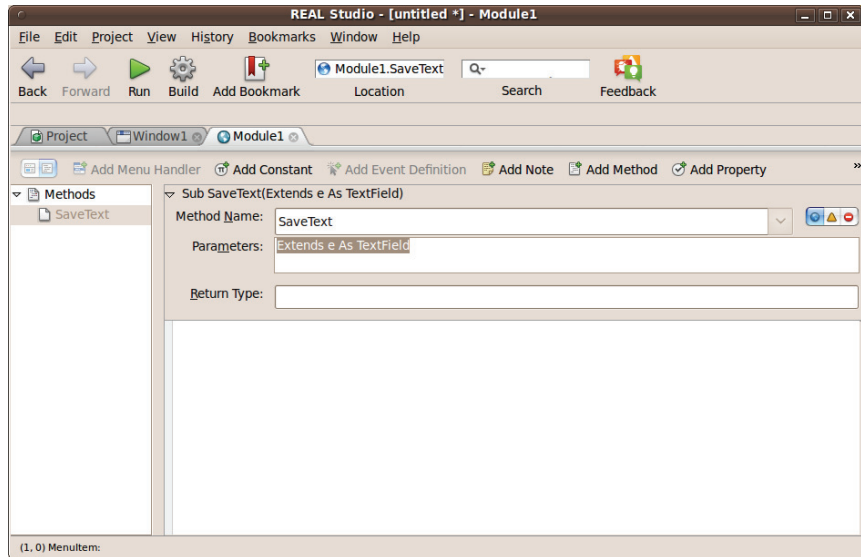
To define a method as a class extension method, use the “Extends” keyword prior to the first parameter in the parameter declaration. The data type of the first parameter is the object type from which the method will be called. In other words, the use of the “Extends” keyword indicates that the parameter is to be used on the left side of the dot (“.”) operator in a calling statement. When you use the Extends keyword, you do not have to pass an object of that type to the method. You can add “normal” parameters to the parameter declaration that follow the Extends parameter.

Writing a Class Extension Method

For example, you can add a method that can be called from a TextArea object. Suppose you want a method that saves the text in a TextArea to a text file. You name it “SaveText”. Create a module in the project if it doesn't already have one and add the SaveText method to the module.

Since the SaveText method doesn't take any "normal" parameters, the parameter declaration uses only the parameter that takes the Extends keyword. This is shown in Figure 416.

Figure 416. Declaring a class extension method of the TextField class.



The Parameters area shows the declaration:

Extends e As TextField

The Extends keyword indicates that the method can be called from any TextField object that is in the project. Note also that the Scope for the class extension method is Global.

Calling a Class Extension Method

You can call this method from anywhere in the project. You can simply use the line:

TextField1.SaveText

This assumes that there's a TextField in the window named TextField1.

To use the class extension method in another project, all you need to do is copy the module that contains the definition of the class extension method to the other project.

Constructors and Destructors

When you create a new object, you will sometimes want to perform some sort of initialization on the object. The *constructor* is a mechanism for doing this. An object's

constructor is the method that will be executed automatically when an instance of the class is created.

Constructors

You add a constructor to a class by adding a method to the class called “Constructor”. It will be called automatically when an instance is created via the New operator. If your constructor accepts parameters, you must pass the required number of parameters and they must be of the correct data type.

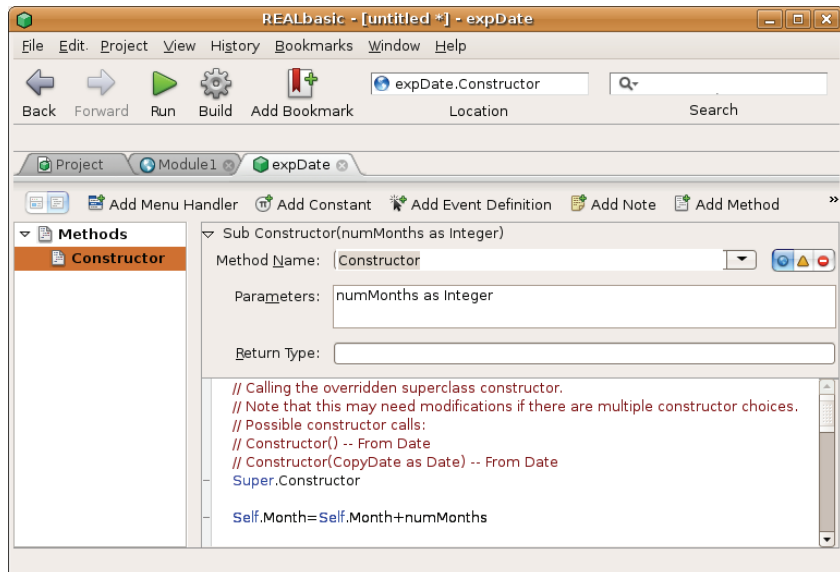
Whenever you create a constructor for a subclass, REAL Studio automatically adds a call to the super class’s constructor for you in the Code Editor and adds comments that explains what it is doing. This is shown in the example in Figure 417 on page 570. This is because the constructor that you are writing overrides its Super class’s constructor, but the new object may not be initialized correctly unless its Super class’s constructor executes. This is how you call a method of a Super class that is being overridden by a method of the subclass.

Here is a simple example of a constructor. Suppose you are managing a service that sells monthly subscriptions. You want a custom version of the Date class the automatically takes the value of the expiration date when an instance is instantiated.

First, add a new class to the project and set its Super class to Date and rename it “ExpDate”. Double-click it to open its Code Editor and create a new method called Constructor. The constructor takes one integer parameter, NumMonths, the number of months the customer has signed up for. The constructor has only one line of code:

```
self.Month=self.month+NumMonths
```

Figure 417. The constructor for the expiration date class.



When an instance of a ‘regular’ Date object is created, it is initialized to the current date, so this line increments the current month number by the number of months that is passed in as a parameter.

When you need to get the expiration date for a customer who signs up for 6 months, you can get it like this:

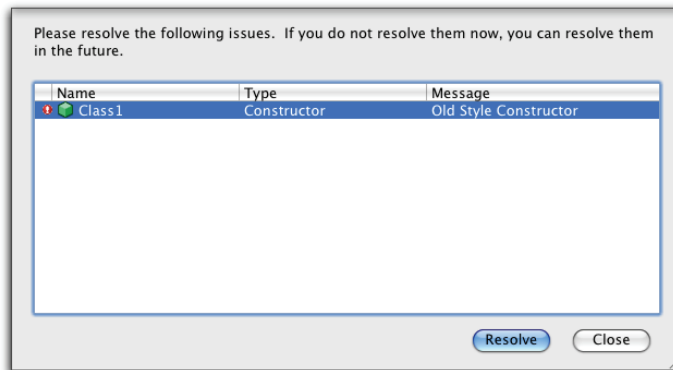
```
Dim ex as New ExpDate(6)
TextField1.text=ex.ShortDate
```

The number months is passed in as a parameter and the new instance holds the expiration date instead of the current date.

Old Syntax

In early versions of REAL Studio, you could also create a constructor by using the name of the class as the name of the constructor (instead of “Constructor”). If you open an old project that contains such a constructor, REAL Studio lets you rename the old-style constructor just before it displays the IDE window. If the old project has such an item, you will see a dialog box such as this.

Figure 418. A Rename old-style Constructor dialog box.



Highlight each old-style constructor and click Resolve. REAL Studio will rename the item using the supported syntax.

Initializing an instance of a control class

If you need to initialize an instance of a control class, don’t use a constructor. Instead, put the code that does the initialization in the Open event of the control. The section “Examples of Subclasses” on page 533 has an example in which a custom ListBox is initialized to display the months of the year and select the current month. This is the functionality of a constructor, but the code instead goes in the control’s Open event.

Destructors

You can also create a destructor. The destructor is called automatically when an instance of the parent class is deleted or goes out of scope — for example, when the

user closes the window. Name the new method “Destructor”. Destructors take no parameters and do not return a value.

Destructors are called when the last reference to an object is removed, even if execution is in a destructor for another object. Note that this means you can cause a stack overflow if your destructor triggers other destructors in a deep recursion. However, such overflow will not happen as long as properties of the object are being cleaned up automatically. So, it is generally preferable to *not* set properties to Nil in your destructor, but instead let REAL Studio clean them up for you.

Overloading

REAL Studio supports what is known as *overloading* of methods. A language that supports overloading allows you to have two or more methods with the same name but have a different number of parameters or parameters with different data types. When that method name is called, REAL Studio will figure out which method you ‘mean’ to call from its parameters.

A good example of a built-in overloaded function is the ‘+’ operator. If its arguments are numbers, it computes the sum; if the arguments are strings, it concatenates the strings.

Overloading Custom Classes

Using the language, you can overload these operators so that you can perform arithmetic and comparison operations on your custom classes. For example, if you want to add two objects that store lists, you need to write a function that defines the “+” operator for that custom class. Your function overloads the built-in “+” operator. REAL Studio has a set of built in keywords that you use for this purpose.

The Operator_ keyword represents a set of reserved words that enable you to implement the standard arithmetic and comparison operators in your custom classes. The supported operators and keywords are shown below:

Operator	Keyword
+	Operator_Add Operator_AddRight
-	Operator_Subtract Operator_SubtractRight
*	Operator_Multiply Operator_MultiplyRight
/	Operator_Divide Operator_DivideRight
\	Operator_IntegerDivide Operator_IntegerDivideRight
Mod	Operator_Modulo Operator_ModuloRight
And	Operator_And Operator_AndRight

Operator	Keyword
Not	Operator_Not
Or	Operator_Or Operator_OrRight
=, <, >, <=, >=	Operator_Compare
(lookup)	Operator_Lookup
(negation)	Operator_Negate
(convert)	Operator_Convert

For example, to add the ability to add two instances of a custom class, you define a function called “Operator_Add” as a method in the custom class. This function defines the addition process for that class. After it is defined, you can simply use the + operator to add two instances of that class. The + operator will automatically call your custom Operator_Add method

For detailed information about operator overloading for custom classes, see the section on each Operator_ keyword in the *Language Reference*.

Assigning a Value to a Method

When you call one of your methods, you can optionally assign a value to it using the syntax that you normally use to assign a value to a property. That is, you can write:

```
objectname.methodname = value
```

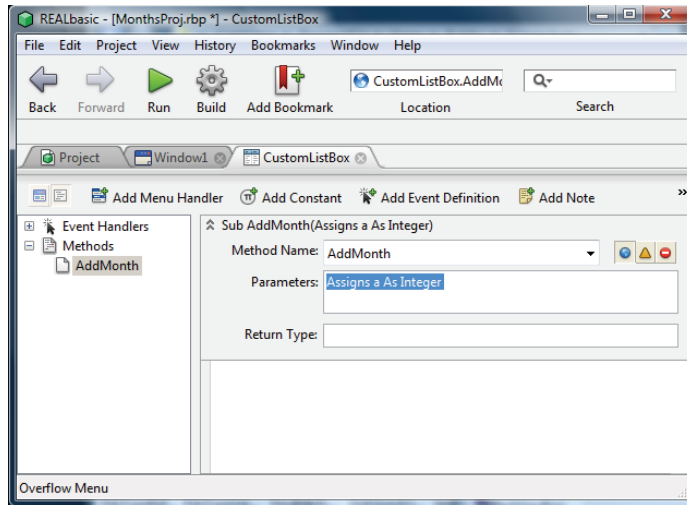
When you do so, the value becomes the value of the last parameter that the method expects. It must be of the same data type as that parameter. If the method expects more than one parameter, you must pass values of all but the last parameter in the normal way. When you want to use this syntax, you must use the “Assigns” keyword when you declare the method. Here is an example.

Suppose you create a custom class based on `ListBox` called `CustomListBox`. You can add a method to this class that adds a row to the `ListBox` based on the number passed to it. The variable, `a`, is declared as an `Integer` parameter, as shown in Figure 419.

```
Select case a
  case 1
    addrow "January"
  case 2
    addrow "February"
  case 3
    addrow "March"
  case 4
    addrow "April"
  case 5
    addrow "May"
  case 6
    addrow "June"
  case 7
    addrow "July"
  case 8
    addrow "August"
  case 9
    addrow "September"
  case 10
    addrow "October"
  case 11
    addrow "November"
  case 12
    addrow "December"
End Select
```

When you declare the method, use the “Assigns” keyword in the following way.

Figure 419. Using the “Assigns” keyword in a method declaration.



When you use the Assigns keyword, you pass the integer value to the AddMonth method using the Assignment operator:

```
CustomListBox.AddMonth = 2
```

If the method takes more than one parameter, the value you assign is used as the value of the last parameter. Include all the parameters in the call except the last one. For example, suppose you write another method of the CustomListBox class called “ChangeCell” that takes *row* and *value* as parameters. *Row* is the cell in the ListBox you want to change and *value* is the new value.

```
Sub ChangeCell(row as Integer, Assigns value as String)
    Cell(row,0)=value
```

If you want to change the value of the third element in the ListBox to “New York”, the “normal” way to call ChangeCell is:

```
CustomListBox.ChangeCell(2, "New York")
```

If you use the “Assigns” keyword, you can also call it using *row* as the parameter and assign the new value using an equals sign:

```
CustomListBox.ChangeCell(2) = "New York"
```

Using Arrays of Classes

As is the case with any object, you can create an array of instances of a class. For example, you can create an array of controls as part of your interface to make it easy to manage the controls. You simply give the controls the same name and

distinguish among them using its Index property. Control arrays are described in the section “Sharing Code Among An Array of Controls” on page 348.

You can also create an array of instances of non-control classes. When you do so, you can take advantage of the object hierarchy or class interfaces. If myClass1 is subclassed from mySuperClass1 or a class interface implemented by myClass1, then an array of myClass1 can be used whenever an array of mySuperClass1 is expected. The array maintains its original element type and each insert, append, and assignment statement to an array element is checked to make sure that the new value matches the element type. If the type does not match, a `TypeMismatchException` occurs. You can manage it via a `Catch` statement in a `Try` block or in an `Exception` block.

Casting

An extremely powerful way of creating generic, reusable code is to take advantage of the object hierarchy, using *casting*. The idea is best illustrated by an example.

Since the objects you create are subclasses of base classes in the REAL Studio language, you can always test to see whether an object is a member of a specific subclass. The `IsA` operator in the language does this.

With the `IsA` operator, you test whether an object is of a specific subclass and, if it is, cast it as that type to do something specific with it. You cast an object by using the classname as a function that operates on the instance.

When you cast an instance, REAL Studio does not do error-checking for you to guarantee that you are casting to a legal object type. The instance that you are casting has to be of the type that you specify. Casting just tells REAL Studio to treat the object as an instance of the class to which it is cast. It doesn't convert it from one class to another.

Here is an example that uses a `For` loop to cycle through all the controls in a window to test whether each control is a `TextField`. If it is, it casts the control, gets its name and the values of its `Text` and `DataField` properties, and assigns the contents of the `TextField` to the field in a database table named *fieldname*.

```
Dim r as DatabaseRecord
Dim fieldname, fieldContents as String
.
.
For i = 0 to Self.ControlCount-1 //number of controls in window
  If Self.control(i) IsA TextField then
    fieldname = TextField(control(i)).DataField //cast it
    //the text property assigned to the contents of that field
    fieldContents = TextField(control(i)).text //cast the control as a TextField
    r.column(fieldname)= fieldContents
  End if
Next
```

As you can see, the code does not refer to any specific windows, TextFields, or even databases. Therefore, you can write this routine once and use it in any window in any project that includes a database.

Managing Menus within Classes

Classes that can receive the focus can control the menus when they have the focus. This makes it even easier to encapsulate code within a control. Classes that you create based on classes that can have the focus will have an `EnableMenuItems` event handler and can have menu handlers for any of the menu items in your project.

When an instance of a class has the focus and the user clicks in the menubar (or presses a keyboard shortcut for a menu item), the class's `EnableMenuItems` event handler is executed. This gives the class the opportunity to enable or disable any menu items. The window's `EnableMenuItems` event handler will be executed next, followed by the application class's `EnableMenuItems` event handler. If a menu item is then selected, REAL Studio first checks the class to see if it has a menu handler for the selected menu item. If the menu handler exists, it is executed, followed by the window's menu handler (if it has one for the selected menu item), followed by the application's menu handler (if it has one for the selected menu item).

The `SecureTextArea` class mentioned in the section "Examples of Subclasses" on page 533 is an example of a class controlling menu items. When the `SecureTextArea` has the focus and the user clicks in the menu bar, the `SecureTextArea`'s `EnableMenuItems` event handler sets the `Enabled` property of the Cut and Copy menu items to `False`, disabling them. These menu items would normally be enabled automatically by REAL Studio.

Another example of a class that manages menus is a class based on the `ListBox` that allows the user to use the Cut and Copy menu items to move menus between `ListBoxes`. See the `ClipListBox` project in the `ListBox` folder in your Examples folder for an example.

Using Classes in Your Projects

Before you can use a class in your project, you must first understand a few concepts and terms. The use of a class in a project involves three items: the class, the instance, and the reference.

The Class

The class is a template set of events, methods, and properties from which you create subclasses and instances.

The Instance

An instance is a place in memory that stores a copy of the properties of the class. Methods are not stored in memory with each instance. Instead, they are loaded from the class into memory when they are called.

The Reference The reference is a value stored in a property or local variable that keeps track of where the instance is in memory. You use the property or local variable holding the reference to access the instance of the class. In the following example, “person” is a local variable storing a reference to the instance of the custom class “Programmer.” The reference is then used to access the value in the name property of the instance created using the New operator.

```
Dim person as Programmer
person=New Programmer
person.name="Jason"
```

You will learn more about using the New operator later in this chapter.

How you use a class in your project depends on whether the class is based on a control.

Subclasses Based on Controls

The easiest way to add a class based on a control to a project is to drag the control from the Controls list to a Window Editor.

A new class, called *ControlNameX*, where X is a sequential number, will be added to the window and the Properties pane will show that it is subclassed from the control’s class. Use the Properties pane to rename the new class.

You can also create a subclass of a control class by clicking the Add Class button from the Project Editor or by choosing Project ► Add ► Class to add a new class to the project and then use the Properties pane to set the Super Class of the new class to a control class.

Since you’ve subclassed the class from a control, you can then add new properties and methods to the class — to create your own custom control — and then add an instance of the custom control to a window. See the section “Creating Custom Interface Controls with Classes” on page 583 for an example.

To add an instance of a class based on a control to a window, use the Controls drop-down list in the Window Editor to choose Project Controls and then drag the class from the Controls list to the window in which you want the new instance—just as if you were to create an instance of a built-in control by dragging its icon from the Controls list.

Classes Based on Classes Other Than Controls

Classes don’t have to be based on controls. You can also create classes based on classes that are not based on the Control class. For example, the Thread class is not based on the Control class. You might need to create a subclass of the Thread class and add properties to it to store information used or created by the thread. You might even need to create classes that have no super class. For example, you could create a class called “People” that had properties like Name, Age, and Height to store information about people. You could then create a subclass of people called “ComputerUsers” which would add additional properties that define a computer user.

To create an instance of a class based on a class other than one of the control classes, you must first have a place to store the instance. You can store the instance in a property or a local variable. The property or local variable must be of the same type as the class or one of the class's super classes. For example, if you need an instance of the Date class to store an individual's birth date, you create a variable of type Date to store a reference to the instance.

You use the Dim statement to create the variable—just as you would create a variable that stores a REAL Studio data type. Use the New operator to create a new instance of the class in memory and then assign a reference to the new instance to the property or local variable you have typed.

```
Dim birthDate as Date
birthDate=New Date
```

In this example, the local variable “birthDate” is typed as class Date. The New operator is then used to create a new instance of Date and assign a reference to this variable.

You can use a simpler syntax. You can place the New operator in the Dim statement as a modifier. When you do so, the statement creates the local variable and the reference in one step:

```
Dim birthDate As New Date
```

Accessing the Properties and Methods of a Class

Once you have created an instance of a class and stored a reference to it in a local variable or property, you can access its properties and methods the same way you access any object's properties and methods. For example, in the code below, the local variable birthDate now has access to the properties of the Date class. You can use them to assign values to properties.

```
Dim birthDate as New Date
birthDate.Year=1967
birthDate.Month=3
birthDate.Day=21
```

In this simple example, the instance is based on one of REAL Studio's built-in classes. If it was based on one of your own custom classes, which in turn was based on a built-in class, the instance would have access to the properties and methods of both its own super class and the built-in class that its super class was based on.

When are Instances of Classes Removed From Memory?

REAL Studio manages memory for you automatically using something called *reference counting*. REAL Studio maintains a counter for each instance that you create and records references to the object. Each reference increments the counter and removing a reference to the object decrements the counter. You remove a reference, for example, by setting it to Nil or by closing the window in which the instance is located. When the reference counter reaches zero, the object is removed from memory immediately.

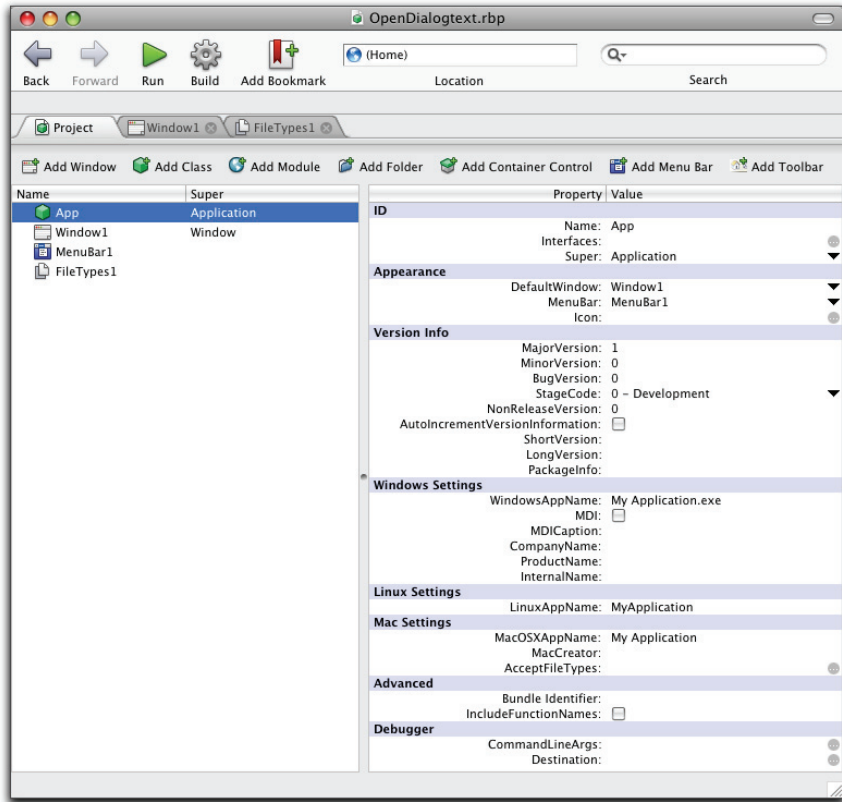
This means that instances of classes are removed from memory automatically when they are no longer used. Suppose you create a class based on a ListBox. You then create an instance of that class in a window. When the window is opened, the instance of the class is created in memory automatically. When the window is closed, the instance of the class is automatically removed from memory. If you store the reference to a class in a local variable, when the method or event handler is finished executing, the instance of the class is removed from memory. If you store a reference to an instance of a class in a property, the instance will be removed from memory when the object owning the property is removed from memory.

The Application Class

The Application class is used to create a subclass that represents your application rather than a window or a control. When you create a new project, a subclass based on the Application class is added to your project automatically. This is the “App” class that is listed in the Project Editor. Its Properties pane shows that it is derived from the Application class and the default menubar is assigned to the application as a whole.

If you wish, you can create more than one subclass based on the Application class. However, there is only one such subclass that plays the role of the “blessed” App class. Also, only one class in a project can be named “App”. The “blessed” App class is the one that has the built application’s properties in the Properties pane of the Project editor. Only one such class can exist in a project.

Figure 420. The default App class.



Special Event Handlers

The Application class has special event handlers. They are:

- **Open**: Executes when you run the application by clicking the Run button in the Toolbar or by choosing Project ► Run (Ctrl+R or ⌘-R) or when launching a standalone version of your application.
- **Close**: Executes when you quit your application either from the Debugging environment or in a stand-alone application.
- **NewDocument**: Executes when the stand-alone version of the application is launched without double-clicking one of the application's documents.
- **OpenDocument**: Executes when one of the application's documents is double-clicked from the desktop.
- **EnableMenuItems**: Executes when the user clicks in the menu bar but before any menu items are displayed. The EnableMenuItems event handler executes after the EnableMenuItems event handler of any classes with instances in the frontmost window and after the window's EnableMenuItems event handler. This is the event

handler that should be used to enable menu items that should be enabled regardless of whether there is a window open or not (This possibility exists on Macintosh applications). Note that if a menu item should always be enabled, you should use its `AutoEnable` property instead of an `EnableMenuItems` event handler.

- **HandleAppleEvent:** Executes when an `AppleEvent` is received by the application.
- **Activate:** The application is being activated. This occurs when the application is opening and when it is being brought to the front.
- **Deactivate:** The application is being deactivated. This occurs when another application or a desktop window is being brought to the front or when the application quits.
- **Unhandled Exception:** Executes when a runtime error occurs that is not handled by an `Exception Block`. This event gives you a “last chance” to catch runtime errors before they cause your application to quit. For more information on runtime errors, see the section “Runtime Exception Errors” on page 652.

Scope of the App Class’s Properties

When you add properties, methods, and constants to the `App` class, you declare the Scope of the new item. Your choices are `Public`, `Protected`, or `Private`. For descriptions of these choices, see the section “Scope of a Class’s Methods, Properties, and Constants” on page 543.

`Public` properties of the `Application` class are accessible to all code in your project. That is, if you want to access the property anywhere in your application, set its Scope to `Public` when you declare the property.

Use the `App` function to access the `App` class’s `Public` properties outside the `App` class. For example, to access the a property called “`Separator`” outside the `App` class, use the syntax:

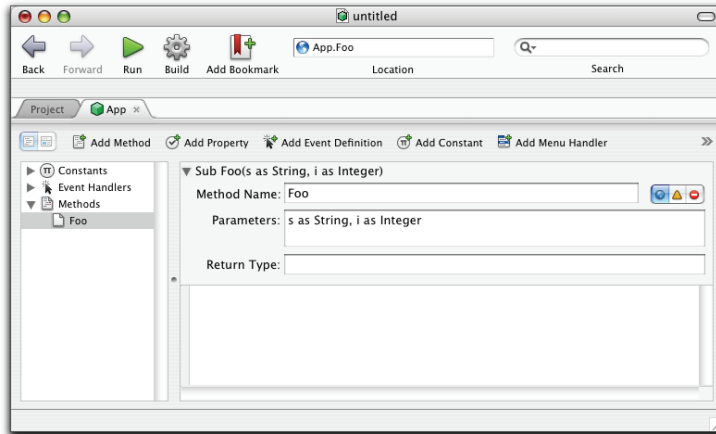
```
App.Separator
```

To access an `App` class property in the `Properties` pane of a window or class outside the `App` class, precede the reference by the number sign, “`#`”. In this example, you would enter “`#App.Separator`” to access this property in the `Properties` pane.

If you set a property’s Scope to `Protected` or `Private`, it can be used only within the `App` class.

Scope of the App Class’s Methods

`Public` methods of the `Application` class are accessible to all code in your project. If you want to access a method outside of the `App` class’s own methods and event handlers, be sure to set its Scope to `Public`. For example, if you add a method to the `App` class called “`Foo`”, set its Scope as shown in Figure 421.

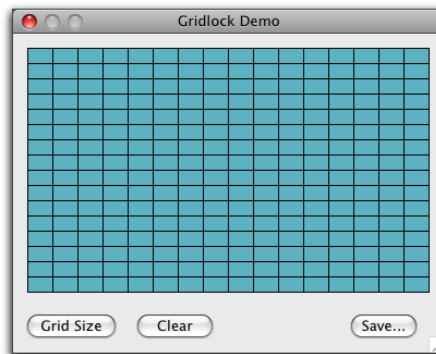
Figure 421. Creating a Public method of the App class.

If you set a method's Scope to Protected, then you can call it only within the App class's own methods and event handlers. Since this method is public, you can call it the App class with the line:

```
App.Foo
```

Creating Custom Interface Controls with Classes

One of the most important uses of classes is for creating custom interface controls. While REAL Studio provides most of the interface controls you will need in your project, you may find you need to create interface controls that are not built-in to REAL Studio. Suppose you need to create a control that displays a grid of cells. You want the user to be able to click on the cells in the grid to select them. Figure 422 shows an example of what such a grid control might look like.

Figure 422. The custom grid control in the GridLock project.

Custom interface controls are created by creating a subclass based on the Canvas control. The Canvas control gives you an area in which you can draw your control

and it receives events allowing you to interact with the user. For example, in the grid control example in Figure 422, the object is an instance of the Gridlock class, which was derived from the Canvas class (See Figure 394 on page 540). This is the Gridlock class example that you can find in the Graphics folder in the REAL Studio Examples folder. The Gridlock class inherits all the properties and events of the Canvas control and has special methods, properties, and events that enable it to present and operate this interface.

The Paint event handler of the Gridlock control is used to draw the grid. The Gridlock class has properties that store the number of rows and columns the programmer wants for a particular instance of the Gridlock. There are also properties that store the selected cell color and the unselected cell color.

When the user clicks in the grid area, the MouseDown event handler for the Gridlock class executes. The code for this event handler determines which cell was clicked and then determines if the cell should now be selected or unselected. A new event called CellClicked has been added to the Gridlock class that is called by the instance of the class when the user clicks on a cell. The purpose of this event is to allow an instance of a Gridlock class to react to a cell click. The CellClicked event handler is passed the row and column numbers of the cell that was clicked. The CellClicked event handler also acts as a function.

If an instance of the Gridlock class returns True in the CellClicked event handler, the Gridlock class assumes the programmer wants to filter the click, so it acts as if the user didn't click in the cell. The line of code that calls the CellClicked event is an If statement that tests whether it returns False before executing the code that changes the color of the cell that was clicked.

Drawing Your Custom Control

The Paint event handler of a Canvas control (or a Canvas control-based subclass) is executed any time the control needs to be redrawn. For example, if a window is covering part of the control and it is then moved to uncover more of the control, the Paint event handler executes to redraw the control. If the look of the control doesn't change at all when it's used, you can do all of the drawing of your control in the Paint event handler. However, if your control changes, you will need to take a different approach. For example, the Gridlock control changes when the user clicks on a cell. The Gridlock control also has a method that allows the number of rows and columns in the grid to be changed on the fly. This requires the grid to be redrawn.

In the Gridlock example, the grid needs to be redrawn at two different times. It needs to be redrawn in the Paint event handler when something like a window positioned over the control has uncovered a portion of the control, and when the grid is redefined to have a different number of rows and columns. Because of this, the code to do the actual drawing is placed in its own method. The method is called DrawGrid and it is passed the Graphics property of the Canvas control that the Gridlock class is based on. The DrawGrid method can then use this property to redraw the grid. By placing this code in a separate method, the same code can be

used by the Paint event handler and by the DefineGrid method. The Paint event handler is passed a reference to the Graphics property of the Canvas so this reference can be passed on to the DrawGrid method when calling it from the Paint event handler. The DefineGrid method calls the DrawGrid method as well since the grid is being resized and needs to be redrawn. The DrawGrid method can be passed the graphics property in this case by using the syntax:

```
DrawGrid Me.Graphics
```

Me is a reference to the instance of the class in the window. So although this code is being called from inside the Gridlock class, the use of Me allows it access to properties of the instance in use.

Class Interfaces

A class interface is a construct that you can use to tie together classes that do not share a super class but have something in common in your application. Class interfaces are used to specify what an object does without specifying how it does it.

In order to understand class interfaces better, it's helpful to think of a class as consisting of two components, the (public) interface to the class and the implementation. The interface consists of the class's Public method calls and the implementation is the code that implements the methods. The interface says what the class does and the implementation says how it does it.

In the object hierarchy, a subclass inherits both the interface and the implementation from its super class. That is, it gets both the method calls and the specific implementation of the method.

Class interfaces enable you to separate the two constructs. If two or more classes need to do the same thing but do it in different ways, you use an interface instead of a super class.

A class interface operates as a “spec” that contains a list of methods that custom classes in your project use. It does not actually contain any code for the methods themselves.

The methods in the class interface are placeholders for methods that are actually contained in each custom class that “implements” the class interface. Also, a custom class can implement more than one class interface.

The term “implement” simply means that the class has methods of the same names and declarations that are found in the class interface. The class interface specifies the methods and their declarations but not the code.

Many class interfaces are built into REAL Studio. You can implement any of these in your classes or add and implement your own. For example, the Readable and Writeable interfaces specify methods for reading and writing data. Each class that uses the Readable or Writeable interfaces supplies the implementations. For exam-

ple, a class that reads data from the Serial port would use a different implementation than a class that reads data from a binary file.

When you specify that a class implements a class interface, the class must implement all the methods in the class interface and the method declarations must match. However, the classes are free to implement the methods in different ways. For example, a method that changes the font in a StaticText would be implemented in a different way than a method that changes the font in a Canvas control that displays text via calls to the Graphics class.

The process involves three basic phases:

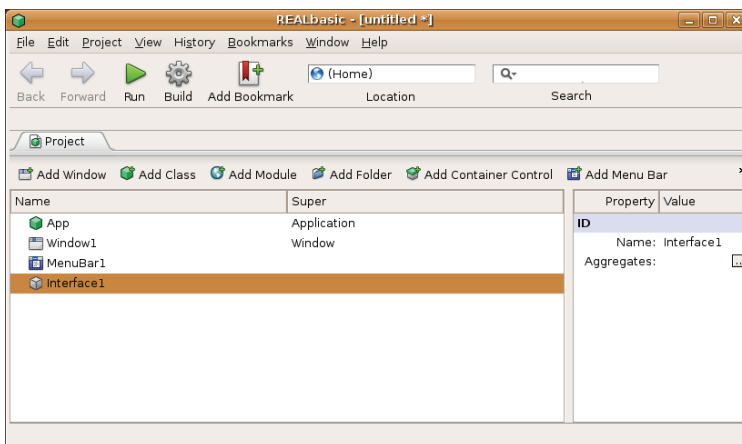
- Creating the class interface,
- Creating the classes that implement the class interface,
- Adding the classes to your project and calling the class interface methods in your program. Typically, that means writing generic code that tests whether a class implements a class interface and executing class interface methods where appropriate.

To create a class interface, do this:

- 1 **If the Project Editor is not displayed, click on its tab and then click the Add Class Interface button or choose Project ► Add ► Class Interface.**

A new class interface is added to the Project Editor. Its icon is hollow, indicating that it doesn't actually hold code. The Properties pane shows that the only property that you can modify is its name. It has no Super Class, as it is not part of the object hierarchy.

Figure 423. The Project Editor with a new class interface.



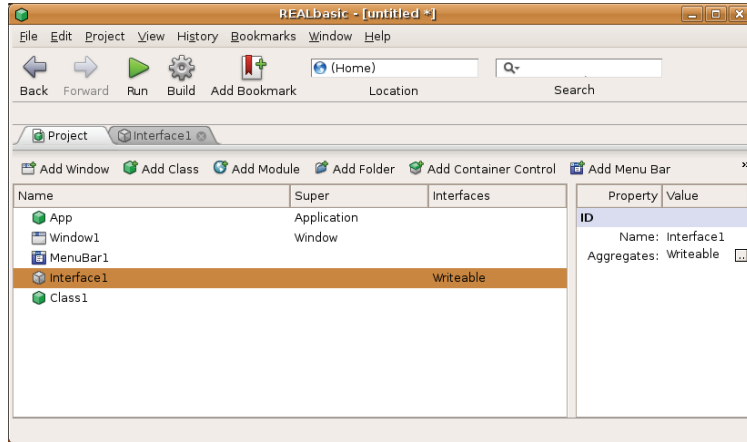
- 2 **If desired, use the Properties pane to change the name of the class interface.**

The Aggregates property enables you to specify one or more interfaces for this interface to implement.

3 If desired, click the three dots and choose one or more interfaces.

The Project Editor displays the chosen interfaces.

Figure 424. A new interface that implements the existing Writeable interface.



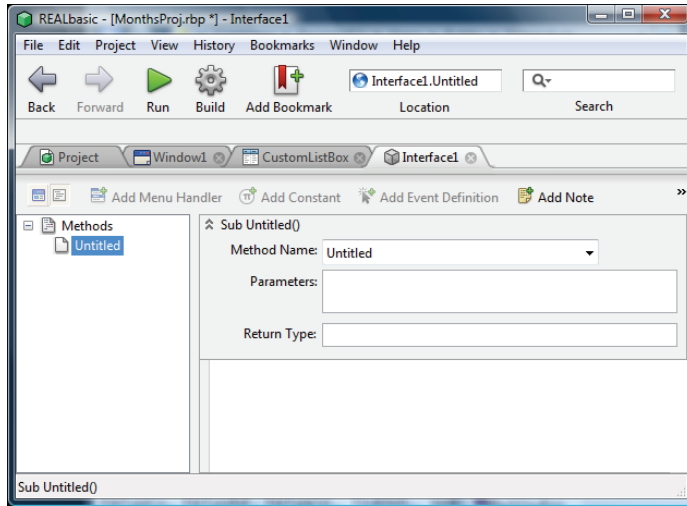
4 Double-click the Class Interface item in the Project Editor to display the Code Editor for the class interface.

The Code Editor for a class interface has items only for methods, shared methods, and notes. You cannot create properties or constants for a class interface.

5 Click the Add Method button or choose Project ► Add ► Method to add a method declaration to the class interface.

The Method declaration area appears above the Code Editor area.

Figure 425. The Method declaration area for a Class Interface.



- 6 Enter the name of the method, its parameters, and, if it will return a value, the data type of the value being returned.**

In other words, declare the method in the normal manner. The only difference is that there is no way to specify the Scope of a method in a class interface. This is because the code for the method does not live in the class interface at all. You use the Method declaration only to provide the ‘spec’ for the methods that will be written (a.k.a., “implemented”) elsewhere. The method will have several implementations, one in each class that implements the class interface.

For more information on declaring a method, see the section “Adding Methods to Windows” on page 329.

- 7 Repeat steps 4 and 5 for each method or function declaration of the Class Interface.**

After you have created your class interface, you must “hook it up” to one or more custom classes. To be non-trivial, we assume it will be two or more custom classes. You add the code for the methods declared in the class interface in *each* class that implements the class interface.

To implement a class interface, do this:

- 1 In the Project Editor, select the class to which you want to add the class interface or, if it does not yet exist, click the Add Class button in the Project Editor or choose Project ► Add ► Class.**

Notice that the Properties pane for the class contains a field for specifying the class interface (or interfaces) for the class.

You can specify the class interface or interfaces that the class implements by entering their names in the Interfaces field in the Properties pane or via the Project pane’s

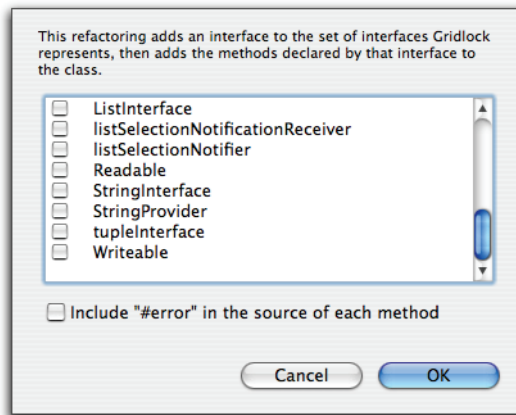


contextual menu. If desired, you can specify more than one class interface for the class.

- 2 **In the Properties pane’s Interfaces field, click on the ellipsis (the box on the right with three dots in it) or right+click (Control-click on Macintosh) on the name of the class in the Project Editor and choose Implement Interface.**

The Implement Interface dialog box appears. It presents a list of all the currently defined class interfaces in the application.

Figure 426. The Implement Interfaces dialog box.



- 3 **Click the checkboxes for the class interface or interfaces you wish to add.**

When you do so, the names of the class interfaces are added to the Interfaces field in the class’s Properties pane. REAL Studio also adds an Interfaces column to the Project Editor that shows the names of the newly added interfaces. This is shown in Figure 430 on page 592.

When you choose a class interface, REAL Studio adds all the method declarations for the interface to the class’s Method Editor. The class’s Method Editor then displays the first such method, ready for you to write the method. Each method that is generated by the Implement Interface dialog has a comment line that explains which Class Interface the method belongs to.

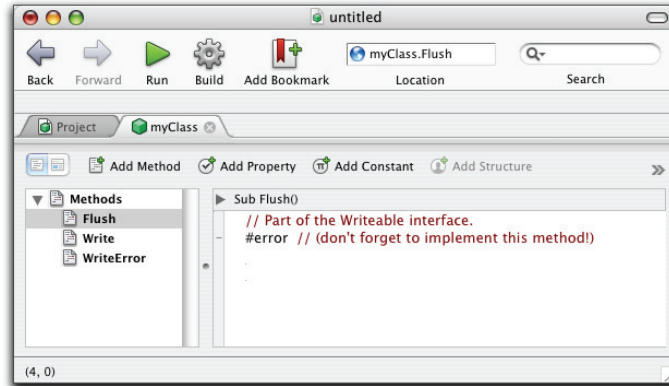
- 4 **If desired, choose the “Include #error” option.**

If you select the “Include #error in the source of each method” option in the Implement Interfaces dialog, it also includes an uncommented line with the directive “#error”. This line causes the compiler to generate a syntax error. The purpose of the line is to remind you to implement the method. If it weren’t there and you forget to implement the method, you would satisfy the technical requirement that the method exists, but it would be an empty method. The resulting compiler error would remind you to implement the method.

When you finish implementing the method, you should remove or comment out this line.

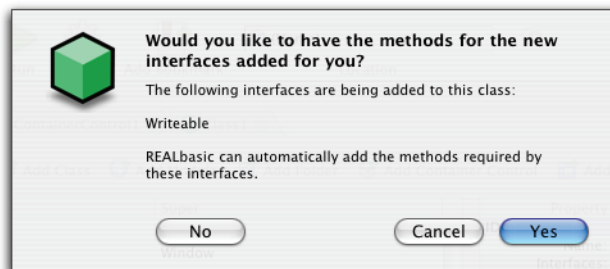
Here is an example method that was generated by the Implement Interface dialog.

Figure 427. The Flush method of the Writeable class interface.



You can also enter the names of class interfaces directly into the Interfaces field in the Properties pane. Click in the text area of the Interfaces field to get an insertion point and enter the name of the interface into the Interfaces field of the Properties pane. When you enter an interface, REAL Studio displays a dialog asking you whether you'd like it to generate all the method declarations for the interface.

Figure 428. The Add Methods dialog.



If you click Yes, it displays the Implement Interface dialog where you can choose the interface you entered.

If you don't accept this choice, you must take care to implement all the methods yourself.

When you add a class interface to a class, the Project Editor adds a third column and it lists the class interfaces for the class's interfaces.

Implementing Methods

The next task is to implement the methods of the class interfaces that you added to the class. If you used the Implement Interfaces dialog box, you have a head start on this task because it adds the method names and declarations of each class interface to the class automatically. If you entered the names of the class interfaces manually, you now need to add the required methods to the class.



To implement a method, do this:

- 1 If the methods are not already added, click the Add Method button or choose Project ► Add ► Method.**
The the Add Method pane appears.
- 2 Enter a method name and the declaration that was defined in the class interface and click OK.**
This time, the Code Editor supports code entry.
- 3 With the Code Editor, write the implementation of each class interface method for this class.**
- 4 Repeat steps 4 to 6 for every method declaration in each class interface that is implemented by this class.**
- 5 Repeat the entire process (steps 1 to 7) for each class that implements the class interface.**

For example, in Figure 430 two classes implement the class interface, `FontInterfaceManager`, shown in the Project Editor. Different classes can implement the same method in different ways.

Modifying and Deleting Interfaces

If you change your mind and want to delete or replace an interface, you do so via the Interfaces field in the Project Editor.

Click in the Interfaces field to get an insertion point and then select the name of the interface you want to modify or delete. Type to make the desired changes.

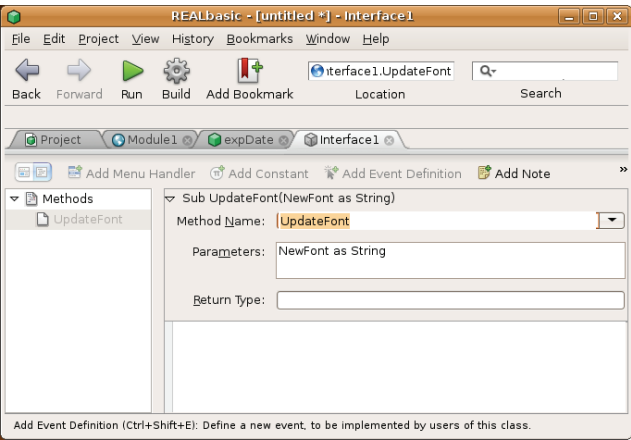
Please note that any methods that were added to the class while the interface was implemented are not modified or deleted when you remove the interface that they belonged to. That is, if you add an interface via the Implement Interfaces dialog, the methods that are specified by that interface remain as part of the class even if you delete the interface itself. You must take care of any “clean up” activities.

A Class Interface Example Project

The following very simple example illustrates the basic concepts. The application has custom classes based on the `TextField` and `StaticText` classes that implement a class interface. The class interface has one method specification for updating a font. The user chooses a font from the popup menu and a message is sent to all controls in a window that implement the class interface.

The class interface, `FontInterfaceManager`, contains one method specification:

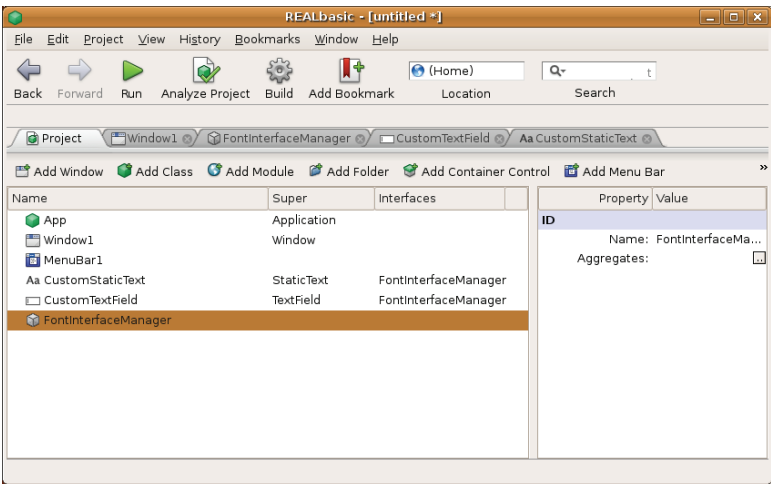
Figure 429. The `UpdateFont` method specification.



This method declaration area specifies that each class that implements this class interface has to have a method called `UpdateFont` and it must take one parameter as a string.

Two custom classes based on `TextField` and `StaticText` implement this method, as is shown in the Project Editor:

Figure 430. The Project Editor with two classes that implement the class interface.



The implementation of the `UpdateFont` method happens to be the same in both classes, but this is not a requirement of class interfaces in REAL Studio. It is simply:

```
Sub UpdateFont (s As String)
  Self.TextFont=s
End Sub
```

With these objects in place, generic code can be written that determines whether a control in a window implements the `FontInterfaceManager` class interface.

Suppose several instances of `CustomTextFields` and `CustomStaticTexts` are added to a window. The following code in the `Change` event of a `PopupMenu` changes the font displayed by all such controls based on the user's selection. The `IsA` operator tests whether a control implements the class interface. If it does, the `UpdateFont` method of the class interface is called, as *implemented by each custom class*. The term `Me.Text` contains the current selection of the `PopupMenu`.

```
Dim i as Integer
For i=0 to Self.ControlCount //number of controls in the window
  If Self.Control(i) IsA FontInterfaceManager then
    FontInterfaceManager(Self.Control(i)).UpdateFont(Me.Text)
  End if
Next
```

As you can see, you can specify that any custom class implements a class interface, regardless of its super class. In this way, class interfaces can group together classes that are not related to one another via the object hierarchy and give them functionality that can be managed in one place in your project.

Creating a new Class Interface from an Existing Class

If an existing class in your project contains methods that you want to reuse as a class interface, you can generate the new class interface from the Project Editor.

To use an existing class as the basis of a class interface, do this:

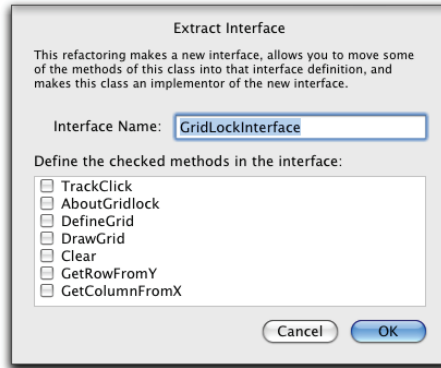
- 1 Right+click (Control-click on Macintosh) on the class you want to use as the starting point for the new class interface.**

The contextual menu for the class appears.

- 2 Choose Extract Interface from the contextual menu.**

The Extract Interface dialog box appears.

Figure 431. The Extract Interface dialog box.



The Extract Interface dialog lists all the methods in the current class.

3 Enter the name of the new class interface in the Interface Name field.

The name of the class you clicked on, followed by *Interface* is entered by default.

4 Click the checkbox for each method you want to include in the new class interface.

5 When you are finished, click OK to save the result.

REAL Studio creates the new class interface, adds it to the project, and makes the current class an implementor of the new interface. That is, the name of the new class interface is now shown in the Interfaces field of the class's Project pane.

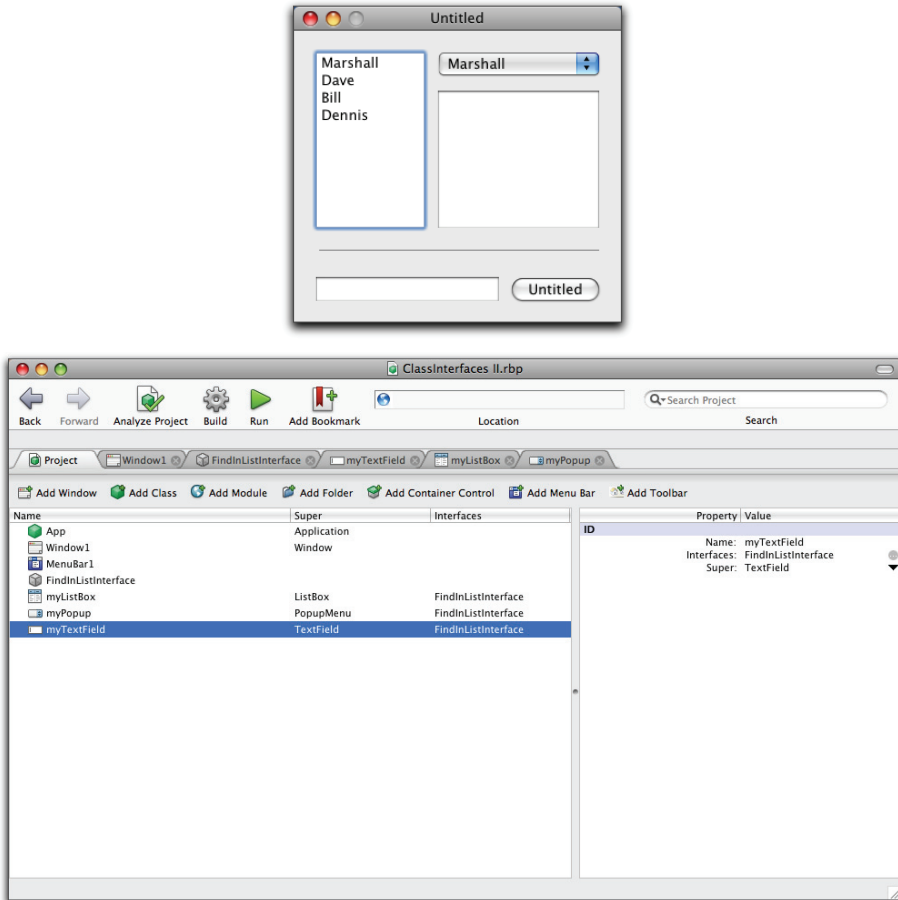
Interface Inheritance

Although Interface Inheritance sounds complicated when described in abstract language, it actually addresses a simple problem. If you have several controls that need to perform the same task but in a different way (depending on the specifics of the types of control) you can write and execute interface-specific code in an elegant way.

Figure 432 on page 595 shows an application that uses interface inheritance. The purpose of this application is to conduct a search for a user-entered string and find the string in the three controls located above the separator: The TextField, ListBox, and PopupMenu are all based on custom classes. Although the task is identical (a find operation), it cannot be done with exactly the same code for all three objects since the three objects store and manipulate data differently. Therefore, each custom class has its own implementations of the methods used to do the search.

The ListBox, TextField, and PopupMenu are all derived from custom classes that use a custom interface, FindInListInterface. They all have a Find function that takes the same parameter, but all implement it differently. The code for the Find button can call all of their Find functions using the same syntax.

Figure 432. An example application that uses interface inheritance.



The user enters a search string in the TextField, FindValue, to the left of the Find button. When he clicks the Find button, the following code is executed:

```
FindIt FindValue.text, listBox1
FindIt FindValue.text, popupMenu1
FindIt FindValue.text, TextField1
```

The same method, FindIt, is called for each of the three controls, but each line of code actually executes a different version of FindIt—the one that is appropriate for that type of control. The second parameter is the name of the control; each control inherits methods from the custom class on which it is based.

The TextField, PopupMenu, and ListBox are all instances of custom classes. The custom classes have two methods, Find and SelectRow, that implement the correct search routines for that object type. This is shown in Table 36.

Table 36. Find functions and SelectRow methods for different controls.

Control	Find Function	SelectRow Method
TextField	<pre>Function Find (FindValue as string) as Integer dim rows, foundPos, foundCRpos as integer rows=-1 foundPos=instr(text,findValue+ chr(13)) do until foundCRpos>=foundPos foundCRpos=instr(foundCRpos +1,text,chr(13)) rows=rows+1 loop return rows</pre>	<pre>Sub SelectRow (Row as Integer) dim counter, startPos, endPos as integer do until counter=row startPos=instr(startPos+1, text, chr(13)) if startPos <> 0 then counter=counter+1 end if loop endPos=instr(startPos+1,text,chr(13)) selStart=startPos selLength=endPos-startPos</pre>
ListBox	<pre>Function Find (FindValue as string) as Integer dim i as integer for i=0 to listcount-1 if list(i)=findValue then return i end if next</pre>	<pre>Sub SelectRow (Row as Integer) listindex=row</pre>
PopupMenu	<pre>Function Find (FindValue as string) as Integer dim i as integer for i=0 to listcount-1 if list(i)=findValue then return i end if next</pre>	<pre>Sub SelectRow (Row as Integer) listindex=row</pre>

The FindIt method itself uses the FindInListInterface:

```
Sub FindIt (findValue as String, source as FindInListInterface)
dim row as integer
source.selectRow source.find(findValue)
```

The FindInListInterface class simply has two blank methods, Find and SelectRow. It simply defines the methods and their parameters. When the FindIt method runs, it actually executes the versions of the methods that are appropriate for the control passed as the second parameter to FindIt.

Introspection

REAL Studio includes a facility for obtaining information about an application's structure at runtime. It is the *Introspection* system. The Introspection system is implemented as a REAL Studio module and consists of the following classes:

Name	Description
AttributeInfo	Provides information about an item's attributes. Attributes are compile-time properties. They are created via the Attributes Editor in the IDE. Each attribute consists of its identifier (a.k.a. name) and optionally a value.
ConstructorInfo	Provides information on a class's constructors. Use the GetParameters method to get the parameters of each constructor, if any and the ReturnType property to get the TypeInfo on the value returned by the constructor, if any.
MemberInfo	The super class for ConstructorInfo, MethodInfo, PropertyInfo, and TypeInfo. Contains a property that contains the datatype's name.
MethodInfo	Provides information on the methods in the datatype. Use the GetParameters function of this class to get information on the parameters of each method in the class.
ParameterInfo	Provides information on the parameters of methods belonging to the class. Use the MethodInfo.GetParameters function to obtain a ParameterInfo array to get the parameter info.
PropertyInfo	Provides information on the properties of the datatype.
TypeInfo	The root of the introspection system and the primary way to access program metadata. It is an abstract class that describes all the attributes or members a datatype might have.

The Introspection classes are Public rather than Global in scope. This means that you need to use the dot notation to access its classes. For example, this code obtains a TypeInfo object for the passed class instance and reports its class name.

```
Dim tcp as New TCPSocket
Dim t as Introspection.TypeInfo
t=Introspection.GetType(tcp)

MsgBox "My class name is "+t.Name
```

The MethodInfo and PropertyInfo classes get the methods and properties belonging to a class. For example:

```
Dim d as New date
Dim myDbMethods() As Introspection.MethodInfo = _
    Introspection.GetType(d).GetMethods

For i as Integer = 0 to UBound(myDbMethods)
    ListBox1.AddRow myDbMethods(i).Name
Next
```

The GetMethods function returns an array of MethodInfo objects, which can be used to get the datatype of each element.

Similarly, the ParameterInfo class is used to get information about a method's parameters, if any. For example,

```
Dim tcp as New TCPsocket

Dim myMethods() As Introspection.MethodInfo = _
    Introspection.GetType(tcp).GetMethods

For i as Integer = 0 to Ubound(myMethods)
    Dim myParameters() as Introspection.ParameterInfo = _
        myMethods(i).GetParameters
    If Ubound(myParameters) > 0 then //if a method has any parameters
        For j as Integer=1 to Ubound(myParameters) //loop over the parameters
            ListBox1.AddRow myMethods(i).Name
            ListBox1.Cell(ListBox1.LastIndex, 1)= _
                myParameters(j-1).ParameterType.Name
        Next
    End if
Next
```

For each element of the myMethods array, it gets the parameters for that method and creates a ParameterInfo array for that method. The ParameterType method gets TypeInfo for the parameter.

See the entries in the *Language Reference* for the Introspection module and its classes for more information on the Introspection system.

Importing Classes From Other Projects

Since classes can be exported, they can also be imported. When a class is exported, it appears on the desktop with a cube icon. Figure 433 shows an example of an exported class. To import a class, just drag the class file into your Project Editor or choose File ► Import and use the open-file dialog box to choose the desired class file.

This copies the class into the Project. If you wish, you can delete the class file if don't need to use it elsewhere. The Project is not dependent upon it.

Figure 433. An exported class file (Macintosh).



If a class you are importing is based on another class, that other class must be present in order for the class you are importing to function. If that class is based on one of the built-in classes (like the TextField for example), this isn't an issue. However, if the class is based on a class that isn't built-in, then that other class must be present in your Project Editor.

Exported classes can be encrypted. This means that, while you can import and use the class, you cannot view or edit the source code for the class. If the class is encrypted, a small key badge appears in the class's icon in the Project Editor.

Encryption is supported only in the Professional and Studio editions of REAL Studio. Decryption is supported in all editions of REAL Studio.

For more information, see the following section "Encrypting Your Source Code" on page 599.

Importing External Project Items

If you need to use the class in more than one project, you can add it to the project as an external project item. An external project item is stored on disk and is referenced by each project that uses it. If a change to the item is made from one project, those changes are made available to all the other projects that reference the external item. Changes to the external project item are saved to disk when you save the project.

To import an item as an external project item, hold down the Alt key (Option key on Macintosh) and the File ► Import item changes to File ► Import as External. Choose the item and it will be imported as an external project item.

For more information, see the section, "External Project Items" on page 542.

Exporting Classes For Use In Other Projects

Classes can be easily exported from your Projects for use in other projects. You can export the class by clicking on the class to select it in the Project Editor and choosing File ► Export Class. To export the class as an external project item, follow the steps in the section "External Project Items" on page 542.

Encrypting Your Source Code

You may want to share classes with other REAL Studio users. If you wish to share a class with other users but you don't want to share the source code itself, you can encrypt the class before you export it. This creates an exported class that can be imported and used but cannot be edited. The user cannot even view the source code.

This is especially important if you plan to create sophisticated classes that you wish to sell as third party add-ons to REAL Studio.

Encryption is supported only in the Professional and Studio editions of REAL Studio. Decryption is supported in all editions.

To encrypt a class, do this:



- 1 Click on the class to be encrypted in the Project Editor and choose Edit ► Encrypt or Right+click on Windows and Linux (Control-click on Macintosh) on the class in the Project Editor and choose Encrypt from the contextual menu.**

You can optionally add an Encrypt button to the Project Editor toolbar. If you have done so, you can encrypt an item by selecting it and clicking the Encrypt button. The Encrypt *Class* dialog box appears, as shown in Figure 434.

Figure 434. The Encrypt *Class* dialog box.



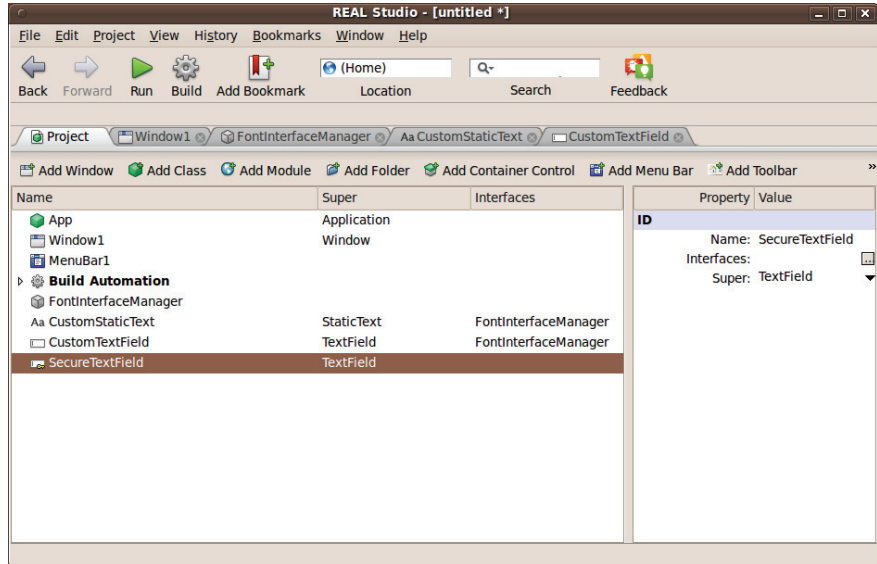
- 2 Enter and confirm a password for encryption.**

Important: Don't forget your password.

- 3 If you want the class to be accessible only to REAL Studio 2006r3 and above, then check the "Use REAL Studio 2006r3 Encryption" checkbox.**

An encrypted class appears in the Project Editor with a small key in the right corner of the class icon. If it is subclassed from a control class, it has the icon of the control.

Figure 435. A project with an encrypted class.



When a programmer tries to open an encrypted class, REAL Studio presents the Decrypt *class* dialog box, shown in Figure 436.



To decrypt an encrypted class, do this:

- 1 Click on the module to be decrypted in the Project Editor and choose **Edit ► Decrypt** or Control-Click on the module in the Project Editor and choose **Decrypt** from the contextual menu or choose **Edit ► Decrypt**.

The Decrypt *class* dialog box appears.

Figure 436. The Decrypt *Class* dialog box.



- 2 Enter the decryption password and click **Decrypt**.

In a few moments, the key will disappear from the class's icon, indicating that it has been successfully decrypted. If you entered an incorrect password, a dialog box will inform you of that fact.



NOTE: Since other users cannot open an encrypted class to view its source code, you will need to provide them with a list of methods and properties if they should have access to them.

Deleting Classes From a Project

Before deleting classes from a Project, make sure you are not using the class in your code anymore. Also be sure to check for other classes that may have this class as their super class.



To delete a class from a Project, do this:

- 1 Click on the class in the Project Editor to select it.**
- 2 Press the Delete key on the keyboard or choose Edit ► Clear or Right+click (Windows and Linux) or Control-click on Macintosh on the class and choose Delete from the contextual menu.**

If you delete a class accidentally, choose Edit ► Undo (Ctrl+Z or ⌘-Z).

Creating Databases with REAL Studio

With REAL Studio, you can create database front-end applications that can be used with a variety of database engines, including REAL Software's own data source.

REAL Software's own single-user database engine, REAL SQL Database, is built into all versions of REAL Studio. REAL Software's multi-user database engine, REAL Server, is sold separately to REAL Studio Professional customers. It is included with the Studio version of REAL Studio.

Database engines that are not included with REAL Studio are supported only in the Professional and Studio versions of REAL Studio.

Contents

- REAL Studio's database architecture,
- Structured Query Language (SQL),
- REAL Studio's database tools,
- Creating and modifying databases from the Project Editor,
- The DatabaseQuery control,
- Using the DataControl control,
- Creating a database programmatically.

REAL Studio's Database Architecture

You can use REAL Studio to build a “front-end” to your database. It works in conjunction with a database “back-end” that actually stores the data itself. The database back-end can be a separate application or it can be REAL Studio's own database back-end, REAL SQL Database. The front-end serves as the user interface — the means by which queries are sent to the source and information is displayed and printed. The end user uses the front-end to view, enter, and modify records, search for and sort records, and print reports.

The database back-end, such as REAL SQL Database, REAL Server Database, PostgreSQL, or Oracle, actually stores the data. The database application that actually holds the data is referred to in REAL Studio as the *data source*.

A great feature of this architecture is that a database front-end that you create in REAL Studio can be adapted to work with any supported data source—or multiple data sources. You can develop a database application with the internal REAL SQL Database and then deploy the system after switching the data source.

A REAL Studio front-end can also use two or more data sources simultaneously. For example, you can access data locally on REAL SQL Database while simultaneously accessing remote data on a SQL or ODBC-compliant database.

REAL Studio uses its plug-in architecture to support multiple data sources. Plug-ins are external files that must be placed in the Plugins folder in order to use the data source. The exception is the REAL SQL Database, which is built into the REAL Studio application and does not require an external plug-in.

You (or a third-party) can add support for additional data sources by writing a plug-in for that back end. REAL Software's plug-in SDK contains information on writing database plug-ins.

Structured Query Language

A REAL Studio front-end uses the Structured Query Language (SQL) to communicate with its data sources. The plug-in for your data source receives a SQL statement from REAL Studio and (if necessary) translates the statement into a form that the data source understands, and sends it to the data source.

The REAL SQL Database is based on SQLite. It uses the most recent version, which is 3.6.6. It supports the subset of SQL described at <http://www.sqlite.org>. REAL Studio simply passes your SQL to the data source. Therefore any valid SQL for that data source will work in a REAL Studio front-end for that source. Please refer to the documentation for your selected data source for further information on supported SQL and specifics on the data types supported by that data source.

If you are unfamiliar with SQL, you will need to learn its basics before implementing your REAL Studio front-end. This manual does not attempt to teach you SQL; rather, it describes the subset of SQL that is currently supported for the

REAL SQL Database data source. For complete information on SQLite, see their web site at <http://www.sqlite.org>.

For other SQL data sources, please consult one of the many good SQL references, such as *SQL for Dummies* by Allen G. Taylor (ISBN: 0-7645-0105-4), *The Practical SQL Handbook*, by Bowman, Emerson, and Darnovsky (ISBN: 0-2014-4787-8).

REAL Studio's Database Tools

A database front-end typically uses a mixture of database-specific and generic controls and commands. There are two database-specific controls, the DatabaseQuery and DataControl controls, and several classes and methods that are database-specific. Beyond that, you will use generic controls such as StaticTexts, TextFields, PopupMenus, ComboBoxes, and ListBoxes to display and edit data, PushButtons and menu items to perform actions, and TabPanel controls and other interface elements to polish the user interface.

All REAL SQL Database tables have an Integer Primary Key column. If you don't explicitly define one, one will be created for you. You can refer to the Integer Primary Key column using the "rowid" keyword. But, if you don't explicitly define your own Integer Primary Key column, you won't get the "rowid" column unless you refer to it in queries.

The REAL SQL Database supports text encodings. When you insert records into the database, the database stores text values using the UTF-8 encoding. If it is not already UTF-8, it will be converted. If you want to defeat this conversion, store text data in a Blob column.

The REAL SQL Database engine supports transactions for both changes to the database design and for changes to the data. A transaction is started automatically when you make any change to the database and ended by a call to the SQL commands COMMIT TRANSACTION or ROLLBACK TRANSACTION.

Selecting a REAL Data Source

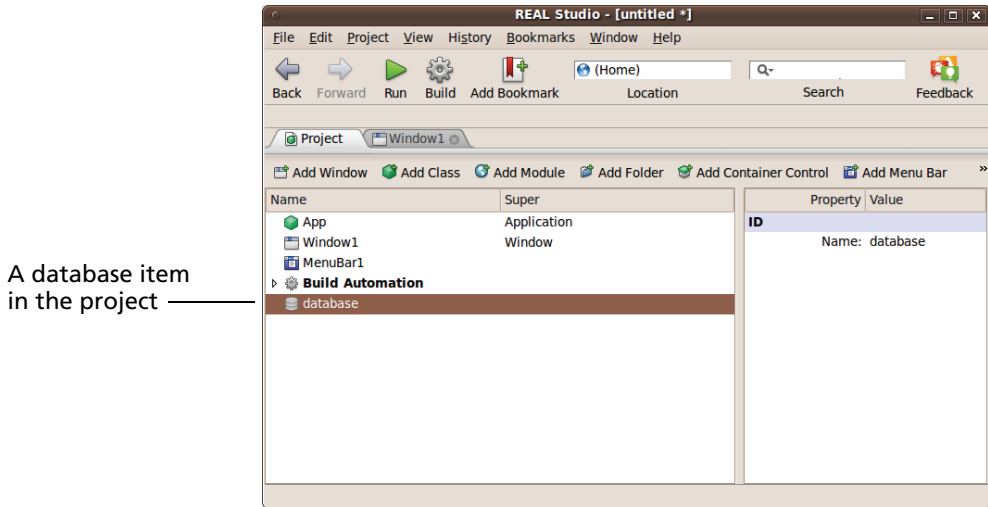
You can select a data source using either the language or the Project ► Add .Database ► New REAL SQL Database or Select REAL SQL Database submenus in the IDE. Except for the REAL SQL Database, a data source will appear in the Database submenu only if its plug-in is in the Plugins folder.

When you choose Select REAL SQL Database, a standard open-file dialog box appears. Navigate to the directory that contains the REAL SQL Database and open it.

When you choose New REAL SQL Database, a standard save-file dialog box appears. You can save the REAL SQL Database on any mounted volume. Name the database file as you would any other file and click Save.

When you choose an existing data source or add a new REAL SQL Database, it appears in the Project Editor as shown in Figure 437.

Figure 437. A database in the Project Editor.



To remove a data source from the Project Editor, highlight the data source and press the Delete key on the keyboard or choose Edit ► Delete. You can also Right+click (Control-click on Macintosh) on the data source and choose Delete from the contextual menu.

You can also create or open a REAL SQL Database using the language. To open an existing REAL SQL Database, create an instance of the REALSQLdatabase class and assign the location to its DatabaseFile property. Then call the Connect method. If Connect returns True, the connection was successful and you can proceed with database operations. Otherwise, you should look at the ErrorMessage and ErrorCode properties to troubleshoot the problem.

Here is an example:

```
Dim dbFile as FolderItem
Dim db as REALSQLdatabase
db=New REALSQLdatabase
dbFile = GetFolderItem("Pubs")
db.DatabaseFile=dbFile
If db.Connect() then
    //proceed with database operations here..
else
    Beep
    MsgBox "Database Error: " + Str(db.ErrorCode) + EndOfLine + _
        EndOfLine + db.ErrorMessage
end if
```

This example opens the “Pubs” database, which is in the same folder as REAL Studio. If the database is not in the same directory as the REAL Studio application, use the Volume function and the Parent or Child properties of the FolderItem class

to navigate to it. Examples of this are shown in the section “Getting a File at a Specific Location” on page 492.

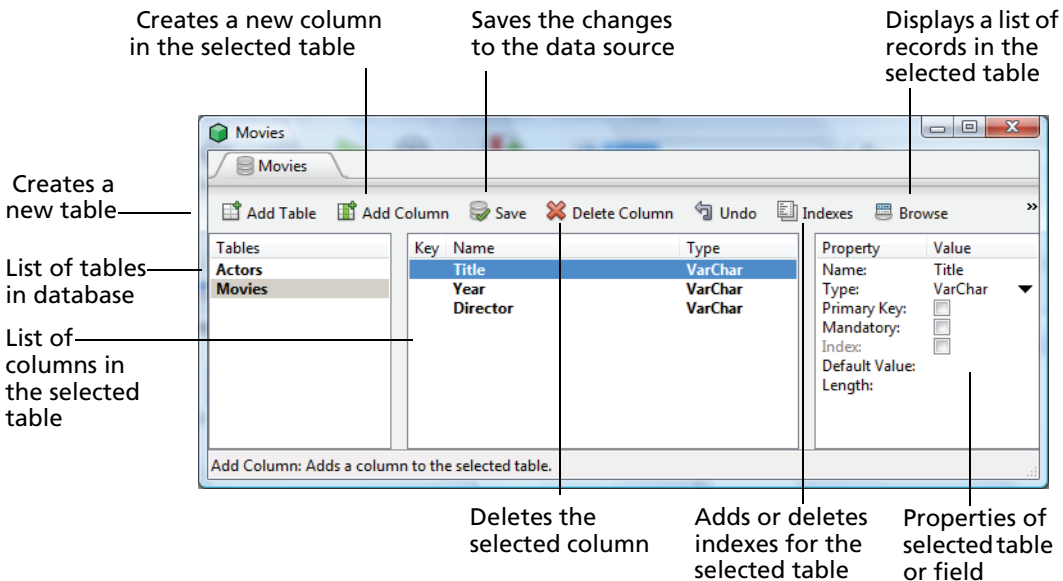
To create a new REAL SQL Database, follow the same procedure except call CreateDatabaseFile instead of Connect. It returns True if the operation was successful. Here is an example:

```
Dim db as REALSQLdatabase
Dim f as FolderItem
f=New FolderItem("mydb")
db=New REALSQLdatabase
db.databaseFile=f
If db.CreateDatabaseFile then
    //proceed with database operations...
else
    Beep
    MsgBox "Database Error: " + Str(db.ErrorCode) + EndOfLine + _
        EndOfLine + db.ErrorMessage
end if
```

Creating and Modifying Databases from the Project Editor

You can double-click a REAL SQL Database in the Project Editor to display its Schema—the list of its tables and each table’s columns. From that list, you can view the data itself and the list of columns and their properties.

Figure 438. A database Schema for an existing database.



If the database is new, the list of tables will be blank; you can add tables by clicking the Add Table button.

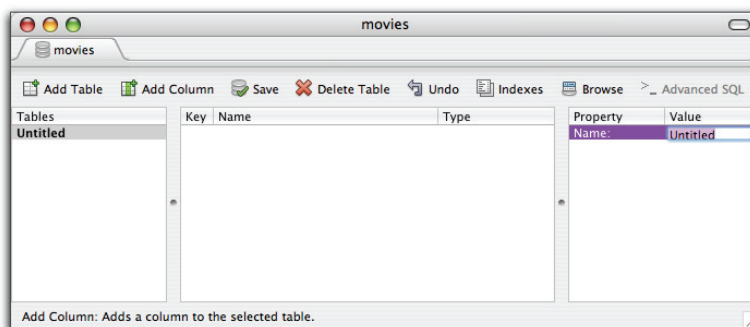


To create a new table, do this:

1 Click the Add Table button.

REAL Studio adds a new table to the database named “Untitled”. The Properties pane shows that it has one property, its Name.

Figure 439. A new table in the Database editor.



2 Enter the Name of the table into the Properties pane.

When you do so, the name of the table changes in the Tables pane.

Each table has to have at least one column. This is a unique identifier (Integer) that you can use to identify each record in relational operations. You don’t need to create a Primary Key column of your own. If you refer to the “rowID” column, it will created for you.

3 Click the Add Column button in the Table editor toolbar to add the first column.

REAL Studio adds a column named “Untitled” in the Columns pane, with the Var-Char data type.

4 Select the new column and use its Properties pane to set its Name and data type properties.

The supported column types are shown in the Type pop-up menu.

5 Assign any other properties to the column, as desired. If the data source does not support a property that is listed in the Properties pane, it is dimmed out.

6 Repeat steps 3 and 4 to add additional columns.

As you add each column, it appears in the column list. Figure 440 shows the Table editor with a new table and one column.

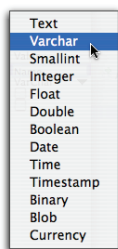
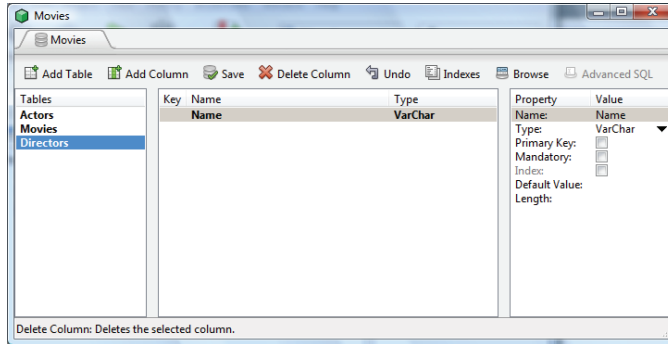


Figure 440. The New Table dialog with one column.

- 7 When you are finished specifying the table, click the Save button in the toolbar to commit all the changes to the data source.**

Adding Indexes

After adding your columns, you will want to index certain columns. Indexes improve the performance of the database. You should index columns on which you plan to do the majority of your searches and sorts. Information retrieval and relational operations among tables are faster when indexed columns are used.

Note Certain data sources do not support indexing. If the data source you are working with does not support indexing, the Indexes button described in this section does not appear in the New Table/Edit Table dialog

Columns that are not good candidates for indexing are those that only take on a few values (i.e., gender or race), columns that are rarely searched on, and any columns in small tables where search and sort times are unlikely to be long.

You should not index too many columns because each index adds to the size of the database and, as records are added and deleted, it takes time for the system to update each index.

You can create indexes in your code via the SQL Create Index statement. If you want to create an index within the REAL Studio IDE, use the following procedure.



To create the indexes for a table, do this:

- 1 Click the Indexes button in the Database Editor toolbar.**

REAL Studio adds an Indexes panel to the editor.

- 2 Click Add Index to add a new index to the list of indexes.**

A new untitled index appears in the Indexes list. You define this index by adding columns to it.

- 3 Rename the index to something more meaningful.**

Usually you will use the names of the column or columns that comprise the index.

- 4 Click the Add Column button in the Indexes toolbar.**

The Select Column dialog box appears, listing all the eligible columns in the selected table.

5 Click on a column name to add it to the index.

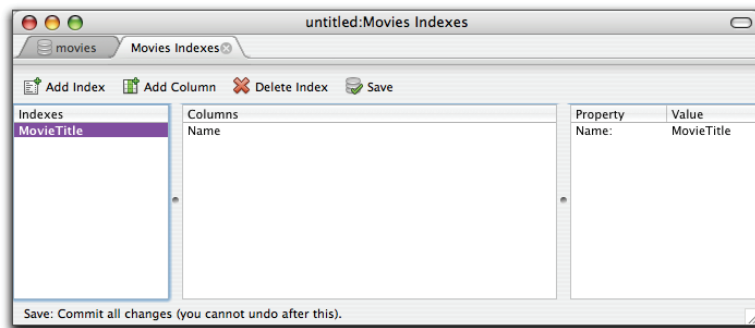
REAL Studio adds the column name to the list of columns in the center of the Indexes editor.

6 (Optional) To build a composite index (an index on two or more columns), click on Add Column once again and add another column to the index.

For example, you may want to create a composite index on LastName and FirstName columns so that the database can quickly work with different people with the same last name.

Creating a composite index is different from creating several distinct indexes (such as the Primary Key index and the Name index). You create several distinct indexes by clicking the Add Index button once per index and adding one column per index. A finished index is shown in Figure 441.

Figure 441. The index for the Movies Name column.



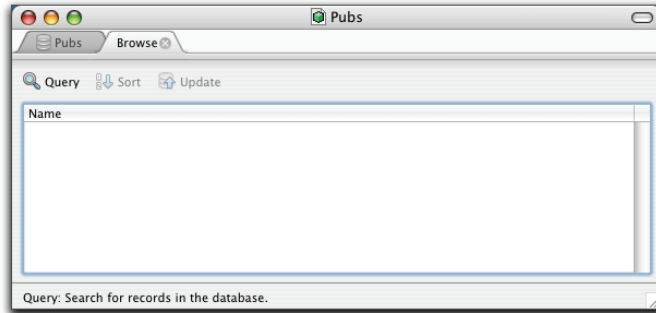
7 (Optional) To create additional indexes, click Add Index once again and repeat the process.

8 Click Save in the Indexes toolbar to save the indexes to the database.

Viewing Data

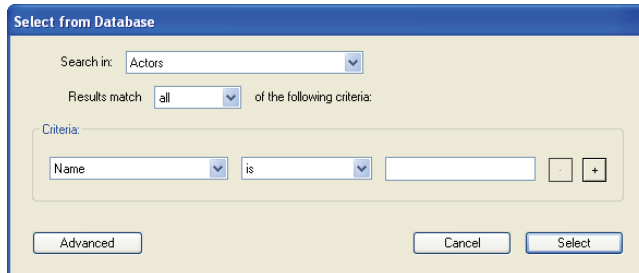
The Database Editor window enables you to view, sort, and update records, but it isn't a full-fledged end-user interface. You use a SQL Select statement to retrieve and sort the records. If the Select statement has no errors, REAL Studio will display the requested records in a list. If it encounters an error, it will display an error code and message. REAL SQL Database returns SQLite error codes. See the entry in the *Language Reference* for the REALSQLiteDatabase class for information on those error codes or <http://www.sqlite.org>.

To view the existing data, click the Browse button in the Browse toolbar. REAL Studio adds a new tab panel to the window for listing records. This is shown in Figure 442.

Figure 442. The Browse panel before running a query.

Initially, no records are shown. You must first select records to view via a Query. Click the Query button in the Browse panel to display the Query dialog box. It enables you to specify a query via a “Find” dialog, shown in Figure 443.

The Select From dialog box builds a SQL Select statement for you from your specifications. You can also do the query by writing a SQL Select statement manually. If you wish to specify the query that way, click the Advanced button. It is described next, in the section “Advanced Select” on page 613.

Figure 443. The Select From Database dialog box (Simple screen).

The simple version of the dialog box enables you to specify your query without knowing about the syntax of a SQL Select statement. The elements of this dialog perform the following functions:

- The **Search In** drop-down list contains the list of all of the tables in the database. With the Simple dialog, you can search in only one table. First choose the table that will be searched. Figure 443 specifies that this query will be in the Actors table.
- The **Criteria area** is where you specify the query. Each row in the Criteria area allows you to query on one column. You can specify multiple queries and combine them with either the AND or the OR operators. The first drop-down list in the Criteria area lists all the columns in the selected table. Choose a column to search on. Figure 443, for example, specifies the Name column in the Actors table.
- The **Value area** is where you specify the value you are looking for in the search.

- The **Operators drop-down list** lets you specify the relationship of the column's contents to the value you entered in the Value area. Your choices are “is” (a.k.a. equal to), “is not” (a.k.a. not equal to), “greater than”, “less than”, “starts with”, “ends with”, and “contains.” Greater Than and Less Than pertain to numeric, date, and time columns, while starts with, ends with, and contains pertain to VarChar columns.

For example, the following dialog specifies “Name equal to Fisher”.

Figure 444. A completed simple search.

The screenshot shows the 'Select from Database' dialog box. The 'Search in:' dropdown is set to 'Actors'. The 'Results match' dropdown is set to 'all'. The 'Criteria:' section contains a single row with 'Name' in the column dropdown, 'is' in the operator dropdown, and 'Fisher' in the value field. There are minus and plus buttons to the right of the value field. At the bottom are 'Advanced', 'Cancel', and 'Select' buttons.

If you want to search on more than one column, you need to add a row to the Criteria area. Do this by clicking the Plus sign (to the right of the Value area). A new blank row is created in the Criteria area. This is shown in Figure 445.

Figure 445. A second criterion row in the Criteria area.

The screenshot shows the 'Select from Database' dialog box with two rows in the 'Criteria:' section. The first row has 'Name' in the column dropdown, 'is' in the operator dropdown, and 'FISHER' in the value field. The second row has 'Movie' in the column dropdown, 'is' in the operator dropdown, and an empty value field. Minus and plus buttons are present to the right of each value field. At the bottom are 'Advanced', 'Cancel', and 'Select' buttons.

Repeat the process of specifying the search criterion on the second column.

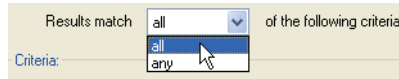
If you want to remove one of the rows in a multiple-column search, click the minus sign (next to the plus sign) in the row you want to remove.

When you have two or more rows in the Criteria area, you must specify whether a record must meet all of the specifications or any one of the specifications. The first type of search uses the AND operator to relate all statements, while the second type of search uses the OR operator.

You do this using the “Results Match” drop-down list. The default selection, “All”, requires that a record is not selected unless all of the statements in the Criteria area

are true; the “Any” selection means that a record will be selected if any one of the statements in the Criteria area are true.

Figure 446. The Results Match drop-down list.

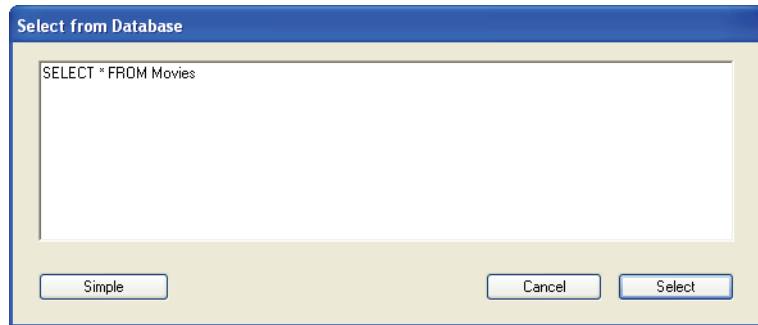


If you search on only one column, the Results Match drop-down list is not relevant. When you are finished specifying the query, you can click Select to do the search.

Advanced Select

If you want to enter the SQL Select statement directly or modify the one that is generated from the “Simple” version of the dialog box, click the Advanced button. The Advanced version of the dialog has only one field, in which you must write a valid SQL Select statement.

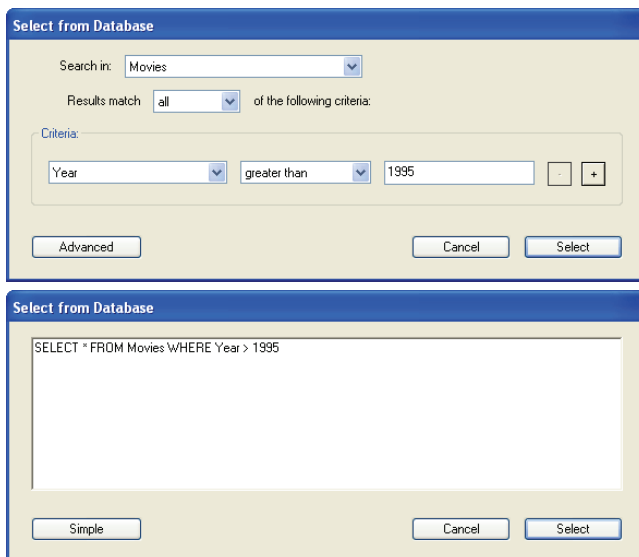
Figure 447. The Advanced version of the Select From dialog box.



If you have already entered search specifications into the “Simple” version of the dialog, those specifications will have been translated into a valid SQL Select statement for you. You can then edit that query or replace it.

For example, here is the same query that expressed on both dialog types:

Figure 448. A query in the Advanced and Simple versions of the dialog box.

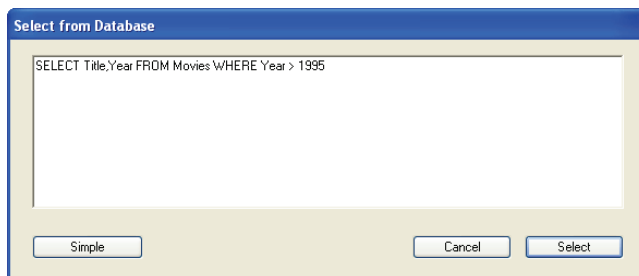


You may, of course, continue editing the SQL statement that REAL Studio generated.

The Simple search screen does not contain interface elements for all of the options that the SQL Select statement supports, so you may want to take advantage of these options. For example, the Simple screen does not enable you to specify the list of columns that will be returned by the query. Any search that is specified on the Simple screen will return all the columns.

If you want fewer columns, you should replace the asterisk in the SQL statement with the column list. For example, Figure 449 shows an edited version of the query in Figure 448 that specifies two columns.

Figure 449. A modified SQL query.



If you switch to the Simple version of the dialog after entering a SQL query, REAL Studio will set the values of the drop-down lists and the Value area to the extent that it can. If you have specified options in the SQL statement that are not available in the Simple view, you will lose that information.

In the Simple view, you can add an **ORDER BY** clause to sort the rows by one or more columns. Figure 450 shows a SQL statement that uses the **ORDER BY** clause to sort all the movies in the table in descending order.

Figure 450. A listing of all records.

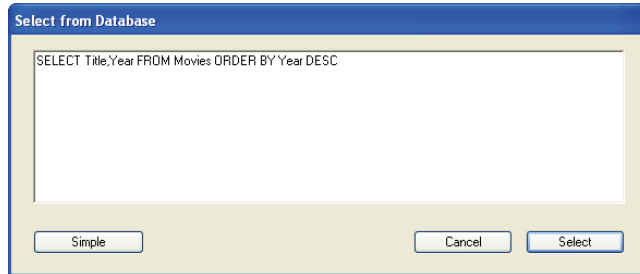
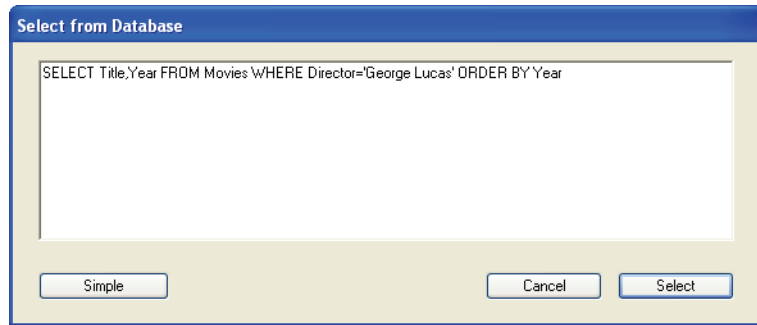


Figure 451 shows another example of a SQL statement that returns two columns and the records sorted in ascending order (the default sort order).

Figure 451. Selecting and sorting rows with a SQL Select statement.



Storage Types and Column Type Affinities

The data types shown in Table 37 are supported by the REAL SQL Database. Except for the Integer primary key column, any column can store data of any type.

If you are using another data source, please consult the documentation for your data source for information about its supported data types.

Table 37: Data Types supported by REAL SQL Database

Data Type	Description
Binary	Stores code, images, and hexadecimal data of any size.
Blob	Stores a binary object. The REAL SQL Database supports blobs of up to any size. A blob can be stored in a column of any declared data type. Use a Blob column in REAL SQL Database and REAL Server Database to store an image as a string.

Data Type	Description
Boolean	Stores the values of TRUE or FALSE. "False" and zero are treated as the boolean value of FALSE and "True" and 1 are treated as the boolean value of TRUE. The value of other values are treated as undefined if retrieved via the DatabaseField.BooleanValue function. If you use DatabaseField.StringValue instead, it will attempt to return the original data if it can't be identified as boolean.
Currency	A 64-bit fixed-point number format that holds 15 digits to the left of the decimal point and 4 digits to the right.
Date	Stores year, month, and day values of a date in the format YYYY-MM-DD. The year value is four digits; the month and day values are two digits.
Double	Stores double-precision floating-point numbers.
Float	Stores floating-point numeric values with a precision that you specify, i.e., FLOAT (5).
Integer	A numeric data type with no fractional part. The maximum number of digits is implementation-specific. The REAL SQL Database supports 8-byte integers. Prior to REAL Studio 2006 Release 4, the REAL SQL Database supported only 32-bit integers. The DatabaseField.IntegerValue function will continue to work as it has but it can now return values up to 64 bits. However, the field type returned by Database.FieldSchema is now 19 rather than 3. If you are using this value, be sure to update your code. If you are using another data source, check the documentation of your data source.
SmallInt	A numeric data type with no fractional part. The maximum number of digits is implementation-specific, but is usually less than or equal to INTEGER. The REAL SQL Database supports 4-byte smallints. The Database.FieldSchema function returns 3 for SmallInts. If you are using another data source, check the documentation of your data source.
Text	Stores alphabetic data in which the number of characters can vary from record to record. The REAL SQL Database uses UTF-8 text encoding. If the text is not already in UTF-8, it is converted. If you want to preserve a different encoding, use a Blob column instead.
Time	Stores hour, minute, and second values of a time in the format HH:MM:SS. The hours and minutes are two digits. The seconds values is also two digits, may include a optional fractional part, e.g., 09:55:25.248. The default length of the fractional part is zero.

Data Type	Description
TimeStamp	Stores both date and time information in the format <i>YYYY-MM-DD HH:MM:SS</i> . The lengths of the components of a TimeStamp are the same as for Time and Date, except that the default length of the fractional part of the time component is six digits rather than zero. If a TimeStamp values has no fractional component, then its length is 19 digits. If it has a fractional component, its length is 20 digits, plus the length of the fractional component.
VarChar	Stores alphabetic data in which the number of characters can vary from record to record. The REAL SQL Database uses UTF-8 text encoding. If the text is not already in UTF-8, it is converted. If you want to preserve a different encoding, use a Blob column instead.

Values that are passed in single or double quotes (as literals) are stored as Text.

The DatabaseQuery Control

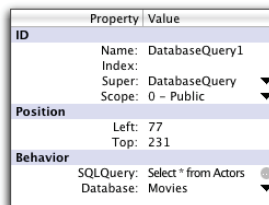
The DatabaseQuery control can be used to send queries to the data source, but this function can also be performed with the language, using the `SQLSelect` function of the Database class. It is up to you.

You add a DatabaseQuery control to a window like any other control, but it is not visible to the end-user. It is used only as an object that performs database queries. It has the following properties:

Name	Description
Database	The data source that will be queried.
SQLQuery	The text of the SQL query to be run against <i>Database</i>

The SQLQuery that you enter in the Properties pane is executed automatically when its window appears. For example, the properties shown in Figure 452 will retrieve all rows and columns from the *Actors* table when the window opens. However, the DatabaseQuery control cannot *display* the rows and columns all by itself.

Figure 452. A DatabaseQuery control's Behavior properties.

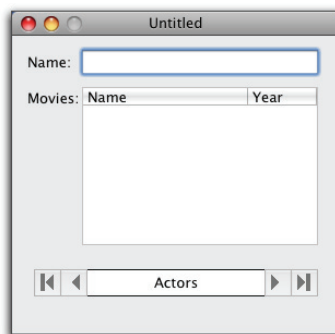


The DatabaseQuery control has one method, `RunQuery`, which executes *SQLQuery* against *Database*. You can call it via your code.

The DataControl Control

The DataControl control gives you a very simple and powerful way to create a data entry screen that works with a database table. The DataControl is a single object that consists of record navigation buttons (First, Last, Next, and Previous records) and a caption. When you place it in a window, it looks like this:

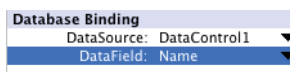
Figure 453. A DataControl in a window.



You use a DataControl by creating a window that uses TextFields, ListBoxes, Popup Menus, ComboBoxes, or StaticText controls to display and edit data. Custom classes based on these controls can also use a DataControl as their data source.

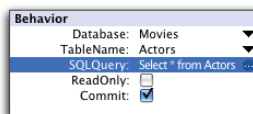
These controls have two properties that are meaningful only when used in conjunction with a DataControl: DataSource and DataField. They are shown in the Database Binding topic at the bottom of the control's Properties pane:

Figure 454. The Database Binding properties of a TextField.



The DataSource property should be assigned the name of the DataControl in the window. It, in turn, is bound to a database table using its Database and TableName properties. When you set the Database, a pop-up menu of tables from the database becomes available for the TableName property.

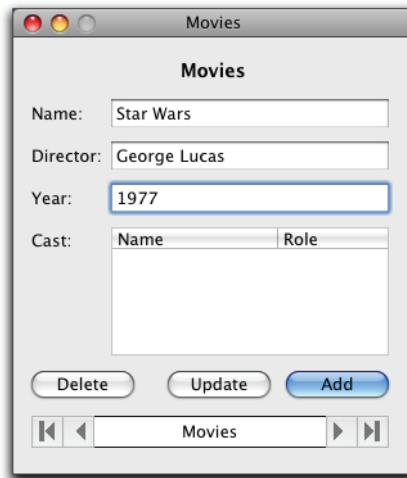
Figure 455. The DataControl's Behavior properties.



These properties specify that the DataControl will manage the records in the Actors table in the Movies database. The SQLQuery property is the SQL query that will run when the form is opened. This SQL statement finds all records in the Movies table.

The following window displays columns in TextFields and uses a DataControl for record navigation.

Figure 456. A simple database that uses a DataControl.



Each TextField is linked to the DataControl in its Properties pane by setting the DataSource and DataField properties.

The TextField for the movie name is bound to the column “Name” in the table to which the DataControl is linked. In this case, it is the Movies table.

Figure 457. The Database Binding properties for the “Name” TextField.



Each field on the form is bound to a column in the Movies table in this manner.

Creating a Database Front End Programmatically

To fully exploit REAL Studio’s database capabilities, you will need to write some code. The commands that are listed in the Database theme in the *Language Reference* provide you with all the necessary tools to build sophisticated database front-ends.

The Examples folder that ships with REAL Studio includes a fully worked out relational database example. The Orders Database is a Products/Orders/Customers relational database that illustrates the process of creating a REAL SQL Database. Please refer to that example and the accompanying documentation for more details on REAL SQL Databases.

Accessing a Data Source

The following classes allow you to choose the database back-end(s) used in your application. Except for the REAL SQL Database, each class requires that the appropriate database plug-in be installed in REAL Studio’s Plugins folder. All the data-

base plug-ins are available from REAL Software. Database plug-ins may be updated more frequently than REAL Studio itself; the latest versions of each plug-in can be found at <http://www.realsoftware.com>.

Data sources are accessed via a group of subclasses of the Database class. Each subclass accesses a specific data source. The database subclasses shipped with REAL Studio are shown in the following table. Third parties may also write plug-ins that support other data sources.

Table 38: Classes used to access data sources.

Database Subclass	Description
MySQLCommunityServer	Accesses MySQL Community Edition databases.
MySQLEnterpriseServer	Accesses MySQL Enterprise Edition databases.
ODBCDatabase	Accesses ODBC-based databases.
OracleDatabase	Assesses Oracle 8i and above databases.
PostgreSQLDatabase	Accesses PostgreSQLDatabase databases.
REALSQLDatabase	Built into REAL Studio and supports the REAL SQL Database. The REAL SQL Database data source is supported in all versions of REAL Studio.
REALSQLServerDatabase	The REAL Server is based on SQLite and is a multiuser database. It is available from REAL Software as a separate product to REAL Studio Professional customers. REAL Studio Studio includes REAL Server at no additional charge. The REALSQLServerDatabase class has its own SQL commands for managing the server. These commands are described in its own Users Guide.

Third-parties may supply plug-ins that work with other data sources. FrontBase can also be accessed from REAL Studio using a third-party plug-in. Contact FrontBase for more information.

Please see the entries for the new Database subclasses in the *Language Reference* for more information.

Creating a Database in Code

Creating the database in code is a little more involved than via the IDE but is more flexible. You can handle the case in which the application cannot find the database and create a new one automatically. You also can manage the user interface without the DatabaseQuery and DataControl controls.

Begin by creating an object of type REAL SQL Database (or REAL Server Database) as a property of the App class. In the Open event of the App class, an instance of the database is created and your code attempts to connect to it. You set the DatabaseFile property to the FolderItem that you expect to be the database file. If the connection

fails, the event handler calls a method that will create the database and its schema (tables and fields).

```
// Create database object
// myDB is a property of the App class
myDB = New REALSQLDatabase

// sets its databasefile property to the file..
myDB.databaseFile = GetFolderItem("myDatabase.rsd")

// try to connect to the database
If myDB.databaseFile.exists = True then
    // The database file already exists, so we want to connect to it.
    If myDB.Connect() = False then
        // there was an error connecting to the database
        // call a method to handle the error
        DisplayDatabaseError( False )
        Quit
        Return
    End if
Else
    // The database file does not exist so we want to create a new one.
    // The process of creating a database will establish a connection to it
    // so there isn't a need to call Database.Connect after we create it.
    // Call a method that creates the database and its tables...
    CreateDatabaseFile
End if

// Set the application to AutoQuit,
// so if all windows are closed then the application will quit.
App.autoQuit = True
```

Opening a Data Source

When opening a data source, you should test whether your connection is successful before proceeding with database operations. Use the `Connect` method of the `Database` class to determine whether the connection was successful. If it was, `Connect` will return a value of `True`. If the connection failed, you should examine the contents of the `ErrorMessage` and `ErrorCode` properties of the `Database` class.

For example, the following code opens a REAL Server database.

```
Dim db as REALSQLServerDatabase
db=New REALSQLServerDatabase
db.host="127.0.0.1"
db.port=4430
db.userName="Mary"
db.Password="Elton"
If db.Connect then
    //proceed with database operations
    If db.error then
        Beep
        MsgBox "Database Error: " + str(db.ErrorCode) + EndOfLine + _
            EndOfLine + db.ErrorMessage
    End if
Else
    MsgBox "Connection failed!"
End if
```

These methods return an object of type Database. Table 39 gives the Database class methods.

Table 39: Database Class methods.

Name	Parameters	Description
Close		Closes the database.
Commit		Commits (saves) changes to records. If you quit the application after making changes to a RecordSet, REAL Studio issues an implicit Commit. Use Commit and Rollback to manage transactions.
Connect		Connects to the database server and opens the database for access. Returns a Boolean. Before proceeding with database operations, do a test to be sure that Connect returns True.
FieldSchema	TableName as String	Returns a RecordSet with information about all columns in the table. See notes, below.
InsertRecord	TableName as String, Data as DatabaseRecord	Inserts <i>Data</i> as the last row of <i>TableName</i> .
Property	Name as String	Returns the database property specified by <i>Name</i> from the data source. This must be supported by the data source.
Rollback		Cancels a set of changes to records.

Table 39: Database Class methods. (Continued)

Name	Parameters	Description
SQLSelect	SelectString as String	The SQL Select statement. It returns a RecordSet. Call SQLSelect for any SQL command that returns a RecordSet.
SQLExecute	ExecuteString as String	The SQL Execute statement. Used to execute SQL commands that do not return a RecordSet.

When you use the language instead of a DatabaseQuery control, you call the SQLSelect method to query the database. Other SQL commands are set via the SQLExecute method. For example, the following type of statement is used to create a table in place of the interactive method discussed in the section, “Creating and Modifying Databases from the Project Editor” on page 607.

```
Dim db as REALSQLDatabase
db=New REALSQLDatabase

db.SQLExecute "create table Customers (FirstName varchar, "+ _
"LastName varchar, Address varchar, Email varchar, "+ _
"Company varchar, Phone varchar, Fax varchar, City varchar, "+ _
"State varchar, PostalCode varchar, "+ _
"ID integer NOT NULL PRIMARY KEY)"
```

Editing Records

You select the record or records you want to edit by passing the SQLSelect function a SQL SELECT statement as a string. It returns an object of type RecordSet. For example, the following selects all records in a table, returning all fields.

```
Dim rs as RecordSet
rs = db.SQLSelect( "SELECT * FROM Movies" )
```

In the terminology of SQL, a RecordSet is a database *cursor*. You may encounter the term cursor instead of recordset when you consult SQL reference material.

You can then display the RecordSet in a ListBox and/or edit them. See the section “Listing Records” on page 626 for a generic method that you can use to display a list of records. If you need to edit the rows, you must process the records one row at a time.

When an SQL statement returns a RecordSet, the user has “possession” of those records for his exclusive use. If the RecordSet contains more than one record, you can use the RecordSet’s properties and methods to cycle through the rows and columns of the RecordSet.

After a query using the SQLSelect method, the RecordSet contains a pointer to the current record (by default, the first record in the RecordSet). You can use the BOF (Beginning Of File) and EOF (End Of File) properties of the RecordSet class to

determine if the current record pointer is before the first record or beyond the last record. You can use these to determine if your query has found any records. If your query finds no records, both BOF and EOF will both be True since the current record pointer points at nothing.

If your query finds one or more records, both BOF and EOF will be False, since the current record pointer is pointing at the first record and not at the beginning or end of the file. For example, if your query found three records, the RecordSet would look like this:

BOF

Record1 (the current record pointer is pointing at this record)

Record2

Record3

EOF

As the record pointer moves to a particular record, you can use the Edit and Update methods to edit the record or the DeleteRecord method to remove the record. To modify the record, call the Edit method and perform the modifications.

To get or set the value of an individual field, use the “Value” property of the DatabaseField class that matches the data type you are working with. There are separate properties for each data type. They are listed below.

Name	Description
BooleanValue	Used to get and set the values of Boolean field types. When you read using BooleanValue, it requires that the value of True is coded as either 'True' or 1 and False is coded as either 'False' or 0. If this is not the case, using StringValue instead should get the original values in the column.
CurrencyValue	Used to get and set the values of Currency field types.
DateValue	Used to get and set the values of Date field types.
DoubleValue	Used to get and set the values of Double field types.
Int64Value	Used to get and set the values of Int64 field types. Support for 64-bit integers is at the framework level. Database plug-ins need to be updated to support 64-bit integers.
IntegerValue	Used to get and set the values of Integer field types.
JPEGValue	Used to get and set the values of Picture field types. The REAL SQL Database and the REAL Server Database do not support this field type. Use a Blob column to store pictures.
MacPICTValue	Used to get and set the values of Picture field types. The REAL SQL Database and the REAL Server Database do not support this field type. Use a Blob column to store pictures.
NativeValue	Used to get and set the values of fields in their native encoding. Useful for reading and writing blobs to and from database fields.

Name	Description
StringValue	Used to get and set the values of String/Character field types. If the field is not of this type, StringValue will try to return the value as a String. REAL SQL Database converts text to the UTF-8 text encoding.
Value	Used to get and set the value of a field of any data type. The properties that get and set the values of specific data types are recommended over Value. Set Value to Nil to set the field to NULL. The NULL value is currently supported in the PostgreSQL plug-in.

For example, code such as the following uses the StringValue property to assign the contents of a group of TextFields to the corresponding database fields.

```
// Update customer information
// Call RecordSet.Edit prior to this code
rs.Field("Company").StringValue = CustomerCompany.Text
rs.Field("FirstName").StringValue = CustomerFirst.Text
rs.Field("LastName").StringValue = CustomerLast.Text
rs.Field("Phone").StringValue = CustomerPhone.Text
rs.Field("Fax").StringValue = CustomerFax.Text
rs.Field("Email").StringValue = CustomerEmail.Text
rs.Field("Address").StringValue = CustomerAddress.Text
rs.Field("City").StringValue = CustomerCity.Text
rs.Field("State").StringValue = CustomerState.Text
rs.Field("PostalCode").StringValue = CustomerPostalCode.Text
// call RecordSet.Update after this code and then call Commit to commit
// the changes to the data source
```

After modifying the columns for a row, call Update to update the RecordSet. When you are finished, call the Database object's Commit method to commit the set of modifications to the database or call the Rollback method to cancel the modifications. The EOF property becomes True when you try to move the pointer past the last record.

If you don't call Commit, REAL Studio issues an implicit Commit when the user quits the application.

Conversely, you use these properties to get values from a record in the database. This code displays some fields in a record in a series of TextFields:

```
// Display the order record data
StatOrderNumber.Text = rs.Field("OrderNumber").StringValue
OrderOrderedOn.Text = rs.Field("DateOrdered").StringValue
OrderShippedOn.Text = rs.Field("DateShipped").StringValue
OrderPurchaseOrder.Text = rs.Field("PurchaseOrder").StringValue
OrderTaxRate.Text = Str(rs.Field("TaxRate").DoubleValue)
StatSubTotal.Text = Format(rs.Field("SubTotal").DoubleValue, _
    "$###,###,##0.00")
StatTotalTax.Text = Format(rs.Field("TotalTax").DoubleValue, _
    "$###,###,##0.00")
StatOrderTotal.Text = Format(rs.Field("Total").DoubleValue, _
    "$###,###,##0.00")
OrderCustomerNumber.Text = rs.Field("CustomerID").StringValue
```

See the examples for the RecordSet, DatabaseField, and Database classes in the *Language Reference* for more information. The separate Orders database example in your Examples folder walks you through the process of saving, modifying, printing, and importing/exporting records.

Listing Records

Most database front-ends include a List View of the data that uses a ListBox to display a selection of records. The easiest way to do this is to add a generic method to the window that contains the ListBox. The method should take as its parameters the ListBox that will display the data and the RecordSet that holds the records.

This Private window method, PopulateListBox, is called when the window opens and whenever the user does a search.

```
Private Sub PopulateListBox(lb as ListBox, rs as RecordSet)

// Populates the passed listbox with the data in the passed recordset
// This will loop through the records in the recordset and add rows
// to the listbox that contain the data in the recordset.

Dim i as Integer
// Clear the passed listbox
lb.DeleteAllRows

// Loop until we reach the end of the recordset
While Not rs.EOF
    lb.AddRow ""// add a new row to the listbox

    // Loop through all of the fields in the recordset
    // and add the data to the correct column in the listbox
    For i = 1 to rs.FieldCount
        // The listbox Cell property is 0-based so we need to subtract 1
        //from the database field number to get the correct correct column
        //number. This means field 1 is in column 0 of the listbox.

        lb.cell( lb.LastIndex, i-1 ) = rs.IdxFld( i ).StringValue
    Next

    rs.MoveNext // move to the next record
Wend

// If the listbox is set to be sorted by a particular column then we want to
// sort the listbox contents after we populate it, so that they appear in the
// correct order.
If lb.SortedColumn > -1 then // the listbox is sorted by a column
    lb.sort // sort the listbox data using the current sort settings
End if
```

Adding Records

You add a new record using the Database object's InsertRecord method. It has two parameters, the database object and an object of type DatabaseRecord.

Name	Type	Description
Column	Name as String	Column in current table.

For example, the following code adds a record to the “authors” table.

```
dim db as REALSQLDatabase
db=New REALSQLDatabase
.
dim rec as DatabaseRecord
rec = New DatabaseRecord
.
rec.Column("au_id")="09"
rec.Column("au_fname")="Oscar"
rec.Column("au_lname")="Wilde"
db.InsertRecord("authors",rec)
```

See the descriptions of the Database, RecordSet, and DatabaseRecord classes in the *Language Reference* for additional discussion and examples.

Storing Pictures

The REAL SQL Database uses the Blob field type to store pictures. One strategy is to write the picture data to a temporary file and then read it into a String variable as a BinaryStream. Then use the DatabaseField’s BlobColumn property to assign it to a Blob field.

This example takes an image stored in an ImageWell control and adds it to the current database record.

```
// Convert the picture to binary data that can be stored in the database.
// We do this by saving the picture to a temporary file and then
// reading it back in as binary data.

Dim imageData as String
Dim bs as BinaryStream

If productImageWell.image <> NIL then
    // Get a temporary file to save the image to
    f = SpecialFolder.Temporary.Child( "Temp_Image.jpg" )

    // Save the image out to the file
    f.saveAsJPEG productImageWell.Image

    // Open the file as a BinaryStream and read the data in
    bs=BinaryStream.Open(f,False)
    If bs <> NIL then
        imageData = bs.read( bs.length )
        bs.close
    End if

    // delete the temporary file if it exists
    If f.exists then
        f.delete
    End if
End if
```

You can then save the image by assigning the string to a Blob column:

```
rec = New DatabaseRecord
rec.BlobColumn("Image") = imageData

db.InsertRecord("products",rec)
```

Wouldn't it be great if every line of code executes just the way you want without a single error? Well, for those times when it doesn't work out that way, REAL Studio provides you with some tools to track down the bugs and fix them.

Contents

- What is debugging?
- Displaying the Debugger by setting breakpoints
- Watching your variables and properties
- Following the execution of methods
- Interrupting code execution at runtime
- Handling runtime exception errors
- Profiling your project
- Remote debugging

What is Debugging?

Debugging means removing errors, both logical and syntactical, from your programming code. Errors in programming code are referred to as “bugs.” You are probably wondering why errors are called “bugs.” Well, back in the 1940’s, the United States Navy had a computer that occupied an entire warehouse. At that time, computers used vacuum tubes and the light from the tubes attracted moths. These moths would get inside the computer and short out the tubes. Technicians would have to go in and remove the bugs to make the computer work again. Since this was a government project, everything had to be logged, so they would put down “debugging computer” in the log. But enough of the history lesson.

Debugging is part of programming. It’s the part of programming most programmers like the least. Fortunately, REAL Studio makes it easy to track down those nasty bugs and squash them like a, well, bug. REAL Studio comes with a Debugger which is a set of windows that help you see what is going wrong.

Logical Bugs

These are bugs in your programming logic. You will know you have found one of these when your code compiles but does not produce the results you were expecting. REAL Studio’s built-in Debugger can help you find these by letting you watch your code execute one line at a time.

Syntactical Bugs

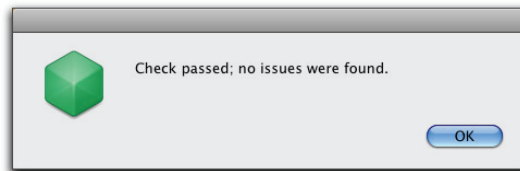
These are bugs where you have mistyped the name of a class, property, variable, or method. You may have also tried to use two values together that don’t go together. For example, if you try to assign a string value to a variable or property of type integer, you will get a Type Mismatch error because they are different data types.

Analyzing the Project

Since REAL Studio cannot compile a project that contains syntax errors, it offers the option of analyzing the project as a preliminary step. Choose Project ► Analyze Project or Project ► Analyze *Item*, where *Item* is the current item in the IDE window. Analyze Project checks for errors and other issues but does not compile the project. Some issues that it identifies are the use of deprecated items and ‘old style’ constructors, unused local variables and parameters, and type conversion issues.

If no errors or issues are found, it displays an alert box.

Figure 458. The Check Passed alert box.

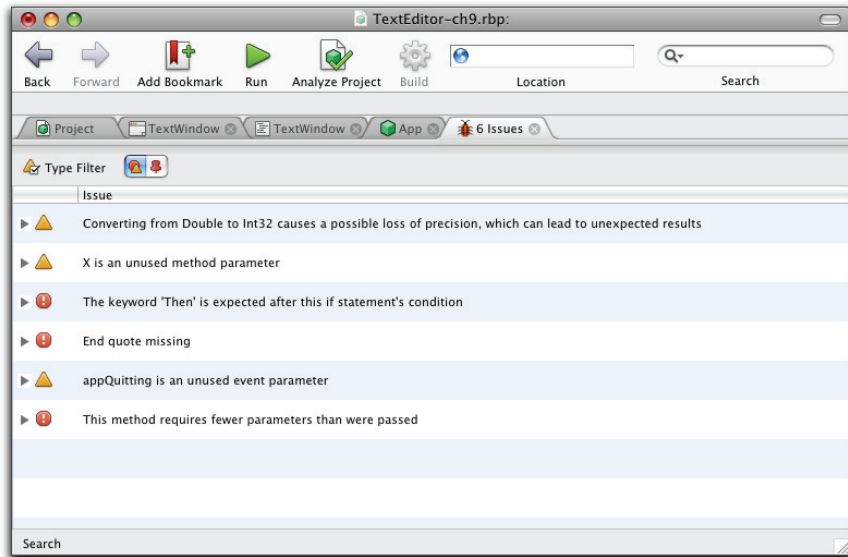


If it finds errors or issues, it opens an Issues screen. The Issues screen reports both the syntactical errors and issues in the same list. It reports one issue per line, even if

there are several issues in the same line. You may find that you need to rerun Analyze Project to catch additional errors in a line.

A typical Issues screen with both errors and issues is shown in Figure 459 on page 633.

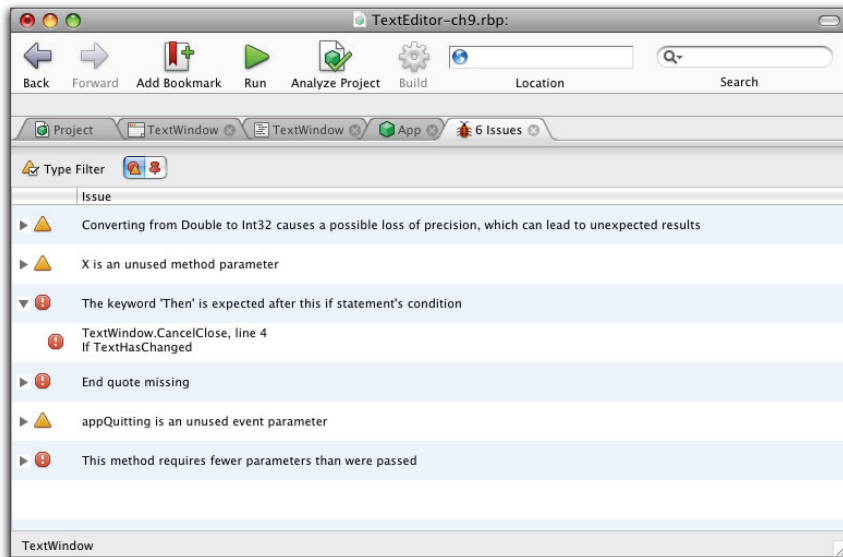
Figure 459. An Issues screen with both bugs and warnings.



Errors are identified in the left column by the alert icon and issues by the caution icon . You have to fix all of the syntax errors before you can compile the project; The issues are presented for your information and can help you optimize your code. You should take them under advisement and follow their recommendations when appropriate for your project.

In this view, the results are organized by error/issue. To the left of the icon, there is a disclosure triangle. This is because there may be several instances of an error or issue. Click the disclosure triangle to reveal the instances of that bug or issue.

Figure 460. The instances of a bug.



The expanded view shown in Figure 460 shows the expanded view of the “missing ‘Then’ keyword” error. It shows that there’s one instance of that type of error. The line of code containing the error is shown; double-click it to go to the Method Editor containing the bug.

Viewing the Issues screen by Method

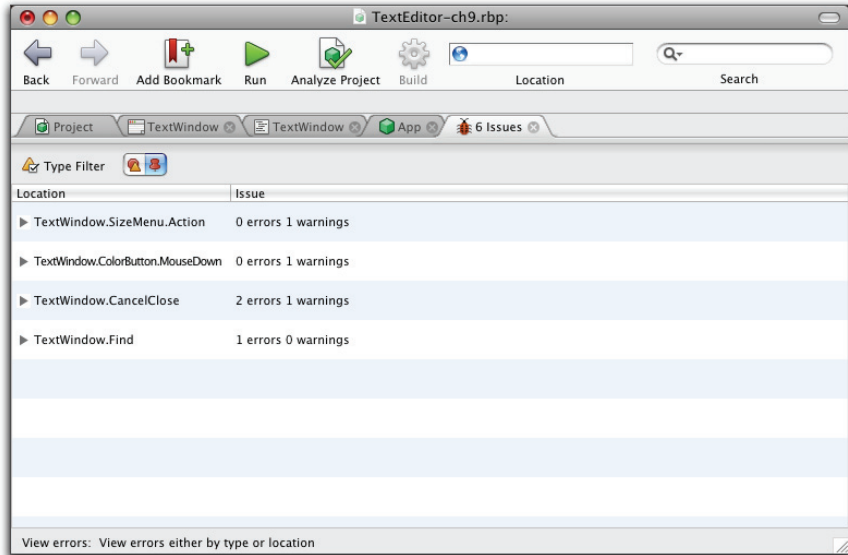
The default view of the Issues screen presents the items organized by error/issue. You can also view the items organized by method. In this view, each item is a method and the disclosure triangle reveals all the errors/issues that Analyze Project found in that method. You control the display via the icons in the Issues toolbar.

Figure 461. The Issues/Methods icons.



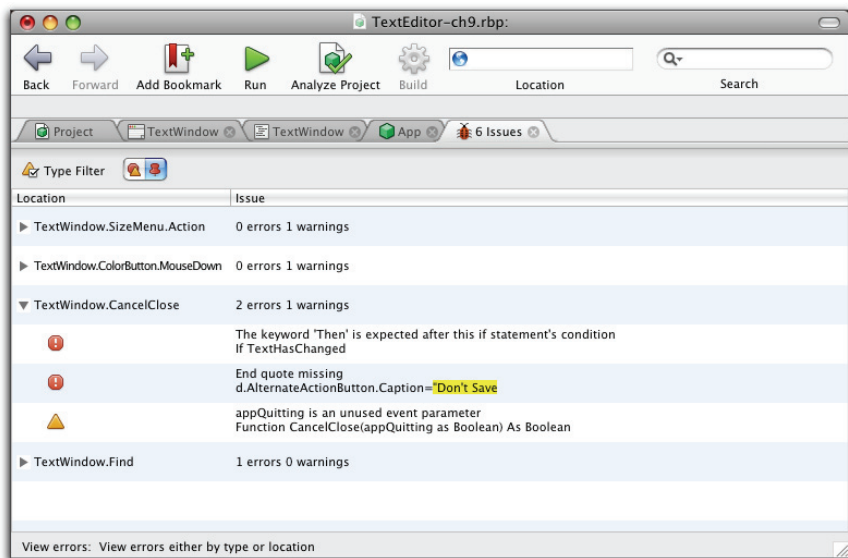
The errors/issues icon organizes the results as shown in Figure 460. The Pushpin icon denotes the listing by the method that contains the error/issue. Figure 462 shows the same results organized by method.

Figure 462. The method view of the Issues pane.



When you expand a method, you see all the errors/issues that were found in that method. Figure 463 shows the method that contain the “missing ‘Then’ keyword” error.

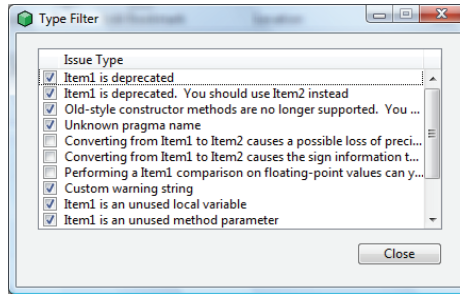
Figure 463. The expanded CancelClose method.



Filtering Types of Issues

You can control the types of issues that the Issues screen will report. Clicking the “Type Filter” button in the Issues toolbar will display a dialog box that lists all the types of issues that it will check. Only the selected issues are reported. If you don’t want to receive reports concerning a particular issue, deselect that issue.

Figure 464. The Type Filter dialog box.



For example, in Figure 459 the Analyze Project system is reporting that an Event Parameter is not used. This project does not have to use that parameter in order to function properly. If you don’t want Analyze Project to check for that issue, you can deselect the “Item1 is an unused event parameter” item. When a type that was deselected occurs, then the Issues tab contains the note “(hidden)”. But it does not list the error.

The Debugger

When you start a debug build, REAL Studio attempts to compile the application. If it finds no syntax errors, it launches the application in a new window and a Debugger panel called “Run” appears in the REAL Studio IDE. It enables you to halt execution of your application and execute your code line-by-line while watching and modifying the values of your variables.

There are three menu commands that initiate a debugging session: Run, Run Remotely, and Run Paused.

- **Run:** Run compiles the application and, if the compiler finds no errors, launches the built application in a new window on your machine. This is the normal way to use the debugger.
- **Run Remotely:** Run Remotely enables you to test your application on other platforms. REAL Studio builds the debug application, but launches it on the remote computer of your choice. Run Remotely is available only in the Professional and Studio versions of REAL Studio. For more information, see the section “Remote Debugging” on page 657.
- **Run Paused:** Run Paused builds the application and starts the debugger but it does not launch the executable automatically. If you have resources that you need to be in a specific location for the debug app to run, you need to use Run Paused instead of Run.

Run Paused allows you to debug the REAL Studio project but have an external entity responsible for launching the executable. You can then copy whatever resources your application needs into “the right location” in the debug app directory and then manually launch the executable. It will then connect to the debugger and you can continue to debug as usual.

It can be used for debugging REAL Studio plug-ins at the same time as debugging REAL Studio code.

Run Paused gives you a debug run that is more similar to what your release build will behave like as files and resources are located in the same places they will be in the final build.

The other choice you have is conditional compilation with different paths for the database in the debug build and the production version.

Breaking into the Debugger

While it is running, you can stop the application and monitor execution in the Debugger panel. There are several ways that REAL Studio halts the compilation process:

You Have Set A Breakpoint In Your Code

A *breakpoint* is a marker you can set at a line of code. It tells REAL Studio to stop execution and display the Debugger when it reaches that line of code. It stops just before it executes it. In the Code Editor, the lines that accept breakpoints are indicated by a dash in the first column, to the left of area that accepts code. You can set a breakpoint in the Code Editor simply by clicking on the dash. You can also add a breakpoint using the contextual menu. Place the insertion point on the line to which you want to add a breakpoint, right+click (Command-click on Macintosh), and choose Turn Breakpoint On from the contextual menu. You can set as many breakpoints as you wish.

In Figure 465 on page 638, a breakpoint is set. A red dot appears to the left of the line of code to indicate that a breakpoint has been set.

To remove a breakpoint, click on the red dot or place the insertion point in the line and chose Turn Breakpoint Off from the contextual menu.

Breakpoints are persistent. This means they will stay in your code until you remove them. You remove a breakpoint by clicking on the red circle. You can also clear all breakpoints by choosing Project ► Clear All Breakpoints.



NOTE: Breakpoints have no effect in stand-alone applications you build by clicking the Build button in the Toolbar or by choosing Project ► Build Application.

You use the Break Keyword

Instead of setting a break point in the Code Editor as shown in Figure 465, you can place the Break keyword in your code. When the compiler reaches this line of code, it breaks into the debugger as if a breakpoint had been set. With the Break keyword, you can set the breakpoint conditionally. For example, you if you want to break into the debugger only when the value of a variable is equal to a certain value,

you can put the Break keyword in an If statement that tests for that value. It's convenient to use the one-line version of the If statement, for example:

```
If ErrorNum=103 then Break
```

breaks into the debugger when a local variable equals 103.

You Have Pressed a Keyboard Equivalent

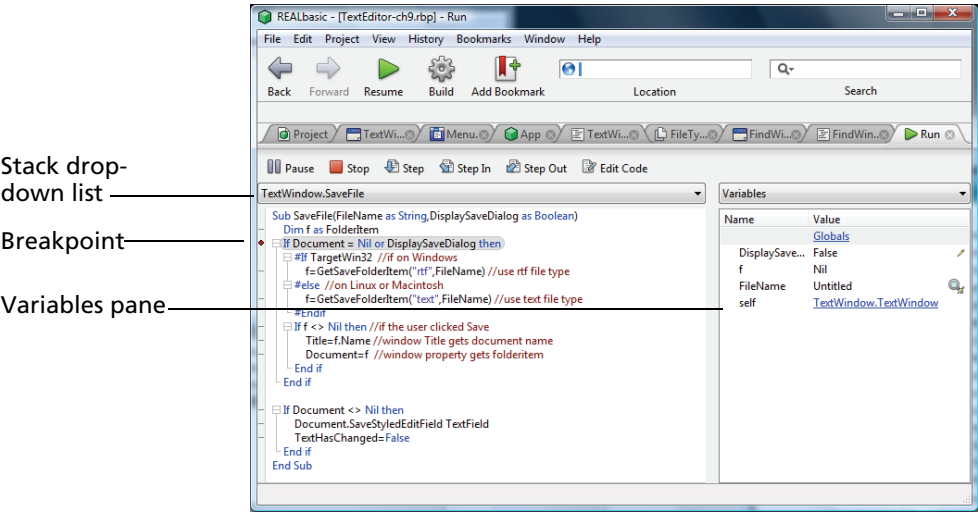
If you are running your project and you need to interrupt the code that is executing, press Control-C on Macintosh or either Ctrl+Break or F5 on Windows or Linux to halt code execution and display the Debugger. This is handy if you find yourself in an endless loop.

You can also switch back to the Development environment by clicking the Stop button in the Debugger Toolbar.

Command-Shift-Period does not work under Mac OS X.

In Figure 465, the line of code at the breakpoint is highlighted. Execution has stopped at the breakpoint. The line in gray is about to be executed.

Figure 465. The Debugger screen.



The Debugger Screen

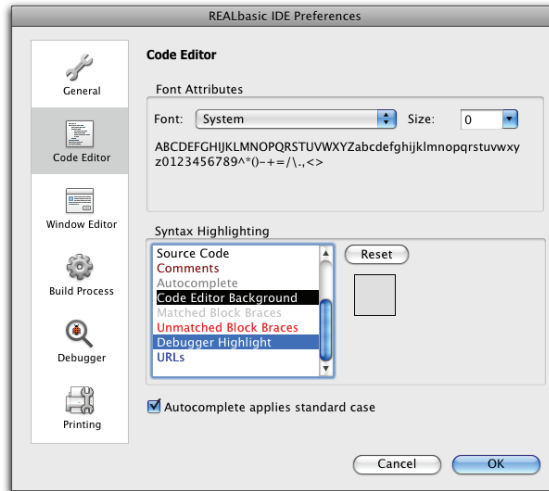
The Debugger screen is divided into two panes. The Code Editor pane shows the method that is currently executing and the Variables pane shows the variables local to the method displayed in the Code Editor pane.

The Stack drop-down list, just above the Code Editor, contains the name of the current method, along with the list of methods that eventually invoked the current method. They are listed in the order that they were called. You can check the Stack drop-down list to verify that methods are actually called when you expect them to be called. You can view the code for any method listed in the Stack drop-down list by clicking on it.

Changing the Highlight Color

The highlight color can be set in the Code Editor panel of the Options dialog box (Preferences on Macintosh). It is called “Debugger Highlight” and is listed in the Syntax Highlighting listbox. Highlight “Debugger Highlight” and then click on the color patch to the right of the list to modify it. A Color Picker will appear. When you select your color and close the Color Picker, the new color will be shown in the Syntax Highlighting groupbox.

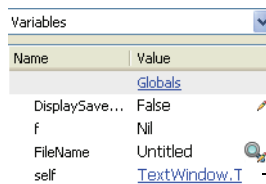
Figure 466. Changing the Debugger Highlight color.



The Variables Pane

The Variables pane contains a list of all the variables local to the method shown in the Code Editor area, along with their current values (if any).

Figure 467. The Variables pane.



Hyperlink to Object Viewers

Any objects (rather than variables) that are defined in the method are shown as hyperlinks rather than values.

Viewing and Changing the Values of String Variables

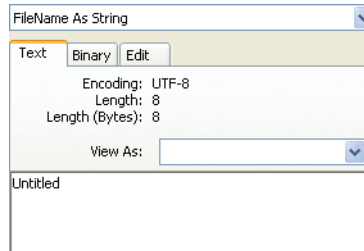
An important feature of the Debugger is that a variable's value can be changed while the application is running. You can change the value of most types of variables in the debugger. Exceptions are variants, enums, multiline strings, and array types.

A variable whose value can be changed has a Write icon (✎) to the right of its current value. A variable whose value can be changed via its Viewer has a combination magnifying glass/write icon (🔍✎). The magnifying glass indicates that it has a separate Viewer. Click it to display the Viewer for the variable. A variable whose value can be viewed in a Viewer but cannot be changed has a magnifying glass icon without the Write icon (🔍).

For example, In Figure 467 the DisplaySaveDialog boolean variable has a Write icon. Click on it to get pop-up that enables you to change its value to True.

The FileName variable has the Viewer/Write icon. Click it to display its Viewer. It is shown in Figure 468.

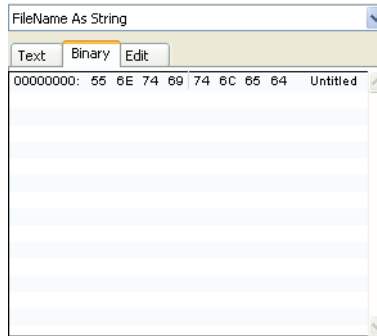
Figure 468. The Viewer for a String variable.



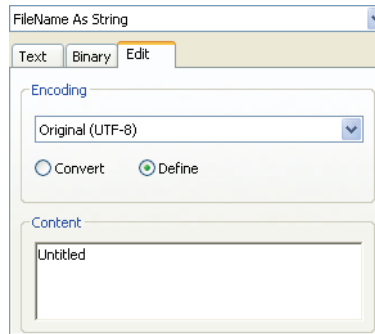
The Viewer for a string variable whose value can be changed has three tab panels, Text, Binary, and Edit. The Text panel displays the current text of the string. The second panel shows the text in Binary, and the last panel, Edit, enables you to edit the value.

In addition to the current value of the string, the Text pane also displays its encoding and length. The “View As” drop-down list enables you to change the encoding used in the display. Normally, a string will be stored with the UTF-8 encoding and would be readable to you. If the string somehow has a Nil encoding or the wrong encoding, you can tell the Debugger to use an encoding that is useful to you. This affects only the encoding used in the Viewer; it does not change the encoding of the variable itself.

The Binary panel displays the string in hex. It enables you to detect the presence of any non-printable characters that may be in the string. Ordinarily, REAL Studio stores strings in UTF-8 but you may get non-printable characters in a string if it represents raw data from a binary stream. For example, if the string has a null byte in the last position, it is easy to see it in the hex representation.

Figure 469. The Binary panel for a String variable.

The Edit panel displays the text of the string in editable format. You can also assign a text encoding to the string.

Figure 470. The Edit Panel for a String variable.

You can change its value by modifying or replacing the text in the Content field.


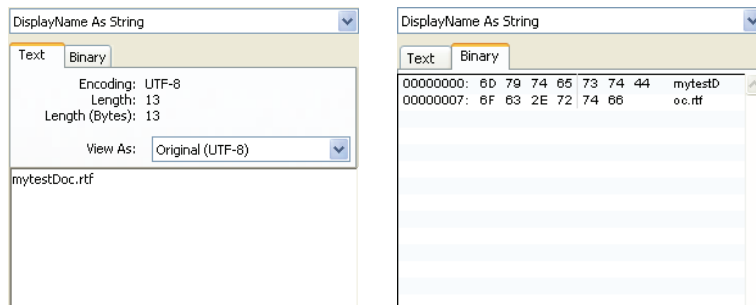
If a variable has the magnifying glass icon , then its Viewer lacks the Edit panel. You can view the Text and Binary panes.

Figure 471. The Viewer panels for a read-only variable.

Displaying Object Viewers

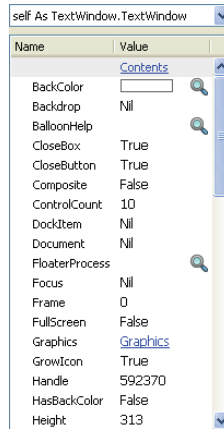
If you click a link in the Variables pane, it changes to show the current properties of the object you clicked. The top object, Globals, links to a pane that shows the values of your global variables, if any.

If you Shift-click (Command-click on Macintosh) on a hyperlink, the requested Object Viewer opens in a separate window. Otherwise, the new Object Viewer replaces the current Object Viewer in the IDE window.

You can also right+click (Control-click on Macintosh) on a hyperlink to display a contextual menu that lets you choose between opening the new Object Viewer in the IDE or in a new window. The latter is equivalent to Shift-clicking (Command-clicking on Macintosh) on the hyperlink.

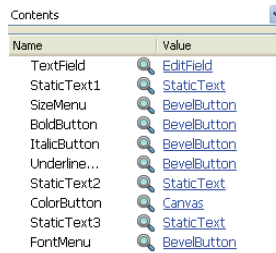
Figure 472, for example, shows the Object Viewer for an application's main window. You get this by clicking the TextWindow link in Figure 467.

Figure 472. The Variables Pane for TextWindow.



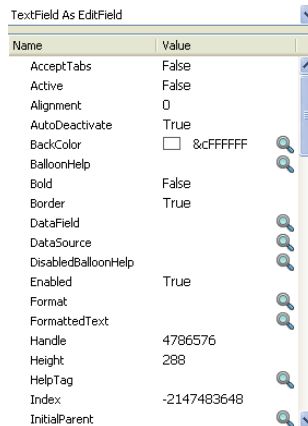
As you execute code line by line in the Debugger, the Object Viewer updates values in real time. In this way, you can see whether a particular value (or lack of a value) is causing the problem.

In Figure 472, the top link, Contents, displays a list of objects in the window. In this case, there are several buttons, a Canvas, and a TextField. Click the Contents link to examine these objects and click an object to display its variables.

Figure 473. The Contents of TextWindow pane.


Contents	
Name	Value
TextField	EditField
StaticText1	StaticText
SizeMenu	BevelButton
BoldButton	BevelButton
ItalicButton	BevelButton
Underline...	BevelButton
StaticText2	StaticText
ColorButton	Canvas
StaticText3	StaticText
FontMenu	BevelButton

Click a control to display its variables and objects, if any. In the case of TextWindow, several controls are contained within the window. They are listed on the Contents pane. For example, clicking the first link, TextField, shows its properties. As you drill down the hierarchy of objects, the pop-up menu above the variables pane enables you to jump back up to an earlier level in the hierarchy.

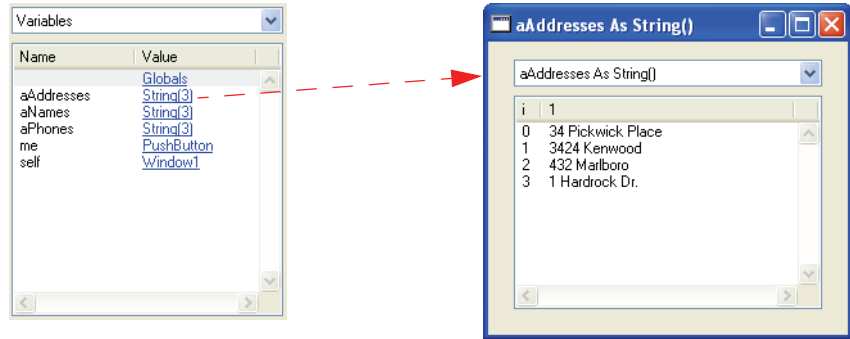
Figure 474. The Contents pane of TextWindow's Object Viewer and the Object Viewer of one of TextWindow's controls.


TextField As EditField	
Name	Value
AcceptTabs	False
Active	False
Alignment	0
AutoDeactivate	True
BackColor	<input type="checkbox"/> &cFFFFFF
BalloonHelp	
Bold	False
Border	True
DataField	
DataSource	
DisabledBalloonHelp	
Enabled	True
Format	
FormattedText	
Handle	4786576
Height	288
HelpTag	
Index	-2147483648
InitialParent	

This sequence is repeated down to the last level of objects. For example, if a window contains an ImageWell control, it is listed on the window's Contents pane. If the ImageWell has a picture assigned to its Image property, the picture is shown on the Contents pane of the ImageWell's Object Viewer.

If the variable that you are investigating is an array, the array's Object Viewer displays each value in the array and its index. For example, the Variables Object Viewer below shows that there are three arrays. Right+clicking on the aAddresses array to display its Object Viewer shows the values of each of its elements.

Figure 475. A Variables Object Viewer and the Object Viewer for one of its arrays.



Viewing the Values of Singles and Doubles

A Single or Double in the Variables pane can be displayed in one of three formats, Decimal, Scientific Notation, or Rounded. The default is Decimal.

To change the format, right+click on the value or the Write icon to display a contextual menu and then choose the desired format. When you click outside the field holding the value, the format changes.

Viewing the Values of Integers

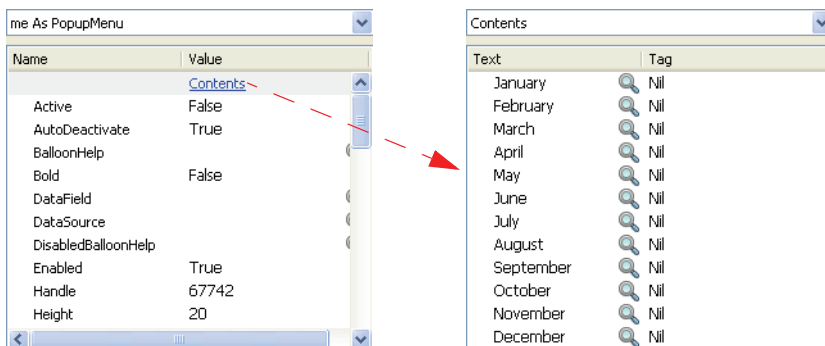
An Integer in the Variables pane can be displayed in one of four formats: Decimal, Hex, Binary, or Octal. The default is Decimal.

To change the format, right+click on the value or the Write icon to display a contextual menu and then choose the desired format. When you click outside the field holding the value, the format changes.

Viewing the Values of Pop-ups and ComboBoxes

A PopupMenu or ComboBox in the Variables pane will have a hyperlink to its Viewer. Click on the link to get the Viewer for the control and then click the Contents link at the top of the Viewer. It is a hyperlink to a viewer for the items in the PopupMenu or ComboBox. Both the text and the rowtag associated with the item are shown. This is illustrated in Figure 476.

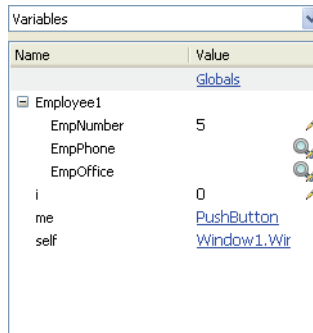
Figure 476. The Viewer for the items in a PopupMenu.



Viewing the Values in a Structure

A structure appears in the Variables pane as an expandable item. Click the plus sign or arrow to expose the fields in the structure.

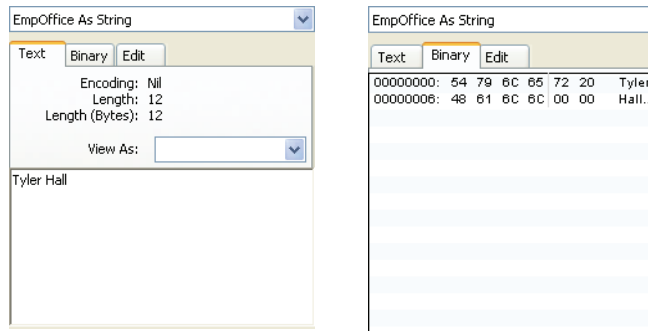
Figure 477. A structure in the Variables pane.



The values of fields can be changed in the debugger. If the value can be changed directly in the Variables pane, it will have the Write icon to the right. Select the current value and replace it.

String fields will have their own viewer just like the viewers in the section “Viewing and Changing the Values of String Variables” on page 640. Unlike string variables, string fields in a structure are of a fixed length. If there are fewer characters in the value than the declared length, then the unused characters are null. This can be seen in the Binary pane in the string field’s viewer.

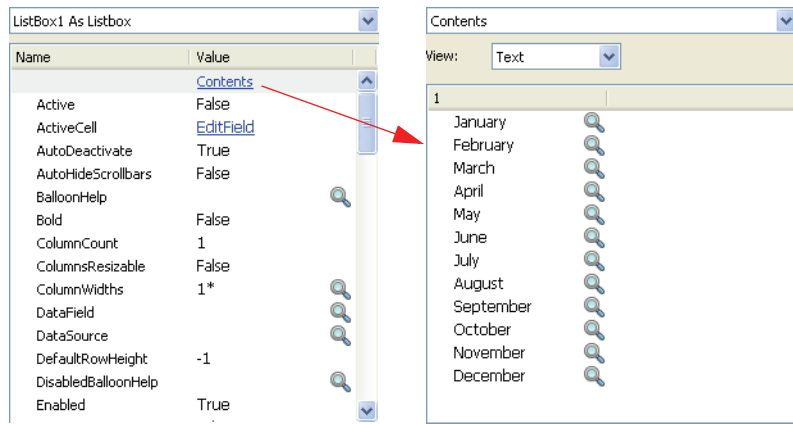
Figure 478. The Text and Binary panes for a string field.



Displaying the items in a ListBox

A ListBox appears in the Contents viewer of a window. Click it to get the Object Viewer for the ListBox and then click its Contents link to see the items in the ListBox.

Figure 479. The items in a ListBox in the Debugger.

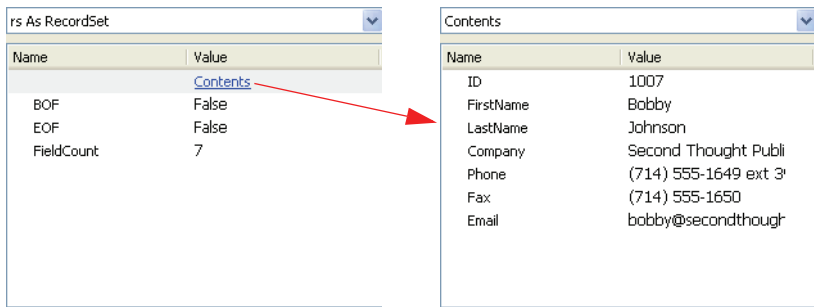


In the Contents pane, the drop-down list above the list of items gives you the choice of viewing the ListBox’s items or the tags associated with each cell. Choose Tag to change the view to the Tags.

Displaying the values of a RecordSet

A RecordSet appears in the Variables pane as an object. Click its link to get its Object Viewer and then click its Contents link to see the current values of the fields in the RecordSet.

Figure 480. A RecordSet in the Debugger.



Viewing Shared Properties

If you have a class with only shared properties (and possibly methods) and you don’t have any instances of that class created in your code but want to see the class shared properties in the debugger, put code such as this in a shared method:

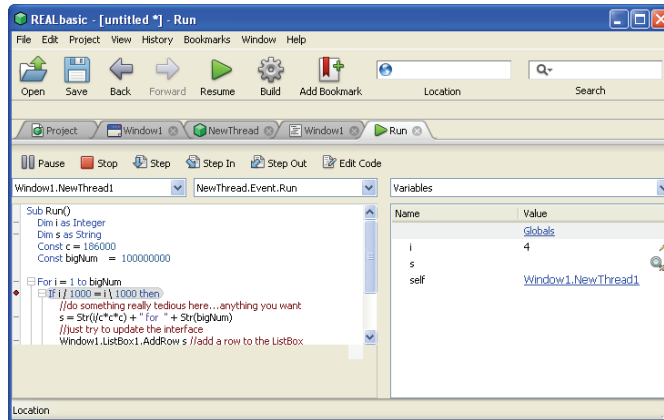
```
#If DebugBuild
Dim tmp as New ClassThatImTryingToDebugSharedMethodsIn
Break
#Endif
```

You now have an instance that you can easily see the shared properties.

Displaying Multiple Threads in the Debugger

If your application has more than one running thread, the Debugger displays an extra drop-down menu above the Code Editor area that allows you to choose the thread you want to watch. The new drop-down menu appears to the right of the Stack drop-down menu.

Figure 481. The Threads menu in the Debugger.

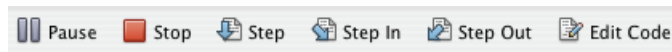


Controlling Execution

Using the Debugger's Toolbar, you can control execution. Instead of just allowing your code to run normally, you can control execution on a line-by-line basis.

The Debugger's Toolbar has five buttons that perform the functions of the Debug menu items:

Figure 482. The Debugger Toolbar.



When the Debugger is active the Project ► Step submenu offers the Over, Into, and Out commands as well. In addition, the Run button in the Main Toolbar has changed to Resume.

- **Resume:** (Main toolbar) Continues execution from the breakpoint line without further interruption. This exits from the Debugger screen.
- **Pause:** Pauses execution without stopping and exiting to the IDE.
- **Stop:** Stops execution and returns to the REAL Studio IDE. This also exits from the Debugger, but without executing any more code.
- **Step:** Executes the current line (indicated by the highlight) and moves on to the next line. If the current line includes one of your methods, the Debugger executes the method but will *not* step through the method's code.

- **Step In:** Executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger displays the method and steps through the method's code.
- **Step Out:** Executes the rest of the method without stopping on each line. This is handy when you have used Step In to step through a method that was called by another method and now wish to continue code execution without stopping on each line.
- **Edit Code:** Jumps to the Code Editor for the method that is currently executing. Use this button to modify code or correct errors in the current method. It does not close the Debugger tab.

Following the Execution of Methods

When your code isn't cooperating or you're just not sure what is executing and when, it's helpful to be able to watch your code as each line executes. The Debugger makes this easy. Once the Debugger is displayed, the line that's highlighted indicates which line of code is about to be executed. When you tell the Debugger to continue, it executes that line and goes on to the next line of code. What it does next depends on the command you give it when you wish to continue. The Debugger Toolbar and the Project menu give you three commands, each of which will execute the current line and then take a different course of action for the next line of code.

Step

Step executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger executes the method but will *not* step through the method's code. When the method is finished executing, the Debugger will continue from the next line of code in the current method. Consider the following code:

```
TextField1.SelBold=True  
TextField1.Text=ToFrench(TextField1.Text)  
TextField1.SelBold=False
```

Let's assume that "ToFrench" is a method that translates English to French. If you step through this code using the Step Over menu item, the second line of code is executed, but the Debugger won't display the code in the ToFrench method. It executes the ToFrench method and continues with the next line of code.

Step In

Step In executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger displays the method and steps through the method's code. When the method is finished executing, the Debugger returns to the calling method or event handler and continues with the next line of code.

Step Out

Step Out executes the rest of the method without stopping on each line. This is handy when you have used Step In to step through a method that was called by another method and now wish to continue code execution without stopping on each line. If you entered the current method or event handler using Step In, then stepping out executes the rest of the method and stops on the next line of code in the method that called the method you are stepping out of.

Tracking Method Execution with the Stack

A method or event handler can call another method or event handler which can call another one. This can go on for a while and you may need to keep track of the path of methods that were executed to get you where you are now. The Stack drop-down list does just that. When code execution begins (for example, when a PushButton is clicked), the Stack lists the PushButton's action event handler. If the action event handler calls a method, that method is added to the top of the list in the Stack when it's called. If that method calls another method, it is added to the top of the list. Once the current method finishes executing, it is removed from the list as REAL Studio returns to the method that called it. It's called the "Stack" because the methods are "stacked" one on top of the other in the order they were called.

If you need to see the code from a method or event handler called earlier in the stack, click on its name in the Stack drop-down list to display the code for that method in the Code Editor area of the Debugger screen.



NOTE: The larger the Stack gets, the more memory is being used. If you run out of memory it could be because your stack is so long that it takes up all the memory that has been allocated to the stack. The solution is try to make fewer method calls and use fewer local variables.

Watching Your Values

Part of debugging is monitoring the conditions under which certain lines of code execute. Another part of debugging is monitoring the values of variables, objects, and properties as your code executes. The Variables pane is used for these purposes. This pane displays any local variables, parameters, the current object, and its super class. It also displays global properties from modules and the Application subclass if there is one.

Local Values

The Variables pane displays the local variables with their current values or, in the case of object, links to their object viewers. Click a hypertext link to display an Object Viewer pane that shows the current values for all the properties of the object.

Negative numbers appear in red in an Object Viewer. Items that appear in blue underline are hyperlinks to their Object Viewers. They update as you step through your code and the code affects the values of properties.



NOTE: The Object Viewer currently only supports viewing single dimension arrays.

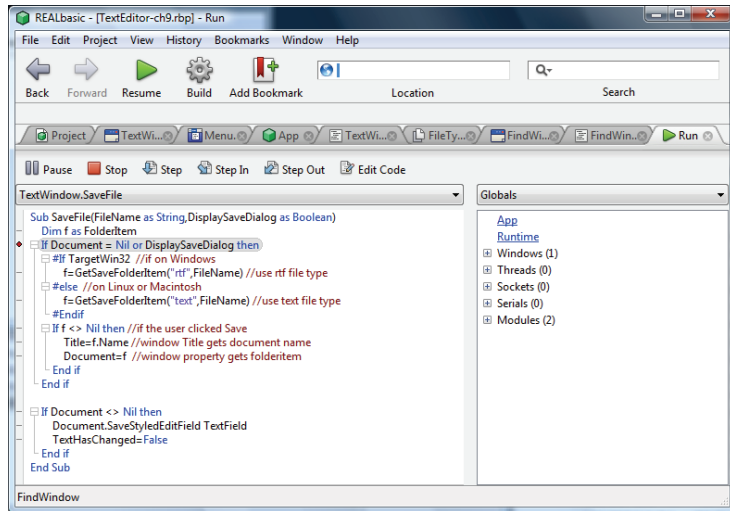
Parameters

If the Variables pane appears while REAL Studio is executing a method that is passed parameters, the values of those parameters are shown in the Variables pane when the method is called.

Global Values

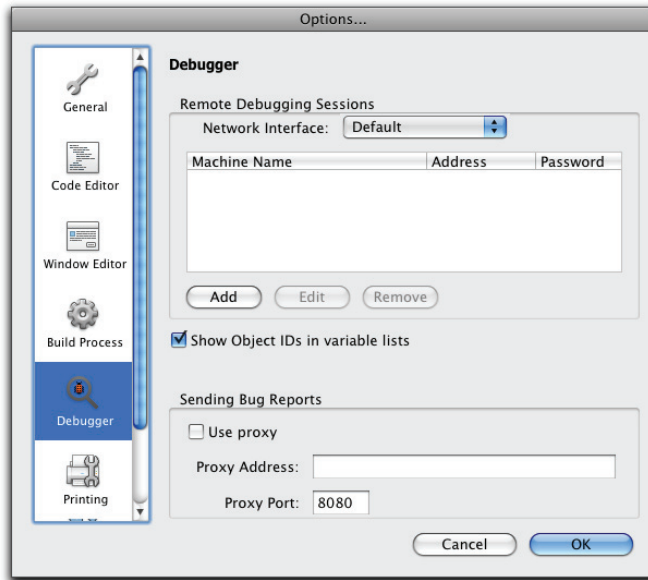
The top entry in the Variables pane is for global variables. It displays a viewer for items global to the application. This includes the App class, modules, threads, sockets, and windows. Any of these items that exist in the current application will have links to viewers for each item.

Figure 483. The Globals pane.

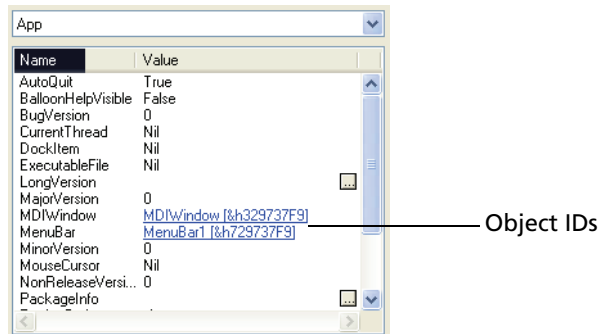


Object IDs

The Debugger can optionally display the internal object IDs within the Debugger. These IDs are used internally by the compiler and are not generally needed to debug REAL Studio applications. For that reason IDs are not displayed by default. To display the object IDs, choose Edit ► Options (Linux and Windows) or REAL Studio ► Preferences (on Mac OS X) and click on the Debugger icon in the browser area. The Debugger Preferences screen is shown in Figure 484.

Figure 484. Debugger Options.

Click the “Show Object IDs in Variable Lists” CheckBox and click OK to save your preference. When you use the Debugger, the object IDs will be shown in brackets following each object’s data type or class, as shown in Figure 485. It shows the viewer for the App class, which will be in the Globals pane.

Figure 485. The Debugger with the Show Object ID preference selected.

Starting and Stopping Your Project

You can switch back to the Development environment while your application is running in the Debugger by clicking the Pause button in the Debugger toolbar or choose Project ► Stop (Ctrl+K or ⌘-K). Click the Resume button in the toolbar to resume the application.

Runtime Exception Errors

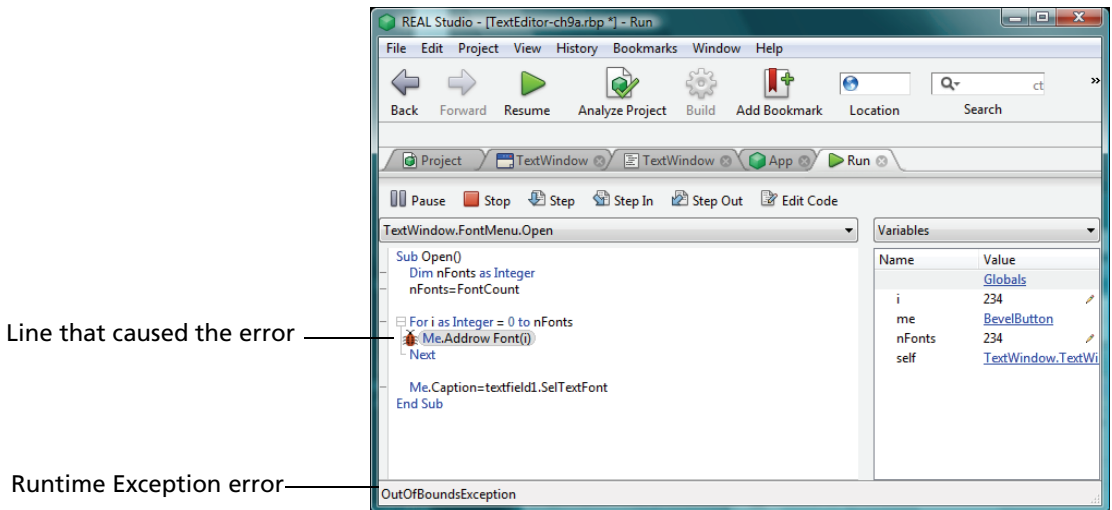
When you test your application in the IDE, REAL Studio performs basic syntax checking and, if it finds a problem, it stops compilation and alerts you to the problem.

There are certain types of errors that can be detected only at runtime, (i.e., when the line of code that contains the problem is actually executed). This is because these errors depend on values that are not known until the code is running. An application containing a *runtime error* compiles without error and may even run without problems for a very long time before the lines of code containing the error are actually called.

But when these lines are executed, REAL Studio has no choice but to stop execution. If you are testing the application in the IDE, the Code Editor will reappear and an error message will be shown below the line that contains the runtime error. An example runtime error message is shown in Figure 486.

To find and fix all such errors via the Debugger, you should select the Break on Exceptions item in the Project menu. With Break on Exceptions on, the Debugger will appear when a runtime error is encountered while you are testing your application. However, the unhandled exception will cause the built (standalone) application to quit.

Figure 486. An Unhandled Runtime Exception error in the Debugger.



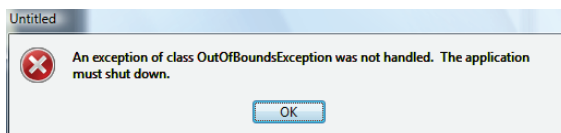
This error occurs when the value of the counter, *i*, reaches the value of *nFonts*. The programmer has forgotten that font numbering starts at zero rather than one; the loop should be from zero to `FontCount-1`. Until the value of *nFonts* is actually reached at runtime, the error cannot be detected. If the error is buried in an obscure method that is very rarely called, the application may survive many hours of testing without encountering the error.

As with syntax errors, the Tips bar contains the error message.

Runtime Errors in Standalone Applications

If the error occurs in a built application, REAL Studio displays a generic message box that identifies the type of problem that it encountered. An example message box is shown in Figure 487.

Figure 487. An Unhandled Runtime Exception Error in a Built Application.



The application will quit when the end-user accepts the message box.

Handling Runtime Errors

REAL Studio provides a way to detect and handle runtime errors that occur in standalone applications. There are several types of so-called *runtime exceptions* that you can detect in your code and handle. Please see the description of each Runtime Exception error in the *Language Reference* for more examples.

You can “catch” and handle runtime errors using the Try or Exception blocks. With either type of block, you can display a more informative message box that will help track down the problem.

Exception blocks always appear at the end of a method (not where you think the error might occur) because every line after the Exception line is considered part of the exception block.

When a block “catches” an exception error, the code in the Exception block runs, allowing you to obtain information on the location of the problem and the values that caused the error.

With an exception block, you can add code to your methods that handle the error in specific ways, depending on the type of error and the context in which it was encountered. This provides more control over the error handling process than the Break on Exceptions menu command.

In the Code Editor, the Exception line has the same level of indentation as the Sub or Function line. You can use **Exception** alone if you wish to handle any type of exception in the Exception block, as shown below:

```
Sub...
.
.
Exception
  MsgBox "Something really bad happened, but I don't know what."
```

The syntax of an Exception block is as follows:

```
Exception errorParameter As errorType
```

Both *errorParameter* and *errorType* are optional; *errorType* cannot be used without *errorParameter*.

The example shown above is sufficient to prevent the application from quitting, but the message is not very informative because you don't have a clue what type of exception occurred.

One way to test *ErrorParameter* is with an If statement in the Exception block:

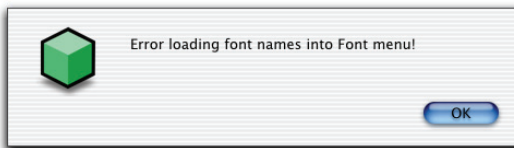
```
Sub...  
.  
.  
Exception err  
  If err IsA TypeMismatchException then  
    MsgBox "Tried to retype an object!"  
  elseif err IsA NilObjectException then  
    MsgBox "Tried to access a Nil object!"  
  .  
  .  
End if
```

For example, the runtime exception error shown in Figure 486 on page 652 can be handled with the following exception handler.

```
Sub Action()  
  Dim i, nFonts as Integer  
  nFonts=FontCount  
  for i=1 to nFonts  
    listBox1.addrow(Font(i))  
  next  
Exception err  
  if err IsA OutOfBoundsException then  
    msgBox "Error loading font names into Font menu!"  
  end if
```

When this method runs, it displays a message box such as shown in Figure 488:

Figure 488. A Handled OutOfBoundsException error.



When the end user accepts this message box, the application does not quit.

Instead of using multiple If statements, you can also use multiple Exception blocks, each of which handles a different runtime exception type:

```
Sub...
.
.
Exception err as TypeMismatchException
    MsgBox "Tried to retype an object!"
Exception err as NilObjectException
    MsgBox "Tried to access a Nil object!"
```

In case you fail to include an Exception block in the method in which the error occurs (or a method that calls the defective method), you have one last chance to “catch” runtime errors before they bring down a built application. This is in the UnhandledException event of the Application class. This event occurs if a runtime error occurs anywhere in the application that was not handled by an Exception block. This event is passed a parameter of type RuntimeException and you can write a function that handles the exception. For safety’s sake, you can write a generic routine that catches all runtime exceptions.

In this example, if the Action event shown here did not include an Exception block, you could catch the runtime error using the following function in an App class (a class derived from the Application class).

```
Function UnhandledException(error as RuntimeException) as Boolean
    If error IsA OutOfBoundsException then
        MsgBox "An OutOfBounds Exception error has occurred!"
    End if
    Return True
```

This method of handling runtime errors cannot provide the specific diagnostic feedback (such as Figure 488) because it will receive all OutOfBoundsExceptions throughout the application.

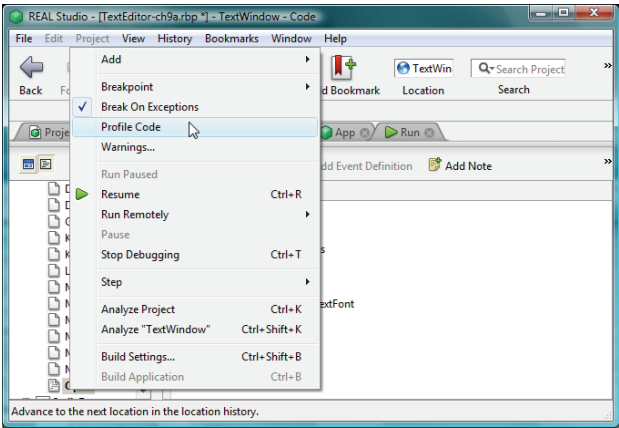
Profiling your Project

The Profiler is a service that monitors the built application while it is running. It measures the amount of time spent in each method, and it also reports how many times the method is called. With the Profiler, you can track down performance issues within your applications quickly without any additional coding.

The Profiler is available only in the Studio version of REAL Studio.

To use the Profiler you must enable it via the Project menu. Choose Project ► Profile Code to enable the Profiler. When enabled, a checkmark appears to the left of the menu item. If you don’t have the Studio version of REAL Studio, this menu command is not available.

Figure 489. Enabling the Profiler.

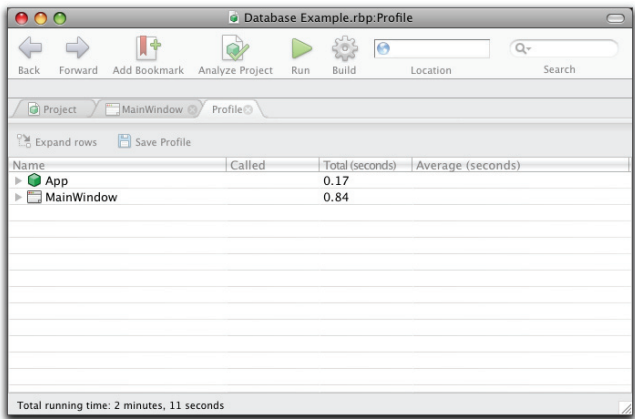


Once the Profile Code menu item is checked, the Profiler will automatically profile any debug application or build. Only code that is actually executed is profiled, and only user-written methods twill be profiled, not built-in methods.

Since the Profiler is an integrated feature of the IDE, the profile data that is gathered will be formatted and presented in the IDE when you debug your application. You have the option to also save this profile data in HTML or CSV (comma separated values) format. A “No Profile Data Was Gathered” error can occur if you are Remote Debugging (this is not yet supported), or if the debugging session ends prematurely. The application must exit properly for the Profiler to gather its data.

When you quit out of the debugger, the Profile screen appears. You can use the Expand Rows item to expand all the items in the profile or expand them individually.

Figure 490. The initial view of a profile.



The expanded view of an item lists all the methods belonging to the item and reports the number of calls, the total execution time, and the average time.

Figure 491. An expanded view of a profile.

Name	Called	Total (seconds)	Average (seconds)
MainWindow		0.84	
ProductsList.Open	1	0	0
CustomersList.Open	1	0	0
ProductImageWell.Open	1	0	0
OrdersList.Open	1	0	0
LineItemsList.Open	1	0	0
Open	1	0.02	0.02
UpdateOrderList	3	0.02	0.0067
EscapeSQLData	10	0.02	0.002
PopulateListBox	7	0.02	0.0029
UpdateCustomerList	2	0	0
UpdateProductList	2	0.02	0.01
EnableMenuItems	3	0	0
CustomersList.Change	4	0.03	0.0075
ClearCustomerFields	4	0	0
EnablePushbutton	19	0	0
SaveCustomer.Action	1	0.02	0.02
CheckMandatoryFields	2	0	0
ShowOrders.Action	1	0.02	0.02
OrderSearchBy.Change	1	0	0
OrderSearchFor.TextChanged	1	0	0
OrdersList.CellTextPaint	36	0	0
OrdersList.Change	1	0.02	0.02
ClearOrderFields	1	0.02	0.02
OrderTaxRate.TextChanged	2	0	0
CalculateSubTotalAndTax	2	0	0
UpdateLineItemsList	1	0	0
LineItemsList.CellTextPaint	20	0.02	0.001
LineItemsList.Change	1	0	0
LineItemProductName.TextChanged	2	0	0
ProductsList.CellTextPaint	69	0.02	0.0003
ProductsList.Change	2	0.55	0.275
ClearProductFields	3	0.02	0.0067
GetThumbnailForProduct	1	0	0
NewProduct.Action	1	0	0
SaveProduct.Action	1	0.02	0.02
MoneyToDouble	1	0	0
Close	1	0	0

Total running time: 2 minutes, 11 seconds

Click the Save Profile button in the Profiler toolbar to save the profile to disk in either HTML or CSV (comma separated values) formats.

The profiler feature also works when you build your application. In this mode the Profiler is compiled into your built application. When your application finishes executing, the profile is saved in CSV format as “Profile.txt” in the same directory as your application. The application must exit properly for the Profiler to generate this file.

Remote Debugging

Remote debugging enables you to test your application on a different computer from your development machine. For example, if you are developing under Mac OS X and need to test the application under Windows, you can do so from your Macintosh. Remote Debugging sends a debug build of the application to the remote machine and launches it automatically.

Without Remote Debugging, you would have to create a standalone Windows build on your Macintosh, move it to the Windows computer, and then launch it from that computer.

Remote Debugging is available in the Professional and Studio versions of REAL Studio. Also, Remote Debugging requires that you open additional ports in both computers' firewalls. This information is given in the section "About Firewalls" on page 663.

We'll call the computer on which the REAL Studio IDE is running the *development machine* and the remote computer on which you will be testing your application the *debugging machine*.

Remote Debugging works with a small utility application that runs on the debugging machine. It is called the *Remote Debugger Stub*. The Remote Debugger Stub is available as a separate download from the REAL Software web site.

You first configure the Remote Debugger Stub on the debugging machine. When you want to test the application on that machine, you use the IDE's Project ► Run Remote menu command instead of the Project ► Run command. If you don't have the Professional or Studio versions of REAL Studio, the Run Remote command is not available.

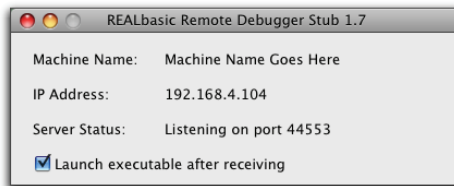


To configure the Remote Debugger Stub, do this:

- 1 Copy the Remote Debugger Stub application to a debugging machine and double-click it.**

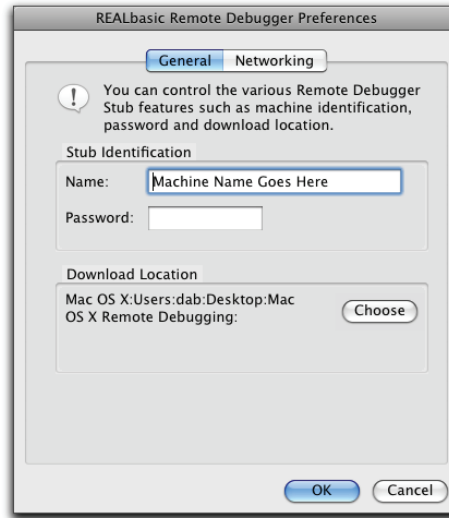
The Stub main screen appears. Initially, the machine name will be "Machine Name Goes Here".

Figure 492. The Remote Debugger Stub main screen.



- 2 Choose File ► Options (Windows and Linux) or Remote Debugger Stub ► Preferences (Mac OS X) to configure the Remote Debugger Stub.**

The configuration dialog box appears:

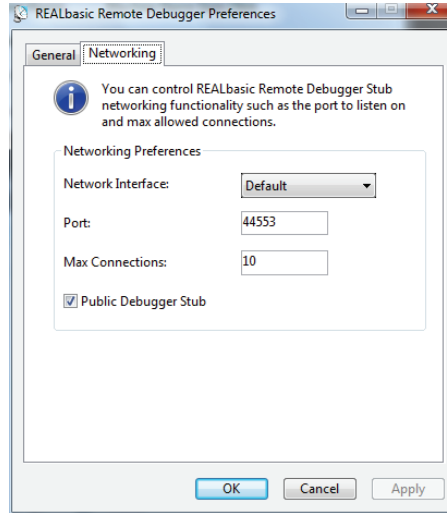
Figure 493. The Remote Debugger General Preferences screen.

It offers the following options:

- **Name:** The name of the debugging machine that will identify it on the development machine. It can be any name you like.
- **Password:** The password you will use from the development machine to access this machine.
- **Download Location:** The local directory where the application will be downloaded for testing. The default is the same directory as the stub.

The second panel of the Preferences screen allows you to set the listening port. Normally, you do not need to change this preference.

Figure 494. The Networking Options panel.



- **Network Interface:** The network interface that the stub will support. Most computers have only one network interface adapter; it is identified as “Default” and by its IP address. If the computer has more than one network interface adapter, then the IP addresses of each adapter will appear as menu items.
- **Port:** The port that all incoming TCP connections should be routed to. Most people will want to use the default value. If you have a firewall, this port may be blocked by default and you will need to modify the firewall to permit communications.
- **Max Connections:** The maximum number of connections that you will allow to this machine.
- **Public Debugger:** If selected, the machine can be auto-discovered by the development machine during the setup process. By default, the debugger stub is public. When it is public, the name of the machine appears in the remote debugging setup dialog automatically. If this option is not selected, the developer will need to enter the IP address of the machine in order to connect.
- **Public Debugger Stub:** If selected, the machine appears in the development machine. Otherwise, the developer can select it only by knowing the machine’s IP address.

The next step is to set up Remote Debugging on your development machine. Before you can use a remote machine for debugging, REAL Studio needs to be configured to send the debug application to the remote machine and launch it automatically.

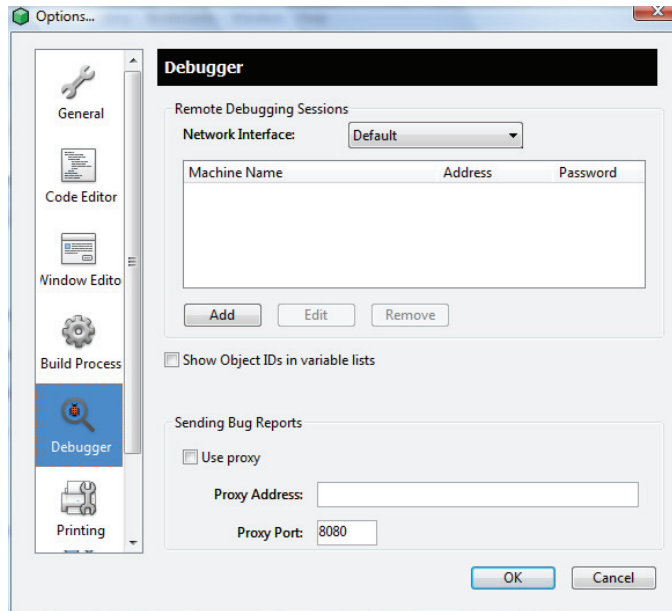
To configure the REAL Studio IDE for Remote Debugging, do this:

- 1 **Choose Project ► Run Remotely ► Setup.**



The Debugging panel of the Options dialog box appears. Alternately, you can open the Options dialog box directly (Preferences dialog on Macintosh) and navigate to Debugger options.

Figure 495. The Debugger Options panel.



The Remote Debugging Sessions Listbox will show all the debugging machines that have already been configured. The Network Interface drop-down list shows all the network interface adapters for the machine. Most computers have only one network interface adapter; it is identified as “Default” and by its IP address. If the computer has more than one network interface adapter, then the IP addresses of each adapter will appear as menu items.

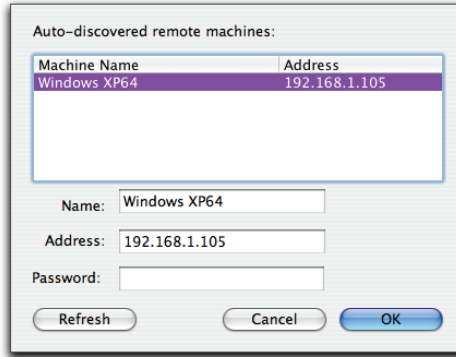
2 (Optional) If the default network interface adapter is not correct, select the correct network interface.

You then need to add the desired machines to this list.

3 Click Add to add a debugging machine to the list.

The Auto-discover dialog box appears.

Figure 496. The Auto-discover Remote Machines dialog.



In Figure 496, one computer is running the Remote Debugger Stub and has the Public Debugger Stub option enabled. It is identified by the Machine Name that had been entered into the Remote Debugger Stub configuration screen.

If you find that not all your remote debugging machines are listed, verify that the Remote Debugger Stub is actually running on those machines and that you have selected the Public Debugger Stub option. This is set by default on the Networking preferences screen. Also, double-check that the firewalls on both machines are configured properly. For more information, see “About Firewalls” on page 663.

4 (Optional) Click Refresh if a machine that you are expecting to be listed does not appear.

5 Highlight a debugging machine.

Its Machine Name and IP address appear in the Name and Address areas in the lower section of the dialog box. You can also add a debugging machine by typing its name and IP address into the entry areas but you may want to figure out why REAL Studio was unable to see it automatically.

6 Enter the password if the machine is password-protected.

7 If Public Debug Stub is off, enter the IP address and password of the machine.

8 Click OK to add the debugging machine.

When you return to the Debugging Preferences panel, it will appear in the Remote Debugger Connections ListBox.

9 If desired, repeat the process to add additional debugging machines.

When you are finished with the Debugging Preferences screen, the selected debugging machines are added to the Project ► Run Remotely submenu.

The most recently used debugging machine will be at the top of the menu and get the keyboard shortcut.



To debug on a remote machine, do this:

1 Verify that the Remote Debugger Stub is running on the desired debugging machine.

You should see the message “Listening on Port XXXXX.” If not, you may need to configure the firewall on that machine.

2 Choose the desired debugging machine from the Project ► Run Remotely ► *DebugMachineName* menu.

REAL Studio creates a debug build for the correct platform, sends it over the network to the debugging machine and launches it automatically. As it is doing so, a progress indicator on the development machine gives you the status of the operation.

3 Reorient yourself in front of the debugging machine and test the application normally.

If you have put a breakpoint in your code or otherwise break into the Debugger, it will appear on the development machine.

When you are finished debugging, you need to quit out of the debug build before resuming work in the IDE on the development machine. You can do so on the debugging machine or, on the development machine, click the Stop button in the Debugger or choose Project ► Stop Debugging.

About Firewalls

If you are behind a firewall, the default ports you need to open are as follows:

Table 40: Development Machine.

44553	UDP
44553	TCP
13897	TCP

Table 41: Debugging Machine.

44553	UDP
44553	TCP

You can change the ports communicated on by the stub (port 44553) in the Networking panel of the Options dialog box. In the IDE, you can connect to the new port by specifying the address as “X.X.X.X:Port” (Figure 496 on page 662). Please note that if you change the default port, the stub can no longer be automatically discovered.

In order for remote debugging to work, the selected UDP and TCP ports must be opened in the OS’s firewall or any third-party firewall applications that are running.

Communicating With The Outside World

Some applications need to communicate with other applications or even serial hardware devices to exchange information. Sometimes this is done automatically while other times it is initiated by the user. For example, when you use your computer to connect to the Internet, you are initiating communications between an application on your computer and an application on another computer at your Internet Service Provider (ISP). Fortunately, REAL Studio provides controls that make communications between applications on different computers, and even communications between a computer and a serial hardware device easy.

Contents

- Communicating with serial devices
- Communicating with other computers via TCP/IP
- Communicating with other computers via UDP

Communicating With Serial Devices

A serial device is a device that communicates by sending and/or receiving data in serial. This means that it is either sending data or receiving data at any one moment. It doesn't send and receive at the same time. The most common serial device is a modem. Some printers are serial devices. Serial communications using REAL Studio are done with the *Serial* control. To communicate with a serial device you configure a Serial control, open the serial port to make the connection, read and/or write data to and/or from the serial device connected to one of your serial ports, and finally close the serial port when you are through to disconnect from the serial device.

Getting Set Up

So the first step is to place a Serial control in one of your project's windows or instantiate a Serial object using code. Before you can begin communicating with a serial device using a Serial control, you need to set up the Serial control so that it will know which serial port your serial device is connected to. You will also need to set the speed at which communications will occur, as well as a few other settings. This can all be done at design time using the Properties pane or at runtime using code.

How you configure the Serial control's behavior properties will depend on what the serial device is expecting. Some devices can only communicate with one specific configuration. Other devices (like modems) can communicate using many different configurations. In the case of a modem, you will not only have to consider what configurations the modem will accept but also what configuration the modem your modem will be connecting to will accept. The Serial control's default configuration should work for most modems. You may need to change the default configuration for other serial devices.

Opening the Serial Port

Once you have configured the Serial control, you can open the serial port to initiate communications with the serial device. This is done by calling the Open method of the Serial control. This method is, in fact, a function that returns True if the connection is opened and False if it is not. For example, suppose you have a Serial control whose name is "Serial1." To open the serial port using this control, you can use the following code:

```
If Serial1.Open then  
    MsgBox "The serial port was opened."  
Else  
    MsgBox "The serial port could not be opened."  
End if
```

Once you have successfully opened the serial port, it will be unavailable to all other applications (and in fact, to other Serial controls as well) until it is closed.

Reading Data

When the serial device sends data back to the Serial control that is connected to it, the Serial control's DataAvailable event handler executes. The data that has been

sent back goes into a place in the computer's memory called a *buffer*. The buffer is simply a place to store the data that has been sent by the serial device because most serial devices don't have much memory of their own. When new data arrives in the buffer, REAL Studio executes the `DataAvailable` event handler of the Serial control.

In the `DataAvailable` event handler, you use the Serial control's `Read` or `ReadAll` methods to get some or all of the data in the buffer. Both of these methods act as functions. Use the `Read` method when you want to get a specific number of bytes (characters) from the buffer. If you want to get all the data in the buffer, use the `ReadAll` method. In both cases, the data returned from the buffer is removed from the buffer to make room for more incoming data. If you need to examine the data in the buffer without removing it from the buffer, you can read the data from the Serial control's `LookAhead` method.

This example appends any incoming data to a `TextField`:

```
Sub DataAvailable()  
    TextField1.AppendText Me.ReadAll()
```

You can clear all data from the buffer without reading it by calling the Serial control's `Flush` method.

Both the `Read` and `ReadAll` methods of the Serial class take an optional parameter that enables you to specify the encoding. Use the `Encodings` object to get the desired encoding and pass it as a parameter. For example, the code above has been modified to specify that the incoming text uses the ANSI encoding, a standard on Windows:

```
Sub DataAvailable()  
    TextField1.AppendText Me.ReadAll(Encodings.WindowsANSI)
```

You may need to specify the encoding when text is coming from another platform or is in another language. For information about text encoding, see the section “Working with Text Encodings” on page 416.

Writing Data

You can send data to the serial device at any time as long as you have opened the serial port with the Serial control's `Open` method. You send data using the Serial control's `Write` method. The data you wish to send must be a string, as the `Write` method accepts only a string as a parameter.

The `Write` method is performed asynchronously. This means that the next line of code following the `Write` method can already be executing before all the data has actually been sent to the serial device. If you need your code to wait for all data to be sent to the serial device before continuing, call the Serial control's `XmitWait` method immediately following a call to the `Write` method.

Changing a Serial Control's Configuration on the Fly

There may be times when you need to change a Serial control's behavior properties while the serial port is open. While you can change these properties, the changes won't take effect until you close the serial port and reopen it. If you need the behavior properties to update immediately, call the Serial control's Poll method. This updates all properties immediately and calls the DataAvailable event handler immediately if there is any data waiting in the buffer.

Closing the Port

Once you are finished communicating with a serial device, you must close the serial port to end the communications session and make the port available to other Serial controls or other applications. To close the serial port, call the Close method of the Serial control that opened the serial port.

Communicating With Modems

Modems have a set of commands you can send them to tell the modem to do things such as dial a particular number. Most of these commands are the same for every modem. Your modem probably came with a guide that lists its commands. Consult that guide for more information.

Communicating with USB and FireWire Devices

These devices use a much higher-level API that requires drivers for interaction. USB doesn't work like serial and requires its own support. Some USB devices have a chip in them that makes them appear as a serial device. Typically this chip is an FTDI chip. If the device has this chip, you can communicate with the device via the SerialPort class. If that does not work for you, then the Monkeybread plug-in has USB support for a handful of specific types of devices.

USB wraps up several things into one:

- A cable interconnect specification (what the cable connectors need to look like),
- A low-level packet oriented protocol, so the OS can figure out what driver to load and talk to the USB device to identify it,
- A vendor API, i.e., HID device like a mouse, keyboard, or mass storage device.

Just because something uses a USB cable doesn't mean you can talk to it.

Some device types are very common so the OS vendors have the drivers built in. This includes HID devices (e.g., mice and keyboards) and mass storage devices like hard disks. If it's one of those, they will work without any additional work.

Almost anything else requires a driver from the vendor. If you'd like to control such a device, you will need to obtain a shared library from the manufacturer or write your own. Try to contact the manufacturer to see if they have a library. If the manufacturer has a library for USB communications, you can talk to the library from REAL Studio using Declare statements.

TCP/IP Communications with the TCPSocket Control

Sometimes applications need to communicate with other applications on the same network. This can be accomplished using the REAL Studio's *TCPSocket* control. The *TCPSocket* control can send and receive data using TCP/IP.



In versions of REAL Studio prior to 5.0, TCP/IP communications were handled with the *Socket* control. The *Socket* control has been superseded by the *TCPSocket* control.

The *TCPSocket* control is derived from the *SocketCore* class. The *SocketCore* class handles the core functionality that both the TCP and UDP protocols provide (the UDP protocol is described in the section “UDP Connections with the *UDPSocket* Control” on page 677). The *SocketCore* class is an abstract class that cannot be instantiated. The *SocketCore* class has properties and methods that are inherited by the *TCPSocket*, *ServerSocket*, and *UDPSocket* controls. If you have used the *Socket* control in your projects, you should update to the *TCPSocket* control.

TCP/IP is the protocol of the Internet. It's the way most data is transmitted via the Internet. In fact, the “IP” in TCP/IP stands for “Internet Protocol.”

The *TCPSocket* control can be used to communicate with other computers on the same network. When you connect to the Internet, you are part of the Internet network. This allows you to communicate with other computers on the Internet via TCP/IP.

Getting Set Up

You can either add a *TCPSocket* control to a window or instantiate a *TCPSocket* object using code. Before you can connect to another computer using the *TCPSocket* control, you must first set the *port*. The port is to TCP/IP what channels are to television or frequency assignments are to radio stations. Ports give an application the ability to focus on specific data rather than receiving all the data transmitted to your computer via TCP/IP. This allows you to browse the web and send email at the same time because the web uses one port and email uses another. The port is represented by a number and there are thousands of available ports. Some have already been designated for specific functions like web browsing, email, FTP, etc. If you are designing an application that will need to communicate with another application, you will need to find out what port the other application is using. For example, if the other application is an SMTP server, it's probably using port 25 since that is the port that is reserved for SMTP (Simple Mail Transfer Protocol).

A *TCPSocket* control has a *Port* property (inherited from the *SocketCore* class) that can be assigned at design time or runtime but it must be assigned a value before you can connect to another computer. If you plan on initiating the connection, you must also assign the IP address of the computer you wish to connect to the *Address* property of the *TCPSocket* control that will make the connection.



Mac OS X and Linux have a built-in restriction regarding port numbers. Ports below 1024 cannot be assigned by a user who is not running with “root” privileges. Mac OS X is configured so that a user cannot gain root privileges via the graphic

user interface. Most users run with Admin privileges—not root—so you should use ports above 1024 for normal TCP/IP communications with Mac OS X computers because the `TCPSocket` cannot access port numbers below 1024. This is not a bug but a security feature that is built into these operating systems.



NOTE: A `TCPSocket` control can only be connected to one application at a time. If you need to maintain multiple connections simultaneously, you should use the `ServerSocket` control, which is designed for this purpose. See the section “Handling Multiple Connections with the `ServerSocket` Control” on page 675.

Making a Connection to Another Computer

Once you have assigned a port and an IP address, you can connect to an application on the computer at that IP address, provided that the application is listening for TCP/IP connections on the port you have specified. To initiate a connection, you call the `TCPSocket` control’s `Connect` method. A connection is not necessarily established immediately after you call the `Connect` method. There may be network connection issues that take some time to resolve before the connection is actually established.

There are two ways to determine that you are connected. You can either wait until you receive a `Connected` event from your `TCPSocket` or test the return value of the `IsConnected` method of the `SocketCore` class. If you don’t wait for this feedback, you either cause the connection process to halt, resulting in a lost connection error (102), or an out of state error (106).

When the connection is established, the `TCPSocket` control’s `Connected` event handler executes. If a connection is not established, an error occurs and the `TCPSocket`’s `Error` event handler is executed.

Once a connection is established, your application can begin sending and receiving data with the application at the other end of the connection.

Listening For a Connection From Another Computer

In some cases you may want your application to wait for another application to connect to it rather than initiate the connection. To do this, you use the `TCPSocket` control’s `Listen` method. For example you have a button that when pressed, causes the application to listen for a TCP/IP connection on the port number that is assigned to the `TCPSocket`’s `Port` property. Let’s assume that the `TCPSocket` control is named “`TCPSocket1`.” In the `PushButton`’s `Action` event handler, you use the following code:

```
Sub Action()  
    TCPSocket1.Listen
```

Once a connection is established, the `TCPSocket` control’s `Connected` event handler executes, letting you know that you have a connection.

Reading Data

When the application at the other end of the connection sends data back to the `TCPSocket` control that it’s connected to, the `TCPSocket` control’s `DataAvailable`

event handler executes. The data that has been sent back goes into a place in the computer's memory called a *buffer*. The buffer is simply a place to store the data that has been sent by the other application. When new data arrives in the buffer, REAL Studio executes the `DataAvailable` event handler of the `TCP Socket` control.

In the `DataAvailable` event handler, you can use the `TCP Socket` control's `Read` or `ReadAll` methods to get some or all of the data in the buffer. Both of these methods act as functions. Use the `Read` method when you want to get a specific number of bytes (characters) from the buffer. If you want to get all the data in the buffer, use the `ReadAll` method. In both cases, the data returned from the buffer is removed from the buffer to make room for more incoming data. If you need to examine the data in the buffer without removing it from the buffer, you can read the data from the `TCP Socket` control's `LookAhead` property.

This example appends any incoming data to a `TextField`:

```
Sub DataAvailable()  
    TextField1.AppendText Me.ReadAll()
```

When you are reading text from an outside source, you may need to specify the text encoding. The text encoding is the scheme that maps each letter in the text to a numeric code. If the text comes from another application, operating system, or is in another language, you may need to tell REAL Studio which encoding was used. For more information on text encoding, see the section “Working with Text Encodings” on page 416.

Both the `Read` and `ReadAll` methods of the `TCP Socket` class take an optional parameter that enables you to specify the encoding. Use the `Encodings` object to get the desired encoding and pass it as a parameter. For example, the code above has been modified to specify that the incoming text uses the ANSI encoding, a standard on Windows:

```
Sub DataAvailable()  
    TextField1.AppendText Me.ReadAll(Encodings.WindowsANSI)
```

From then on, REAL Studio stores the encoding with the text. The text will display and print properly and string operations will work as expected.

Writing Data

You can send data to the application you are connected to at any time. You send data using the `TCP Socket` control's `Write` method. The data you wish to send must be a string, as the `Write` method accepts only a string as a parameter. In this example, the text from a `TextField` is being sent via a `TCP Socket` control:

```
TCP Socket1.Write TextField1.Text
```

If you need to send the text to an application that is expecting a specific text encoding, then you should convert the text to that encoding prior to sending it. Use the `ConvertEncoding` function to do this. Its parameters are the text to be converted

and the text encoding to use. For example, the following line writes the text in the TextField using the MacRoman encoding:

```
TCPSocket1.Write ConvertEncoding(TextField1.text,Encodings.MacRoman)
```

When you call the Write method, you begin the process of sending data across the network. Certain low-level socket service providers have limits on the maximum amount of data the socket can send in one batch. This is dependent on a few factors; among them are which library is providing the TCP/IP services (such as WinSock on Windows), and how much data you are trying to send.

You might think that the provider will only affect transfers of large data, but this is not true. Never assume how much data REAL Studio will send between calls to the SendProgress event. It is normal to see this fluctuate. Each provider specifies the minimum and maximum amount of data it will send.

If you are trying to send data larger than the maximum, it will not be sent all in one chunk. Instead, REAL Studio will loop until your data is completely sent, giving you periodic SendProgress events. If you try to send too little data, the internet service provider (ISP) will queue your data up. This doesn't always mean your data has been sent although you will receive SendProgress and SendComplete events.

This is due to the ISP implementing the Nagle algorithm, which helps network productivity. For every chunk of data that is sent across the network, there is a header attached to the beginning of that data. You never will have to deal with these headers, because they are taken care of for you by the ISP. The reason this is important, though, is that if you are sending one and two bytes at a time across the network, you are also attaching these 40 or so bytes of header to each send. This can bog down a network unless the Nagle algorithm is implemented.

Currently, REAL Studio does not allow the user to turn this feature off, and leaves it set to the default. Note that, if you send only one byte of data, and never send anymore, the system will still send your one byte out even though the Nagle algorithm is enabled.

The conclusion is this: Do not assume that you know when your data has been completely sent. Rely on the SendComplete event to tell you when the send has finished. Also, do not expect the bytesSent parameter of the SendProgress event to be the same value every time. This value will change based on how many bytes of data the ISP was able to send.

Note: If you are going to be sending small chunks of data across a network (especially a small network), it might make more sense to use the UDPSocket class instead.

Handling Errors

Errors can occur while attempting to connect or while sending or receiving data. Errors are not always what they seem. For example, when the other computer closes the connection, an error is generated. When an error occurs, the TCPsocket control's Error event handler is executed. Errors are represented by numbers. The TCP-

Socket control's `LastErrorCode` property will contain the number of the last error that occurred. See the `SocketCore` class in the *Language Reference* for a complete list of error numbers.

Errors are simply ways to alert your application to conditions it may not have anticipated or be able to anticipate.

Orphaning a Socket

One of the new features of the new socket architecture is the ability to orphan a socket. This feature is needed to support the functionality of the `ServerSocket`.

To make this new functionality possible, we have introduced some new behavior to the socket class. When you call the `Connect` method of the `SocketCore` class, the `Listen` method of the `TCP Socket` class, or add a socket using the `AddSocket` method of the `ServerSocket` class, your socket's reference count is incremented.

This means that the socket does not have to be owned by the window in order for it to continue functioning. This is helpful in certain circumstances. For example, suppose you write your own socket subclass that implements all of the events for the socket, called `MySpiffySocket`. In the action event for a `PushButton` you use the following code:

```
Dim s as MySpiffySocket
```

```
s = New MySpiffySocket  
s.port = 7000  
s.address = "somecool.server.com"  
s.connect
```

The socket will continue to stay connected, even though there is nothing owning a reference to it except within the `PushButton`'s Action event.

In other words, a socket will continue to live until you tell it to die. If you have dragged a `TCP Socket` to a window and then called the `Connect` or `Listen` methods before closing the window, there will be two references to the socket, whereas in previous versions of REAL Studio there was only one reference—the window's reference. In this case, the socket will continue to function until its connection is terminated, even after the window has been closed.

The termination can be done either locally, by calling the `Close` method of `TCP Socket` or remotely, with the remote host terminating the connection. If you have not called the `Connect` or `Listen` methods, then there will be only one reference to it (the window's), and it will be destroyed appropriately when the window closes. In either case, once the application terminates, all sockets are released gracefully, and your application will not leak memory.

Maximum Number of Sockets

There is a limit to the number of sockets your application can have opened concurrently on Mac OS X (prior to Mac OS X 10.3). This is because BSD sockets use a file descriptor for each open socket (one that is currently bound to any port on

the machine). The standard limit on Mac OS X is set to 256 file descriptors, but this limit can fluctuate based on the amount of RAM in your machine. This means that you can have, at most, 256 sockets connected at once per application. In practice, this number tends to be less than 256, because your application might have files open, or the underlying API calls might be using a file descriptor for their purposes. This is not an issue on Windows or (to a certain extent) Linux, and it is not a bug in REAL Studio. It is a characteristic of the underlying BSD system.

Note that you can run into this issue on Linux, but it tends to be far less likely.

Closing the Connection

When you are finished communicating and wish to disconnect from the other application, you do so by closing the connection. The connection is closed by calling the `TCPSocket` control's `Close` method. Suppose you have a `Socket` named "Socket1" that has established a connection. To close the connection, you can use the following code:

```
Socket1.Close
```

Sending and Receiving Email via TCP/IP

REAL Studio includes subclasses of the `TCPSocket` class that are designed specifically to build an email client application—an application that sends and receives email, like Apple's Mail program, Eudora, or Microsoft's Entourage.

A typical email client program actually uses two protocols, POP, which stands for Post Office Protocol and SMTP, which stands for Standard Mail Transfer Protocol. The POP protocol is used to receive messages and SMTP is used to send messages. An email client program uses both to communicate with a mail server that supports POP and SMTP, such as 4D Mail.

REAL Studio provides direct support for these two protocols via the `POP3Socket` and the `SMTPSocket` classes. Both are subclassed from `TCPSocket` but have special methods and properties to make it easy to handle email.

For example, the `POP3Socket` class has properties for the Username and Password sent to the mail server and methods to poll the mail server for unread messages, retrieve messages, delete messages, and disconnect from the mail server.

The `SMTPSocket` class has a property to hold the messages in queue to be sent and methods to add email messages to the queue, send messages, and disconnect from the mail server.

To manage an email message, REAL Studio uses three additional classes, `EmailMessage`, `EmailHeaders`, and `EmailAttachment`. The `EmailMessage` class holds the body, header, and attachments, which are created or stored using the `EmailHeaders` and `EmailAttachment` classes.

For detailed information on the methods and properties of these classes, see the Language Reference. For an example email client, see the example email application on the REAL Studio CD or the REALsoftware web site.

The Professional and Studio versions of REAL Studio support secure POP3 and SMTP communications via the POP3SecureSocket and SMTPSecureSocket classes. These classes are subclassed from the SSLSocket class, but are otherwise identical. You can support secure communication by setting the Secure property of the SSLSocket class to True.

HTTP Communications

The HTTP protocol is the protocol that web browsers use. REAL Studio supports the HTTP protocol via the HTTPSocket class, derived from the TCPSocket class, and the HTTPSecureSocket class, derived from the SSLSocket class. It contains methods and properties that enable you to retrieve a URL or post a form using HTTP.

With the Professional and Studio versions of REAL Studio, you can support secure communications (https) by invoking the Secure property of the SSLSocket class. When you do so, the default port changes from 80 to 443.

See the *Language Reference* for information about the methods, events, and properties of these two classes.

Handling Multiple Connections with the ServerSocket Control

If you need to communicate with more than one application via the same port, it is difficult to do using the TCPSocket because each TCPSocket can manage only one connection at a time. To use the TCPSocket, you would have to implement a system for managing multiple TCPSockets, as connections are received.

To handle this situation, you should use the ServerSocket control. A ServerSocket is a permanent socket that listens on a single port for multiple connections. When a connection attempt is made on that port, the ServerSocket hands the connection off to another socket, and continues listening on the same port. Without the ServerSocket, it is difficult to implement this functionality due to the latency between a connection coming in, being handed off, creating a new listening socket, and restarting the listening process. If you had two connections coming in at about the same time, one of the connections may be dropped because there was no listening socket available on that port.

To initiate the ServerSocket's listening process, set the port to listen to by assigning a value to the Port property and call the ServerSocket's Listen method.

On Mac OS X and Linux, attempting to bind to a port less than 1024 will cause a SocketCore.Error event to fire with an error 105 unless your application is running with root permissions. This is a built-in security feature of Unix-based operating systems. This is not a bug, but a security feature that prevents problems that can arise from allowing sockets to listen on privileged ports.

The ServerSocket control automatically manages a "pool" of TCPSockets available for use. You don't create the TCPSockets explicitly; instead you can set the size of this pool using the MinimumSocketsAvailable and MaximumSocketsConnected

properties after establishing the listening socket. If you change the `MaximumSocketsConnected` property, it will not kill any existing connections. It just may not allow more connections until the existing connections have been released). If you change the `MinimumSocketsAvailable` property, it may fire the `AddSocket` event of the `ServerSocket` to replenish its internal buffer.

When you call the `ServerSocket`'s `Listen` method, it first fills its internal pool of handoff sockets. It does this by calling the `AddSocket` event. This event will be called until it has enough sockets in the internal pool of available sockets. It adds the number specified by the `MinimumSocketsAvailable` property, plus ten extra sockets. (Note that if you return `Nil` from this event, it will cause a `NilObjectException`.) The `ServerSocket` is not ready to hand off connections until this process is completed. Connections that come in while the server is populating its pool are rejected. To determine when the `ServerSocket` is ready to accept incoming connections, check its `IsListening` property.

A `ServerSocket` can only return a `TCPSocket` (or a subclass of `TCPSocket`) in its `AddSocket` event. Since UDP is a connectionless protocol (see “UDP Connections with the `UDPSocket` Control” on page 677), it does not make sense for a `ServerSocket` to deal with `UDPSockets`.

Reference Counting

The reference count isn't incremented when you return a socket with the `AddSocket` method. Instead, the socket is pooled internally and its reference count is incremented when the server hands off a connection to that socket. If the `ServerSocket` is destroyed before it uses one of these pooled sockets, the unused sockets get destroyed as well. Until that time, the `ServerSocket` is the parent of the `TCPSocket`, and so the `TCPSocket` will remain. If a socket returned from the `AddSocket` event has been handed a connection and then the `ServerSocket` is destroyed, the socket will remain connected and continue to function.

The new functionality described here tells you that a socket can be orphaned. This does not hold true for a `ServerSocket` or a `UDPSocket`. Each of these sockets must have a reference holder. If it does not, then once the socket goes out of scope in your code, it is destroyed. However, when the `ServerSocket` is destroyed, it will not terminate any of your already-made connections, if there are any. It will only destroy `TCP.Sockets` that have not been connected.



The `ServerSocket` is available only in the Professional and Studio versions of REAL Studio.

Handling Secure TCP Connections with the `SSL.Socket` Control

The `SSL.Socket` control is used to do secure communications via TCP/IP using Secure Sockets Layer (SSL) technology. SSL is an internet protocol that specifies how to pass secure communications over the internet. This provides world-class security, allowing you to accept credit card numbers, serve medical records, human resources information, or other sensitive data without fear of interception.

There are currently four different protocols supported by REAL Studio. They are shown below:

SSL v2	SSSL version 2
SSLv23	SSL version 3, but can roll back to version 2 if needed
SSLv3	SSL version 3
TLSv11	TLS (Transport Layer Security version 1

The default protocol is SSLv23; this is compatible with most SSL servers. You must set this property before you establish the connection by calling the Connect method of the SSLSocket class. Trying to set the protocol after you have begun the connection process will have no effect.

Not all servers will accept a connection with the default protocol (SSLv23). This is server-specific, and may not be known beforehand. When you don't know whether the server will accept SSLv23 connections, try making multiple connection attempts to the server. If the initial attempt is rejected and returns a 102 error, then try again with a different ConnectionType. Be sure to have a way to terminate this process if none of the connection types works, or if you get an error other than 102.

To establish an SSL connection, set the Secure property to True and use the Connect method.

Like the ServerSocket control, the SSLSocket control does not have an icon of its own in the Controls list. Since it is not derived from the Control class, you can instantiate it via code. Or, you can add it to a window using the window's contextual menu by choosing Add ► SocketCore ► TCPSocket ► SSLSocket or drag a TCPSocket to a window and then change its Super Class to SSLSocket.

The SSLSocket control is a subclass of TCPSocket and is available only in the Professional and Studio versions of REAL Studio.



UDP Connections with the UDPSocket Control

The User Datagram Protocol, or UDP, is the basis for most high speed, highly distributed network traffic. It is a connectionless protocol that has very low overhead, but is not as secure as TCP. Since there is no connection, it does not take as many steps to prepare when you wish to use a UDP socket.

Like the TCPSocket control, the UDPSocket control is derived from the SocketCore class. Since it is derived from the SocketCore class rather than the Control class, you don't need to add a UDPSocket control to a window to instantiate it. You can instantiate it with code via the New operator.

In order to use a UDPSocket, it must be bound to a specific port on your machine. Once the bind has occurred, the UDPSocket is ready for use. It will immediately begin accepting any data that it sees on the port that it has bound to. It also allows you to send data out, as well as set UDP socket options.

Datagrams

In order to differentiate between which machine is sending you what data, a `UDPSocket` uses a data structure known as a *Datagram*. A Datagram has two properties, *Address*, which is the IP address of the remote machine that sent you the data, and *Data* — the actual data itself. When you attempt to send data out, you must specify information in the form of a Datagram. This information is the remote address of the machine you want to receive your packet, the port it should be sent to, and the data you wish to send to the remote machine.

UDPSocket Modes

UDP sockets can operate in various modes which are all very similar, but have vastly different uses. The mode that most resembles a TCP communication is called “unicasting.” This occurs when the IP address you specify when you write data out is that of a single machine. An example would be sending data to `www.realsoftware.com`, or some other network address. It is a Datagram that has one intended receiver.

The second mode of operation is called “broadcasting.” As the name implies, this is a broadcasted write. It is akin to yelling into a megaphone. Everyone gets the message, whether they want to or not. If the machine happens to be listening on the port you specified, then it will receive the data on that port. This type of send is very network intensive. As you can imagine, broadcasting can create a huge amount of network traffic. The good news is, when you broadcast data out, it does not leave your subnet. Basically, a broadcast send will not leave your network to travel out into the world. When you want to broadcast data, instead of sending the data to an IP address of a remote machine, you specify the broadcast address for your machine. This address changes from machine to machine, so REAL Studio provides a property of the `UDPSocket` class that tells you the correct broadcast address.

The third mode of operation for UDP sockets is “multicasting.” It is a combination of unicasting and broadcasting that is very powerful and practical. Multicasting is a lot like a chat room: you enter the chatroom, and are able to hold conversations with everyone else in the chatroom. When you want to enter the chatroom, you call `JoinMulticastGroup`, and you only need to specify the group you wish to join. The group parameter is a special kind of IP address, called a *Class D IP*. They range from `224.0.0.0` to `239.255.255.255`. Think of the IP address as the name of the chatroom. If you want to start chatting with two other people, all three of you need to call `JoinMulticastGroup` with the same Class D IP address specified as the group. When you wish to leave the chatroom, you only need to call `LeaveMulticastGroup`, and again, specify the group you wish to leave. You can join as many multicast groups as you like, you are not limited to just one at a time. When you wish to send data to the multicast group, you only need to specify the multicast group’s IP address. All people who have joined the same group as you will receive the message.

Multicasting has some extra features that make it an even more powerful utility for network applications. If you wish to receive the data you sent out with a multicast send, you can set the `SendToSelf` property (also known as loopback) of the `UDPSocket`. If it is set to `True`, then when you do a send (to a multicast group) you will get that data back. You can also set the number of router hops a multicast datagram will take (also known as the Time to Live). When your Datagram gets sent out, it runs through a series of routers on the way to its destinations. Every time the Datagram hits a router, its `RouterHops` property gets decremented. When that number reaches zero, the Datagram is destroyed. This means you can control who gets your datagrams with a lot more precision. There are some “best guesses” as to what the value of `RouterHops` should be.

Value	Description
0	Same host
1	Same subnet
32	Same site
64	Same region
128	Same continent
255	Unrestricted

Note that if your Datagram runs through a router that does not support multicasting, it is killed immediately.

Due to the connectionless functionality of UDP, it does not provide any guarantees that your data will reach its destination. You can work around this by creating your own protocol on top of the UDP protocol which acknowledges receives.

The `UDPSocket` operates in asynchronous mode, but the `Connect` method works synchronously. If the connect fails, you will get an error event immediately, and the `IsConnected` property will be set immediately.

Making Networking Easy

If you only need to establish network communications among REAL Studio applications on a network, you can use a group of classes that are specialized for this task. They are based on the TCP or UDP protocols but they can only communicate among REAL Studio applications. If you need to communicate with another application, such as an FTP server, you need to use the more generic classes.

In exchange for this limitation, it is extremely easy to set up communications. Only a few lines of code are needed, and you don't need to know a lot about the TCP or UDP protocols.

The AutoDiscovery Class

The AutoDiscovery class is perfect for managing network communication among REAL Studio applications. As its name implies, the AutoDiscovery class can automatically “discover” all of the computers on the network that are running the “easy” UTP protocol that’s used as the basis for network communications with this class. With the AutoDiscovery class, you can send and receive messages with individual users or multicast to a group of users.

Joining a MultiCast Group

To join a multicasting group of users, all you need to do is pass the name of the group to the Register method of the AutoDiscovery class. Everyone who joins the group uses the same name. Internally, the class takes care of assigning a proper multicasting IP address for you. For example, the line:

```
AutoDiscovery1.Register(" myMulticastingGroup")
```

is all you need to join the group. To get the list of all group members, call the GetMemberList method. It returns a string array of the IP addresses of all computers currently in the group. At any time you can call the UpdateMemberList method to refresh this list.

This code gets the list of member computers and displays them in a ListBox.

```
Dim s(-1) as string //declare the array and let GetMemberList resize it
Dim i as Integer
s= AutoDiscovery1.GetMemberList
For i=0 to Ubound(s) //Ubound gets index of the last element of s
    ListBox1.AddRow s(i)
Next
```

Sending a Message

To send a message to the group, you use the SendMessageToGroup method. It takes an integer command code (which you can assign arbitrarily) and the text of the message. For example, the following sends the contents of TextField2 as the message with a command ID of 50:

```
AutoDiscovery1.SendMessageToGroup(50,TextField2.Text)
```

To send a message to an individual, you need the computer’s IP address, which you can get from the GetMemberList method, the command ID, and the message text. Call the SendMessageToIndividual method. For example, the following sends the same message to the person at 192.168.1.118:

```
AutoDiscovery1.SendMessageToIndividual("192.168.1.118",_
,50,TextField2.Text)
```

This example uses the underscore character to continue the line of code in a second line in the Code Editor.

The EasyUDPSocket and EasyTCPSocket Classes

The EasyUDPSocket and EasyTCPSocket classes are “easy” versions of the UDP-Socket and TCPSocket classes, respectively, that are specialized for REAL Studio-to-REAL Studio network communication. They both eliminate the need to deal with some lower-level networking issues in order to provide a simple solution to a particular problem.

The main difference between the EasyTCPSocket class and the regular TCPSocket class is the message-based aspects of the protocol. The connection process is identical to a regular TCPSocket. However, when you want to send data to a remote machine you must use the SendMessage method to do so. When you receive a message sent via this protocol, you will get a ReceivedMessage event.

Receiving a Message

Receiving a message couldn’t be easier. The AutoDiscovery class has a ReceivedMessage event that fires when AutoDiscovery receives a message from anyone in the group (or an individual), including yourself. The event returns the IP address of the sender’s computer, the Command ID, and the message text. An event handler like this is all you need:

```
Sub ReceivedMessage(FromIP as String, Command As Integer, data as String)
    MsgBox FromIP + " sent us " + Str(Command) + ": " + data
```

Understanding Protocols

Any kind of communication requires that all parties involved agree on a method of communication and a language. For example, if you want to communicate with a friend, you might talk to them face to face, call them on the phone, or send them email. Both of you must be able to communicate using the same language or you won’t be able to communicate at all. Communications via TCP/IP work the same way. The language used is called a *protocol*. A protocol is simply an organized way of sending and/or receiving information.

If you are writing an application that will communicate with another application via TCP/IP, you will need to understand the protocol the other application will be expecting in order to communicate with it. For example, on the Internet, the protocol for the world wide web is called HTTP (HyperText Transfer Protocol), the protocol for sending email is called SMTP (Simple Mail Transfer Protocol), and the protocol for receiving email mail is called POP3 (Post Office Protocol 3). Complete descriptions of these Internet protocols and others are available on the Internet. The descriptions of these protocols are called RFCs (Request For Comments). The easiest way to find information on RFCs is to go search for “RFC”. This will give you a list of links to various web sites that explain all of the various Internet protocols.

If you are writing an application that communicates with other applications you have written, then you can define your own protocol. Your protocol will simply be a set of commands you define that allow the applications to understand what the other wants.

Extending the Capabilities of REAL Studio

One of the things that makes REAL Studio easy to learn and use is that it abstracts you from the inner workings of the operating system. You don't have to know any of the thousands of commands that make up the API (application programming interface) used to work with the Windows, Macintosh, or Linux operating systems. This also means that REAL Studio may not have a particular capability that you require. Fortunately, REAL Studio provides several ways to extend its capabilities, allowing you to add just about any functionality you need.

Contents

- Making API calls to the operating system
- Calling AppleScripts
- Communicating with AppleEvents
- Using and Writing REAL Studio Plug-ins
- Using Shared and Dynamically Linked Libraries
- Office Automation
- ActiveX components

Making API calls to the Operating System

Using the Declare statement, you can access the API on the Macintosh, Linux or Windows platforms. Both PPC and Intel machines are supported. You need to use the conditional compilation feature to isolate your Declare statements for each platform. With the Declare statement, you specify the name of the toolbox call and its shared library, and the parameters the call uses. If the call returns a value, you specify the data type of the value that is returned.

If the functionality is available on both platforms, you can use the same name for both platforms. However, often the parameters for the call will be different. Use conditional compilation to isolate your calls as well.

The following button Action uses the Macintosh Speech manager to speak the text in a TextField:

```
dim s as string
dim i as integer
#if TargetMacOS then
  Declare Function SpeakString lib "SpeechLib" (SpeakString as pstring) as Integer
#endif
s=TextField1.text
#if TargetMacOS then
  i=SpeakString(s)
#else
  MsgBox "Speech is supported only on Macintosh!"
#endif
```

If the name of the toolbox call is the same as a REAL Studio method, use the Alias keyword to refer to the call. For example, if SpeakString was the name of a REAL Studio method, you could not use the above syntax. You could use, for example:

```
Declare Function MySpeakString lib "SpeechLib" Alias "SpeakString"
(SpeakString as pstring) as Integer
```

You would then use MySpeakString in your code to invoke the toolbox call.

See the description of the Declare statement in the *Language Reference* for more information and examples.

Calling AppleScripts



AppleScript is Apple's system-level scripting language that makes controlling applications easy. REAL Studio supports AppleScript. You can write a script in AppleScript and then call that script in your REAL Studio project.

Note: AppleScript is available for use only on Macintosh.

Preparing an AppleScript to Work in REAL Studio

In order for REAL Studio to run an AppleScript, the entire script must be enclosed in an “on run” handler like this:

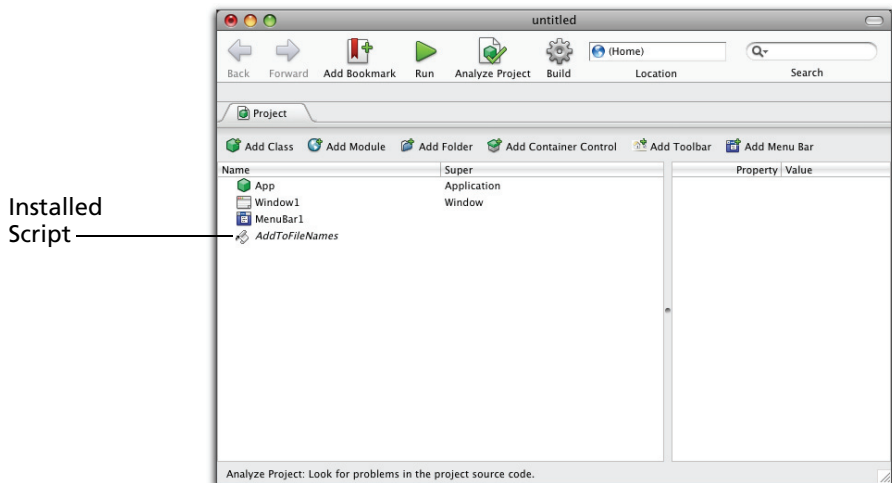
```
on run
  //your script code goes here
end run
```

Next, your script must be saved as a compiled script. In the Script Editor supplied by Apple, choose File ► Save As. Then choose “Script” from the File Format pop-up menu.

Adding an AppleScript to a Project

To include an AppleScript, just drag your compiled script file into the REAL Studio Project Editor. The script will appear with a script icon next to it. Figure 497 shows an example of a project with a script installed.

Figure 497. A compiled AppleScript in Project Editor



When you drag a compiled script into your Project Editor, REAL Studio creates an alias to the AppleScript on disk. Its name is shown in italics, as indicated in Figure 497. When you create a standalone application, the AppleScript is included in the built application.

Passing Values To an AppleScript

If you are writing a script you want to pass parameters to, the parameters must be enclosed in curly braces following the on run statement. In the following example, the x and y are parameter variables that will hold the values of the two parameters passed to the script:

```
on run {x,y}
  //your script code goes here
end run
```

Returning Values From an AppleScript

You write a script to act as a function by having it return a value. To return a value from a script, simply use the return command in AppleScript followed by the value you wish to return. This simple example takes a number of days and returns the equivalent number of years:

```
on run {daysOld}
  return daysOld/365
end run
```

Calling an AppleScript

Scripts are called just like the built-in methods and functions. Type the name of the script as it appears in the Project Editor. If the script requires parameters, the parameters follow the name of the script just as they do with any of the built-in commands. This example calls a script that sets the sound level of the Macintosh to 5:

```
SetSoundLevel 5
```


Scripts that return values (acting as functions) work just like the built-in REAL Studio functions. This script gets the current sound level and assigns it to a variable:

```
Dim level as Integer
level=GetSoundLevel()
```

Removing an AppleScript

To remove a script from a project, click on the script in the Project Editor to select it then press the Delete key or Control-click on the Applescript and choose Delete from the contextual menu.

Communicating with AppleEvents

AppleEvents is the core communications system between applications on the Macintosh. As a matter of fact, when you are calling AppleScript code, AppleScript is actually performing all of its magic with AppleEvents. When you choose  ▶ Restart on Mac OS X, the Finder sends a “Quit” AppleEvent to any open applications. This particular AppleEvent is one that all applications are required to support.

You can perform some very fast and powerful actions with AppleEvents. You create AppleEvent objects in REAL Studio using the AppleEvent constructor. The constructor takes as its parameters class of AppleEvent, the Event ID of the AppleEvent you wish to send, and the bundle id of the target application of the AppleEvent. AppleEvents have three parts: an event class, an event ID, and the creator code of the target application.

The Event Class and Event ID together uniquely define a particular AppleEvent. The EventClass acts as a category for logically grouping events together. While there are many standard (and even some required) AppleEvents, many applications have several custom AppleEvents for performing actions specific to the application. Consult the application’s documentation or its author to get information on what custom AppleEvents may be available.

Sending AppleEvents

Once you create the AppleEvent object and populate the necessary parameters with data, you then send the AppleEvent to the target application using the AppleEvent object’s Send method.

In this example, an AppleEvent is created to tell the Finder to restart the Macintosh. “FNDR” is the class of AppleEvent and “rest” is the event ID. “rest” is short for “restart”. Finally, “com.apple.finder” is the bundle ID for the Finder. The AppleEvent class has a Send method. This method is a function that returns True if the AppleEvent is successful and False if it fails.

```
dim ae as AppleEvent
ae=New AppleEvent("FNDR","rest","com.apple.finder")
if not ae.send then
    MsgBox "The computer couldn't be restarted."
end if
```

Receiving AppleEvents

In order to receive AppleEvents your project must have a subclass that has Application as its Super property value. That’s because the Application class is the only class with a HandleAppleEvent event handler. The App class that is included in the Desktop Application template has this event handler. When your application receives an AppleEvent, the application class’s HandleAppleEvent event handler is executed and the AppleEvent is passed as a parameter to the event handler.

This event handler, when called, is passed an AppleEvent object, the event class, and the event id. There are required AppleEvents that your application should support. One of the them is the Quit AppleEvent.

In this example, if the HandleAppleEvent event handler receives a quit AppleEvent from the Finder, it calls the Quit method.

```
Function HandleAppleEvent(Event as AppleEvent, eventClass as String,  
    EventID as String) as Boolean  
    if eventClass="aevt" and eventID="quit" then  
        //the Finder wants the app to quit  
        beep  
        msgBox "I must quit now."  
        quit  
    end if
```

You can create your own set of AppleEvent classes and event IDs for your application that represent various actions your application can take in response to them.

If your application needs to know if the Mac has gone to sleep or when the Mac wakes up from sleep, you can get this information easily. It turns out that the Mac OS sends a “wake” AppleEvent to all applications when the computer wakes up from sleep.

The HandleAppleEvent event handler of the Application class fires anytime your application receives an AppleEvent. This event handler is passed the AppleEvent, the EventClass, and the EventID. To determine whether you have received the wake event, check to see if the EventClass is “pmgt” and the eventID is “wake” in the HandleAppleEvent event handler. If you receive an AppleEvent with these values, the Macintosh has just woke up from sleep.

Keep in mind that AppleEvent classes and IDs are case-sensitive. Also, you won't receive these events when running your application in the IDE. You will receive them only in your built applications.

Sophisticated AppleEvents

AppleEvents can actually contain quite a bit of very specific data. AppleEvents for example, can be used write to applications that process data for web servers. For more information on AppleEvents, see the AppleEvent class in the *Language Reference*.

Using and Writing REAL Studio Plug-ins

Many applications have their own plug-in format. Netscape Navigator, Adobe PhotoShop, 4th Dimension, are just a few examples of applications that have a plug-in formats. Plug-ins are a way for an application to be extended by other programmers. For example, there is a plug-in for Firefox that allows it to play QuickTime movies that have been embedded into web pages.

REAL Studio also has its own plug-in format. Plug-ins are written in languages like C and C++. For example, James Milne of Essence Software wrote a plug-in for REAL Studio that plays a particular type of music file. REAL Studio also uses plug-ins to manage connectivity to database back ends. You can add support for other database engines simply by writing (or obtaining from a third-party) the plug-in for that database engine.

If your application is being designed to run on Mac OS X, plug-ins must be carbonized.

Loading Plug-ins

Loading plug-ins is easy. Simply create a folder called “Plugins” in the same folder that contains REAL Studio. Then drop your plug-in files into that folder. Any plug-ins in this folder will automatically be available to your projects.

Using Plug-ins

Some plug-ins are in the form of controls similar to those that appear in the REAL Studio Controls list. When you have this type of plug-in in your Plug-ins folder, a new control will appear in the Controls list. Plug-in controls appear in the Plug-in Controls category in the Window Editor’s Controls list.

You use a plug-in control the same way you use any other control in the Controls list, by dragging it to a window. The Properties pane will then display any properties that can be set from the Design environment.

Plug-ins can also be a set of methods that has no interface whatsoever. Plug-ins of this type do not appear anywhere in the interface. You must have some documentation to know which methods exist in the plug-in, what the methods do, and how to use them.

Including Plug-ins in Your Stand-Alone Applications

When you build a stand-alone application from your project, any plug-ins you are using in your project will automatically be built-in to the stand-alone application.

Writing Your Own Plug-ins

If you know C or C++, you can write REAL Studio native plug-ins. The REAL Studio Plug-in Software Development Kit (SDK) is available on the REAL Software web site. This kit contains all the information you need to write plug-ins including sample plug-ins and include files for Metrowerks CodeWarrior.

Using PowerPC Shared Libraries

PowerPC shared libraries are files that have subroutines that can be called and passed parameters. These parameters are referred to as “entry points.”

In REAL Studio 2005 and above, shared and dynamically linked libraries are accessed via the Declare statement in the language. See the Language Reference for information on the Declare statement.

Microsoft Office Automation

REAL Studio includes four classes that enable you to program Microsoft Office applications directly from REAL Studio. Office automation is supported only on the Windows platform. Obviously, Microsoft Office must also be installed on the machine that will run the built application.

The five classes are as follows:

Class	Description
ExcelApplication	Inherits from OLEObject and is used to automate Excel.
Office	Contains all the enums you need for Office Automation.
OLEObject	Used to automate COM servers and the base class for ExcelApplication, PowerPointApplication, and WordApplication.
PowerPointApplication	Inherits from OLEObject and is used to automate PowerPoint.
WordApplication	Inherits from OLEObject and is used to automate Word.

Office Automation is the Component Object Model (COM) technology that makes Microsoft Office applications programmable.

The language that you use to automate Microsoft Office applications is documented by Microsoft and numerous third-party books on Visual Basic for Applications (VBA). Microsoft Office applications provide online help for VBA. To access the online help, choose Macros from the Tools Menu of your MS Office application, and then choose Visual Basic Editor from the Macros submenu. When the Visual Basic editor appears, choose Microsoft Visual Basic Help from the Help menu. The help is contextual in the sense that it provides information on automating the Office application from which you launched the Visual Basic editor.

The help files for VBA may not have been installed by default. If VBA Help does not appear, you will need to perform an optional install before you can access the help files. On Windows Office 2003, Office prompts you to install the VBA Help files when you first request VBA help by following the procedure described in the previous paragraph.

Microsoft has additional information on VBA at <http://msdn.microsoft.com/vbasic/> and have published their own language references on VBA. One of several third-party books on VBA is “VB & VBA in a Nutshell: The Language” by Paul Lomax (ISBN: 1-56592-358-8).

Uses of Office Automation

With Office Automation, you can create methods to automatically create and format documents in Word. You can use Automation to run other applications from within a REAL Studio application. For example, a REAL Studio application can run a hidden instance of Excel to perform mathematical and analytical operations on data in a REAL SQL database. Or, you could use Word to create mail-merge letters using data in a REAL SQL database.

Here is a simple example of office automation that creates an OLEObject, launches, MS Word, and opens a blank document.

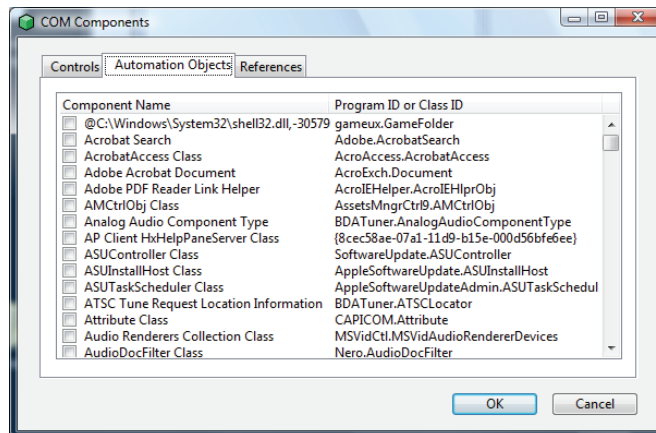
```
Dim word as New OLEObject("word.Application")
Dim wordcopy as OLEObject
wordcopy=New OLEObject(word)
wordcopy.Visible=True
wordcopy.Documents.Add
```

For some examples, see the entries for the ExcelApplication, WordApplication, and PowerPointApplication classes in the Language Reference.

ActiveX Components

ActiveX components are standard user interface elements or programmable objects that enable you to build a highly customized interface rapidly. With the Windows version of REAL Studio, you can add ActiveX controls and components to your application using the Project ► Add ► ActiveX Component menu item. It presents a dialog box with the list of ActiveX controls and components that are installed on your PC. That is, the list will differ depending on what is currently installed and available and would not necessarily agree with what is installed on your users' computers.

Figure 498. The ActiveX Components dialog box.

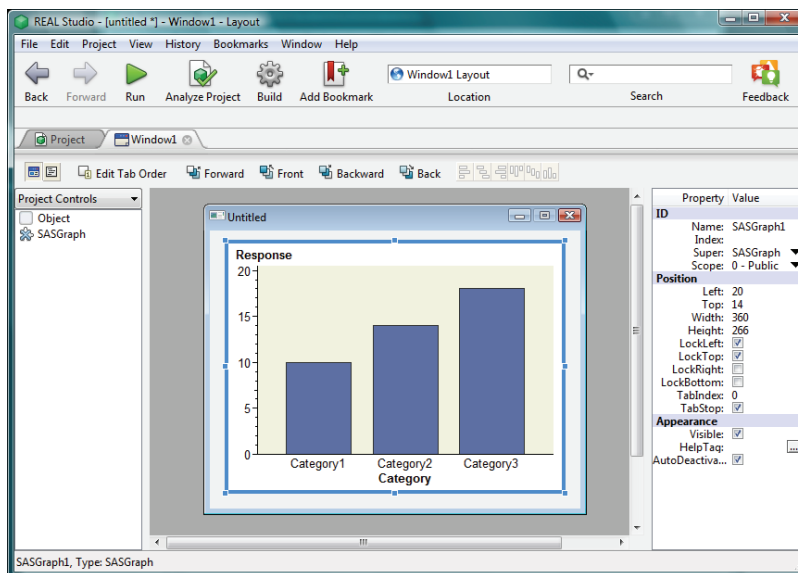


Click the CheckBox corresponding to each ActiveX component that you want to add and click OK. If you added any ActiveX controls, they will appear in the Project Controls list in the project's Window Editors and your Project Editor. If an

ActiveX component is not a control, it will appear only in the Project Editor. Once added, you can incorporate the control or component into your REAL Studio application as if it were a standard REAL Studio object. You work with it just as if it were a REAL Studio control or class.

For example, if the machine has a licensed copy of SAS Graph installed (<http://www.sas.com>) you can add programmatic statistical graphing capabilities to REAL Studio via the SAS Graph control, as shown below.

Figure 499. The SAS Graph control added to a REAL Studio window.



For information about programming ActiveX components and specific Microsoft ActiveX components, refer to Microsoft documentation in the MSDN library at:

<http://msdn.microsoft.com/library/>

Building Stand-Alone Applications

When you are ready to turn your project into a standalone application, there are a few things you will need to know. This chapter will help you understand what finishing touches your application may need to make it complete.

REAL Studio can create two types of standalone applications: fully-functional and demo. The fully-functional version is the same as the application that runs when you click the Run button in the Toolbar or choose Project ► Run in the IDE, except that no copy of REAL Studio is needed (hence the name “standalone”). A demo version is the same as one built with the fully-functional version, except that it quits automatically after five minutes.

The Professional and Studio versions of REAL Studio build fully-functional standalone applications for the Windows, Mac OS X, and Linux platforms. The Personal version allows you to build fully-functional applications only for the platform on which REAL Studio is running. It can only build demo versions for each platform on which REAL Studio is *not* running.

Contents

- Choosing a Target Platform
- Building Your Application
- Project Editor Items
- Assigning Custom Icons
- Registering Your Creator Code
- Using and Writing REAL Studio Plugins

Choosing a Target Platform

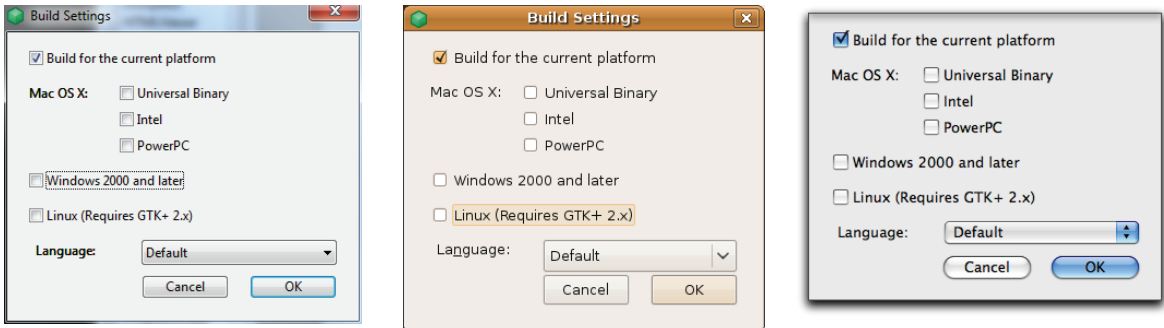
Before you build your application, you choose the platform or platforms on which it will run. REAL Studio can build for Windows, Linux, and Mac OS X. You can build for any or all of these platforms simultaneously.

To choose a target platform, do this:

1 Choose Project ► Build Settings.

The Build Settings dialog box appears.

Figure 500. The Build Settings dialog box.



By default, the operating system that you are currently running is selected. You can choose as many other targets as you wish.

Note that the Personal version of REAL Studio only builds demo versions of the application for platforms other than the one on which the IDE is running.

Your choices are:

- **Windows 2000 and later:** Builds a Windows application that will run on Windows 2000, XP, and Vista.
- **Linux with GTK+ 2.x:** Builds a Linux application that runs on x86-based machines. Linux builds of REAL Studio desktop applications require GTK+ 2.8 or above, glibc-2.3 (or higher), the CUPS (Common UNIX Printing System) and libstdc++.so.6. For information on GTK, see <http://www.gtk.org>. Linux builds of console applications do not require GTK.

- **Mac OS X Universal Binary:** Builds for Mac OS X for computers that use either the PowerPC or the Intel architecture. Mac OS X 10.2 or above is required.
- **Mac OS X PowerPC:** Builds for PowerPC Macintoshes in the Mach-O format. Mac OS X 10.2 or above is required.
- **Mac OS X Intel:** Builds for the Intel-based Macintoshes in the Mach-O format. Mac OS X 10.2 or above is required.

All three Macintosh choices use the Mach-O format. Mach is the Unix kernel on which Mac OS X is based and Mach-O is the object file format for Mach. Mach-O is the native executable format for Mac OS X.

Building Your Application

Building a standalone version of your project as an application couldn't be easier. Just click the Build button in the Toolbar or choose Project ► Build Application (Ctrl+B on Windows and Linux or ⌘-B on Macintosh). REAL Studio will create standalone applications for all the platforms that you selected in the Build Settings dialog.

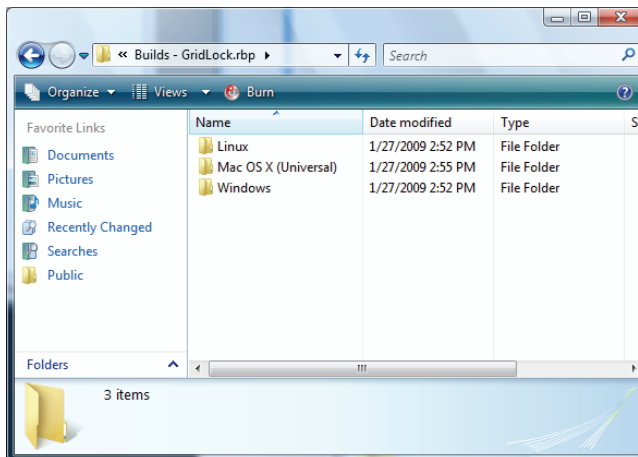
REAL Studio uses a folder called "Builds - *ProjectName*" that stores all of the builds for that project. Inside this folder, there will be folders for each selected target platform. If you build for all possible target platforms, the Builds folder will contain the following subfolders:

- Windows
- Linux
- Mac OS X (Intel)
- Mac OS X (PowerPC)
- Mac OS X (Universal)

This folder structure prevents possible naming conflicts when building for multiple targets.

For example, here is the Builds folder for the "GridLock" project. The Builds folder is named "Builds - GridLock.rbp" and the selected target platforms for the build are: Windows, Mac OS X (Universal), and Linux.

Figure 501. The Builds folder for the GridLock project.



Each subfolder will contain the build for that platform. If you rebuild for a target, then the new build will overwrite the older build without notifying you. When building several targets at once, the compiler will halt the build process for the first target that encounters errors.

If you haven't changed the application's name for that platform, each build will use the default name of "MyApplication" and use the default generic application for the platform.

Figure 502. A standalone application (Linux¹, Windows, and Macintosh) that uses the default names and icon.



See the section "Customizing the Standalone Application's Properties" on page 700 for information on changing the application's name and icon.

Building for Windows

The build process for the Windows platform differs from the other platforms. In general, the Windows build process results in a *folder* that contains two items: the .exe file, as shown in Figure 502, and a folder that contains the .dlls that the application uses. A .dll will be written to this folder for each REAL Studio plug-in that the application uses. This includes any internal plug-ins that the application calls and any third-party plug-ins that you have installed. If dlls are included in the build, the build folder will be nested inside the Windows folder shown in Figure 501. Otherwise, the Windows folder will contain only the .exe file.

1. Different Linux distributions may have different icons.

REAL Studio implements a number of items as internal plug-ins. They do not appear on the desktop as plug-ins, but are treated as plug-ins for purpose of the application build. Whether or not an item is implemented as an internal plug-in is an implementation detail that may change in subsequent releases. The complete list of internal plug-ins is found in the “Internal Plug-ins” topic in the *Language Reference*. You will get a .dll if the application uses an item in its list.

If an application does not use any plug-ins, then the compiler generates only the .exe file. If it does use a plug-in, it generates a folder that uses the name that you gave the Windows version of the application. The folder will contain a folder called *MyApplication Lib*, where *MyApplication* is the application’s name. This is the folder that contains the required .dlls.



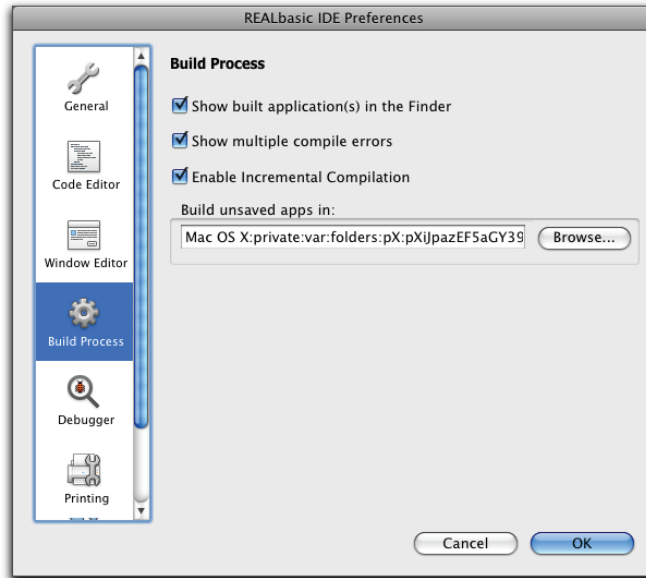
Since the .dlls are an integral part of the application, it is mandatory that you do not move or rename them. Also, you cannot share a .dll library among several builds. If you distribute the built application, you must distribute the entire folder, not just the .exe file. This marks a change in procedures from releases of REAL Studio prior to 2008 Release 2.

Incremental Compilation

When you compile various versions of a project repeatedly, REAL Studio uses incremental compilation by default. When it does a compilation, REAL Studio compiles each project item individually and saves the compiled code in a cache. The next time you compile the project, it analyses what has changed. It reloads as much of the saved code as it can and compiles only the aspects of the project that have been added, modified or depend on other items that have been modified. For example, if you change a class, REAL Studio recompiles that class and everything that refers to that class and its subclasses.

If desired, you can disable incremental compilation and force a full compilation. To do so, chose Edit ► Options (REAL Studio ► Preferences on Macintosh) and display the Build Process screen. Deselect the Enable Incremental Compilation option.

Figure 503. The Enable Incremental Compilation option.



By default, when REAL Studio is finished building the application, it will bring the window containing the application to the front for you. This will be a subfolder in the Builds folder for that project. If you have specified more than one target platform, the Build folder will be opened. You can double-click the standalone application for your platform to launch the built application.

If you don't want REAL Studio to bring the window containing the application to the front, you can deselect this preference in the Build Process screen of the Options dialog box (Preferences on Mac OS X). This panel is shown in Figure 503.

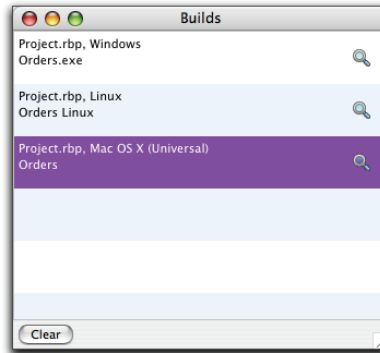
Deselect the “Show Built Applications on *Desktop*” preference to disable this feature. On Windows, the desktop is called “Windows Explorer” on Macintosh, it is called the Finder, and on Linux, it is called the Desktop.



When you build an application, REAL Studio does not ask before overwriting an existing build in the Builds folder. If you want to keep an earlier build around, move it into another folder before rebuilding the application.

The Build Process screen in the Options dialog box allows you to change the directory used to store debug builds of the application (those that are created when you click the Run button in the toolbar). If you wish, you can change this by changing the path shown in the “Build unsaved apps in” field.

When you click the Build button in the Main toolbar or choose Project ► Build Application, REAL Studio displays the Build Progress dialog box. It shows the progress of each build you specified. If you are building for multiple targets, each target is shown in its own line.

Figure 504. The Build Progress dialog.

If you want to display one of the files resulting from a build, click the magnifying glass on the right side of the desired row.

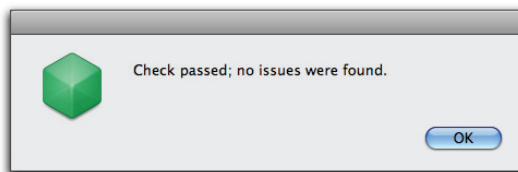
If the attempt to build failed, the magnifying glass is replaced by an alert icon and an “Errors” panel opens in the IDE. It lists all the errors that prevented compilation.

If the Builds window is closed and you want to reopen it without creating a new build, choose **Window ► Builds**.

Analyzing the Project

When you attempt to compile an application, REAL Studio must first check your code for errors. If it finds an error, it cannot compile the application. It will stop the compilation process and display the error or errors in a new “Errors” panel in the IDE (If you have tabbed editing off, the Errors screen will open in a new window).

if REAL Studio finds no errors, it displays an alert box indicating that the project passed all its tests.

Figure 505. The “Check Passed” alert box.

Before you attempt to compile the application, you can use the **Project ► Analyze Project** command to review the project. Analyze Project checks the project for both syntax errors and other issues. For example, it checks for deprecated items, for the obsolete syntax for declaring constructors, and unused variables and parameters. For more information on the Analyze Project command, see the section “Analyzing the Project” on page 632.

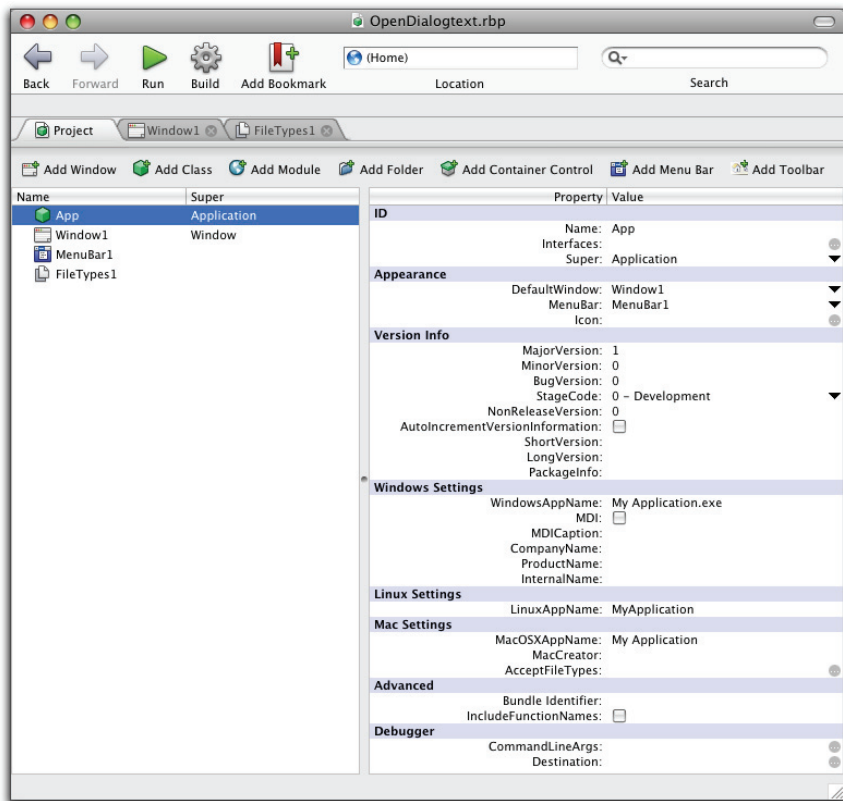
Customizing the Standalone Application's Properties

Of course, you will want to replace the default name of your application and you may also want to build for other platforms, set version information, and platform-specific options.

You choose the target platforms for the build and the default language in the Build Settings dialog box.

You customize your build options by setting the properties of the App class. The Properties pane for the App class contains themes for the application's Appearance, Version Info, Windows, Macintosh, and Linux build settings, and Advanced settings.

Figure 506. The properties of the App class.



After you're finished entering these properties, REAL Studio will use these settings when you build the application.



To customize your application's properties, do this:

- 1 If the Project Editor is not displayed, click its tab in the IDE.
- 2 Select the App class in the Project Editor.

The Properties pane changes to show the Application object's properties, shown in Figure 506. The properties are organized into seven groups: ID, Appearance, Version Information, Windows, Linux, and Macintosh settings, Advanced, and Debugger.

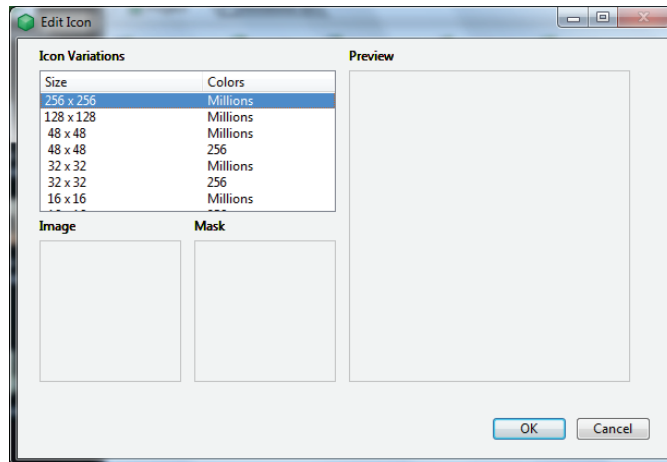
Appearance Settings

In the Appearance group, you choose the default window and menubar and install the application's icon. The default window and menubar are shown when the application starts. The drop-down lists offer all the project's windows and menubars. The default selections are the window and menubar that are added to a Desktop Application project when you first created the project with the File ► New Project menu command or started the REAL Studio IDE application.

You can also paste in or import the custom icon for the application. If no custom application is provided, REAL Studio will use the generic application icon for each platform on which the application is built. Custom icons are recognized only if the application has its own Creator code. Click the "..." button for the Icon to display the Edit Icon dialog box for the application.

REAL Studio does not include an icon editor; you must design your icons in an icon editor program or image creation application and import or paste them into REAL Studio.

Figure 507. The Edit Icon dialog box.



The Edit Icon dialog box allows you to install icons of various depths and sizes. The procedure for adding an application icon is the same as for document icons. For more information, see the section “Adding a Document Icon” on page 485.

The Edit Icon dialog can be resized and the areas for the Image, Mask, and Preview will resize proportionally.

You need to provide at least the following items:

- For Mac OS X, a 128 x 128 image (or larger), and smaller sized icon images for other platforms. Even for Mac OS X it is best to provide smaller sized image files at 32 x 32 and 16 x 16.
- A mask that defines the icon's edges so that the operating systems can determine which regions in the square image are clickable.

If you are going to distribute your application on Macintosh, we highly recommend that you set a Creator code and register it with Apple, Inc.



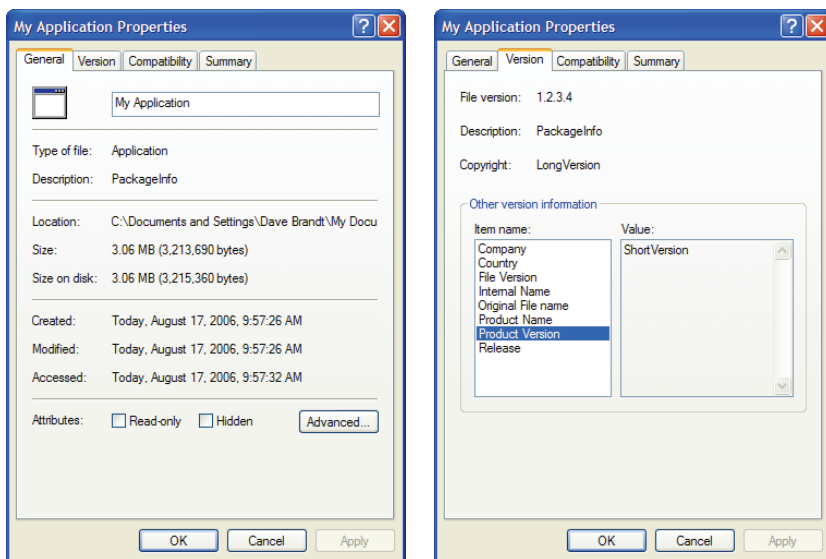
Another way of adding custom icons to your built application is to put your icon data (including 'icns' 32-bit icons) in a Resources file that you add to your project. These icons will overwrite the icon resources supplied by REAL Studio itself. However, it is easier to accomplish the same thing using the Edit Icon dialog box.

For information on using the Edit Icon dialog box, see the section “Adding a Document Icon” on page 485.

Version Information

Some of the information in the Version Info area will be displayed when the user clicks on your application icon and chooses File ► Get Info (⌘-I) on Macintosh or chooses Properties from the application's contextual menu on Windows. The text you enter in the Package Info area appears directly below the application's name. The text you enter in the Long Version entry area appears in the Version area below the modification date. Here is the General panel of the Properties window on Windows XP.

Figure 508. General and Version properties panels on Windows XP.



The File Version field on the Version panel concatenates the MajorVersion, MinorVersion, BugVersion, and NonReleaseVersion properties. The Description field contains the value of the PackageInfo field and the Copyright field contains the contents of the LongVersion property.

The Other Version Information Listbox contains the following properties:

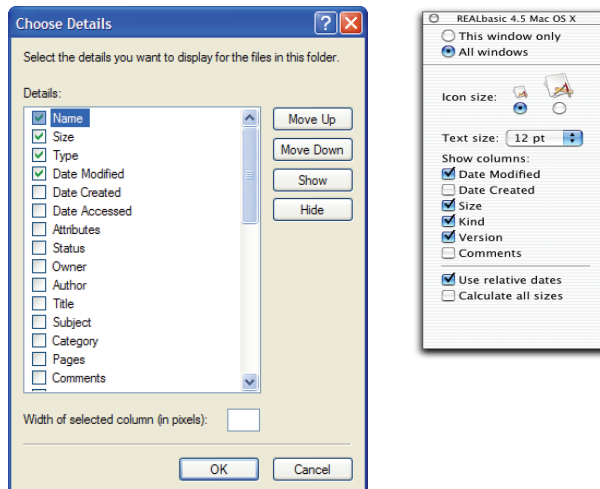
Table 42: App class properties displayed in the Other Version Info Listbox.

Item	Property
Company	CompanyName
Country	
File Version	MajorVersion.MinorVersion.BugVersion.NonReleaseVersion
Internal Name	InternalName
Original File name	WindowsAppName
Product Name	ProductName
Product Version	ShortVersion
Release	StageCode

On Macintosh, the Version number will appear in the Get Info dialog box, the Finder's List view, and written to the 'vers' resource of your application.

The version information can be displayed in a Windows List view if you select the "Details" view option and right+click on the column headings to display the Choose Details dialog (Figure 509). Scroll down and select the Product Version option. On Mac OS X, the comparable option is available using the View ► Show View Options menu command. If you are using a List view in a window, this command displays the dialog box shown on the right in Figure 509.

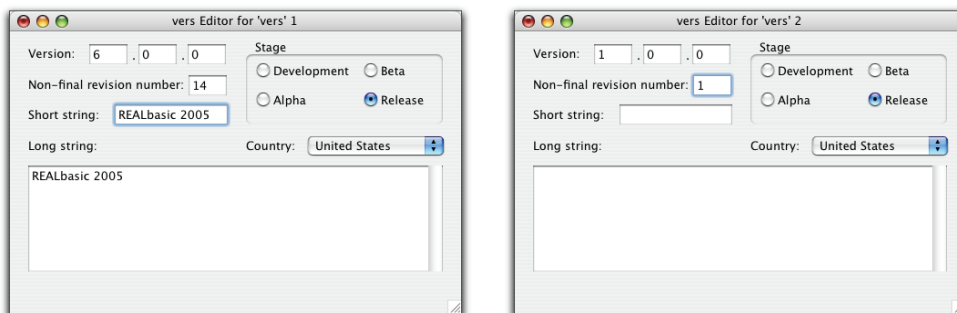
Figure 509. The View Options dialog box for List Views (Windows XP and Mac OS X).



The remaining Get Info settings are not visible but are stored in your built application's 'vers' resource. The 'vers' resource can be used to store information about a individual application or, if it is part of a set of files, to the group of files. The 'vers' resource with an ID of 1 specifies version information for the application; the version resource with an ID of 2 specifies version information for a set of files.

Figure 510 shows the relationship between the Version Information settings and the 'vers' resource settings:

Figure 510. Macintosh 'vers' resources 1 and 2.



The string in the Long Version area for vers ID2 corresponds to the Package Info field in the App class's properties.

The following elements are stored:

- **Major Version level:** In binary-coded decimal format. On Macintosh, accessible only from the 'vers' resource in the first byte.
- **Minor Version level:** In binary-coded decimal format. On Macintosh, accessible only from the 'vers' resource in the second byte.
- **BugVersion:** In binary-coded decimal format. On Macintosh, accessible only from the 'vers' resource in the third byte.

Stagecode. The levels are coded as follows:

Table 43: Values corresponding to Stagecode.

Value	Description
0	Development
1	Alpha
2	Beta
3	Final

This information appears as the Release level in Windows Properties.

- **NonreleaseVersion:** Accessible from the 'vers' resource as the fourth byte. The Major, Minor, Bug, and Nonrelease version numbers are concatenated in the Windows Preferences dialog and shown as the "File Version."

- **AutoincrementVersionInformation:** If this option is selected, the `NonreleaseVersion` will be incremented automatically for each build. It does not increment for a debug run in the IDE.
- **Short version:** Identifies the version number of the software. This may be displayed, at the end-user's option, in the Finder's List views.
- **Long version/Package Info:** Long Version contains the version number and other identifying information about the company, developer, or group of files. For 'vers' ID 1, the Long Version is displayed in the version field of the built application's Get Info window. For 'vers' ID 2, the string is displayed under the application's name and next to the application's icon at the top of the Get Info window. It is referred to as 'Package Info' in the Build Application dialog box.

Using the `GetResource` method of the `ResourceFork` class, you can access the information contained in the 'vers' resource of your built application.

Windows Settings

In the Windows Settings group you give the Windows build of your application a name and, if desired, choose to run it as a Multiple Document Interface (MDI) application.

If you select the Multiple Document Interface option, the application's windows will be enclosed in the "parent" MDI window. If you select this option, you can enter the caption for the MDI window that appears in its title bar.

Select Multiple Document Interface if you want your Windows application to run inside a "master" window. Enter the text of the master window title in the `MDICaption` area.

If you deselect Multiple Document Interface, your application's windows will appear alongside other applications' windows (like the REAL Studio IDE) and have their own menubars. If your application can open more than one document window, it will launch another copy of itself.



The `MDIWindow` class in the language allows you to set several properties of the MDI window, such as its initial location and size, minimum size, and title. For more information, see the entry for the `MDIWindow` class in the *Language Reference*.

There are three optional fields in the Windows Settings group:

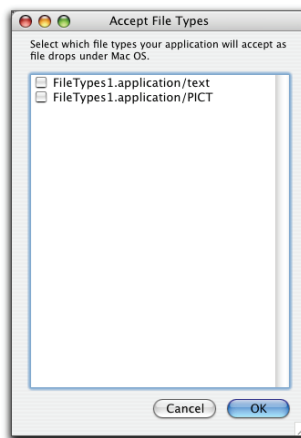
- **Internal Name:** This is useful when your product has a different internal name than its external name. If you leave Internal Name blank, it will be set to the name of the application.
- **Company Name:** The name of the executable file. Use this if the name of the executable is different from the (public) product name that is visible in the Start menu. For example, "winword.exe" versus "Microsoft Word".
- **Product Name:** The name of the product as installed in the Windows Start ► All Programs menu.

Linux Settings The Linux Settings group enables you to give the Linux build a name.

Mac Settings Use the Mac OS Settings group to set the name of the Mac OS X applications, set a Creator code for the built application, and specify the file types that the application will accept via drag and drop. The Creator code can be set using a global constant (that is declared in a module) or a public constant using the syntax *modulename.constantname* or *windowname.constantname*.

File Types for Macintosh Drag and Drop To specify the acceptable file types for drag and drop, first define the file types in a File Types Set Editor. Then click the “...” button for the AcceptFileTypes property in the App object’s Properties pane. The following dialog box will appear, listing the file types that you have defined.

Figure 511. The Accept File Types dialog box.



The AcceptFileTypes dialog will list all of the currently defined File Types. It uses the syntax *FileTypeSetName.FileTypeName*. To indicate that the Macintosh build of the application will accept drag and drop for the file type, select the file type’s CheckBox by clicking its CheckBox.

Registering Your Creator Code Each application’s creator code should be unique. This is because the Finder uses these codes to determine which application to launch when a file is double-clicked. The Finder simply locates the first application it can find with a matching creator code.

You can register your application’s creator code with Apple to be reasonably sure that it’s unique. For more information about registering Creator codes, see the Apple developer web site at <http://developer.apple.com>.

Debugger The properties in the Debugger group provide the following capabilities:

- **CommandLineArgs:** Enables you to pass command line arguments to debug builds. Enter the arguments in this field.
- **Destination:** Enables you to specify a destination directory for debug builds. Clicking the “...” button opens a file browser that enables you to select the desired directory. When you make your selection, the full path to the directory is displayed in this field.

Advanced

The properties in the Advanced group offer the following options:

- **Bundle Identifier:** For Mac OS X bundled applications, you can enter the CFBundleIdentifier in the Bundle Identifier field. This will add the CFBundleIdentifier to the application’s plist. A bundle is a directory structure that organizes executable and resources belonging to the application.
- **Include Function Names:** The RuntimeException.Stack property is a String array that contains a list of all of the methods in the stack from the main entrypoint to the point at which the exception was invoked.

The first element (element 0) contains the current function. The methods in the stack continue from there. This feature only works if the Include Function Names property is set to True in the “blessed” App class’s properties.

Default Language

If you have provided support for more than one language via constants (see “Using Constants to Localize your Application” on page 378), you can choose the default language for the build from the Language pop-up menu. The values stored in the ‘vers’ resource are given in Table 44 on page 720.

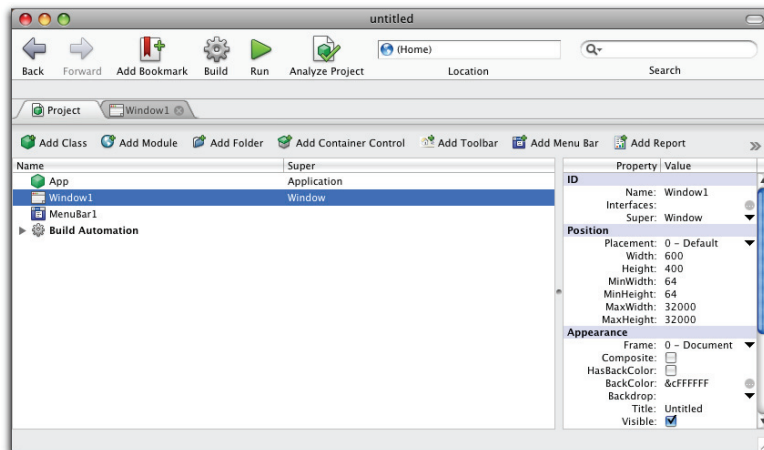
You can select the default region for the application as well.

Build Automation

If you wish, you can choose to automate the build process. For example, you can specify the default language, the target platform, and set property values like build version of the target application. This is entirely optional; you can choose to build your application manually using the normal IDE controls.

You automate the build process using the Build Automation item that is added to every project. The item has blank placeholders for every target platform; you automate the build process by adding script items to the desired platform item.

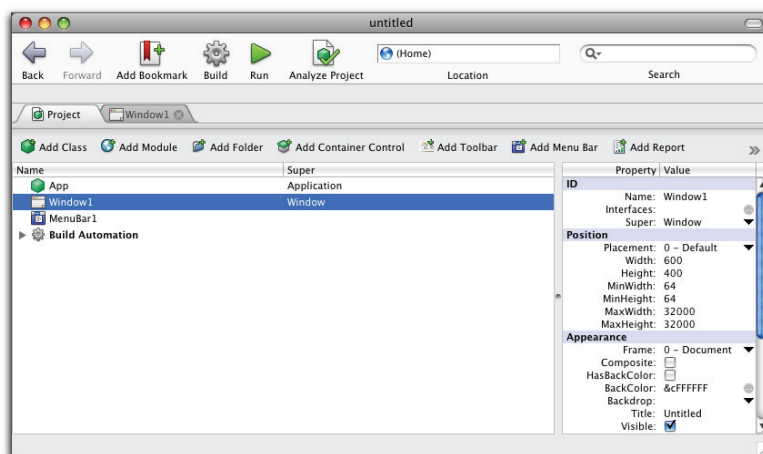
Figure 512. The Project Editor window with the Build Automation item.



Each project can have its own pre and post-build IDE script. With these scripts you can add 'hooks' for third-party tools to run IDE scripts. The scripts are placed in the Build Automation item in the Project Editor window. This item is included in all Desktop Application projects, regardless of whether you want to automate the build process.

Each platform group is populated with one item that represents the build. This item cannot be modified or deleted. Instead, you add script items to the platform group.

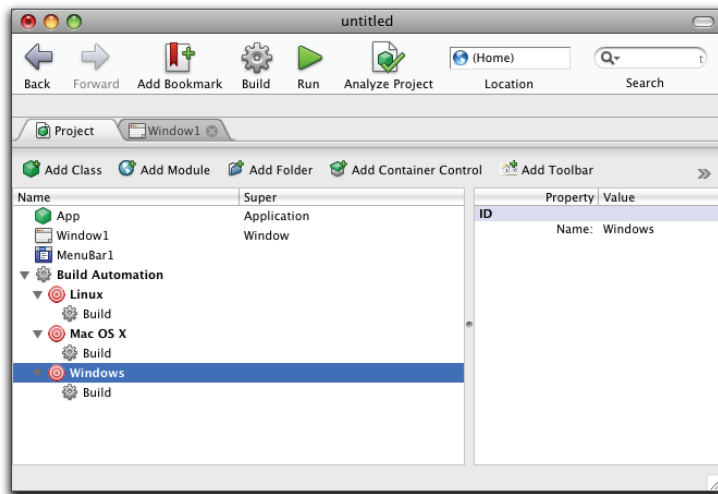
Figure 513. The Project Editor window with the Build Automation item.



Each target contains one item. It represents the build project step for that platform. This step cannot be added, removed, or renamed. It marks the point in the list of build steps where the compilation of the application will occur. It also serves as the

marker that determines what steps happen before the application is compiled and what steps occur after the application is compiled.

Figure 514. The empty build steps in the default desktop application.



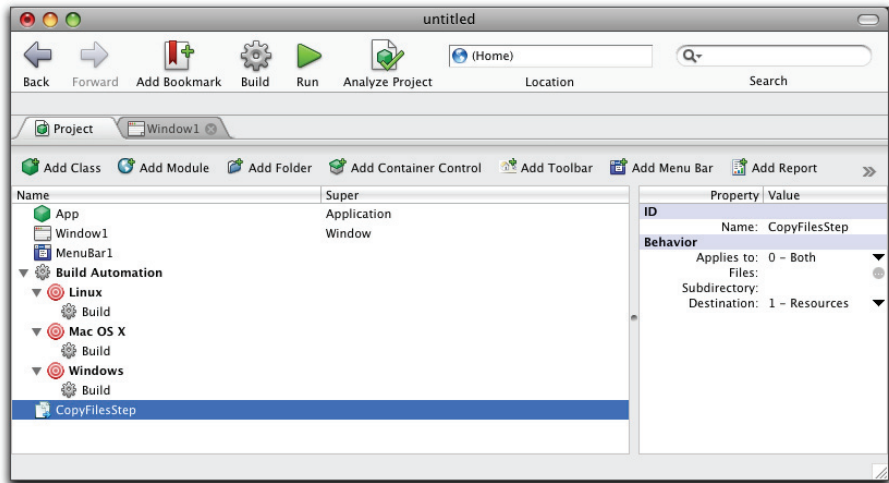
You can add separate build scripts for each target platform (Windows, Mac OS X, and Linux) and you can use the feature to specify what happens before the application is compiled and what happens after the application is compiled.

You add build steps using either the Project > Add > Build Automation submenu or the contextual menu. In the latter case, choose Add to Project > Build Automation to display the submenu items. In both cases, you will see three items:

- Copy files
- New IDE script
- External IDE script

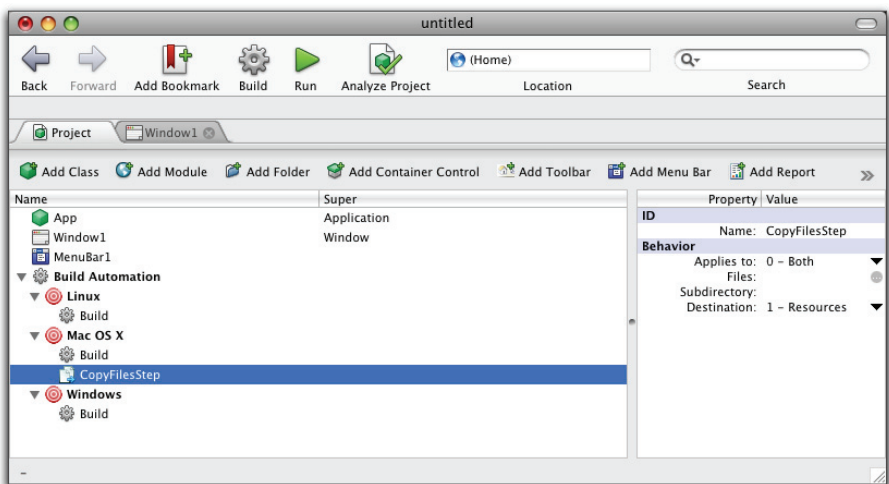
Adding an item does not nest the item inside one of the target platform groups. When it is external to all the Build Automation items, it is disabled from all targets. However, you can configure the item from this position. In all cases, you set the properties of the step to configure it.

Figure 515. A new Copy Files step.



Drag it into a target platform group to make it active:

Figure 516. An enabled Copy Files step.



Here are descriptions of all the script types.

- Copy Files Step** The CopyFiles step copies a list of files to a selected destination. The destination for the copy is specified by the Destination property of the CopyFilesStep. The values are:
- App Parent Folder
 - Resources Folder
 - Framework Folder

■ Bundle Parent Folder

Since REAL Studio is a cross-platform development tool, the destinations are specified in as cross-platform compatible way as possible. This table explains what happens on each platform.

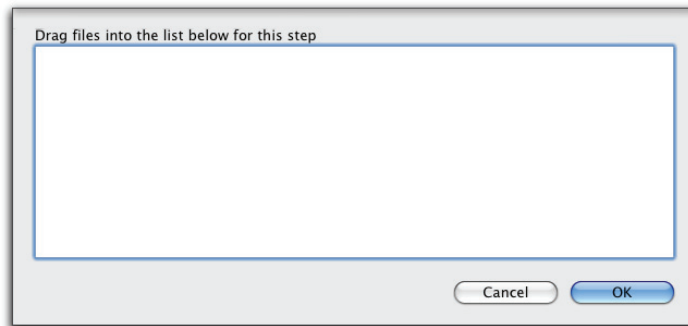
Destination	Mac OS X	Windows	Linux
App Parent Folder	Next to the executable in Bundle > Contents> Mac OS	next to EXE	Next to EXE
Resources Folder	In Bundle > Contents > Resources	Directory containing EXE > Resources	Directory containing EXE
Framework Folder	In Bundle > Contents > Frameworks	Directory containing EXE <i>appname</i> Libs	Next to EXE
Bundle Parent Folder	Directory containing the bundle	Directory containing EXE	Directory containing EXE.

Use the Properties pane for a Copy Files step to specify the Destination from the pop-up menu.

The Subdirectory can also be entered as a string. The specified directory will be created and files copied into it. It will not create an entire relative directory path at present.

Use the Files property to add files. It offers a dialog in which you can add the files.

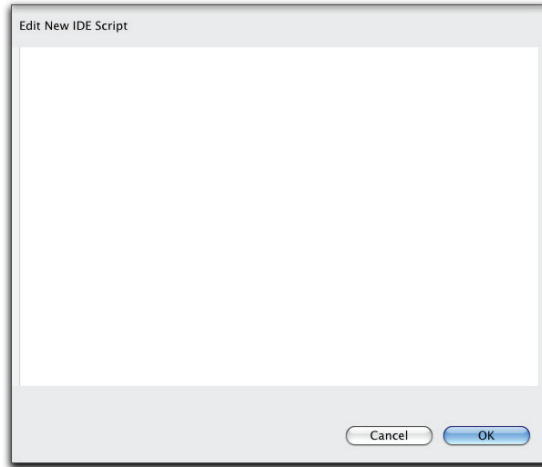
Figure 517. The Files dialog box.



New IDE Script

A new IDE script is created inline. It is not stored externally to your project like other (external) IDE scripts. If you send a project that contains an internal IDE script, the script will be included as part of the project. You add the internal script as the Script item in the Properties pane. Click the Script property to open a dialog in which you can enter the script.

Figure 518. The New IDE Script dialog.



There are two new variables available for IDE build scripts, `CurrentBuildLocation` and `CurrentBuildTarget`. `CurrentBuildLocation` is the shell path to the item being built. It is not valid in pre-build steps and will be blank. This location will be wherever [GetFolderItem](#) would try to locate the file. `CurrentBuildTarget` is an [Integer](#) value that you can use to determine what kind of executable is being built. It uses the same constants as `BuildApp`.

External IDE Script

An external IDE script is stored externally to your project. It is just a reference to an external file. You specify the script file using the File property of the External Script's Properties pane. Click in the File area to open a browser area.

Since an external script is a reference, they are not included as part of the project. If you send such a project to another person, you will need to include the external script file or files.

Preparing your Application for Compilation

Although the process of creating a standalone application is very simple, you should take into account any special features or limitations of the target operating system so that you can optimize your application's performance.

REAL Studio offers a very convenient way of testing your application under a different operating system: remote debugging. If you have a computer that is running a second target operating system, you can use remote debugging to send it a debug build of your application and launch it automatically. Remote debugging is available in the Professional and Studio versions of REAL Studio.

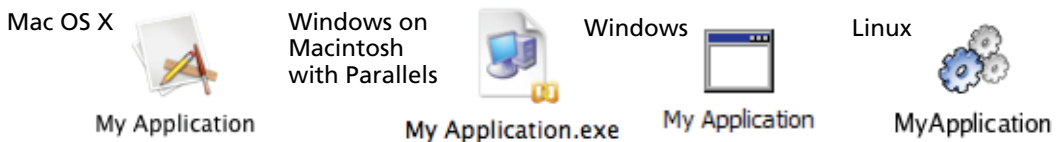
For information on how to configure your computers for remote debugging, see the section "Remote Debugging" on page 657.

This section covers some of the issues affecting the behavior of an application on each of the platforms.

Compiling for Windows

To compile for Windows, check the Windows CheckBox in the Build Settings dialog box area and enter the name of the .exe file in the Windows group in the App object's Properties pane. The compiler will create a single executable application for the Windows environment. On Windows, the application will have a generic application icon unless you have provided a custom icon. On Macintosh, the Windows application will have a Parallels badge if Parallels is installed. You can double-click the Windows build to open your Windows application in Parallels (assuming you have a Windows virtual machine).

Figure 519. Windows builds of a REAL Studio application.



As discussed in the section “Building for Windows” on page 696, a windows build will, in general, result in a folder that contains the .exe file plus a folder that contains all the .dlls that the application uses. The build will produce only the .exe file only in the special case in which the application does not use any plug-ins.

Windows Considerations

Before building an application for deployment on Windows, you should check the following issues for compatibility:

- **End of Line Characters:** When inserting text into TextFields or TextAreas via code, keep in mind that Returns are followed by Line Feeds. Use the EndOfLine function rather than hardcoding end of line characters.
- **Non-ASCII Codes:** Characters above ASCII 127 are not identical on all platforms and also differ by language. For example, the bullet character on the Macintosh is 165 but on Windows it is 149 (US operating systems). When you read in text from an outside source, you can use the optional Encoding parameter of the Read, ReadLine, or ReadAll methods to specify the encoding that the incoming text uses. Internally REAL Studio stores the encoding with each text string. If you need to write text out to an application or platform that expects a particular encoding, you can call the ConvertEncoding method to convert the text to the desired encoding.
- **Windows GUI:** If you are developing your application on Macintosh or Linux, it's important to consider Windows user interface guidelines. Otherwise, your application will look like a port from another platform to Windows rather than a genuine Windows application.
- **Control Order:** If you are developing the application on Macintosh for deployment on other platforms you should check the control order since controls like

CheckBoxes and PushButtons can get the focus on Windows and Linux. Make sure you test the control order on other platforms before building your application.

- **Multiple Document Interface:** Some MDI (multiple document interface) applications on Windows maximize the MDI frame window when launched. The MDI frame window is the parent window in which the applications windows open. If you are compiling a Win32 version of your project and would like to maximize the MDI frame window when your application launches, you can call the Maximize method of the MDIWindow class in the Open event in the App object. The the App object is added to your Project Editor automatically. Be sure to specify that you want the application to run in an MDI window in the Windows settings group of the App class's properties.
- **Mac-only Controls:** Some control are unique to the Macintosh and don't have a Windows or Linux counterpart. For example, the SpotlightQuery control supports the Macintosh-only Spotlight API.
- **Windows Menus:** The text of certain standard menu items on Windows are different. For example, "Exit" is used instead of "Quit". You can use the REAL Studio constants system to localize your application's interface for Windows in this respect (see "Using Constants to Localize your Application" on page 378). The Exit or Quit menu in REAL Studio is already handled via constants in the App class.
- **Mac-only Features:** A few REAL Studio capabilities are Macintosh-specific. For example, AppleEvents and AppleScript, and the DockItem class. Another example is the Permissions class, which is for Linux and Mac OS X only. You can use conditional compilation to isolate this code. It uses the structure:

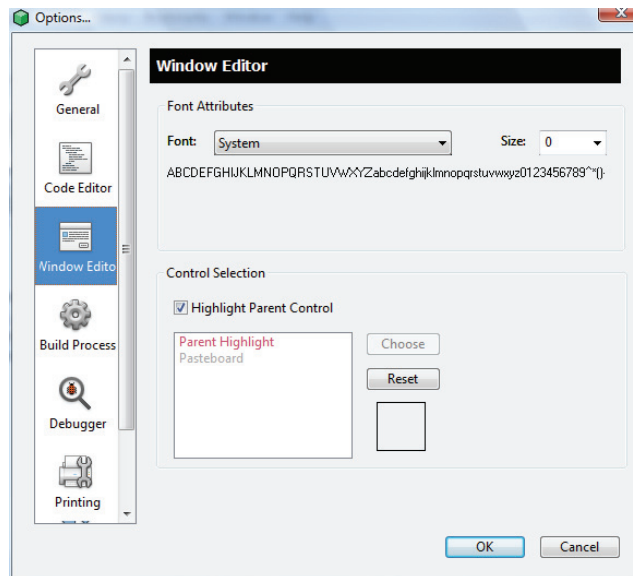
```
#If TargetBoolean then
//platform-specific code, included in
//the built app when TargetBoolean is True
#elseif TargetBoolean2 then
//platform-specific code for TargetBoolean2
#endif
```

TargetBoolean is a boolean constant or constant expression that lets you selectively include code that will be included only in a particular build. You can use the built-in boolean target compiler constants TargetMacOS, TargetMachO, TargetHasGUI, TargetCarbon, TargetLinux, TargetWin32, or Targetx86. These constants test *which type of code is compiling*. You can also use constant expressions that evaluate to True or False. For example, you can use the RBVersion and the RBVersionString constants to determine whether the user is using a particular version of REAL Studio or at least a minimum version level, e.g., version 2005 and above.

See the section in the *Language Reference* on Cross-Platform Development and the descriptions of the conditional compilation constants for more information.

- **Platform-specific FolderItems:** Some of the functions that return references to Mac OS-specific or Linux-specific folders will return Nil on Windows and vice versa. See the entry for SpecialFolders in the *Language Reference* and access special OS FolderItems via this module.
- **API Calls:** With the Declare statement, you can make API calls for Macintosh, Linux, or Windows platforms. Of course, the nature of the call will differ by platform. Use conditional compilation to isolate both the Declare statements and your usage of your toolbox calls later in your code.
- **Font sizes for controls:** The three target platforms supported by REAL Studio have their own conventions regarding default system fonts and font sizes. Often, a font size that looks good on one platform is too large or too small on another platform. For that reason, the REAL Studio Options dialog box has an option for choosing the default font size for the target platform to use as the default font size for all controls that use text. This, in effect, allows you to choose more than one default font size for controls simultaneously. Choose a font size of zero to use this option.

Figure 520. Choosing the default font size for the target platform for controls.



You can also set the TextSize property of any control to zero to invoke this option for the particular control (if the option is not selected globally, as shown in Figure 520). You can also specify the System font or the SmallSystem font rather than a specific font to instruct REAL Studio to use the system (or small system) font for the target platform.

Mac OS X Considerations

Before building your application for Mac OS X, you should carefully test all the interface elements to make sure that they look right (i.e., buttons, Popup menus, ListBoxes, etc. are properly sized and look good with the fonts used in Mac OS X). Also any shared libraries that the application uses must be carbonized.

If you are a Windows or Linux user and are cross-compiling for Mac OS X, you should familiarize yourself with the Apple's Human Interface Guidelines. It is a detailed specification for Apple-recommended user interface design. The complete user interface guidelines are at:

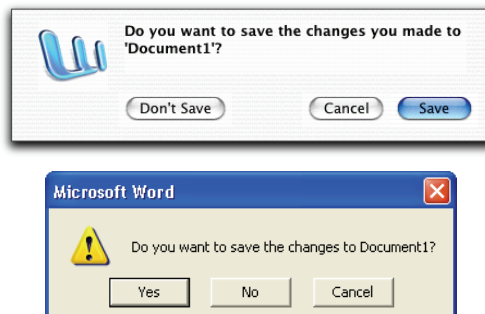
<http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/>

The following are some of the major differences between Apple interface conventions and those of other platforms. Some differences can be handled via careful interface design while others may require alternate versions of windows, dialogs, and other interface elements. To manage alternate versions of objects, you can use the `#If` statement to conditionally compile for different target platforms. See the *Language Reference* for information about `#If` and the target platform constants used for conditional compilation.

- Macintosh uses a single menubar that is always at the top of the screen. Individual windows do not have menubars. Although REAL Studio supports multiple menubars that can be associated with different windows, the use of this feature is contrary to Apple user interface guidelines.
- The Exit menu item on Windows is named “Quit” on Macintosh. You can use the REAL Studio constants system to localize your application's interface for Windows in this respect (see “Using Constants to Localize your Application” on page 378). This is done within REAL Studio, for example. The `App` class contains a constant that is used to set the text property of this menu item.
- An application's preferences menu item is located in the application's own menu, rather than under the File, Edit, or Tools menu. REAL Studio has a special class for handling this issue, the `PrefsMenuItem` class. Use this class rather than the `MenuItem` class for your Preferences menu item.
- Macintosh does not support Windows' Multiple Document Interface. This makes windows document-centric rather than application-centric because a document window is never enclosed in a parent window that identifies the application and holds the application's menubar. Moreover, each Mac OS X window exists in its own layer in the Finder. That is, clicking on one of an application's windows does not bring all of the application's windows to the front. As a result, a Macintosh user can interleave one application's windows with other application's windows. If you designed your application as an MDI application, be sure to check out its behavior in a non-MDI environment.

- Historically, the standard Macintosh mouse is a one-button mouse, although Apple and third-party multiple-button mice are now available and supported by Mac OS X. However, you can't expect that all your users will have a three-button mouse. On Macintosh, the equivalent of the Windows' right+click gesture is Control-click. Be sure that your application does not rely on mouse gestures that are not supported by the one-button Apple mouse.
- Mac OS X uses the Dock as an application and document launcher. REAL Studio's DockItem property of the Application class enables you to control the appearance of your application's Dock icon and the DockItem property of the Window class enables you to control the appearance of the icon representing one of your application's documents (When a Mac OS X user minimizes a document window, it appears as an item in the Dock). These icons should be designed as 128 x 128 pixel icons.
- In dialog boxes, PushButton captions use verbs rather than "Yes" and "No" and are arranged differently. Typically, the positions of the "validate" and "cancel" buttons are reversed. For example, the two dialogs shown in Figure 521 are the "save changes" dialogs from Microsoft Word for Mac OS X and Windows XP. As you can see, the arrangement and wording of the PushButtons are significantly different. Also, modal dialogs such as this are implemented on Mac OS X as sheet windows when they pertain to a particular window.

Figure 521. "Save Changes" dialogs on Macintosh OS X and Windows XP.



- Toolbars that contain many, many small icons are discouraged by Apple. Apple favors the use of a few "high quality, larger" icons in the toolbar which are labeled with text below the icon rather than via a hidden "tips" window. REAL Studio's Toolbar class can be used to create such toolbars. Mac OS X applications also use the Drawer window to reveal additional choices or options when a toolbar icon is clicked. Apple also recommends the use of floating palettes as an alternative to extensive toolbars.
- There is a standard list of reserved keyboard shortcuts for Macintosh. Your application should not try to override them. Aqua user interface guidelines also specify certain recommended keyboard shortcuts for common operations, such as the

keyboard equivalents for the standard File and Edit menu items. These keyboard shortcuts are listed in the Aqua Human Interface Guidelines, available at the URL given above.

- Carbon applications include an automatically generated 'plst' resource which describes your application to the Mac OS X Finder. If you wish, you can override this by including your own 'plst' 0 resource in a resource file in the project.

Linux Considerations

If you are building your application for Linux from another platform, you should carefully check out the appearance of the Linux version of your interface and adjust the font size and size of controls for maximum readability. For example, controls such as PushButtons are too short to display text properly when drawn at the standard height for Macintosh and Windows.

Here is a method that examines all the RectControls on a layout and enlarges the appropriate ones for the Linux version of the application:

```
// For Linux we are going to set the new control height to 28
const kNewHeight = 28

Dim i as Integer
Dim ctl as RectControl

// loop through all of the controls on the window
For i = 0 to ControlCount - 1
    If NOT control( i ) isa rectControl then
        Continue // move to the next control in the loop
    End If

    // Cast it as a RectControl that has a Height property
    ctl = RectControl( control(i) )

    If ctl.height < kNewHeight then

        If (ctl isa pushButton) OR (ctl isa staticText) OR (ctl isa TextField) OR _
            (ctl isa popupMenu) OR (ctl isa comboBox) then
            ctl.height = kNewHeight

        End if

    End if
Next
```

You should keep in mind the following information.

Requirements Linux applications run on only on x86 machines and REAL Studio desktop applications require GTK+ 2.8 or above (which has its own requirements, such as GDK, Pango, Atk, etc.), glibc-2.3 or above, CUPS, and libstdc++.so.6. Depending on

your distribution of Linux, you may not already have this pre-installed. However, you can download and install the GTK+ libraries from <http://www.gtk.org>. You may also find pre-built libraries from your Linux distributor's home page or GTK may be included on your distribution CD as an optional install item.

General Information

Some controls and operations that work on Macintosh and/or Windows have limitations under Linux. This is detailed below.

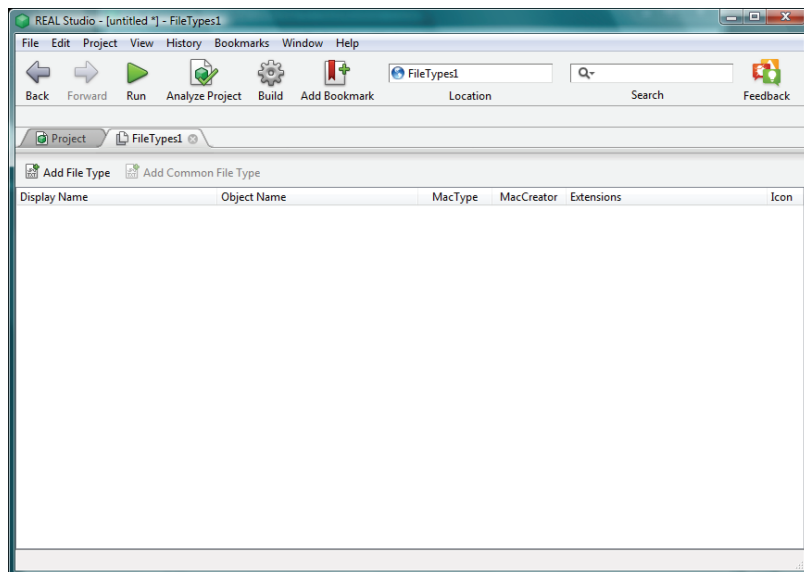
- **Sounds:** Requires `libsndfile` (<http://www.zip.com.au/~erikd/libsndfile/>) but is weak linked to your app, so your app should still run even if the end user doesn't have this library installed, but he or she can't play any sounds.
- **Canvas:** Scroll will redraw the Canvas's contents if there are certain controls on top of the Canvas, such as a `PushButton`.
- **Drag and Drop:** Dragging over controls that have mouse over effects, such as `PushButtons`, do not update the drag bounds properly.
- **TextField and TextArea:** The `SelCondense`, `SelExtend`, `SelOutline`, and `SelShadow` styles are not supported. These are nonstandard text styles that were introduced with Mac OS "classic" and not supported on more recent OSs.
- **FolderItem:** Setting the creation date of a file or folder is not supported.
- **Menus:** A window cannot be assigned the same menubar as another. If you want two windows to have the same menus you need to duplicate the menubar.
- **QuickTime** related classes are not supported.
- **PushButton, CheckBox, and RadioButton:** You can't suppress the depressed state of buttons, `CheckBoxes`, and `RadioButtons` when returning `True` from the `MouseDown` event.
- **ScrollBars:** `ScrollBars` behave a little bit differently than on Macintosh and Windows. Clicking on the arrow buttons without `LiveScroll` does not trigger the `ValueChanged` event until you release the mouse button.
- **Speak:** The `Speak` method is not supported in the initial release.
- **RadioButtons:** In a group of `RadioButtons`, one must always be selected.
- **PopupMenu:** There is no support for `Separators`.
- **Printing:** Printing requires `libgnomeprint 2.2` (or above) and `CUPS` installed. Printing is supported only to PostScript printers. On Linux, there is no Page Setup dialog. Calling the `PrinterSetup` method returns `False` and no dialog is shown to the user.

Assigning Custom Document Icons

As you already know, you can assign a custom application icon in the App class's properties pane. If your application creates documents, you will probably want to

assign icons that match the theme of your application icon, to the documents it creates. This can be done in the File Type Sets Editor. To add the appropriate document icons, click the Icon button in the File Types Sets Editor (Figure 522) to display the custom file type dialog box. For more information, see the section “Adding a Document Icon” on page 485.

Figure 522. The File Types Set Editor.



Region Codes

The following table gives the Region Codes that are used in the ‘vers’ resource for built applications.

Table 44: Region codes used in the ‘vers’ resource.

Code	Value	Code	Value
00	US	22	Malta
01	France	23	Cyprus
02	Britain	24	Turkey
03	Germany	25	Yugoslavia
04	Italy	33	India
05	Netherlands	34	Pakistan
06	Belgium-Lux.	36	It. Swiss
07	Sweden	40	Anc. Greek
08	Spain	41	Lithuania
09	Denmark	42	Poland

Table 44: Region codes used in the 'vers' resource.

Code	Value	Code	Value
10	Portugal	43	Hungary
11	Fr. Canada	44	Estonia
12	Norway	45	Latvia
13	Israel	46	Lapland
14	Japan	47	Faeroe Isl.
15	Australia	48	Iran
16	Arabia	49	Russia
17	Finland	50	Ireland
18	Fr. Swiss	51	Korea
19	Gr. Swiss	52	China
20	Greece	53	Taiwan
21	Iceland	54	Thailand

Converting Visual Basic Projects to REAL Studio

Because of the similarities between REAL Studio and Visual Basic, creating a Macintosh version of a Visual Basic application is fairly easy. REAL Studio can save you hours of time by handling the tedious job of recreating the interface and pasting in your code into all the various event handlers and methods.

Contents

- VB Migration Assistant
- Database Options

VB Migration Assistant

REAL Studio has a utility that makes the conversion much easier. The VB Migration Assistant is available as an optional download at the REAL Software web site. Use the Migration Assistant as the first step in converting the VB application.

The VB Migration Assistant (VBMA) is a cross-platform REAL Studio application. It saves you time by doing a lot of the conversion involved in porting from Visual Basic to REAL Studio.

VBMA creates a REAL Studio project from the contents of your Visual Basic project. Specifically, it moves over forms, modules, and classes. Rather than having to copy and paste forms and code, VBMA handles this part for you. Since the controls in REAL Studio are not identical to those in Visual Basic, VBMA provides a mapping of Visual Basic controls to REAL Studio controls. The common controls are mapped by default. Controls that VBMA does not recognize can be mapped to REAL Studio controls by you before VBMA converts your project to a REAL Studio project.

What doesn't it do?

While Visual Basic code and REAL Studio code are very similar in some respects, they are very different in others. VBMA does not do any code conversion. It does make an effort to move code over, but it makes no attempt to convert Visual Basic code into REAL Studio code. In fact, it comments out all the code that it moves over. This is done to make it easier for you to convert your code one method or event at a time. You simply uncomment your code then make the necessary changes to make it work in REAL Studio. Active X/COM controls are not cross-platform and VBMA makes no attempt to make them work cross-platform. In fact, at the moment REAL Studio doesn't support visual COM controls. However, some COM controls have cross-platform equivalents built right into REAL Studio or available from third party developers.

Supported Versions of VB

Visual Basic versions 5 and 6 are supported. VB.NET and Visual Basic Express are not currently supported.

Third-party Controls

The fewer third party controls your project uses, the better candidate it is for porting to REAL Studio. Projects with lots of third party controls can be ported but VB Migration Assistant will be less useful and you will have to do more work than with a project that uses the standard Visual Basic 5/6 controls.

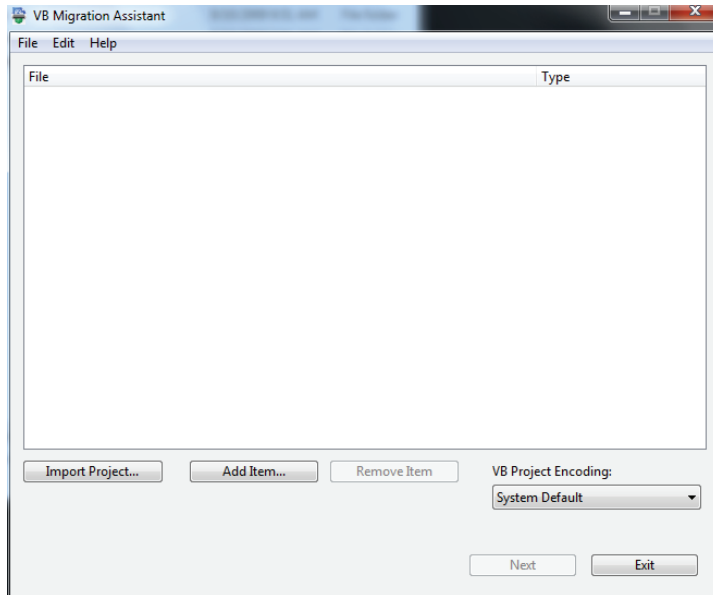
Converting a VB Project

To convert as VB project, do this:

1 Launch VB Migration Assistant.

The VB Migration Assistant window appears.

Figure 523. The VB Migration Assistant window.

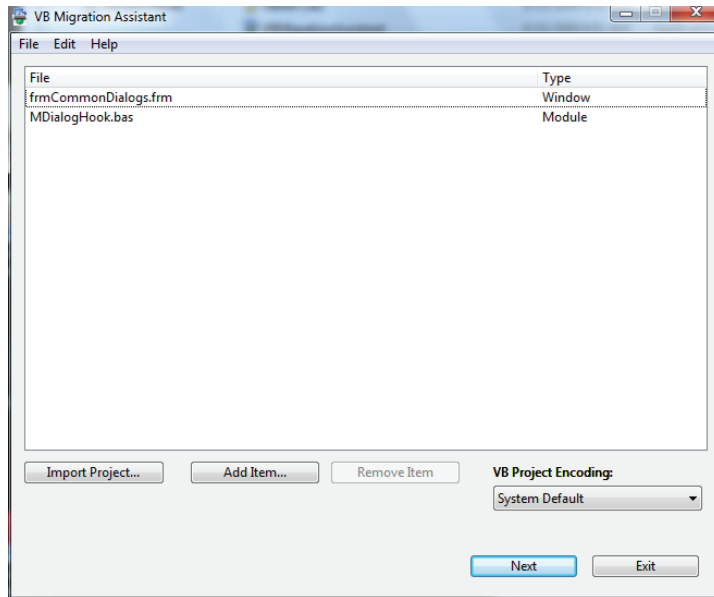


2 Click the Import Project button and choose your Visual Basic project file.

Alternatively, you can drag your .frm, .bas, .cls and .ctl files into the file list in the window.

The items to be converted appear in the window.

Figure 524. An opened VB project.

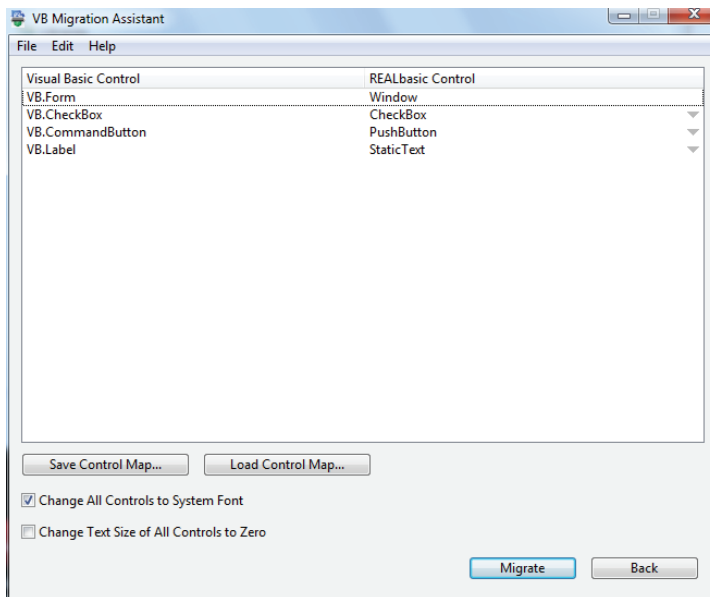


3 Set the VB Project Encoding if necessary.

4 Click the Next Button.

The proposed mapping appears. Each mapping can be changed using its pop-up menu.

Figure 525. Setting the mapping.



- 5 Adjust the control mapping if necessary.**
- 6 Click the Migrate button.**
- 7 When VB Migration Assistant is finished, save your REAL Studio project.**
 VB Migration Assistant will attempt to launch REAL Studio and open your migrated project.

Encoding Issues on Windows

If your Visual Basic project was created using a version of Windows for a different language than the one you are running VB Migration Assistant, you will need to set the VB Project Encoding combobox to the language of the system upon which you created your VB project.

Encoding Issues on Macintosh and Linux

If you are running VB Migration Assistant on the Macintosh or Linux, you will need to set the VB Project Encoding pop-up menu/combobox to the language of the Windows OS upon which your project was created.

Non-English File Names

If some of your VB files have non-English names, you will need to choose the proper VB Project Encoding before you migrate your project. If VB Migration Assistant can't find the files from your project, try importing them individually by clicking the Add Item button.

Auto-opening your Project VB Migration Assistant should automatically open your project using REAL Studio or REAL Studio once it finishes migrating it. If this doesn't happen, choose Options from the Edit menu (Preferences from the VB Migration Assistant menu on Macintosh) and select the REAL Studio application you wish to use when opening migrated projects.

Database Options

Visual Basic applications often use Microsoft Access or the Jet database engine that comes with Visual Basic to provide single-user database capabilities. You can convert these applications to use the built-in REAL Studio database engine or any other supported data source. The REAL SQL Database data source is included in all versions of REAL Studio.

For multi-user applications, you can use REAL Server, available as a separate product from REAL Software. The REAL Server is available to REAL Studio Professional customers and is included as part of REAL Studio Studio. The REAL Server runs on Linux, Macintosh, or Windows as a background application. It can be administered by a separate Admin application (included with the product). It also includes a plug-in that adds the `REALSQLServerDatabase` class to the REAL Studio framework. With the plug-in, you can develop full-featured multi-user database applications.

Index

Symbols

&c literal 216, 378
#error option 391, 589
#If statement 262

Numerics

4th Dimension 604

A

About REAL Studio menu command 73
absolute path 492
 creating an 492
accelerator
 entering a 380
accelerators
 creating 198
AcceptFocus property 134
AcceptTabs property 132, 134
Access Scope 225, 324, 331, 370, 400
Action event handler 272, 280, 344, 349, 350, 538
ActionButton 111
Activate event handler 582
Active X Help 176
ActiveX components 174–176, 691–692
ActiveX controls 174–176
Add Bookmark button 40, 57
Add Bookmark dialog box 60
Add Bookmark Folder dialog box 72
Add Bookmark Folder menu command 71
Add Bookmark menu command 71
Add Constant declaration area 326, 380, 554
Add Folder button 79
Add Index screen 609
Add Menu button 195
Add Menu command 65
Add Menu Item button 199
Add Method button 555
Add Method declaration area 556
Add Note button 248
Add Separator button 205
Add to Class menu command 82
Add to ModuleName Menu command 81
Add to Project Menu command 81
Add Window button 90, 102
AddPicture method 526, 528
AddRawData method 453
AddResource method 528
Address property 669
AddRow method 241
Aggregates property 587
aliases
 importing 306
 of folderitems 491
Align menu command 64
alignment guides 183
alignment icons 183
All Controls list 114
AlternateActionButton 111
Always Show Tabs option 54
Analyze Item menu command 68
Analyze Project command 632
Analyze Project menu command 68, 632, 699
And operator 250
App class 580–583
 event handlers 581
 properties of 700
 Scope of methods 582
 Scope of properties 582
App class properties 105
Append method 232
AppendToTextFile method 512
AppleEvent object
 Send method 687
AppleEvent objects
 creating in REAL Studio 687
AppleEvents 714
 communicating with 687–688
 receiving 687–688
 required 688
 sending 687
 sophisticated 688
AppleScript 714
 adding to a project 685
 adding to REAL Studio 685
 calling 686
 passing values to 686
 preparing an 685
 returning values from 686
AppleScripts
 calling 685–686
 importing 306
Application class 527, 529, 580–583, 655, 687
 event handlers 581
 methods of 582
 NewDocument event handler 529
 properties of 582
Application menu
 for Mac OS X 202

- ArcShape class [442](#)
- Arrange menu command [64](#), [131](#)
- array
 - assignment [232](#)
 - declaring an [229–230](#), [316](#), [374](#)
 - definition of [229](#)
 - index of [229](#)
 - passing as a parameter [242](#), [330](#)
 - passing by reference [245](#), [337](#)
 - resizing an [230](#)
 - returned by a function [557](#)
 - returning from a function [331](#), [371](#)
- array element
 - referring to an [230](#)
- Array function [231](#), [439](#)
- arrays [229–243](#)
 - Append method [232](#)
 - converting [233–234](#)
 - declared as a property [545](#)
 - declaring [229](#)
 - in structures [395](#), [562](#)
 - initializing [231](#)
 - Insert method [232](#)
 - Join function [234](#)
 - multi-dimensional [230](#), [243](#), [330](#)
 - of classes [575](#)
 - one-based [229](#)
 - ParamArray keyword [330](#)
 - Redim statement [233](#)
 - Remove method [233](#)
 - resizing [232–233](#)
 - Split function [233](#)
 - zero-based [229](#)
- ASCII [416](#)
- ASCII character codes [416](#)
- ASCII encoding [416](#)
- Assigns keyword [340](#), [573](#)
- AttributeInfo [597](#)
- attributes [84](#), [237–239](#)
 - accessing [239](#)
- Attributes Editor [84](#), [238](#)
- Auto Adjust TabIndexes menu [64](#)
- Auto Adjust TabIndexes menu command [70](#)
- Auto Adjust TabIndexes menuitem [183](#)
- autocomplete [291–293](#)
 - in Location area [57](#)
- Autocomplete applies standard case [276](#)
- Autocomplete shows details [276](#)
- AutoDiscovery class [167](#)
- AutoEnable property [193](#), [582](#)
- automating
 - REAL Studio [62](#)
- B**
- Back and Forward buttons [40](#)
- Back button [55](#), [70](#)
- Back menu command [70](#)
- Backdrop property [519](#)
 - of Canvas control [430](#)
 - to display a picture [428](#)
- BASIC
 - compiled [20](#)
 - disadvantages of interpreted [212](#)
 - history of [20](#), [212](#)
 - interpreted [20](#)
 - object-oriented [20](#)
- Behavior in Window Editor dialog [547](#)
- BevelButton
 - adding items to a [155](#)
 - as popup menu [154](#)
 - No Bevel option [141](#)
- big endian byte order [263](#)
- binary file
 - reading a [522–523](#)
 - writing to [523–524](#)
- binary files
 - benefits of [522](#)
 - compared to text files [514](#)
 - definition of [521](#)
 - random read/write access to [522](#)
- BinaryStream [523](#)
 - compared to TextInputStream [522](#)
 - definition of [522](#)
 - encoding of [523](#)
- BinaryStream class [522](#)
- bitwise comparisons [251](#)
- BOF property [624](#)
- bookmark folders [57](#)
 - adding [71](#)
- bookmarks
 - adding [71](#)
 - customizing [60](#)
 - global [57](#), [60](#)
 - local [57](#), [60](#)
 - modifying [60](#)
- Bookmarks bar [40](#)
 - adding a bookmark to [59](#)
 - adding items to [57](#)
 - adding methods and properties to [60](#)
 - contextual menu [60](#)
 - hiding the [41](#), [68](#)
 - IDE window [59](#)
- Bookmarks bar menu command [70](#)
- Bookmarks dialog box [57](#)
- Bookmarks menu [57](#), [71](#)
- Boolean
 - data type [215](#)
- branching
 - definition of [260](#)
- Break keyword [637](#)

Break on Exceptions 66, 652
 Break statement 262
 breakpoint
 definition of 637
 removing a 637
 breakpoints
 in stand-alone applications 637
 Bring All to Front menu command 72
 broadcasting
 using UDPSocket 678
 Browser 273–280
 contents of 273
 contextual menus in 297
 expanding and collapsing categories 296
 Find and Replace window 299–301
 hiding the 294
 use of bold in 280
 using to access code 279
 viewing items in 278
 buffer
 definition of 667
 bug reports 73
 bugs
 logical 632
 reporting 30
 syntactical 632
 Build Application menu command 68, 698
 build automation 707–712
 Build button 56, 68, 698
 Build Process preferences 698
 Build Progress dialog box 698
 Build Settings dialog box 378, 490, 707
 Build Settings menu command 68
 Build unsaved apps in option 698
 Builds menu command 72
 Builds window 699
 Built-in controls 113
 Bundle Identifier 707
 Bundle Identifier property 707
 ByRef
 passing arrays 337
 ByRef keyword 244, 337
 Byte data type 214
 ByVal keyword 244

C

Can't Undo menu command 63
 CancelButton 111
 Canvas control 132, 147, 161–162, 560
 AcceptFocus property 134
 Backdrop property 430, 516, 519
 copying a picture in a 432
 creating custom controls with 440, 583–585
 drawing in a 437
 getting the focus 133
 MouseDown event 440, 518
 Paint event 437, 441
 Paint event handler 584
 redrawing 441
 saving image drawn in 518
 carriage return
 used in writing to text files 512
 casting 576
 CellClicked event definition 560
 CellClicked event handler 584
 CStringRef data type 219
 character
 getting a non-ASCII 419
 character set
 defined 416
 CheckBox 142–143
 Checkbox
 getting the focus 135
 child control 185
 Child method 492
 Chr function 243, 419
 CICN resource 526
 class
 accessing properties and methods of 579
 based on a control 578
 creating a 539–540
 definition of 577
 extending a 401
 with no super class 578
 class constants 221
 class extension methods 401, 568
 class interface 387–392, 585–593
 creating a 387–389, 586–588
 definition of 585
 example 591–593
 implementing a 389–392, 588–591
 modifying a 66
 class interfaces
 #error option 391, 589
 deleting 591
 Extract Interface dialog 84
 Implement Interfaces dialog 83, 390, 589
 classes
 adding event definitions to 392, 558–561
 adding methods to 555–557
 adding new events to 392, 558–561
 adding properties to 544
 arrays of 575
 built-in 533
 casting 576
 constructors 569
 custom 596
 definition of 531
 deleting 602
 destructors 571

- encrypting 599
- exporting 542, 599
- exporting protected 599
- extending 568–569
- extension methods 401
- importing 306, 598–599
- not based on controls 578–579
- removing from memory 580
- saving 542
- saving as an external project item 542
- shared among projects 80
- Clear All Breakpoints menu command 66
- Clip method 439
- Clipboard
 - getting data from 452
 - putting data on 452–453
 - testing for data types 451
 - transferring text and graphics with 451–453
- Clipboard class 451, 452
 - AddRawData method 453
 - Picture property 452
 - PictureAvailable method 451
 - RawData property 452
 - RawDataAvailable method 451
 - SetText method 452
 - Text property 452
 - TextAvailable method 451
- Close event handler 581
- Close Tab menu command 62
- Close Window menu command 62
- CMY color model 216
- CMY function 445, 446
- code
 - commenting 247
 - copying and pasting 304
 - encrypting exported 307
 - exporting 307
 - line by line execution 648–649
 - protecting 599
 - step in 648
 - step out 648, 649
 - step over line of 647, 648
- Code and Window Editors Share a Tab option 48
- Code Editor 46, 273–304
 - accessing the 369
 - autocomplete 291–293
 - auto-completion in 283
 - breaking up long lines in 288
 - Browser 273–280
 - contextual menus 297
 - defined 46
 - entering code in 283
 - font size 304
 - Notes 248
 - opening 274
 - opening its window 296
 - Options 275–276
 - parameter line 282
 - printing code in 304
 - resize bar 294, 295
 - resizing 294
 - resizing panels in 294
- code execution
 - Step In option 648
 - Step option 647
 - Step Out option 648, 649
- color
 - assigning to a property 130
 - constants 378
 - data type 215
 - inserting into the Code Editor 287
 - working with 445–448
- Color Picker 378, 446
 - RGB 447
- COM Component help 176
- ComboBox control 132, 145, 154
 - adding items to 154
 - getting the focus 132
- CommandLineArgs property 707
- Comment menu command 64
- comments
 - adding to code 247
 - multiline 247
- comparison operators 249–250
- compilation
 - incremental 697
- compilation process
 - halting the 637
- compiler constants 263
- composite index 610
- computed properties 321, 549
- computed property
 - example of 551
 - Get method 321, 550
 - Set method 321, 550
 - writing a 321, 549
- conditional compilation 262, 714–715
- Console Application 76
- Console application 695
- console application
 - creating a 76
- Console Application project 76
 - contents of 76
- Console Application template 76
- ConsoleApplication 76
- Const statement
 - scope of 237
- constant
 - class 221
 - global 129

- Local 375
- constants 236–237
 - adding to a class 553–555
 - class 221
 - color 378
 - copying and pasting 304
 - dynamic 554
 - dynamically localizable 379
 - entering into Properties pane 128
 - for localizing an app 379–382
 - in Dim statements 230
 - local 324
 - module 375–383
 - window 324–327
- ConstructorInfo 597
- constructors 228, 341, 569
- Container Control
 - modifying a 66
- Container Control control 177
- contextual menu 103
 - accessing in Code Editor 297
 - Add Control item 117
 - Edit item 274
 - Find Item 303
 - in Browser 297
 - Make External command 83
 - Make Internal command 83
 - Project Editor 81–84
 - Select All item 118
 - Select item 118
 - selecting controls using 118
- control
 - adding to a window 38
 - child 185
 - locking a 125
 - parent 185
 - position of 121
- control array 348–350
- control class
 - adding a 578
- control hierarchy 184–189
- control layers 131
- control order
 - changing the 179
- controls 345–349
 - ActiveX 174–176
 - adding to a window 116
 - aligning 121, 183
 - alignment guides 121
 - All Controls list 114
 - appearance of 139
 - BevelButton 154
 - built-in 39, 113
 - Canvas 161–162
 - changing properties of 126–131
 - CheckBox 142–143
 - ComboBox 154
 - creating custom 583–585
 - creating new instances on the fly 346
 - creating subclasses of 533
 - custom 39, 440–442
 - DatabaseQuery 173, 605
 - DataControl 143, 173
 - default font size for 715
 - defined 38
 - definition of 345
 - Disclosure Triangle 165
 - distributing evenly 184
 - dragging from Tools window 116
 - duplicating 139
 - events for 345
 - favorite 39
 - Favorites list 114
 - GroupBox 155
 - Line 160
 - ListBox 149–152, 605
 - lock properties 122–125
 - MoviePlayer 163–164
 - moving 121
 - object hierarchy 139
 - OLEContainer 176
 - Oval 161
 - PagePanel 157
 - plug-in 39, 113
 - pop-up menu 39
 - PopupArrow 164
 - PopupMenu 153
 - Position properties 121
 - ProgressBar 148
 - ProgressWheel 165
 - Project 39, 113
 - PushButton 140, 605
 - RadioButton 143
 - RbScript 165
 - receiving the focus 132
 - rectangle 160
 - removing 131
 - RoundRectangle 161
 - ScrollBar 147
 - selecting 118
 - selecting all 119
 - selecting in reverse order 118
 - selecting invisible 119
 - selecting several 119
 - Serial 166
 - ServerSocket 166
 - Slider 147–148
 - SSLSocket 166
 - StaticText 143, 605
 - TabPanel 156–157, 605

- TCPSocket 166
- TextField 144, 605
- Timer 173
- Toolbar 168–173
- UDPSocket 167
- UpDownArrows 165
- Controls drop-down list 37
- Controls list 37, 38
 - annotated illustration of 113
 - defined 38
 - minimizing the 68
- Convert To Menu button 205
- Convert to Method menu command 289–290
- ConvertEncoding function 419, 513, 672
- coordinates system
 - description of 427
- Copy menu command 63
- CopyFiles 710
- CopyFiles step 710
- counter
 - choice of "i" as 256
 - incrementing a 256
- counter variables
 - in loops 256
 - in nested loops 257
 - typing as Integer 255
- CreateResourceFork method 525
- CreateTextFile method 512, 513
- Creator code 706
 - entering the 490
 - registering 490, 706
- CString data type 218
- CueText property 144
- Currency data type 215
- CurrentExecutingMethodName 245
- CURS resource 526, 526–528
- cursors 306
 - custom 526–528
 - custom, in Windows 527–528
- CurveShape class 442
- custom application icons 701–702
- custom classes
 - creating 539–541
 - overloading 572
- custom control
 - drawing 584–585
- custom controls 39, 440–442
 - creating 583–585
- custom document icons 719, 720
- custom icons 489
 - adding to custom file types 490
- Customize Main Toolbar dialog 58
- Customize Project toolbar dialog box 78
- Cut menu command 63

D

- data source
 - adding and removing a 606
 - selecting a 605
 - specifying a 619
- data sources
 - two or more 604
- data type
 - changing 219
- data types
 - Boolean 215
 - Byte 214
 - CFStringRef 219
 - Color 215
 - CString 218
 - Currency 215
 - declaring with Dim statement 223
 - definition of 213–219
 - Delegate 218
 - Double 215
 - Int16 214
 - Int32 214
 - Int64 214
 - Int8 214
 - Integer 214
 - OSType 219
 - PString 218
 - Ptr 218
 - Single 215
 - String 213
 - Structures 219
 - UInt16 214
 - UInt32 214
 - UInt64 214
 - UInt8 214
 - WindowPtr 218
 - WString 218
- data typing
 - in parameter line 283
- DataAvailable event handler 666
- database
 - creating a table 608–610
 - plug-ins 604
 - queries 623
- Database Binding properties 618
- Database class 622
- database schema 607
- DatabaseQuery control 173, 605, 617
 - properties 617
- databases
 - adding records 627
 - back-end 604
 - building interface for 605
 - choosing a data source 619
 - creating a new REALSQLdatabase 607–610

- creating in IDE 607–610
- data source 604
- DataControl control 618
- editing records 623
- field types 615–617
- in Project panel 306
- indexing fields 609–610
- Mandatory field attribute 609
- modifying records 624
- overview of 604
- plug-in architecture 604
- primary key field 609
- selecting a data source 605
- SQL 604
- Unique field attribute 609
- viewing data in 610
- viewing Schema 607
- DataControl control 143, 173, 618
- Datagram class
 - UDPSockets 678
- datagrams 678
- Date class 218, 227–228
- dates
 - formatting 421–423
- Deactivate event handler 582
- DebugBuild constant 263
- Debugger
 - Edit Code command 648
 - Object IDs in 650
 - Pause command 647
 - Resume command 647
 - Stack drop-down list 649
 - Step command 647
 - Step In command 648
 - Step Out command 648
 - Stop command 647
 - Variables pane 649
- Debugger panel 636
- debugging
 - defined 632
 - definition of 632
 - remote 657–663
 - Stack drop-down list 638
 - Variables pane 639
- debugging machines
 - configuring 661
- declaration statement 223
 - for arrays 229
 - for objects 227
- Declare statement 684, 715
- Decrypt menu command 64, 82
- Decrypt Window dialog box 108
- Default Comment Style 276
- default language 707
 - setting the 378
- default window
 - setting the 104
- Define Missing Method menu command 335
- Delegate data type 218
 - data types
 - delegate 393, 566
- Delete menu command 63, 82
- Delete method 496
- DeleteRecord method 624
- Deselect All menu command 64
- Desktop Application 76
- Desktop Application project 76, 192
 - contents of a 76
- Desktop Application template 76
- Destination property 707
- destructors 341, 571
- dialog icons
 - drawing 435
- dictionaries 234
- Dim statement 223
 - for arrays 229
 - for declaring an array 229
 - for objects 227
 - using constant in 230
- Disclosure Triangle control 165
- Do loop 253–254
- document icons 719
- Document window 91–92
- documentation
 - conventions 22
- dot syntax
 - defined 220
- Double
 - data type 215
- double-clicking
 - to open a file 529
- drag and drop 350–353
 - implementing 350
 - multiple items 350
 - PrivateRawData 356–357
 - RawData 356–357
 - RawData property 350
- dragging
 - from a ListBox 351
- dragging text
 - in TextFields 350
- DragItem object 351, 352
- DragRow event handler 351, 357
- DrawBlock method 450
- DrawCautionIcon method 436
- Drawer window 98
- DrawNotelcon method 436
- DrawPicture method 431
- DrawPolygon method 438
- DrawStopIcon method 436

DropObject event handler 352, 353, 355, 356, 357
dropping
 implementing 352–353
Duplicate menu command 64, 82
Dynamic checkbox 379
dynamic constants 383, 554

E

EasyTCPSocket class 167
EasyUDPSocket class 167
Edit Bookmarks dialog box 60
Edit Code command 648
Edit menu command 63, 82
Edit Mode buttons 274
Edit Tab Order button 180
Edit Window menu command 82
EditableMovie class 520
Editor Only command 294
Editor Only menu command 41, 68
Editor toolbar
 hiding the 41, 68
Editor Toolbar menu command 70
Else clause
 in If statement 261
Elself statement
 in If statement 262
Email
 sending and receiving 674
empty events
 showing and hiding 280–282
Enabled property 577
EnableMenuItems
 event handler 581
EnableMenuItems event handler 360, 361, 534, 577
enabling
 menu items 193
encoding 416–420, 671
 ASCII 416
 default 418, 419
 defined 416
 determining the 417
 MacJapanese 417
 MacRoman 417
 of a binary stream 523
 specifying the 418
 specifying when writing text files 513
 text 416–419, 671, 672
 Unicode 416, 417
 UTF-16 416
 UTF-8 416
Encoding function 417, 418
Encoding property 418, 511
encodings
 reading and writing files 511

Encodings object 418, 419, 420, 512, 523, 667, 671, 672
Encrypt dialog box 107, 308, 600
Encrypt menu command 64, 82
encrypted classes
 documenting 601
 importing 599
encrypting
 classes 599
end of file
 checking for 511
endless loop
 escaping from an 254
EndOfLine class 413, 713
Enter License Key menu command 73
entry filters 408
enum 397–399, 564–566
 casting 399, 566
 declaring an 399, 566
enumeration 564
EOF property 511, 523, 624
Equals keyword 250
Event Definition declaration 560
event definitions 392, 558–561
 reasons for adding 560
event handler 222, 280, 282, 283, 315, 320, 335, 344, 345, 349, 350, 353, 354, 355, 356, 357
 Activate 582
 Close 581
 Deactivate 582
 default 279
 definition of 272
 EnableMenuItems 581
 HandleAppleEvent 582
 object-oriented programming 278
 Open 581
 OpenDocument 581
 opening an 274
 passing parameters to 282
 selected 280
event handlers 206, 274, 278, 309–310, 320, 361
 default 274
 for controls 345
 Unhandled Exception event 582
event-driven programming 32, 272, 309
 definition of 271
events
 copying and pasting 304
 defined 32
 examples of 272, 309
 indirect 272
ExcelApplication class 691
Exception block 495, 653–655
 syntax of 653
Exists property 495

- Exit menu command 63
- Export Localizable Values command 63
- Export menu command 63, 83
- Export Source 307
- exported class
 - desktop icon of 598
- exporting
 - classes 542, 599
 - items 307
 - menubars 207
 - modules 402
 - source code 307
- Extends keyword 401, 568–569
- External IDE script 712
- external project item 80–81, 305, 402, 542, 599
 - locked 81
- external project items
 - Make External command 83
 - Make Internal command 83
 - saving to disk 84
 - showing 82
- Extract Interface dialog box 84, 593
- Extract Superclass dialog box 83
- Extract Superclass menu command 541

F

- Favorites controls 39, 114–115
- Favorites list 114
- Feedback 30
- field attributes
 - Mandatory 609
 - Unique 609
- field types 615–617
- fields
 - indexed 609–610
- Figure 244 on page 291 291
- Figure 45 on page 75 76
- FigureShape class 442
- file
 - accessing a 492
- File menu 61
 - New Window command 103
- File Path menu command 82
- file type
 - definition of 480
 - deleting 488
 - editing 488
- File Type Sets Editor 428
- file types
 - APPL 480
 - custom 488–490
 - overview of 480
 - PICT 480
 - specifying several 502
 - TEXT 480
 - using 488
- File Types Set Editor 480, 484, 720
- files
 - creating new 529–530
 - opened by dropping 529
 - opening from desktop 529
 - ownership 497
- FillPolygon method 438
- Find 65
 - recent 302
- Find Implementors menu item 84
- Find Item command 302
- Find Item contextual menu 303
- Find menu command 64
- Find scope 300
- Find/Replace window
 - in Browser 299–301
- Floating Properties Palette option 44, 45
- Floating windows 94–95
- flow of control 260
- focus
 - Canvas control 133
 - Checkbox control 135
 - ComboBox control 132
 - controls that receive 577
 - definition of 132
 - ListBox control 132
 - on Linux 132
 - on Macintosh 132
 - on Windows 132
 - PushButton control 134, 135
 - Slider control 136
 - TextField 132
- FolderItem
 - Child method 493
 - definition of 491
 - Delete method 496
 - deleting a 496
 - Exists property 495
 - getting for application's folder 500
 - getting info on a 496
 - Item method 500
 - locked 497
 - OpenAsSound method 520
 - ownership 497
 - Parent property 493
 - relative paths 500
 - uses of a 491
- FolderItem class 491, 492, 493, 496, 497, 500, 501, 503, 505, 509, 511, 512, 513, 514, 515, 516, 520, 524, 525, 529
- AppendToTextFile method 512
- CreateTextFile method 512
- OpenAsMovie method 520
- OpenAsPicture method 430, 433, 519

- OpenAsVectorPicture method 445
- OpenStyledEditField method 514
- SaveAsPicture method 445, 516
- SaveStyledEditField method 514
- FolderItems 491
 - aliases of 491
 - properties of 491
 - representing System folders 494
 - shortcuts to 491
- folders
 - bookmark 57
 - project 79
- font
 - SmallSystem 406
 - System 406
- font attributes
 - determining the 410–411
 - setting 411–412
- Font function 406
- font size 406, 409, 411, 412, 442
- font style 411
 - determining the 410
- font styles
 - toggling the 412
- FontAvailable function 411
- FontCount function 406
- fonts
 - determining available 406–407
 - setting attributes of 406–412
- For Each statement 258
- For loop 254–257
- ForeColor property 445, 446
- Format function 420
 - FormatSpec 421
- formatting
 - numbers, dates, and times 420–423
- formulas for properties in 127
- FORTTRAN
 - historical note on 256
- Forward button 40, 55, 70
- Forward menu command 70
- full keyboard access 137–139
 - enabling 137
 - selecting a control using 139
- full path
 - creating a 492
- function 283
 - declaring a 557
 - defined 244
 - returning an array 557
- Function statement 283
- functions
 - adding to a module 371
 - definition of 243–244, 330
 - returning arrays 331, 371

- specifying the return type 557

G

- Get method
 - computed property 321, 550
- GetCin method 526
- GetFolderItem
 - relative paths 500
- GetFolderItem function 492, 493, 496, 500, 501, 520
- GetIcl method 526
- GetNamedPicture method 526
- GetOpenFolderItem function 430, 433, 502, 503, 511, 512, 514, 520, 525
- GetParameters 597
- GetPicture method 526
- GetResource method 528
- GetSaveFolderItem function 508, 509, 515, 516, 518
- GetSaveFolderItem method 513
- GetSound method 526
- getter properties 549
- GetTrueFolderItem function 492
- GLDBit (Set Group bit)
 - permissions 499
- global
 - constant 129
 - properties, methods, and constants 370
- global bookmarks 71
- Global Bookmarks option 57, 60
- Global Floating window 94, 97
- global methods 372
- global properties 374
- Global scope
 - for localization constants 325
 - in Class Extension methods 569
- global variables 374
- Go to Location menu command 71
- Go to Search menu command 65
- graphical user interface 76
 - characteristics of 32
- Graphics class 431
 - Clip method 439
 - DrawCautionIcon method 436
 - DrawLine method 437
 - DrawNotelcon method 436
 - DrawOval method 437
 - DrawPicture method 431
 - DrawPolygon method 438
 - DrawRect method 437
 - DrawRoundRect method 437
 - DrawStopIcon method 436
 - FillOval method 437
 - FillPolygon method 438
 - FillRect method 437

- FillRoundRect method [437](#)
- ForeColor property [437](#), [445](#)
- PenHeight property [437](#)
- PenWidth property [437](#)
- Pixel property [436](#), [448](#)
- grid
 - drawing a [437](#)
- Gridlock class example [584](#)
- Group
 - FolderItem [497](#)
- GroupBox [155](#)
 - for organizing RadioButtons [155](#)
- GTK+ 2.0 [694](#)
- GUI
 - defined [76](#)
- H**
- HandleAppleEvent event handler [582](#), [687](#)
- help
 - context-sensitive [23](#), [290–291](#)
 - online [23](#)
- Help menu [73](#)
 - adding a [196](#)
- Hide Empty Events menu command [69](#), [281](#)
- Hide Properties menu command [43](#), [72](#)
- Highlight Parent Control preference [186](#)
- History menu [41](#), [70](#)
 - navigating via the [55](#)
- Home and End Keys [277](#)
- Home and End Keys options [277](#)
- Home menu command [70](#)
- HSV color model [216](#)
- HSV function [445](#), [446](#)
- HTMLViewer control [146](#)
- HTTP protocol [681](#)
- HTTPS protocol [675](#)
- I**
- icons [489](#)
 - custom document [719](#)
- IDE [33](#)
 - panel dividers [41](#)
- IDE options [294](#)
- IDE Script window [62](#)
- IDE scripts [62](#), [165](#)
- IDE Scripts menu command [62](#)
- IDE window [33](#), [35–57](#), [61](#)
 - Bookmarks menu [71](#)
 - Edit menu [63](#)
 - File menu [61](#)
 - Help menu [73](#)
 - History menu [70](#)
 - Location area [37](#), [40](#)
 - Main Toolbar [40](#), [55](#)
 - menus [61–73](#)
 - Project Editor [40](#)
 - Project menu [65](#)
 - tabs [37](#)
 - View menu [68](#)
 - Window Editor panel [36](#)
- If statement
 - one line [262](#)
- If...Then structure [260–266](#)
- ImageWell [162](#)
 - displaying a picture in [429](#)
- Implement Interface dialog [83](#), [390](#), [589](#)
 - #error option [391](#), [589](#)
- Implement Interface dialog box [390](#), [589](#)
- ImplicitInstance property [313](#)
- Import as External menu command [63](#), [80](#), [599](#)
- Import menu command [62](#)
- importing
 - classes [598–599](#)
 - external project items [599](#)
 - into a project [305–307](#)
 - menubars [207](#)
 - pictures [306](#)
 - project items [62](#)
- importing as external
 - project item [599](#)
 - project items [63](#)
- Include Function Names [707](#)
- incremental compilation [697](#)
- index
 - composite [610](#)
 - of an array [229](#)
- Index parameter [349](#)
- Index property [205](#)
 - used to differentiate multiple instances [348](#)
- inheritance [140](#)
- Insert Color menu command [287](#)
- Insert method [232](#)
- InsertRow [243](#)
- installation requirements
 - for Linux [21](#)
 - for Windows [21](#)
- instance
 - definition of [577](#)
- instance properties [552](#)
- instances
 - removing from memory [580](#)
- Int16 data type [214](#)
- Int32 data type [214](#)
- Int64 data type [214](#)
- Int8 data type [214](#)
- Integer
 - data type [214](#)
- integrated development environment (IDE) [33](#)
- interface controls [583](#)

- interface inheritance 594
- interfaces
 - character based 32
 - deleting 591
 - Implement Interfaces dialog 390, 589
- Internet 166
- interrupting execution 638
- Introspection module 597–598
- invisible controls
 - finding 119
- IP address 669
- IsA operator
 - in casting 576
 - in class interface 593
- Issue Type dialog box 67

J

- Jet database engine 728
- Join function 234

K

- keyboard accelerators 380
 - creating 198
- keyboard shortcuts 196
 - for menu items 50
- KeyDown event handler 537

L

- Language reference
 - online 23
- Language Reference menu command 73
- LastErrorCode property 673
- line continuation character 288
 - shortcut for 289
- Line control 160
- lines 160
 - drawing 437
- Lingua 63, 383
- Linux applications
 - requirements for 694, 719
- Linux builds 718
- Linux Settings group 706
- ListBox 132, 149–152, 241, 243
 - AddRow method 241
 - custom borders 150
 - custom shading 150
 - custom sorts 152
 - dragging a row 153
 - editable 152
 - getting the focus 132
 - hierarchical 149
 - implementing drag and drop 351
 - multicolumn 257

- resizing columns in 152
 - selection in 149
 - sorting a 151
- little endian byte order 263
- Local Bookmarks option 57, 60
- local scope 315
- local variables 639
 - declaration of 223
- Localization table 325
- localizing 379–384
- Location area 37, 40, 57, 71
- Lock Position command 125
- LockBottom property 122
- Locked property 497
- locking a control 125
- LockLeft property 122
- LockRight property 122
- LockTop property 122
- loops 252–257
 - endless 638
 - lengthy 254
 - nested 257
 - optimizing speed of 256

M

- Mac OS
 - built application name 706
- Mac OS classic
 - memory requirements for 706
- Mac OS X
 - Application menu 202
 - Color Picker 447
 - compiling for 716
 - number separators 420
- Macintosh
 - text services 293
- MacJapanese 417
- MacProcID 99
- MacRoman encoding 417, 418
- MacType string 451
- Main Toolbar 40, 55
 - Add Bookmark button 57
 - Back button 55
 - Build button 56
 - customizing the 58
 - Forward button 55
 - hiding the 41, 68
 - Location area 57
 - Run button 55
 - Search area 58
- Main Toolbar menu command 70
- Make External command 83
- Make External contextual menu item 542
- Make External menu command 80
- Make Internal command 83

- Mask property 409
- mask property 408
- mathematical operators 235
- mathematical precedence 235
- MDI application 705
- MDI window
 - maximizing 714
- MDIWindow class 705, 714
- Me function 441, 585
- MemberInfo 597
- memory errors
 - caused by excessive calls 649
- memory management 580
 - examples of 580
- MemoryBlocks 396, 564
- Menu Editor 49, 189–196
 - Convert To Menu command 205
 - defined 49
 - Properties pane 50
 - View Mode buttons 50
- Menu Editor toolbar 50
- menu handler 193, 282
 - adding a 359
- menu handlers
 - definition of 358
- menu item separators
 - adding 205
- menu items
 - adding 196–200
 - creating on the fly 205
 - dynamic 205, 362
 - enabling 193, 360
 - enabling or disabling 577
 - handling from controls 361
 - handling when a window is open 361
 - handling when no windows are open 361
 - implementing 205–207, 357–363
 - keyboard shortcuts for 50, 196
 - moving 204
 - removing 205
 - removing programmatically 207
- Menu Layout menu command 70
- menubar
 - default 190, 192
 - previewing 50
- menubars
 - adding to a project 192
 - assigning to a window 192
 - assigning to the application 192
 - importing 306
 - importing and exporting 207–208
- menus
 - adding 193–202
 - managing within classes 577
 - moving 205
- MessageBox class 109–111
 - icons in the 435
- MessageBoxButton 111
- Metal window 98
- method
 - adding to a module 327, 370–372
 - assigning a value to 573–575
 - creating a 335–336
 - definition of 241
 - deleting a 334
 - parameter line 282
- Method declaration area 332
- MethodInfo 597
- methods
 - adding to Bookmarks bar 60
 - adding to windows 329–334
 - associated with objects 310
 - built-in 241
 - class extension 401, 568
 - components of 282
 - constructors 341, 569–571
 - copying and pasting 304
 - default values for parameters 338–339
 - destructors 341, 571
 - finding in Code Editor 302
 - initialization of 341
 - optional parameter for 340
 - parameters passed to 241
 - passing values to 241
 - referring to in subclasses 538
 - Return Type 332
 - returning value from 330
 - scope of 331
 - setter 340–341
 - Stack drop-down list 638
 - tracking execution of 649
 - values returned from 243
- Microsoft Access 728
- Microsoft Office
 - automation 690
- Modal Dialog windows 93–94
- Modeless Dialog window 99
- modems
 - communicating with 668
- module
 - adding a 368
 - compared to class based on Application object 374
 - encrypted 402
 - exporting a 402
 - importing a protected 402
 - nested 399–400
- module namespace 367
- modules
 - decrypting 404

- encrypting 403, 403–404
- importing 306
- importing and exporting 402
- role of 367
- scope of items 370, 400
- shared among projects 80
- moths
 - role of in history of computing 632
- MouseCursor constructor 526
- MouseCursor property 527, 528
- MouseDown event 440
- MouseDown event handler 351, 352, 441, 561, 584
- MouseEnter event handler 527
- MouseExit event handler 527
- MouseMove event 437
- Movable Modal window 92–93
- Movie class 520
- movie controller
 - default appearance of 163
- MoviePlayer
 - assigning movie to 164
- MoviePlayer control 163–164, 520, 521
- Movie property 520
- movies
 - importing into projects 79
- MsgBox function 109, 435, 503, 505, 509
- multicasting 678
- Multiline property
 - in TextArea 409
- multiple connections
 - TCP/IP 670
 - with ServerSocket 675
- Multiple Document Interface (MDI) 705
- MySQLCommunityServer class 620
- MySQLEnterpriseServer class 620

N

- nested loops 257
- New IDE script 711
- New IDE Script menu command 62
- New Implementor menu item 84
- New operator 324, 552, 578, 677
 - creating custom class with 579
 - in Dim statement 579
 - to open a window 313
- New Project dialog box 76
- New Project menu command 61, 74, 87
- New Subclass menu command 83
- New Window menu command 61
- NewDocument event handler 529, 581
- NewPicture function 433, 518
- Next Tab menu command 72
- NextPage method 449
- Nil

- checking for 495
- Nil object 418, 419, 495, 496, 503, 505, 509, 511, 512, 514, 516, 518, 520, 524, 525
- NilObjectException error 495, 503
- Not operator 251
- Notes
 - in Code Editor 248
- numbers
 - formatting 420–421

O

- object hierarchy 117, 139
- object inheritance 140
- Object Viewer 649
 - opening in a new window 642
- Object2D class 442–443
- object-oriented
 - BASIC 20, 212
 - menus 357
 - methods 310
- object-oriented programming 20
 - advantages of 20
 - definition of 310
 - event handlers 206, 222, 272, 274, 280, 282, 283, 309–310, 315, 320, 335, 344, 345, 349, 350
 - interface inheritance 594
 - menu handlers 282, 358
 - Protected properties 315
- ODBCDatabase class 620
- Office automation 690
- OLEContainer control 176
- on run handler (AppleScript) 685
- on run statement 686
- one-based array
 - definition of 229
- online help 23
- Online Reference 23
 - searching 25
- Open Application AppleEvent 529
- Open event handler 352, 354, 355, 357, 536, 581
- open file dialog
 - limiting file types displayed in 502
- Open File dialog box 501
- Open File menu command 82
- Open menu command 61
- Open Recent menu command 61
- Open shared method 510
- OpenAsMovie method 520
- OpenAsPicture method 433, 516, 519
- OpenAsSound method 520
- OpenAsTextFile method 511
- OpenAsVectorPicture method 445
- OpenDialog class 502, 503
- OpenDocument

- event handler 581
- OpenDocument event handler 529
- OpenGLSurface 162
- opening windows 313–314
- OpenOracleDatabase Class 620
- OpenPrinter function 448, 449, 450
- OpenPrinterDialog function 448, 449, 450
 - passing SetupString to 450
- OpenResourceFork method 524, 525
- OpenStyledEditField method 514
- operator precedence 236
- Operator_ keywords 572
- operators
 - comparison 249
 - mathematical 235
- Optional keyword 340
- optional parameters
 - Optional keyword 340
- options
 - Autocomplete applies standard case 276
 - Autocomplete shows details 276
 - Code Editor 287
 - printing 304
 - Syntax Highlighting 275
 - Window Editor 186
- Options menu command 65
- Or operator 251
- Oracle 604
- order of mathematical operations 235
- OSType data type 219
- Others
 - FolderItem 497
- OutOfBoundsException error 654
- Oval control 161
- ovals
 - controlling "ovalness" of 161
 - drawing 437
- OvalShape class 442
- overloaded function
 - example 572
- overloading 251
 - definition of 572
- Owner
 - FolderItem 497

P

- Page Setup dialog box 448
- Page Setup menu command 63
- PagePanel 157
- PageSetupDialog method 448
- Paint event 431, 437, 441
 - Canvas control 431
- Paint event handler 356, 448, 518, 584, 585
- pane
 - definition of 41

- panel
 - definition of 41
- panel divider 41
- panes
 - resizing 41
- ParamArray keyword 330
- parameter
 - passing an array as 242
- parameter declaration 568
- parameter line
 - data typing 283
 - in method 282
- parameter passing 336
 - in parameter line 282
- ParameterInfo 597
- parameters
 - declaring data type of 329
 - default values for 338
 - definition of 241
 - more than one 242
 - optional 340
 - passing arrays as 330
 - passing by reference 336–337
 - passing by value 336
 - passing to methods 329
- parent class 533
- parent control 185
- Parent function 492
- Parent Highlight 186
- password field
 - creating a 408
- Paste menu command 63
- pasteboard 119, 120
 - changing the color of 187
 - Window Editor 187
- path
 - absolute 492
 - full 492
 - to application's folder 500
- Pause command 647
- Pause menu command 67
- permissions
 - as octal value 497
 - Execute 498
 - folderItem 497–500
 - Read 498
 - Sticky bit 499
 - Write 498
- Permissions class 498
 - constructor 500
- PICT file
 - saving a 516–518
- PICT resource 526, 528
- picture
 - copying a portion of 431

- displaying a 431
- displaying in a portion of a window 430
- displaying in a window 428–430
- scaling a 432
- pictures
 - creating 431
 - importing into projects 79
 - opening 518–519
- Pixel property
 - of a Graphics object 448
- pixels
 - drawing 436
- PixmapShape class 442
- Plain Box windows 95
- Play menu command 82
- plug-in controls 39, 113
- plug-ins 689
 - database 604
 - formats of 689
 - including in stand-alone apps 689
 - loading 689
 - used to create custom controls 689
 - using 689
 - writing 689
- polygons
 - drawing 438–439
 - filled 439
- POP3 protocol 681
- PopupArrow control 164
- PopupMenu control 153
 - adding items to a 153
 - changing the select item in 154
 - getting the focus 134
- port
 - definition of 669
- port numbers
 - on Mac OS X and Linux 669
- Port property 669, 670
- PowerPointApplication class 691
- precedence
 - operator 236
- preferences
 - Build Process 698
 - debugger 661
 - Highlight Parent preference 186
 - remote debugger sessions 661
 - Window Editor 186, 715
- Preferences menu command 65
- PrefsMenuItem class 202
- Previous Tab menu command 72
- primary key 609
- primary key index 610
- Print dialog box
 - displaying the 449
- Print menu command 63
- PrinterSetup class 448
 - SetupString property 448
- PrinterSetup class objects 449
- PrinterSetup settings 448
- printing 448–450
 - in Code Editor 304
 - options 304
 - overview of process 448
 - sending a page to the printer 449
 - without the Print dialog box 450
- Private scope 316, 325, 331, 533, 544
- PrivateRawData property 357
- Profile Code menu command 67
- Profiler 655–657
- ProgressBar 148
 - Barber Poles 148
 - indeterminate 148
- ProgressWheel control 165
- project
 - adding and removing items from a 79
 - creating a new 74
 - defined 40
 - removing items from 81
 - saving a 62, 84
 - starting and stopping 651
- Project controls 39, 113
 - adding to a window 578
- Project Editor 40–43
 - adding classes to 533
 - adding data source to 606
 - contextual menu 81–84, 104, 107, 540, 541, 606
 - creating a subclass in 540
 - dragging items to 79
 - importing files into 305
 - items included in 40
 - removing data source from 606
- Project Editor toolbar
 - configuring the 77
- project item
 - removing a 81
- project items
 - decrypting 82
 - encrypting 82
 - organizing 79
- Project Menu
 - Step Out command 649
- Project menu 65
 - Break on Exceptions command 652
 - debugging with the 648
 - Step command 648
 - Step In command 648
- project templates 76
 - creating 87
- projects
 - items in 73–74

- properties 639
 - adding to a class 544–545
 - adding to a module 373–374
 - adding to a window 314–320
 - adding to Bookmarks bar 60
 - assigning values to 220
 - computed 321, 549
 - copying and pasting 304
 - data type mismatches 225
 - defined 42
 - definition of 213
 - formulas for 127
 - getter and setter 549
 - getting value from 222
 - Global 374
 - global 370, 374
 - Private 374
 - protected 315
 - Public 374
 - referring to in subclasses 538
 - setting via popup menu 101
 - setting via text entry area 102
 - shared 552
 - show in Properties pane 315
 - Properties list
 - customizing 545
 - Properties pane 40, 42, 127
 - defined 42
 - displaying in a floating window 45
 - entering Boolean properties 127
 - entering constants 128
 - entering numeric expressions 127
 - floating 44
 - for Menu Editor 50
 - hiding 43
 - Menu Editor 195
 - minimizing the 68
 - setting a color property 130
 - setting Interfaces property 592
 - showing 44
 - using choice lists 130
 - using the 100, 126
 - property
 - definition of 226
 - documenting a 247, 319, 374, 545
 - scope of 315
 - Property List Behavior dialog 64, 83, 546
 - PropertyInfo 597
 - Protected methods 372
 - protected properties 374
 - Protected scope 315, 324, 331, 370, 400, 543
 - protecting
 - exported code 599
 - protocol
 - defining your own 681
 - definition of 681
 - PString data types 218
 - Ptr data type 218
 - Public methods 372
 - Public properties 374
 - Public scope 315, 324, 331, 343, 370, 400, 543
 - PushButton control 140, 272
 - getting the focus 135
- ## Q
- QuickTime 163
 - QuickTime file
 - opening a 520
 - QuickTime movies 306, 520–521
 - Quit menu command 63
 - QuitMenuItem class 190, 203
- ## R
- RadioButton control 143
 - RadioButtons
 - compared to Checkbox 143
 - RaiseEvent statement 561
 - RawData property 357
 - RawDataAvailable method 451
 - RbScript control 165
 - RBVersion constant 263
 - Read method 667
 - ReadAll method 510, 511, 511, 667
 - ReadLine method 510
 - REAL Software
 - contacting 29
 - REAL SQL Database
 - adding tables 607
 - primary key 605
 - REAL Studio
 - 2006r3 Encryption 403
 - adding AppleScripts to 685
 - advantages of compiled 212
 - books and magazines 28
 - built application name 706
 - checking for updates 28
 - Code Editor 46
 - controls 113–184
 - Controls list 38
 - converting Visual Basic apps to 723–728
 - coordinates system 427
 - Debugger 67, 636
 - debugging in 33
 - development process in 32
 - differences from BASIC 212
 - document icons 719
 - electronic documentation 27
 - entering license key 73
 - Feedback 73

- Feedback menu command 73
- hardware requirements 22
- IDE 33–50
- IDE window 35–57
- Info menu command 73
- installation requirements 21
- installing 21
- integrated development environment 33
- Interface Assistant 89, 208
- Introspection 597–598
- localizing an app 379–384
- mailing lists 28
- memory management 580
- Menu Editor 49
- menus 61–73
- object hierarchy 117
- operating system requirements 21
- Options 275, 287, 294, 304
- overview of 20
- plug-ins 689
- preferences 186, 698, 715
- project templates 87
- projects 40, 73
- reporting bugs 30
- reserved words in 240
- saving as XML 85
- scripting 62
- SDK 689
- syntax error messages 33
- technical support 28, 29
- third-party web sites 28
- using code examples 26
- using database with 728
- Version Control Project format 85
- VM Migration Assistant 724
- web page 28
- Window Editor 36
- working with a VCS 85
- REAL Studio 2006r3 Encryption 403
- REAL Studio Feedback 30
- REALbasic
 - magazine about 28
- REALbasic Developer 28
- REALSQLServerDatabase class 620
- REALStudio
 - development cycle 32
 - IDE window 61
- RecordSet class 623
- Rectangle control 160
- rectangles
 - drawing 437
- RectShape class 442
- Redim statement 233
- Redo menu command 63
- reference
 - definition of 578
 - reference counting
 - definition of 580
 - Refresh button 299
 - RegEx class 424
 - regular expressions 424–427
 - Remote Debugger Stub
 - configuring 658
 - remote debugging 67, 657–663
 - connections 661
 - remote machines
 - configuring 661
 - Remove method 233
 - RemoveResource method 528
 - Report Editor
 - Body area 457
 - data source 460
 - Footer section 462
 - GroupByField 461
 - Grouping section 460
 - Page Footer area 458
 - Page Header area 457
 - previewing to a Canvas 475
 - printing 468
 - SummaryType field 462
 - using a text file with 470
 - with interface 470
 - with SQL SELECTstatement 465
 - Report Layout Editor 456
 - reserved words 240
 - resize bar
 - in Code Editor 295
 - resource file
 - adding to a project 525
 - resource fork 524
 - adding a 525
 - adding to a project 525
 - contents of 524
 - opening a 524–525
 - resource forks
 - definition of 524
 - resource types
 - supported 526
 - ResourceFork class 525, 526, 528
 - resources
 - importing 306
 - reading 526–528
 - writing to 528
 - Resume command 647
 - Return character
 - used in writing to text files 512
 - Return type 557
 - reusable code 532
 - Revert to Saved menu command 62
 - RGB color

- specifying via the & operator 216
- RGB color model 216
- RGB function 441, 445, 446, 448
- RGB values
 - getting the 446
- root namespace 367
- Rounded windows 96
- RoundRectangle 161
- RoundRectShape class 442
- RTFData property 515
- Run button 55, 67
- Run menu command 67
- Run panel 636
- Run Paused menu command 67, 636
- Run Remotely menu command 67
- runtime exceptions 653–655

S

- Save As dialog box
 - managing the 507–510
- Save As menu command 62, 84, 85
- Save menu command 62, 84
- SaveAsDialog class 508, 509
- SaveAsJPEG method 516
- SaveAsPicture method 445, 516, 518
- SaveStyledEditField method 514
- schema
 - database 607
- Scope 331
 - for a class 543–544
 - for a module 370–372
 - Global 226, 370
 - Local 225, 237, 315, 375
 - of a constant 377, 554
 - of a control 343
 - of a method 331, 332, 557
 - of a module's items 370, 400
 - of a property 318, 374, 545
 - of class extension method 569
 - of constants 237, 375
 - of window constants 324
 - Private 226, 316, 325, 331, 533, 544
 - Protected 226, 315, 324, 331, 370, 400, 543, 582
 - Public 226, 315, 324, 331, 343, 370, 400, 543, 582
- scripting
 - REAL Studio 62
- ScrollBar control 147
- search
 - via Spotlight 298
- Search area 40, 58
- Search Results 298–299
 - Refresh button 299
- searches
 - favorite 302
 - recent 302
- secure email 675
- secure TCP connections 676
- Select All menu command 63
- Select Case statement 264–266, 349
- selected file
 - getting folderitem for 502
- selected folder
 - getting folderitem for 504
- selected text
 - determining attributes of 410
 - working with 407–408
- SelectFolder function 504, 505
- SelectFolderDialog class 504, 506
- selecting
 - controls 119
- Self function 320, 344, 528
- SelfTextSize property 411
- Separator control 155
- separators
 - in menus 205
- Serial control 166, 666
 - changing configuration of 668
 - Close method 668
 - configuring 666
 - DataAvailable event handler 667
 - Flush method 667
 - LookAhead method 667
 - Open method 667
 - overview of use 666
 - placing in a window 666
 - Poll method 668
 - reading data with 666
 - Write method 667
 - writing data 667
 - XmitWait method 667
- serial device
 - definition of 666
- serial devices
 - communicating with 666–668
- serial port
 - closing the 668
 - opening the 666
- ServerSocket class 166, 669, 673, 675
- service application
 - creating a 76
- Set Breakpoint menu command 66
- Set method
 - computed property 321, 550
- setter methods 340, 573–575
- setter properties 549
- SetupString property
 - storing the 449
- Shadowed Box window 95
- shared libraries 306

- shared methods 557
- shared properties 552
- shortcuts
 - of folderitems 491
- Show all Bookmarks menu command 71
- Show Built Applications on Disk preference 698
- Show Code menu command 69
- Show Empty Events menu command 69, 281
- Show Empty Events menu item 281
- Show Layout menu command 69
- Show Object IDs in Variable Lists option 651
- Show on Disk menu command 82
- Show Properties menu command 44, 72
- Single data type 215
- Slider
 - tick marks 148
- Slider control 147–148
 - getting the focus 136
- SmallSystem font 304, 406
- SMTP protocol 681
- SMTP server 669
- snd resource 526
 - getting sounds from 520
 - reading 526
- socket
 - orphaning a 673
- Socket control 670
 - Close method 674
 - Connect method 670
 - Connected event handler 670
 - DataAvailable event handler 670
 - Error event handler 672
 - Listen method 670
 - Write method 671
- SocketCore class 669
- Sound class 520
- sound file
 - opening a 519
- sound files 519–520
- sounds
 - importing 306
 - importing into projects 79
- source code
 - encrypting 307
 - exporting 307
- Space Horizontally command 184
- Space Vertically command 184
- Split function 233
- Spotlight 174, 298
- SQL 604
- SQL queries 173
- SQLSelect method 623
- SSL communication 166
- SSL protocols 166, 677
- SSLSocket class 166, 675, 676
- stack
 - in debugger 638
- Stack drop-down list 638
 - Debugger 649
- Stack window
 - viewing code from 649
- stand-alone application
 - naming the 706
- standalone application
 - creating a 56
 - naming the 705, 706
- standalone applications
 - building 695–702
- Standardize Format command 289
- Startup screen
 - Options dialog 61
- StaticText control 143
- Step command 647, 648
- Step In command 648
- Step In option 648
- Step menu command 67
- Step option 647
- Step Out command 648, 649
- Step Out option 648, 649
- Step statement
 - for incrementing a counter 256
- Sticky bit 499
- Stop 56
- Stop command 647
- Str function 243
- String
 - data type 213
- StringShape class 442
- structs 219
- structure alignment 397, 564
- structure fields
 - declaring 394, 562
- Structured Query Language 604
- Structures 219
- structures 219, 394–396, 561–563
 - alternative to MemoryBlocks 396, 564
 - using 396, 563
- Styled property
 - in TextArea 514
- styled text
 - definition of 409
 - handling 409–412
 - reading into a TextArea 514
 - saving as RTF 515
 - writing to a file 514
 - writing to disk 515
- styled text files 514–515
- StyledText
 - RTFData property 515
- StyledText class 412–415, 515

- AppendStyleRun method [413](#)
- InsertStyleRun method [413](#)
- Paragraph method [413](#)
- ParagraphAlignment method [414](#)
- ParagraphCount method [414](#)
- RemoveStyleRun method [413](#)
- StyleRun method [413](#)
- StyleRunCount method [413](#)
- StyleRunRange method [413](#)
- Text method [413](#)
- StyledTextPrinter class [410, 450](#)
 - DrawBlock method [450](#)
- StyleRun class [413](#)
- Sub statement [282](#)
- subclass
 - based on a control [578](#)
 - creating a [540](#)
 - creating via contextual menu [83](#)
 - customizing a [533](#)
 - definition of [533](#)
- subclasses
 - ease of debugging [537](#)
 - examples of [533, 533–538](#)
- submenu
 - adding a [200–202](#)
- Super Class [536](#)
 - defined [140, 533](#)
- Super keyword [570](#)
- superclass
 - extracting from a class [541](#)
- Switch To contextual menu item [281](#)
- syntax errors [33](#)
- Syntax Highlighting [275](#)
- System font [304, 406](#)

T

- tab
 - dragging a [54](#)
- Tab Order [69](#)
 - changing the [179](#)
- Tab order [118](#)
 - changing the [179](#)
- tab order
 - changing the [179](#)
- tab panel
 - definition of [41](#)
- Tab Panel Editor [157](#)
- tabbed editing
 - defined [51](#)
- Tabbed Editing option [54](#)
- table
 - creating a [607–608](#)
- Table 9 on page 304 [306](#)
- TabPanel [156–157](#)
- TabPanels
 - advantages of [156](#)
 - tabs
 - reordering [51](#)
 - Tabs bar [34, 36, 37, 40, 41](#)
 - contextual menu [53](#)
 - dragging an item to [79](#)
 - hiding the [53](#)
 - Target property [344](#)
 - TargetBigEndian constant [263](#)
 - TargetCarbon constant [263, 714](#)
 - TargetHasGUI constant [263, 714](#)
 - TargetLinux constant [263, 714](#)
 - TargetLittleEndian constant [263](#)
 - TargetMachO constant [263, 714](#)
 - TargetMacOS constant [263, 714](#)
 - TargetMacOSClassic constant [263](#)
 - TargetPowerPC constant [263](#)
 - TargetWin32 constant [263, 714](#)
 - TargetX86 constant [263, 714](#)
 - TCP/IP [166, 669](#)
 - multiple connections [675](#)
 - supporting multiple connections [670](#)
 - TCP/IP communications [669–681](#)
 - TCP/IP connection
 - closing [674](#)
 - error handling [672](#)
 - listening for a [670](#)
 - reading data [670](#)
 - to another computer [670–674](#)
 - writing data [671](#)
 - TCP/IP protocols [681](#)
 - TCP/Socket control [166, 669](#)
 - for communicating via the Internet [166, 167](#)
 - Port property [669](#)
 - technical support [29](#)
 - templates [76](#)
 - text
 - getting and selecting [407](#)

- text encoding [416–420](#)
 - ASCII [416](#)
 - default [418](#)
 - defined [416](#)
 - determining the [417](#)
 - MacJapanese [417](#)
 - MacRoman [417](#)
 - specifying the [418](#)
 - specifying when writing text files [513](#)
 - Unicode [416, 417](#)
- text encodings [416–419](#)
 - in Serial communications [667](#)
 - in TCP/IP communications [671](#)
 - reading and writing files [511](#)
 - writing [671](#)
- text file
 - creating a [509](#)

- reading a [418, 510](#)
 - writing to [419, 512–513](#)
 - text files
 - compared to binary files [514](#)
 - limitations of [514](#)
 - specifying a text encoding [511](#)
 - working with [510–515](#)
 - text services [293](#)
 - TextArea [132](#)
 - MultiLine property [409](#)
 - Styled property [514](#)
 - TextArea control [145](#)
 - TextEncoding class
 - Chr method [419](#)
 - TextField [132, 144](#)
 - AcceptTabs property [132](#)
 - CueText property [144](#)
 - cut, copy, and paste in [451](#)
 - determining the font in [410](#)
 - determining the font style [410](#)
 - dragging text in [351](#)
 - filtering entries [408](#)
 - Format property [408](#)
 - formatting text in [408](#)
 - getting the focus [132](#)
 - implementing drag and drop [350](#)
 - LimitText property [408](#)
 - Mask property [409](#)
 - mask property [408](#)
 - masking entries [409](#)
 - MultiLine property [414](#)
 - Password property [408](#)
 - SelBold property [411](#)
 - SelChange event handler [408](#)
 - SelItalic property [411](#)
 - SelLength property [407](#)
 - SelStart property [407](#)
 - SelText property [407](#)
 - SelTextFont property [410](#)
 - SelTextSize property [410, 411](#)
 - SelUnderline property [411](#)
 - setting font attributes in [411–412](#)
 - setting the font style [411](#)
 - Styled property [409, 414](#)
 - subclass of [533](#)
 - ToggleSelectionBold method [412](#)
 - ToggleSelectionItalic method [412](#)
 - ToggleSelectionUnderline method [412](#)
 - tooggling font styles [412](#)
 - TextInputStream
 - definition of [510](#)
 - TextInputStream class [418, 511, 512](#)
 - Encoding property [418, 511](#)
 - Open shared method [510](#)
 - TextOutputStream
 - definition of [512](#)
 - TextOutputStream class [419, 512, 513](#)
 - WriteLine method [512](#)
 - Thread class [254](#)
 - Timer object [173](#)
 - times
 - formatting [423](#)
 - TLS protocol [166](#)
 - Toolbar
 - Back button [70](#)
 - Build button [68](#)
 - Forward button [70](#)
 - Location area [40, 71](#)
 - Main [40](#)
 - Run button [67](#)
 - Search area [58](#)
 - Toolbar control [168–173](#)
 - Toolbar Editor [168](#)
 - ToolButton [168](#)
 - ToolItem [168](#)
 - Type Filter dialog box [636](#)
 - type selection [133](#)
 - TypeInfo [597](#)
 - TypeMismatchException runtime error [576](#)
- ## U
- Ubound function [231](#)
 - UDP socket modes [678](#)
 - UDPSocket
 - instantiating a [677](#)
 - UDPSocket class [167, 672, 677–679](#)
 - UDPSockets
 - broadcasting [678](#)
 - IP addresses for [678](#)
 - multicasting [678](#)
 - unicasting [678](#)
 - UDT [394, 561](#)
 - UIDbit (Set User bit)
 - permissions [499](#)
 - UInt16 data type [214](#)
 - UInt32 data type [214](#)
 - UInt64 data type [214](#)
 - UInt8 data type [214](#)
 - underscore character
 - as line continuation character [288](#)
 - Undo menu command [63](#)
 - UnhandledException
 - event handler [582, 655](#)
 - unicasting [678](#)
 - Unicode encoding [416, 417](#)
 - UpDownArrows control [165](#)
 - user interface
 - importance of [89](#)
 - UserCancelled function [436](#)
 - user-defined types [394, 561](#)

UTF-16 encoding 416
 UTF-8 encoding 416, 418, 419

V

vacuum tubes
 use of in computers 632
 values
 changing the data type of 219
 debugging 649–650
 getting and setting in variables 223
 getting from properties 222
 variable
 definition of 213
 variables
 declaration of 223
 defined 223
 definition of 213
 getting and setting values 223
 global 370, 374
 local 225
 Variables pane 639
 Debugger 649
 Variant data type 216
 VB Migration Assistant 724
 VBA 690
 VBA Help 690
 VCS 85
 vector graphics 442–445
 saving 445
 vector object
 definition of 442
 displaying a 443–445
 drawing a 443
 opening 445
 Version Control Project format 85
 version control system 85
 View menu 68, 281
 Maximize Editor menu command 41
 View Mode buttons 50, 296
 virtual method 568
 Visual Basic
 converting to REAL Studio 723–728
 Visual Basic for Applications 690
 Volume function 492

W

Warnings menu item 67
 While loop 253
 While...Wend loop 511
 window
 accessing properties of a 320
 adding a property to a 316, 332
 adding controls to a 116
 default 104

 deleting a method 334
 deleting a property 320
 editing a method 334
 editing a property 319
 managing multiple instances of a 344
 Paint event 431
 types of 90
 Window class
 MacProclD property 99
 Paint event 437
 Window editing area 37
 Window Editor 36
 alignment icons 183
 defined 36
 displaying the 36
 Edit Mode buttons 274
 preferences 186, 715
 View Mode buttons 296
 Window Editor toolbar 180
 customizing the 105–107
 Window menu 72
 WindowPtr data type 218
 windows
 accessing controls, methods, and properties of
 other 342
 adding methods to 329–334
 adding properties to 314–320
 creating 103
 creating new 102
 default type 92
 deleting 103
 events 311–313
 Frame property 90, 101
 Implicit Instance property 313
 importing 306
 multiple instances of 344
 opening 313–314
 opening with New operator 313
 removing 103
 shared among projects 80
 Title property 101
 windoids 94
 Windows Media Player 520
 WordApplication class 691
 WriteLine method 512
 WString data type 218

X

XML
 opening 85
 saving as 85
 XML format 61

Z

zero-based array
 definition of [229](#)