XOJO

User Guide

Book 3
Framework

2015 Release 4

Xojo, Inc.

Preface

Xojo Guide

Book 3: Framework

Version 2015 Release 1

Section 1

About the Xojo User Guide

This *Xojo User Guide* is intended to describe Xojo for both developers new to Xojo and those with significant experience with it.

The *User Guide* is divided into several “books” that each focus on a specific area of Xojo: Fundamentals, User Interface, Framework and Development.

The *User Guide* is organized such that it introduces topics in the order they are generally used.

The Fundamentals book starts with the Xojo Integrated Development Environment (IDE) and then moves on to the Xojo Programming Language, Modules and Classes. It closes with the chapter on Application Structure.

The User Interface book covers the Controls and Classes used to create Desktop and Web applications.

The Framework book builds on what you learned in the User Interface and Fundamentals books. It covers the major framework areas in Xojo, including: Files, Text, Graphics and Multimedia, Databases, Printing and Reports, Communication and Networking, Concurrency and Debugging. It finishes with two chapters on Building Your Applications and then a chapter on Advanced Framework features.

The Development book covers these areas: Deploying Your Applications, Cross Platform Development, Web Development, Migrating from Other Tools, Code Management and Sample Applications.

Copyright

Trademarks

Conventions

The Guide uses screen snapshots taken from the Windows, OS X and Linux versions of Xojo. The interface design and feature set are identical on all platforms, so the differences between platforms are cosmetic and have to do with the differences between the Windows, OS X, and Linux graphical user interfaces.

- **Bold type** is used to emphasize the first time a new term is used and to highlight important concepts. In addition, titles of books, such as *Xojo User Guide*, are italicized.
- When you are instructed to choose an item from one of the menus, you will see something like “choose File ↠ New Project”. This is equivalent to “choose New Project from the File menu.”
- Keyboard shortcuts consist of a sequence of keys that should be pressed in the order they are listed. On Windows and Linux, the Ctrl key is the modifier; on OS X, the ⌘ (Command) key is the modifier. For example, when you see the shortcut “Ctrl+O” or “⌘-O”, it means to hold down the Control key on a Windows or Linux computer and then press the “O” key or hold down the ⌘ key on OS X and then press the “O” key. You release the modifier key only after you press the shortcut key.
- Something that you are supposed to type is quoted, such as “GoButton”.
- Some steps ask you to enter lines of code into the Code Editor. They appear in a shaded box:

```
ShowURL(SelectedURL.Text)
```

When you enter code, please observe these guidelines:

- Type each printed line on a separate line in the Code Editor. Don’t try to fit two or more printed lines into the same line or split a long line into two or more lines.
- Don’t add extra spaces where no spaces are indicated in the printed code.
- Of course, you can copy and paste the code as well.

Whenever you run your application, Xojo first checks your code for spelling and syntax errors. If this checking turns up an error, an error pane appears at the bottom of the main window for you to review.

Table of Contents

Chapter 1

Files

Most applications work with files of some kind. This chapter shows you how to access files in general and process text and binary files. You'll also learn about XML and JSON data formats, File Type Sets and how to use Virtual Volumes.

CONTENTS

Accessing Files

A FolderItem is anything that can be stored on a drive such as volumes, folders, files, applications, and documents.

Using the FolderItem class, you can get a reference to any such items on your drives. To read from a file, you need a FolderItem for it. To write to a file, you need a FolderItem. When you ask users to select a file using one of the file selectors, you get a FolderItem referring to the file they selected.

Once you have a FolderItem, you can refer to its properties (such as Name or path) and perform actions on it such as deleting or copying it.

Folder Item

Class: FolderItem

Properties

NativePath, ShellPath, URLPath
Returns the path of the FolderItem. NativePath uses the operating system conventions, ShellPath is the POSIX path, URLPath is a file:/// path.

Alias
Indicates if the FolderItem is really an alias or shortcut.

Count
If the FolderItem is a folder, then this returns the number of FolderItems in it.

CreationDate, ModificationDate
The creation and modification date of the file or folder.

Directory
Indicates if the FolderItem is a folder.

DisplayName, Name
The DisplayName is the name usually seen by the user (and may omit an extension for example). The Name is the actual name of the file or folder.

Exists
When True, the FolderItem exists on the drive.

ExtensionVisible
Allows you to check the OS setting to see if extensions are visible.

Group, Owner, Permissions
Used by OS X and Linux to change the file attributes.

LastErrorCode
Returns a non-zero error code if an error occurred with a FolderItem operation.

Length
The size of the file.

Locked
A file that is locked cannot be modified.

VirtualVolume
If the FolderItem is a VirtualVolume, this returns the VirtualVolume used to access it. Refer to the section on Virtual Volumes for more information.

Visible
Indicates if the file is visible or hidden on the drive.

Methods

Child
If the FolderItem is a folder, then this returns a FolderItem in it with the specified name.

CopyFileTo
Copies the FolderItem to the destination (specified as a FolderItem).

CreateAsFolder
Creates a folder using the location and name of the FolderItem.

CreateVirtualVolume
Creates a Virtual Volume file. Refer to the section on Virtual Volumes for more information.

Delete
Deletes the FolderItem immediately. The Trash can or Recycle Bin is not used.

GetRelative, GetSaveInfo
GetRelative returns a FolderItem based on the GetSaveInfo supplied to it.
Use GetSaveInfo to create a string that can be used to recreate a FolderItem without relying on an AbsolutePath. This method is able to find a file if it moves, something that cannot be done with an AbsolutePath.

Item
If the FolderItem is a directory, the method allows you to iterate through its contents (1-based).

Launch
If the FolderItem is an application, this method starts the application. If the FolderItem is a document, the document is opened using its default application.

TrueChild, TrueItem
Used to resolve FolderItems that point to aliases.

Shortcuts and Aliases

Shortcuts (aliases on OS X) are files that actually represent a volume, application, folder, or file stored in another location and possibly under another name. The FolderItem class contains properties and methods 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 a shortcut when it encounters it, while the GetTrueFolderItem function works with the shortcut itself.

Accessing a File from 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 your application. The relative path starts with the folder your application is in. The GetFolderItem global function can be used to quickly get a reference to a file as seen in the following code:

```
Dim f As FolderItem
f = GetFolderItem("Schedule")
```

To get the folder where your application resides:

```
Dim f As FolderItem
f = GetFolderItem("")
```

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) // 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%20Sett
ings/" _
   + "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 System Folders

Operating systems have specific locations for various folders that contain information, such as the Documents folder for the user.

Use the SpecialFolder module to get FolderItems representing these special system folders. The benefit of using this module rather than attempting to recreate the path manually, is that SpecialFolder always works and is correct across platforms (in most cases) and languages.

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 OS X and the Application Data directory on Windows:

```
Dim f As FolderItem
f = SpecialFolder.ApplicationData
```

Refer to SpecialFolder in the Language Reference for the complete list of supported functions and the FolderItems that are available for OS X, Windows and Linux.

***Note**: 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 OS X 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 the FolderItem

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 crash.

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("File does not exist!")
  End If
Else
  MsgBox("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 Exception Handling section in the Debugging chapter.

Creating New FolderItems

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 your 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, New 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
```

CREATING FILES ON WEB SERVERS

If your web app creates files (or folders) on web servers, including Xojo Cloud, you have to ensure you set the appropriate permissions to that you can later write to (or delete) them.

To do so, use the Permissions property of the FolderItem class. By default, files created on the web server get the permissions of the parent folder. This setting gives a file read/write access for all users:

```
Dim f As FolderItem("myFile.txt")
f.Permissions = &c666
```

Refer to the FolderItem.Permissions property in the Language Reference for specifics on how you can set permissions.

Deleting FolderItems

Once you have a FolderItem that represents an item that can be deleted, you can call the Delete method. The following example deletes the file represented by the FolderItem returned:

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.

```
Dim f As FolderItem
f = SpecialFolder.Documents.Child("Schedule")
If f <> Nil Then
  If f.Exists Then
    f.Delete
  End If
End If
```

Finding Your Application's Default Folder

Passing an empty string (two quotes with no characters in between them) to the GetFolderItem function returns a FolderItem representing the folder your application 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
f = GetFolderItem("")
For i As Integer = 1 To f.Count
  ListBox1.AddRow(Str(i))
  ListBox1.Cell(ListBox1.LastIndex, 1) =
f.Item(i).AbsolutePath
Next
```

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")
```

Iterating Through FolderItem Contents

You may find that you need to iterate through a folder item (that is a folder or directory) in order to process all the files in it. Here is an example that deletes all the files in a folder:

```
Dim itemsToRemove() As FolderItem
Dim n As Integer = dir.Count

If n > 0 Then
  For i As Integer = 1 To n
    If dir.Item(i).Exists And Not
dir.Item(i).Directory Then
      itemsToRemove.Append(dir.Item(i))
    End If
  Next
End If

For i As Integer = 0 To Ubound(itemsToRemove)
   itemsToRemove(i).Delete
Next
```

This code saves the files that are to be deleted in an array so that they can be deleted after all the files have been processed in order to improve performance.

Prompting the User for Files and Folders

There are several methods available to allow the user to select files while using your applications. You may want to allow your user to specify a file to open, to specify the name of a file to save or you may want to let them choose a folder.

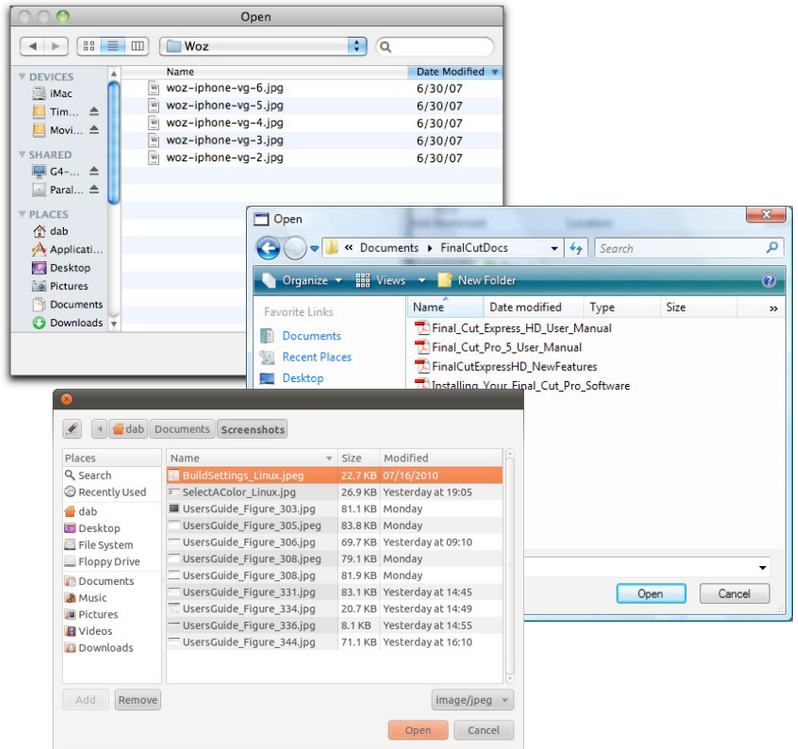
Opening Files

The simplest method for prompting the user to select a file to open is to use the GetOpenFolderItem global function as follows:

```
Dim f As FolderItem
f = GetOpenFolderItem(FileTypes1.jpeg)
MsgBox(f.ModificationDate.ShortDate)
```

The GetOpenFolderItem function displays the Open File selector 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

Figure 1.1 Open File Selectors on OS X, Windows and Linux



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.

If the user clicks the Cancel button rather than the Open button in the Open File selector, GetOpenFolderItem returns Nil. You will need to make sure the value returned is not Nil before using it. If you don't, your app will crash with a NilObjectException. 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
```

For more precise control, you can use the OpenDialog class to create an Open File selector. The class 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)

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 New OpenDialog
dlg.InitialDirectory =
SpecialFolder.Documents

dlg.Title = "Select a MIF file"
dlg.Filter = FileTypes1.pdf

Dim f As FolderItem
f = dlg.ShowModal
If f <> Nil Then
  // Proceed normally
Else
  // User Cancelled
End If
```

Saving a File

The Save As selector 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 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 appears in the Name field in the Save As selector). 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 (which can happen if the user clicked Cancel).

This example displays a Save As selector with a default filename of "Untitled":

```
Dim f As FolderItem
f =
GetSaveFolderItem(FileTypes1.jpeg,"Untitled")
If f <> Nil and f.Exists Then
  MsgBox(f.Name)
End If
```

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 New OpenDialog
dlg.InitialDirectory =
SpecialFolder.Documents
dlg.Title = "Select a pdf file"
dlg.Filter = FileTypes1.Pdf

Dim f As FolderItem
f = dlg.ShowModal
If f <> Nil Then
  // Proceed normally
Else
  // User Cancelled
End If
```

Selecting a Folder

Sometimes you need to have the user select a Folder rather than a file using the Folder selector. You can do this using the SelectFolder global method:

```
Dim f as FolderItem
f = SelectFolder
If f <> Nil Then
  MsgBox(f.Name)
End If
```

If you need more control over this selector, you can use the SelectFolderDialog class instead. The class has properties to modify the:

- Action button caption
- Cancel button caption
- Initial Folder
- Position on screen
- Prompt text
- Title
- Default folder name

This example displays a customized Folder selector:

```
Dim dlg As New SelectFolderDialog
dlg.ActionButtonCaption = "Select"
dlg.Title = "Title Property"
dlg.PromptText = "Prompt Text"
dlg.InitialDirectory =
SpecialFolder.Documents

Dim f As FolderItem
f = dlg.ShowModal
If f <> Nil Then
  // Use the FolderItem here
Else
  // User cancelled
End If
```

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 by your applications. Text files are convenient since they can also 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.

Text Streams

Text Streams are used to read or write data to text files. There are two text stream classes: TextInputStream and TextOutputStream. As you might expect, use TextInputStream to read data from text files. Use TextOutputStream to write data to text files.

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 TextInputStream.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
```

Dealing with Encodings

If you are reading and writing text files with only your 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 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 to Windows.ANSI. The string containing

the text will use the specified encoding until it is specifically changed to something else. The code assumes that there is a File Type Set in the application named TextTypes:

```
Dim text As String
Dim f As FolderItem
Dim stream As TextInputStream
f = GetOpenFolderItem(TextTypes.All)
If f <> Nil Then
  stream = TextInputStream.Open(f)
  stream.Encoding =
Encodings.WindowsANSI
  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.

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 prompts the user to select a text file and then writes the contents of three TextFields to a 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 fs As TextOutputStream
file =
GetOpenFolderItem(TextTypes.Text)
If f <> Nil Then
  fs = TextOutputStream.Append(f)
  fs.WriteLine(NameField.Text)
  fs.WriteLine(AddressField.Text)
  fs.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
```

Dealing with Encodings

As is the case with reading text files, you 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
```

Random Access 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. Conveniently, Binary Files are discussed in the next section.

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 PNG format is public. Other formats are not. Many software vendors do not publish the binary formats that their applications use to create documents.

Binary Streams

Data read from or written to a binary file travels through a BinaryStream. A BinaryStream is a class 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 of the built-in data types.

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 Selector, 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. The EOF property is automatically set to True 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 on Encodings in the Text chapter.

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 of the built-in data types.

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
```

Web Files

When working with web apps, you will often need to be able to make a file available for upload or download.

The WebFile class is used to create a special file that can be processed by the web browser for uploading and downloading.

Downloading Files

To start a download on a WebFile, you use the ShowURL method to show its URL:

```
ShowURL(MyWebFile.URL)
```

Keep in mind that MyWebFile must remain in scope in order for the download to complete.

In general, the easiest way to create a WebFile is to use a FolderItem as the starting point:

```
Dim localFile As FolderItem
localFile = GetFolderItem("localfile.txt")

// mTextFile is a property of the web page
// so that it does not go out of scope
// while the file is downloading.
mTextFile = WebFile.Open(localFile)

ShowURL(mTextFile.URL)
```

The above example ends up loading the file from disk into memory for each session that initiates the file download. This could quickly use up a lot of memory on your web server.

A better approach is to store a single reference to the WebFile on the global App class so that multiple copies are not created.

```
Dim localFile As FolderItem
localFile = GetFolderItem("localfile.txt")

// App.TextFile is a property of App
// so that it does not go out of scope
// while the file is downloading and so that
// it can be reused by multiple sessions.
App.TextFile = WebFile.Open(localFile)

ShowURL(App.TextFile.URL)
```

The URL property gets a URL that is specific to the Session while using the same instance of the WebFile, saving RAM on the server.

Both of the above examples are using the default method of the Open event to load the entire file into memory. To save even more memory on your web server, you can instead have the file read directly from disk (in 64K chunks) by specifying False as the second parameter:

```
Dim localFile As FolderItem
localFile = GetFolderItem("localfile.txt")

// App.TextFile is a property of App
// so that it does not go out of scope
// while the file is downloading.
// The False parameter loads the file from
// disk in 64K chunks instead of loading it all
// into memory at once.
mTextFile = WebFile.Open(localFile, False)

ShowURL(App.TextFile.URL)
```

But an in-memory WebFile can also be useful, especially if you are creating a file to directly download.

You create a WebFile in memory and supply it with data by using the Data property:

```
// mTextFile is a property of the web page
mTextFile = New WebFile
mTextFile.MimeType = "text/plain"

// Ensure the browser downloads the file rather
// than trying to display it.
mTextFile.ForceDownload = True
mTextFile.FileName = "TextFile.txt"

mTextFile.Data = "Hello, world!"

// This causes the file to be downloaded
ShowURL(mTextFile.URL)
```

The above example also demonstrates the usage of the MimeType to specify the type of data contained in the WebFile and ForceDownload to ensure the browser always downloads the file. Some browsers may try to display certain file types (such as text, PDF, etc).

Uploading Files

To upload files to the server using a web app, you can use the WebFileUploader control described in the User Guide Book 2: User Interface.

This control allows the user to add one or more files to be uploaded. When the upload starts (by calling the Upload method), all the files are sent to the web app on the server where you can process them in the UploadCompleted event handler.

Uploaded files larger than 256K are written directly to the temporary folder if it is writeable. The file is kept in memory if the temporary folder is not writeable or the file is 256K or smaller.

For files kept in memory, be aware of the maximum available RAM you have on your web server.

Once the file are uploaded, if the temporary folder is writeable the files in memory are also copied to disk and the UploadCompleted event handler is called.

If the temporary folder was not writeable then all files are in memory.

Regardless, you must process the files in the UploadCompleted event handler in order to save them. Files in the temporary folder are deleted and ones in memory are released when the UploadCompleted event handler returns.

The code below processes each uploaded file (which is a WebUploadedFile) in memory and saves its data to an actual file on disk by using a FolderItem and BinaryStream.

```
Dim bs As BinaryOutputStream
Dim f As FolderItem

For Each file As WebUploadedFile In Files
  f = New FolderItem(file.Name)
  Try
    bs = BinaryStream.Create(f, True)
    bs.Write(file.Data)
    bs.Close
  Catch e As IOException
    // Error, skip file
  End Try
```

However, it is simpler and uses less memory to just call the Save method to save the file to a permanent location:

```
Dim saveFile As FolderItem

For Each file As WebUploadedFile In Files
  saveFile = New FolderItem(file.Name)
  Try
    file.Save(saveFile)
  Catch e As IOException
    // File Error, skip file
    Continue
  End Try
Next
```

XML

XML (Extensible Markup Language) is a human-readable common text file format that can be used for many purposes. It also happens to make a great file format for use by your applications.

You can create, open, modify and manage XML files using the XmlDocument and related classes.

XML is a document format that uses tags, similar to what you may see in HTML. These tags create XML nodes that contain your data. The tags are case-sensitive.

Here is an example XML document that describes three teams in a fictional baseball league:

```
<?xml version="1.0" encoding="UTF-8"?>
<League>
    <Team name="Seagulls">
        <Player name="Bob" position="1B" />
        <Player name="Tom" position="2B" />
    </Team>
    <Team name="Pigeons">
        <Player name="Bill" position="1B" />
        <Player name="Tim" position="2B" />
    </Team>
    <Team name="Crows">
        <Player name="Ben" position="1B" />
        <Player name="Ty" position="2B" />
    </Team>
</League>
```

A tag is anything that is between the brackets, such as <Team>. Within a tag you may have attributes, which is what name is within the Team tag. So in this example:

```
<Team name="Seagulls">
```

Team is a tag and name is an attribute.

The XML structure is very specific. Every tag must have a closing tag, which is the tag name prefixed with a slash. The closing tag for <Team> is </Team>. It is possible to have a single line tag that has its closing tag embedded in it. You can see this here:

```
<Player name="Bob" position="1B" />
```

Because this tag ends in "/>" it is considered to close itself.

Tag names can be anything you want. Because this is your file format, you define the tags and other specifics of its format.

There are many classes available to help you read, process and create XML files, including: XMLDocument, XMLNode, XMLElement and XMLNodeList.

XMLDocument is the primary class you use to work with XML documents. XMLDocument is used to create new XML documents, modify XML documents and load XML documents.

To create XML documents, you create a new instance of XMLDocument and then add the Nodes for the tags you have defined.

Creating an XML Document

To create the XML shown at the beginning of this section, you first create your XMLDocument instance:

```
Dim xml As New XMLDocument
```

Now you can add the topmost node (called the root) to the XML document:

```
Dim root As XMLNode
root =
xml.AppendChild(xml.CreateElement("League"))
```

And now you can add the first team to the root:

```
Dim team As XMLNode
team =
root.AppendChild(xml.CreateElement("Team"))
```

The team has an attribute containing its name:

```
team.SetAttribute("name", "Seagulls")
```

Now you can add the players to the team:

```
Dim player As XMLNode
player =
team.AppendChild(xml.CreateElement("Player"))
player.SetAttribute("name", "Bob")
player.SetAttribute("position", "1B")

player =
team.AppendChild(xml.CreateElement("Player"))
player.SetAttribute("name", "Tom")
player.SetAttribute("position", "2B")
```

Now you are done with the first team and its players. Repeat the code to do the next two teams:

```
team = root.AppendChild(xml.CreateElement("Team"))
team.SetAttribute("name", "Pigeons")
player = team.AppendChild(xml.CreateElement("Player"))
player.SetAttribute("name", "Bill")
player.SetAttribute("position", "1B")

player = team.AppendChild(xml.CreateElement("Player"))
player.SetAttribute("name", "Tim")
player.SetAttribute("position", "2B")

team = root.AppendChild(xml.CreateElement("Team"))
team.SetAttribute("name", "Crows")

player = team.AppendChild(xml.CreateElement("Player"))
player.SetAttribute("name", "Ben")
player.SetAttribute("position", "1B")

player = team.AppendChild(xml.CreateElement("Player"))
player.SetAttribute("name", "Ty")
player.SetAttribute("position", "2B")
```

Lastly, add the code to save the file:

```
Dim f As FolderItem
f = GetSaveFolderItem("", "league.xml")
If f <> Nil Then
  xml.SaveXml(f)
End If
```

This prompts you to save the file (with a default name of league.xml). After the file is saved, you can open it (using a text editor) or most web browsers to see the XML.

Loading an XML Document

Speaking of loading an XML file, here is how you would load the contents of this XML file into a Text Area.

First, you prompt to select the XML file:

```
Dim f As FolderItem
f = GetOpenFolderItem("")
If f = Nil Then Return
```

Now you have a valid file, so try to open it as an XML file:

```
Dim xml As New XMLDocument
Try
  xml.LoadXML(f)
Catch e As XMLException
  MsgBox("XML Error.")
  Return
End Try
```

If the selected file is not an XML file, this raises an XMLException. The exception is caught and an error message is displayed.

Now you have a valid XML file which you can process using the methods and properties of the XMLDocument and XMLNode classes.

This code has three parts to it. Part one verifies that the XML file has a root node called “League”. Part two is a loop that processes each team. And part three is an inner loop that processes each player on each team. The XML data is output to a Text Area.

```
Dim root As XmlNode
root = xml.FirstChild

If root.Name = "League" Then
  Dim team, player As XmlNode
  team = root.FirstChild

  While team <> Nil
    XMLArea.AppendText(team.GetAttribute("name") +
EndOfLine)
    player = team.FirstChild
    While player <> Nil
      XMLArea.AppendText("--->" +
player.GetAttribute("name") + " " +
player.GetAttribute("position") + EndOfLine)
      player = player.NextSibling
    Wend
    team = team.NextSibling
  Wend
Else
  MsgBox("Not a League XML file.")
End If
```

Processing Large XML Files

When you use XMLDocument to load XML files, the entire XML gets loaded into memory at once. This can become a problem for extremely large XML files. In these situations, you can use the XMLReader class to process the XML file in smaller pieces.

XMLReader has a large set of event handlers that are called as parts of the XML file are loaded. You can use these event handlers to look at the data and process it yourself, perhaps only saving the parts you need.

Refer to the Language Reference for information about the various event handlers available with XMLReader.

Extending an XML File Format

XML is a great file format that you should consider using instead of plain text files. A major advantage of using an XML file over a plain text file is that XML makes it much easier to update your file format.

For example, if you wanted to update the format to add a “coach” attribute to the Team tag, you can easily do so when the file is saved and you can modify any loading code to only process this “coach” attribute if it exists.

In the saving code, add a new attribute after each Team node is created, such as this:

```
team.SetAttribute("coach", "Coach Mark")
```

With this change, create a new XML file and load it using the existing loading code. The XML file loads properly, but the new coach attribute is ignored. You have just extended your XML file format without breaking its ability to be loaded.

Of course, you'll want to update the loading code so that it can display the coach. To do that, simply add another line to get the coach attribute after getting the team name attribute:

```
XMLArea.AppendText(team.GetAttribute("name") +
EndOfLine)
XMLArea.AppendText(team.GetAttribute("coach") +
EndOfLine)
```

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. JSON is not as verbose as XML and is often used for Internet and web data communication because its data files are smaller.

You can create and modify JSON data using the JSONItem class. With the JSONItem class, you can manage data similarly to Dictionaries or Arrays.

Here is an example of a JSON document that describes three teams in a fictional baseball league:

```
{
  "Seagulls":{
    "players":{
      "Bob":{
        "position":"1B"
      },
      "Tom":{
        "position":"2B"
      }
    }
  },
  "Pigeons":{
    "players":{
      "Bill":{
        "position":"1B"
      },
      "Tim":{
        "position":"2B"
      }
    }
  },
  "Crows":{
    "players":{
      "Ben":{
        "position":"1B"
      },
      "Ty":{
        "position":"2B"
      }
    }
  }
}
```

This JSON example has a lot of white space to make it easier to read. Without all this white space, the JSON is much smaller:

```
{"Seagulls":{"players":{"Bob":{"position
":"1B"},"Tom":{"position":"2B"}}},"Pigeo
ns":{"players":{"Bill":{"position":"1B"}
,"Tim":{"position":"2B"}}},"Crows":{"pla
yers":{"Ben":{"position":"1B"},"Ty":{"po
sition":"2B"}}}}
```

Creating JSON Data

To create the JSON data shown at the beginning of this section you first create a JSONItem object to hold everything:

```
Dim league As New JSONItem
Dim team, players, player As JSONItem
```

Now you can populate the data for each team. This is how you populate the first team:

```
team = New JSONItem
players = New JSONItem

player = New JSONItem
player.Value("position") = "1B"
players.Value("Bob") = player

player = New JSONItem
player.Value("position") = "2B"
players.Value("Tom") = player

team.Value("players") = players
league.Value("Seagulls") = team
```

Do the same thing for the next two teams:

```
team = New JSONItem
players = New JSONItem

player = New JSONItem
player.Value("position") = "1B"
players.Value("Bill") = player

player = New JSONItem
player.Value("position") = "2B"
players.Value("Tim") = player

team.Value("players") = players
league.Value("Pigeons") = team
```

```
team = New JSONItem
players = New JSONItem

player = New JSONItem
player.Value("position") = "1B"
players.Value("Ben") = player

player = New JSONItem
player.Value("position") = "2B"
players.Value("Ty") = player

team.Value("players") = players
league.Value("Crows") = team
```

Now all the JSON data is created. You can convert this to a string to save (using a TextOutputStream) or to display. This code displays the JSON data in a Text Area:

```
JSONArea.Text = league.ToString
```

Loading JSON Data

JSON data can also be parsed and loaded using the JSONItem class.

When you have the JSON data as a string, you can convert it to a JSONItem class simply:

```
Dim jsonData As String = JSONArea.Text
Dim league As New JSONItem
league.Load(jsonData)
```

With the data in a JSONItem, you can now parse it for display.

You loop through all the teams, displaying them and then for each team display its players.

```
Dim teamName, playerName As String
Dim team, players, player As JSONItem
For i As Integer = 0 To league.Count-1
  teamName = league.Name(i)
  team = league.Value(teamName)

  JSONOutputArea.AppendText(teamName + EndOfLine)

  players = team.Value("players")

  For p As Integer = 0 To players.Count-1
    playerName = players.Name(p)
    player = players.Value(playerName)

    OutputArea.AppendText("--->" + playerName + " ")
    For a As Integer = 0 To player.Count-1
       OutputArea.AppendText(
player.Value(player.Name(a)) + " ")
    Next
    OutputArea.AppendText(EndOfLine)
  Next
Next
```

This code gets the name of each team and displays it in the TextArea. For each team, it then gets the name of each player, looks up their position and displays it all in the TextArea.

Extending the File Format

Much like XML, you can extend your JSON format fairly easily.

For example, if you wanted to update the format to add a “coach” for each team, you can easily do so when the file is saved and you can modify any loading code to only process the “coach” if it exists.

In the saving code, add a new value after each team JSONItem is created, such as this:

```
team.Value("coach") = "Coach Mark"
```

With this change, create new JSON data and display it using the existing loading code. The JSON displays properly, but the new coach value is ignored. You have just extended your JSON data format without breaking its ability to be loaded.

Of course, you’ll want to update the loading code so that it can display the coach. To do that, simply add another line to get the coach after getting the team:

```
team = league.Value(teamName)

Dim coachName As String
If team.HasName("coach") Then coachName =
team.Value("coach")

JSONOutputArea.AppendText(teamName + " "
+ coachName + EndOfLine)
```

File Type Sets

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 PNG 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. PNG files are so named because “PNG” is the file type of a PNG file. Applications are also files but all applications have a file type of “APPL” that tells the Mac OS that this file is an executable and not just data.

Rather than writing code that deals directly with all of these file types, creator codes, and file suffixes, Xojo abstracts you and your code from them with file types. A file type 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.

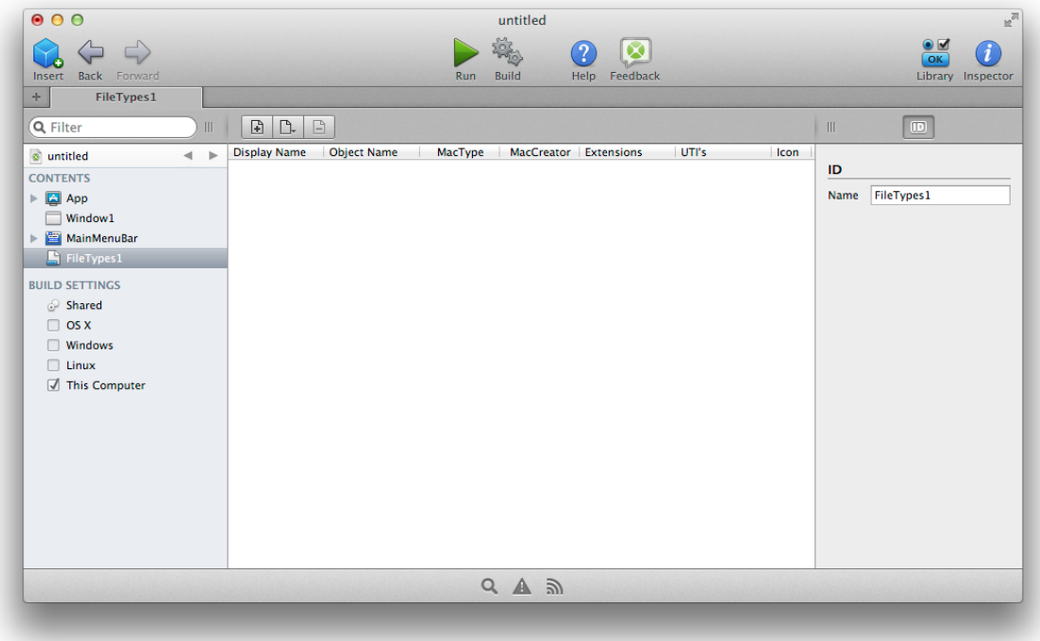
The File Type Sets Editor or the FileType class are used to create File Types. The File Type Sets Editor is described in the following section and the FileType class is described in the Language Reference. The attributes that you give to file types in the editor map directly to properties of FileType class objects.

The File Type Set Editor

You use the File Type Set Editor to create the items that 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 your project: choose Insert ↠ File Type Set. An item called “FileTypes1” is added to your project and displayed in the Navigator and automatically selected to display the File Type Set Editor.

The File Type Set toolbar has these items: Add File Type, Add Common File Type and Remove File Type. Normally you will use

Figure 1.2 File Type Sets Editor



the Add Common File Type button. It enables you to chose a commonly used file type from a list.

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 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.

- Object Name: The name used in your application 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 MacType and MacCreator 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. These settings are no longer relevant in current versions of OS X, which make use of Uniform Type Identifiers (covered in *User Guide Book 4: Development*).

- Extensions: The file extensions used on OS X, Windows, and Linux to identify file types. You can specify more than one extension per file type. Separate multiple extensions with semicolons.

- UTIs: The Uniform Type Identifier. Used by OS X to identify file types. It was developed as a replacement for the OSType (type and creator codes). Introduced in Mac OS X 10.6 Snow Leopard, it was designed to eliminate problems with the File Type and Creator code system. For information on Apple's

UTIs, see http://developer.apple.com/macosx/uniformtypeidentifiers.html. Also refer to the Uniform Type Identifiers section in *User Guide Book 4: Development*.

- Icon: The document icon for that file type. When the user double-clicks such an icon, your 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, your 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. Click the Add Common File Type Button in the File Type Set Editor  to display a list of common file types. Select the one you want.

2. Click "More..." in the list to display additional File Types. This opens an area at the bottom of the File Type Set Editor where you can select more types:

3. After you have chosen the File Type, it appears in the File Type Set Editor. You can click on any of the values to change them if necessary.

Adding a Custom File Type

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.

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. Click the Add File Type Button in the File Type Set Editor  . This adds a blank row to the editor.

2. Enter the values for the File Type.
   To enter more than one file extension, separate them with semicolons.

Adding a Document Icon

You can define a custom document icon for the each file type that the application can open and save. You can import, paste or drag icons into the editor. You can also drag icons from one size to another and they will rescale automatically.

It is best to provide icons and masks for each supported size. Currently, only OS X supports the 512x512 icon size.

***Note**: If you generate an OS X application on Windows or Linux, the 512 x 512 and 256 x 256 icons will not be included because Windows and Linux do not have the necessary JPEG2000 support.*

The document icons are used when your application saves documents in that file type. Double-clicking the document icon will start the application.

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")
```

This statement displays an open-file dialog box that allows the user to view and open only QuickTime movies:

```
f = GetOpenFolderItem("video/quicktime")
```

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, this would refer to the file type with the ObjectName of "JPEG" in the "ImageTypes" File Type Set:

```
f = GetOpenFolderItem(ImageTypes.JPEG)
```

You can concatenate two file types using the "+" operator to specify two file types.

For example, this refers to both the "JPEG" and "PNG" file types in the ImageTypes set:

```
f = GetOpenFolderItem(ImageTypes.JPEG +
ImageTypes.Png)
```

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. This automatically

returns all the file types concatenated together. 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 PNG, GIF, and JPEG files:

```
f = GetOpenFolderItem("image/png;image/
jpeg;image/gif")
```

Virtual Volumes

A Virtual Volume is a special file object that acts as a container for other files. It allows you to have a single file that consists of multiple files.

Virtual Volumes are manipulated using FolderItems and support reading and writing of Text and Binary files.

Virtual Volumes can be used to easily create custom file formats that need to contain other files.

Creating Virtual Volumes

You create a Virtual Volume using the CreateVirtualVolume method of a FolderItem.

```
Dim vFile As FolderItem
vFile = GetFolderItem(“TestVolume.vv”)
Dim vv As VirtualVolume
vv = vFile.CreateVirtualVolume
```

CreateVirtualVolume returns Nil if the Virtual Volume could not be created.

Once you have a Virtual Volume, you can access or add files to it using FolderItems.

Accessing Data in Virtual Volumes

Extending the previous example, you can add a text file to the Virtual Volume by creating a FolderItem in it and then writing text to it using a TextOutputStream:

```
If vv <> Nil Then
  Dim testFile As FolderItem
  testFile = vv.Root.Child("Test.txt")
  Dim output As TextOutputStream
  output = TextOutputStream.Create(testFile)
  output.Write("Hello, world!")
  output.Close
End If
```

Chapter 2

Text

There is a lot to know about text, from how to select it, handle fonts, styling and encodings. In this chapter you will also learn about formatting, regular expressions and how to use the clipboard.

CONTENTS

Text Comparison

All text comparison is case-insensitive by default. This means that text such as “Hello”, “HELLO”, “hello” and “hELLO” are all treated the same when you compare using the comparison operators (=, <, >, <>, <=, >=).

```
If "Hello" = "hello" Then
  MsgBox("They match!") // Displayed
End If

If "Hello" <> "hello" Then
  // Not displayed
  MsgBox("They do not match.")
End If
```

If you need to do case-sensitive text comparisons, then use the StrComp function to test the text values.

StrComp takes three parameters (string1, string2 and mode) and returns an integer result indicating the result of the comparison.

```
If StrComp("Hello”, "hello", 0) = 0 Then
  // Not displayed
  MsgBox("They match!") // Displayed
End If

If StrComp("Hello”, "hello", 0) <> 0 Then
  // Displayed
  MsgBox("The do not match.")
End If
```

The mode parameter indicates the type of comparison to do. When mode = 0, the comparison is done strictly using the bytes of the string. This means that if the string have two different encodings, they will always be different. In most cases you will want to use mode = 0.

When mode = 1, the strings are only compared for case sensitivity it they match exactly other than the case. For example, “Hello” and “HELLO” would be different strings

because they are the same other than the case. However, it does mean that text such as “hello” and “Today” may not compare exactly how you would expect. In the strictest sense, “hello” is greater than “Today” because “h” has a higher character code than “T”. But with mode = 1, that detail is not used because the two strings are not the same.

Selected Text

The term “Selected Text” refers to text that is selected (or “highlighted”) in Text Fields and Text Areas that currently have the focus. Text Fields and Text Areas have three properties that can be used to get and/or set the selected text. Text Fields and Text Areas 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.

***Note**: Selected text only works with desktop applications.*

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 Text Field or Text Area.

The following properties are used to manage selected text within Text Fields and Text Areas.

Figure 2.1 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 Text Field was selected, this property would be 4. Setting this value to 0 (zero) will start the selection at the beginning of the Text Field.
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).

Fonts

Fonts with Desktop Apps

You have the ability to set the font, font size, and font style of many of the objects and controls in your application. Text Areas support multiple fonts, styles, and sizes (collectively referred to as styled text) and List Boxes support multiple styles. Desktop 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. Additionally, many controls also have checkboxes to specify if the font should be Bold, Italic or Underlined.

Text Areas have a TextFont property but they can also display multiple fonts. For information on this, refer to the next section on Styled Text.

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. This font varies by operating system and the user's preferences.

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 Inspector. If you also enter zero as the TextSize, the font size that works best for the platform on which the application is running is used. Because of differences in screen resolution, different font sizes are often required for each operating system platform. This feature enables you to use different font sizes on different platforms without having to create separate windows for each platform. Use the Inspector to set the TextFont to "System" and the TextSize to zero.

Figure 2.2 Setting the Font

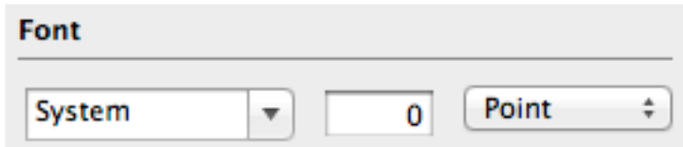


Figure 2.3 System and SmallSystem Font on OS X

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 OS X, both System font and the Small System font are supported. Figure 2.3 illustrates the difference between the two fonts. The TextSize is zero in both cases.

Available Fonts

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. There are 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 is installed:

```
Function FontAvailable(FontName as
String) As Boolean
  Dim i, nFonts As Integer
  nFonts = FontCount-1

  For i = 0 to nFonts
    If Font(i) = FontName Then
      Return True
    End If
  Next

  Return False
End Function
```

Building a Font Menu Dynamically

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, refer to the User Interface Guide. The Desktop chapter has a Menu section that shows you how to do this.

Cocoa Fonts

On OS X Cocoa applications, not all fonts appear as Bold, Italic or Underlined even if those properties are selected. Because of the way Cocoa draws fonts, Bold, Italic and Underline are only displayed for fonts that include it as part of the font definition. These settings cannot be applied to any font as they can be on other platforms.

If you are not seeing the font setting you expect, be sure to verify that the font itself supports it, which you can do using TextEdit or the Font Book app.

Web Fonts

For web apps, you use Web Styles to specify font settings for your controls. For more information on Web Styles, refer to *User Guide Book 2: User Interface, Chapter 3: Web, Section 12: Styles*.

Styled Text

The term styled text refers to text that can have more than one font, font size, and/or font style. The desktop Text Area control supports styled text. In order for a Text Area to support styled text, its MultiLine property must be ON and its Styled property must also be ON, both of which are the defaults. In order to print styled text, you use the StyledTextPrinter class which is discussed in the Printing and Reports chapter.

Determining 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 item.SelTextFont = "" Then
    Start = item.SelStart
    Length = item.SelLength
    For i = Start To Start + Length
      Field.SelStart = i
      Field.SelLength = 1
      If InStr(fonts, Field.SelTextFont) = 0 Then
        If fonts = "" Then
          fonts = Field.SelTextFont
        Else
          fonts = fonts + ", " + Field.SelTextFont
        End If
      End If
    Next
    Return fonts
  Else
    Return Field.SelTextFont
  End If
End Function
```

The SelTextSize property is used to determine the font size of the selected text and works the same way as the SelTextFont

property. If all characters of the selected text are the same font size, the SelTextSize property will contain that size. If different sizes are used, the SelTextSize property will be 0.

There are also boolean properties for determining if all of the characters in the selected text are the same font style. Since text can have multiple styles applied to it, these properties determine if all of the characters in the selected text have a particular font style applied to them. For example, if all of the characters in the selected text are bold but some are also italic, a test for bold returns True. On the other hand, a test for italic returns False since some of the selected text is not in the italic font style. For all of these properties, you test to see if the property is True or False. 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 Font function that determines which fonts are in use in the selected text.

Figure 2.4 Properties Available for Font Styles

Property	Style
SelBold	Bold
SetItalic	Italic
SelUnderline	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. A TextField can also support the Bold, Underline, and Italic styles for all the text in the TextField.

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 have your application 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
```

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:

Figure 2.5 Methods for Toggling Text Styles in Text Areas and Text Fields

Method Name	Style
ToggleSelectionBold	Bold
ToggleSelectionItalic	Italic
ToggleSelectionUnderline	Underline

```
TextArea1.ToggleSelectionBold
```

Styled Text 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, there are also classes 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.

***NOTE**: The StyledText object is not supported for web applications. You can use a WebStyle object to create customized styles for styling controls, including text. Although these concepts are similar, they are not identical. For example, a WebStyle cannot style arbitrary text style runs in the way that the StyledText class described in this section can. Refer to User Guide Book*

2: User Interface, Chapter 3: Web, Section 12: Styles or the WebStyle class entry in the Language Reference for more information about web styles.

Figure 2.6 Properties for Getting or Setting Style Attributes

Property Name	Description
Bold	Gets or sets the Bold style to the selected text in Text.
TextFont	Gets or sets the font for the selected text in Text.
Italic	Gets or sets the Italic style to the selected text in Text.
TextSize	Gets or sets the font size to the selected text in Text.
TextColor	Gets or sets the color of the selected text in Text.
Underline	Gets or sets the Underline style to the selected text in Text.

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
```

Property Name	Description
AppendStyleRun	Appends a StyleRun to the end of Text.
InsertStyleRun	Inserts a StyleRun at a specified position.
RemoveStyleRun	Removes a specified StyleRun from Text.
StyleRun	Provides access to a particular StyleRun in Text. The StyleRun class has its own properties that describe the style that's applied to all the characters in the StyleRun.
StyleRunCount	Returns the number of StyleRuns that make up Text.
StyleRunRange	Accesses the starting position, length, and end position of the StyleRun.
Text	The text that is managed by the StyledText object. Technically, Text is a method, but you can get and set its value as if it were a property.

This sets the first word of the text, "How," to 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.

Property Name	Description
Paragraph	Provides access to a particular Paragraph in Text. The Paragraph class has its own properties that return the start position, length, end position, and alignment of the paragraph.
ParagraphCount	Returns the number of Paragraphs that make up Text.
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: 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 `StyledText1.ParagraphAlignment(0) = Paragraph.AlignRight`

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).

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 Inspector.

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, this code sets the Text property of the StyledText object and displays it in the TextArea:

Figure 2.7 StyledText Display


```
TextArea1.StyledText.Text = "Here is
my styled text." + EndOfLine _
  + "Impressive. Most impressive."
```

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

// Get positions of second paragraph (0-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
```

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.

Encodings

About 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. Your applications can support 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 your own applications, you shouldn't have to worry about encoding issues because the encoding used is stored along with the content of the string.

***Note**: Strings in Structures do not contain encoding information.*

If you are creating applications that open, create, or modify text files that are created outside of your application, 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 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 MacOS 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. For example, if it is coming from an English-speaking user of Windows, 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:

```
theEncoding = Encoding(myString)
```

where the variable myString contains the string whose encoding is to be determined and theEncoding is a TextEncoding object. If the encoding is not defined, the Encoding function returns Nil.

Text Encoding and Files

When dealing with text data in files, it is particularly important to handle encodings properly. Refer to the Text Files section of the Files chapter for information on how to handle this.

Getting Individual Characters

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

There are many ways to control the display and formatting of numbers, dates, and times.

Numbers

Numbers are stored unformatted. Fortunately, there are two functions that can format a number: Format and Str.

The Format function makes providing formatting to numbers easy. The resulting number is formatted to use the number formatting of your OS settings.

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 Str function works similarly to Format, but it does not use the OS settings to format the number.

```
result = Str(Number, FormatSpec)
```

For example, the decimal "." is always used as the decimal separator when Str is used to format the number regardless of what the OS number format setting specifies.

Use the Format function for numbers that should be displayed to the user. Use Str function for numbers that should be stored (perhaps in a database or XML file).

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.

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 OS X, these characters are specified on the Formats panel of the International system preference.

Figure 2.8 Characters used in FormatSpec

Format 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 parenthesis.
)	Displays a closing parenthesis.
+	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.
\	Displays the character that follows the backslash.

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.

Figure 2.9 Examples of Various FormatSpecs

Format Syntax	Result
Format(1.784, "#.##")	1.78
Format(1.3, "#.0000")	1.3000
Format(5, "0000")	x0005
Format(.25, "#%")	25%
Format(145678.5, "###,###.##")	145,678.5
Format(145678.5, "#.##e+")	1.46E+05
Format(-3.7, "-#.##")	-3.7
Format(3.7, "+#.##")	+3.7
Format(3.7, "#.##; (#.##); \z\e\r\o")	3.7
Format(-3.7, "#.##; (#.##); \z\e\r\o")	-3.7
Format(0, "#.##; (#.##); \z\e\r\o")	zero

Dates

Dates are objects and have properties that hold the date in various different formats. To get a date as a string formatted in a specific way, you simply access the appropriate property.

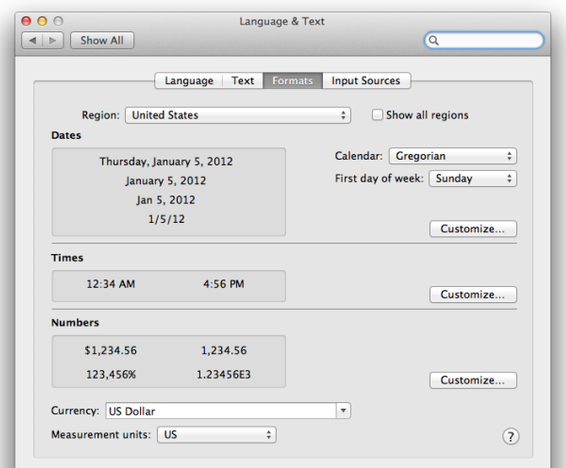
Figure 2.10 Formatting Properties of Dates

Property	Example (default)
ShortDate	10/31/12
LongDate	Wednesday, October 31, 2012
AbbreviatedDate	Wed, Oct 31, 2012

Date formats are controlled by the user's Date Properties (Windows) or Date Formats (OS X) system settings. On OS X, the Date Formats dialog is accessed from the Formats panel of the "Languages & Text" system preference (Click Customize... in the Date area in the Formats panel). 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.

The Date class's ShortDate, AbbreviatedDate, and LongDate properties use whatever format that the user has set in these system settings and may not match what is in the table.

Figure 2.11 Date Format Preferences on OS X



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 now As New Date
Dim theDate As String
theDate = now.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 Jan 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. Figure 2.12 lists the two properties and shows examples of how the time is returned.

Figure 2.12 Formatting Properties to Display Time

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 now As New Date
Dim theTime As String
theTime = now.LongTime
```

As is the case with date formats, several aspects of the ShortTime and LongTime formats are controlled by the user via the Time screen in the Regional Settings Control panel on Windows or the Time Formats panel in “Languages & Text” preferences (OS X).

Regular Expressions

Searching using Regular Expressions

Your applications can search and replace text using regular expressions (often called RegEx), a pattern that describes specific text to find in a string.

You use the properties of the RegEx, RegExOptions and RegExMatch classes to define a regular expression and search or replace text using regular expressions.

Regular Expressions can be a bit tricky to get the hang of, but they are fast and efficient ways to parse text.

RegEx

Class: RegEx

Properties

Options

Assign a RegExOptions instance to adjust search settings.

Figure 2.13 Common Characters Used to Create RegEx Patterns

Matching Character	Description
.	wildcard, matches any character except /r and /n
\r	return
\n	newline
\d	Matches a string of digits
\w	Matches a string of word characters
\s	Matches whitespace
*	Repeat pattern zero or more times
+	Repeat pattern one or more times

ReplacementPattern

The RegEx pattern to use when replacing text.

SearchPattern

The RegEx pattern to use when searching.

SearchStartPosition

The start position to start searching.

Methods

Replace

Finds text using *SearchPattern* and replaces it using *ReplacePattern*, returning the resulting string.

Search

Searches the supplied strings using the pattern specified by SearchPattern.

Use the RegEx class to process regular expressions. The SearchPattern property contains the regular expression to use and you call the Search method (supplying the text to search).

The Search method returns a RegExMatch that contains information about what was found. If nothing is found, then Nil is returned.

You can search again by calling Search multiple times without supplying the text to search.

RegExMatch

Class: RegExMatch

Properties

SubExpressionCount

The amount of subexpressions from the search.

SubExpressionString

An array of subexpressions. Index 0 returns the entire matched string.

Methods

Replace

Used to replace the matched result.

When you call RegEx.Search, an instance of this class is returned if the pattern matched (there were results).

RegExOptions

Class: RegExOptions

This class is used to specify various search options such as case sensitivity, line endings, greediness and more. Refer to the Language Reference for details.

Examples

This example finds the first word in the text “Software development made easy”, returning “Software”:

```
Dim re As New RegEx
Dim match As RegExMatch

re.SearchPattern = "\w*"
match = re.Search("Software development
made easy")
If match <> Nil Then
  MsgBox(match.SubExpressionString(0))
End If
```

See the related Figure 2.14 for examples of other search patterns on the same text “Software development made easy”.

Remember that a RegEx result (RegExMatch) may return more than one match. You should check

RegExMatch.SubExpressionCount to see how many matches were returned.

Figure 2.14 Example RegEx Patterns and Results

Pattern	Result
made	made
simple	Text Not Found
^.	S
[wb]are	ware
..sy$	easy

Replacement

Regular Expressions can also be used to replace strings in text.

This example simply replaces “Software development” with “Programming”:

```
Dim re As New RegEx

re.SearchPattern = "Software develop-
ment"
re.ReplacementPattern = "Programming"
Dim result As String
result = re.Replace("Software develop-
ment made easy")

MsgBox(result)
```

The *result* variable now contains “Programming made easy”.

This example does a simple removal of HTML tags from source HTML:

```
Dim re As New RegEx
re.SearchPattern = "<[^<>]+>"
re.ReplacementPattern = ""

Dim html As String = "<p>Hello.</p>"
Dim plain As String = re.Replace(html)

While (StrComp(html, plain, 0) <> 0)
  html = plain
  plain = re.Replace()
Wend

MsgBox(plain)
```

Clipboard

The Clipboard class is used to get or add data to the system-wide clipboard. 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/raw data. Binary data is represented in string form and is marked with a type you specify so you can tell what the binary data represents.

Getting Text from the Clipboard

Since the Clipboard can contain text, picture and binary data, you should ask it what type of data it contains before you attempt to use it. To check for text, you use the TextAvailable method:

```
Dim c As New Clipboard
Dim clipText As String

If c.TextAvailable Then
  clipText = c.Text
End If
c.Close
```

Adding Text to the Clipboard

This example adds text to the clipboard:

```
Dim c As Clipboard
c.Text = "Hello!"
c.Close
```

For examples on using the Clipboard with graphics and binary (raw) data, see the Clipboard section in the Graphics and Multimedia chapter.

Cryptography

Using the Cryptography functions, you can encrypt or hash your text for security purposes.

The Crypto module contains these methods:

- Hash
- HMAC
- PBKDF2

For each of these you specify the data and the algorithm to use with the Crypto.Algorithm enumerations:

- MD5
- SHA1
- SHA256
- SHA512

In addition, there are MD5, SHA1, SHA256, and SHA512 functions which are convenience methods for Crypto.Hash.

There are also methods for RSA public/private key encryption:

- RSADecrypt
- RSAEncrypt
- RSAGenerateKeyPair
- RSASign
- RSAVerifyKey
- RSAVerifySignature

Examples

This example calculates the hash of the supplied text using SHA256:

```
Dim value As String
value = Crypto.SHA256("DataToEncrypt")
```

RSA Public Key Encryption

With Public Key Cryptography there are two keys: a public key and a private key. The person who wants to receive an encrypted

message generate both of these keys. This can be done in Xojo using the Crypto.RSAGenerateKeyPair function:

```
Dim privateKey As String
Dim publicKey As String
If Crypto.RSAGenerateKeyPair( 1024, _
  privateKey, publicKey ) Then
  // 1024-bit private and public keys
  // were generated
End If
```

The private key is not shared with anyone. The public key can be shared with anyone. To make the public key more presentable, converting it to Base64 is a good idea:

```
viewablePublicKey = EncodeBase64(publicKey)
```

So if you created both a private and public key and shared the public key, others can now create encrypted messages that only you will be able to decrypt. These people create the encrypted message for you by encrypting it using the public key:

```
Dim publicKey As String =
DecodeBase64(PublicKeyArea.Text)
Dim textMessage As String = "Top-secret
message."
Dim msg As MemoryBlock
msg = textMessage
// Encrypt msg using the publicKey
Dim encryptedData As MemoryBlock =
Crypto.RSAEncrypt( msg, publicKey )
If encryptedData <> Nil Then
  MsgBox("Successfully encrypted.")
End If
```

This encrypted message can be sent to you, although again converting it to Base64 can make it simpler to send:

```
Dim msgToSend As String = EncodeBase64(encryptedData)
```

When you receive the message, you can decrypt it using your private key:

```
encryptedData = DecodeBase64(encryptedMsg)
Dim decryptedData As MemoryBlock =
Crypto.RSADecrypt( encryptedData, privateKey )
Dim msg As String = decryptedData
MsgBox(msg)
```

Keep in mind that these “messages” that are being encrypted have to be pretty short (usually just a couple hundred characters, but it depends on the number of bits you use to create the keys).

So typically you use the messages to communicate a “secret key” of some kind that can be used to decrypt an actual message that was encrypted using some other technique (such as AES).

As an example, here is how two people might send a large amount of encrypted data using an encrypted database:

- Julie creates a SQLite database, adds data to it and encrypts it using a secret password.
- Paul creates an RSA Public/Private key pair and gets the Public key to Julie.
- Julie encrypts the secret password using the Public Key from Paul to get an encrypted message that she sends to Paul.
- Paul can decrypt the message from Julie using his Private Key to get the secret password.
- Julie sends the encrypted database to Paul.
- Paul accesses the database using the secret password he previously decrypted.

This is secure because the database cannot be accessed by anyone that does not have the secret password and only the person with the RSA Private Key pair for the Public Key used to encrypt the secret password will be able to decrypt it to open the database.

There is more to RSA encryption, including padding techniques that further improve security. You can learn more about RSA from its Wikipedia topics.

Chapter 3

Graphics and Multimedia

Graphics can make any application look even better. In this chapter you will learn about color, pictures, vector graphics, movies, animation and the use of the clipboard.

CONTENTS

Color

Color is a data type. It consists of three values that define a color. A color can be specified using 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 are Integer values that 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 RGB values that make up 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 &c literal. The &c literal holds a color value. It uses 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, the hex value is inserted into the Value area of the dialog box.

In this example, the ForeColor property of a Graphics object is set to blue so the text drawn will be in that color:

```
Sub Paint(g as Graphics)
  g.ForeColor = RGB(9, 13, 80)
  g.DrawString("Hello World", 50, 50)
End Sub
```

Transparency

Transparency is specified using an optional Integer parameter that varies from 0 (opaque) to 255 (fully transparent). If omitted, it defaults to 0 (no transparency), as in the examples above. For the &c color literal, the transparency parameter is also an Integer, but is specified in hexadecimal. Thus, maximum transparency is the value of “FF” (255). For example, this is valid:

```
&cff0000ff

RGB(255, 0, 0, 255)
```

Transparency Example

The following example draws sample color patches in a Canvas at varying levels of transparency. The code is in the Paint event:

```
Dim red As Color
For i As Integer = 0 To 4
  red = RGB(255, 0, 0, i * 50)
  g.ForeColor = red
  g.FillRect(0, i*20, 200, 20)
  g.ForeColor = &c000000
  g.DrawString("Transparency = " +
Str(i*50), 0, i*20+15)
Next
```

Note: Transparency requires GDIPlus to be enabled for Windows applications and requires Cocoa for OS X applications.

Figure 3.1 Varying Levels of Transparency

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 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 the color (c)
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.

Figure 3.2 Color Pickers on OS X, Windows and Linux

In the 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, the RGB values appear.

The Windows version of the Color Picker uses only one format. 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. 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
End Sub
```

You can see that the code to check the color of a pixel and set the color of a pixel is basically the same.

Pictures

Your applications can contain existing pictures or you can draw your own pictures. In some cases, you can add the pictures you want without writing any code. When you do need to write code, a variety of classes are available to you.

To draw a picture in your application, you use the Canvas (for desktop applications) or the WebCanvas (for web applications) controls.

Since all drawing is done using coordinates to specify the location of things, it is important to understand how they work.

Understanding Drawing Coordinates

Most of the graphics methods require you to indicate the location inside the 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 multiple screens, this is the upper-leftmost corner of the leftmost screen. For a window or web page, the origin is the upper-left corner of the window or web page, and for a control, it's the upper-left corner of the control. The X axis is the horizontal axis. It increases in value moving from left to right. The Y axis is the vertical axis and it increases in value moving from top to bottom.

So, a point that is at 10, 20 (within a window, for example) 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 Canvas control and 20 pixels down from the top edge of the control.

Displaying Pictures

Windows have several ways to display pictures.

Using the Entire Window

With a Window, you can use either the Backdrop property or the Paint event to draw a picture directly to the window background.

The simplest way to do this is to add a picture into your project and then assign it to the Backdrop property in the Inspector.

You can also assign a picture in code:

```
Self.Backdrop = PictureName
```

You can also prompt the user for a picture to display:

```
Dim f As FolderItem
f = GetOpenFolderItem("")
If f <> Nil Then
  Self.Backdrop = f.OpenAsPicture
End If
```

You should only consider using the backdrop for pictures that do not change. If you find you need to change the picture being displayed, then you should use one of the next techniques.

Using a Portion of the Window or Web Page

If you want to display a picture in a portion of the user area (window or web page), you can do so using the platform-specific control, either ImageWell or ImageView.

Both of these controls are simple and only support displaying the picture. They do not support any drawing and you cannot alter the size of the picture being displayed.

ImageWell has an Image property to which you assign a Picture to display it:

```
LogoImageWell.Image = PictureName
```

Of course, you can also prompt the user for a picture and assign it as well.

WebImageView has a Picture property and a URL property that can be used to display a picture.

The URL property displays the picture at the specified URL:

```
LogoImageView.URL =
"http://www.website.com/picture.jpg"
```

You can assign a picture as well:

```
LogoImageView.Picture = PictureName
```

In order to improve performance of you web applications, you should cache your pictures locally in the browser. To do this, simply assign the picture to a property of the web page (or one of its controls) before you use it in your ImageView. This causes the

picture to be sent to the browser only once (when the page with the property is loaded). If you then use it again, it will already be cached in the browser and available for immediate use.

Creating Pictures

In addition to adding pre-existing pictures to your projects, you can also create pictures dynamically.

To do so, you need to have an instance of a Graphics or WebGraphics class. But you cannot instantiate these classes yourself. One way to get an instance of Graphics is to create an instance of a Picture:

```
Dim p As New Picture(100, 100, 32)
```

This creates a picture of the size 100x100 pixels with a 32-bit color depth.

Now that you have a new picture object, you can use its Graphics to draw. To draw an existing picture that has been added to the project:

```
Dim p As New Picture(100, 100, 32)
p.DrawPicture(PictureName, 0, 0)
```

You can then assign your picture object to anything that accepts a picture, such as:

```
LogoImageWell.Image = p // Desktop
LogoImageView.Picture = p // Web
```

Graphics and WebGraphics

All drawing is done using either Graphics or WebGraphics. Graphics can be used in all types of applications, including both desktop and web applications.

As mentioned above, you can always access Graphics by creating a new Picture object. Additionally, in desktop applications any control with a Paint event provides an instance of a Graphics object (as a parameter called g) that you can use for drawing. This is most commonly done using the Canvas control.

In web applications, the WebCanvas control provides you with a WebGraphics instance in its Paint event that you can use for drawing.

***Note**: The example code on the next pages can be used within the Paint event handler of a Canvas or WebCanvas control.*

Scaling Pictures

Often the picture you have is not the right size to display in your application. Using the DrawPicture method on Graphics and

WebGraphics, you can scale your picture to any size. This example, in a Paint event, scales a picture from its original size down to whatever size the Canvas is:

```
g.DrawPicture(PictureName, 0, 0,
g.Width, g.Height, 0, 0,
PictureName.Width,
PictureName.Height)
```

Copying a Portion of a Picture

Sometimes you may need to extract just a portion of a picture. You can easily do that using the DrawPicture method by specifying the coordinates for the portion of the picture you want:

```
g.DrawPicture(PictureName, 0, 0, 30,
30, 10, 10, 40, 40)
```

Scrolling a Picture

***Note**: Scrolling pictures cannot be done using WebCanvas.*

A picture that is drawn into a Canvas with the DrawPicture method can be scrolled by calling the Canvas 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(PictureName, XScroll, YScroll)
```

Drawing Pixels

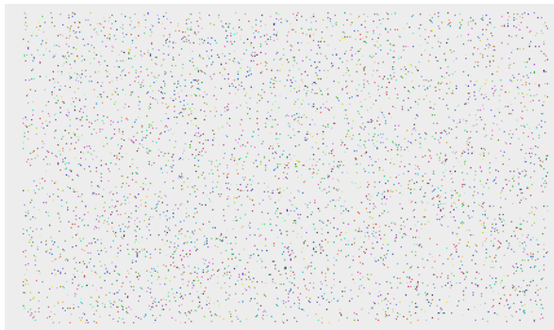
You can get and set the color of individual pixels using the Pixel method. You pass it X and Y coordinates for the pixel and assign it a color.

This example draws 5,000 pixels at randomly selected coordinates within the Graphics of a Canvas:

```
Dim c As Color
For i As Integer = 1 To 5000
  c = RGB(Rnd*255, Rnd*255, Rnd*255)
  g.Pixel(Rnd*Me.Width, Rnd*Me.Height) = c
Next
```

***Note**: Pixel is not available for WebGraphics, so this example does not work with a WebCanvas.*

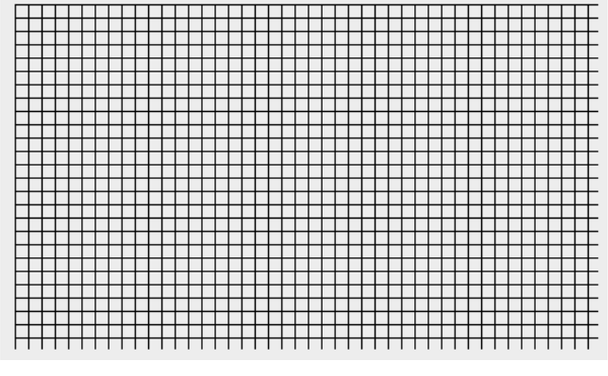
Figure 3.3 Pixels drawn in a Canvas



Drawing Lines

Lines are drawn using the DrawLine method. The color of the line is the color stored in the ForeColor property. To use the DrawLine method, you pass it starting coordinates and ending coordinates of the line.

Figure 3.4 Lines drawn in a Canvas



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 kBoxSize constant:

```
Dim i As Integer
Const kBoxSize = 10
For i = 0 To Me.Width Step kBoxSize
  g.DrawLine(i, 0, i, Me.Height)
Next

For i = 0 To Me.Height Step kBoxSize
  g.DrawLine(0, i, Me.Width, i)
Next
```

Drawing Rectangles

Rectangles are drawn using the DrawRect, FillRect, DrawRoundRect, and FillRoundRect methods.

You supply the X and Y coordinates for the upper-left corner of the rectangle as well as the width and height of the rectangle. Use the ForeColor property to specify the color of the oval.

Figure 3.5 Rectangle drawn in a Canvas

The Draw versions draw a rectangle with just a border. The Fill versions fill the rectangle with the specified ForeColor.

RoundRectangles are rectangles with rounded corners. Therefore, DrawRoundRect and FillRoundRect require two additional parameters: the width and height of the curve of the corners.

This example draws a rectangle and fills it with the color red:

```
g.DrawRect(0, 0, 150, 100)
g.ForeColor = &cFF0000 // Red
g.FillRect(0, 0, Me.Width, Me.Height)
```

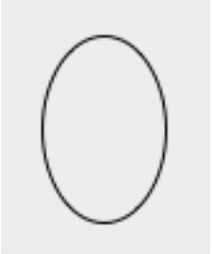
Drawing Ovals

Ovals are drawn with the DrawOval and FillOval methods. You supply the X and Y coordinates for the top-left corner of the oval

and the width and height of the oval. Use the ForeColor property to specify the color of the oval.

DrawOval draws only the border of the oval (use PenWidth and PenHeight to specify the width of the border). FillOval fills the oval with the ForeColor.

Figure 3.7 Oval drawn in a Canvas



This example draws an oval:

```
g.DrawOval(0, 0, 50, 75)
```

Drawing Polygons

Polygons are drawn using the DrawPolygon and FillPolygon methods. You supply an integer array that contains each point in the polygon. This is a 1-based array where odd numbered array elements contain X coordinates 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.

Figure 3.6 Triangle drawn in a Canvas

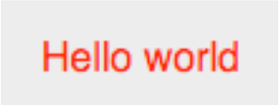
This example draws a triangle:

```
Dim points() As Integer
points = Array(0, 10, 5, 40, 40, 5, 60)
g.DrawPolygon(points)
```

Drawing Text

The DrawString method is used to draw text. You supply the X and Y coordinates for the bottom left of the text and optional parameters to specify whether the text should wrap or be condensed if it cannot fit in the specified area.

Figure 3.8 Text drawn in a Canvas



This example draws text:

```
g.ForeColor = &cff0000
g.TextFont = "Helvetica"
g.TextSize = 16
g.DrawString("Hello world", 10, 130)
```

Drawing Standard Dialog Icons

The MsgBox method and the MessageDialog class allow you to display dialog boxes with a standard dialog icon.

However, there may be times when you need to design your own dialog box and need to use the icons. Using these methods you

can draw the standard dialogs when you need them: DrawCautionIcon, DrawNoteIcon, DrawStopIcon.

***Note**: These methods are not available for WebGraphics.*

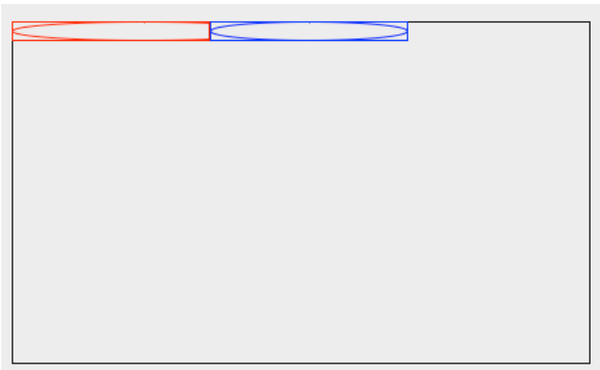
Drawing into Regions

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 and perhaps cause unwanted flicker.

Figure 3.10 Standard Dialog Icons for OS X, Windows and Linux

You create a child region within the parent area with the Clip method. 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.

Figure 3.9 Clipped graphics drawn in a Canvas



Here is an example of how this works. 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, but because the drawing is clipped, there is no overlap.

```
Dim clip1 As Graphics = g.Clip(0, 0, 150, 15)
Dim clip2 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
clip1.ForeColor = &cff0000
clip1.DrawRect(0, 0, clip1.Width, clip1.Height)
// Try to draw outside its clip
clip1.DrawOval(0, 0, 200, 15)

// Draw into the second area in blue
clip2.ForeColor = &c0000ff
//draw the border
clip2.DrawRect(0, 0, clip2.Width, clip2.Height)
clip2.DrawOval(0, 0, 150, 15)
```

***Note**: Clip is not available for WebGraphics.*

Custom Controls

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 supplement the built-in controls.

Suppose you wanted to create a simple custom control like a rectangle whose fill color toggles from black to gray when clicked.

To do so, follow these steps:

1. Drag a Canvas onto a window.
2. Add a property to the Window called "`mFilled As Boolean`".
3. Select the Canvas and add the MouseDown event handler. In this event handler, you toggle mFilled and tell the Canvas to redraw itself.

```
mFilled = Not mFilled
Me.Invalidate
```

4. Finally, in the Paint event, you draw a black rectangle if mFilled = True, otherwise you draw a white rectangle.

```
Dim fillColor As Color
If mFilled Then
  fillColor = &c000000 // Black
Else
  fillColor = &caaaaaa // Gray
End If
g.ForeColor = fillColor
g.FillRect(0, 0, Me.Width, Me.Height)
```

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 can be resized to display at any size and do not pixelate or otherwise “decompose” like bitmap graphics can. The Object2D class is the base class for all the classes that create primitive objects, which include: ArcShape, CurveShape, FigureShape, OvalShape, PixMapShape, RectShape, RoundRectShape and StringShape.

Each of these classes allow you to specify borders, fill and fill colors, rotation, scale and positioning.

***Note**: Vector graphics only work in desktop applications.*

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 defines a RoundRectShape:

```
Dim r As New RoundRectShape
r.Width = 120
r.Height = 120
r.Border = 100
r.BorderColor = RGB(0,0,0) // black
r.FillColor = RGB(255,102,102)
r.CornerHeight = 15
r.CornerWidth = 15
r.BorderWidth = 2.5
```

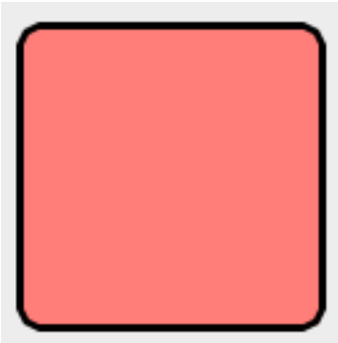
The only problem with this is that the shape doesn’t appear anywhere. It’s just “defined” — ready for your use. You need to add a command to draw the vector object in another object. As you learned in the previous section, a Canvas is a great place to draw stuff.

Use the DrawObject method of Graphics to draw Vector graphics. Adding DrawObject to the above code and adding it to

the Paint event of a Canvas control:

```
Dim r As New RoundRectShape
r.Width = 120
r.Height = 120
r.Border = 100
r.BorderColor = RGB(0,0,0) // black
r.FillColor = RGB(255,102,102)
r.CornerHeight = 15
r.CornerWidth = 15
r.BorderWidth = 2.5

g.DrawObject(r, 100, 100)
```

Figure 3.11 A simple RoundRectShape

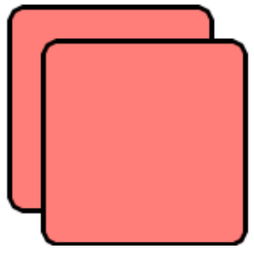


Combining Vector Objects

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.

To illustrate this, take the above code and add a second RoundRectShape object offset from the original by 20 pixels. Use Append to add the two RoundRectShapes to the Group2D object and then draw it in the Canvas:

Figure 3.12 Combining Two Vector Objects



```
Dim r As New RoundRectShape
r.Width = 120
r.Height = 120
r.Border = 100
r.BorderColor = RGB(0, 0, 0) // black
r.FillColor = RGB(255, 102, 102)
r.CornerHeight = 15
r.CornerWidth = 15
r.BorderWidth = 2.5

Dim s As New RoundRectShape
s.Width = 120
s.Height = 120
s.Border = 100
s.BorderColor = RGB(0,0,0) // black
s.FillColor = RGB(255, 102, 102)
s.CornerHeight = 15
s.CornerWidth = 15
s.BorderWidth = 2.5
s.X = r.X + 20
s.Y = r.Y + 20

Dim group As New Group2D
group.Append(r)
group.Append(s)
```

Combining Bitmap and Vector Graphics

You can combine both bitmap and vector graphics by first loading the bitmap graphic into a vector object using the PixmapShape class.

Figure 3.13
Combining Bitmap and Vector Graphics

You then group this with your other vector objects to create a single image. This example combines a bitmap graphic that was added to the project with a StringShape:

```
Dim px As PixmapShape
px = New PixmapShape(PictureName)

Dim s as StringShape
s = New StringShape
s.Y = 70
s.Text = "Hello, world!"
s.TextFont = "Helvetica"
s.Bold = True
Dim d as New Group2D
d.Append(px)
d.Append(s)
g.DrawObject(d, 100, 10)
```

Saving and Loading Vector Graphics

Vector graphics can be saved using the Save method of the Picture class. Vector graphics are stored as PICT files on OS X and as EMF files on Windows.

***Note**: Saving and Loading vector graphics is not supported on Linux.*

Since you need a picture object to call save, you should draw your vector graphics to a picture first and then draw the picture to the Canvas. In this example, mSavePicture is a Picture property on the window:

```
Dim r As New RoundRectShape
r.Width = 120
r.Height = 120
r.Border = 100
r.BorderColor = RGB(0,0,0) // black
r.FillColor = RGB(255,102,102)
r.CornerHeight = 15
r.CornerWidth = 15
r.BorderWidth = 2.5

mSavePicture.Graphics.DrawObject(r,
100, 100)
g.DrawPicture(p, 0, 0)
```

To save this picture, add a button called Save to the window and use this code:

```
Dim file As FolderItem
If TargetMacOS Then
  file = GetSaveFolderItem("", "vector.pict")
Else
 file = GetSaveFolderItem("", "vector.emf")
End If

If file <> Nil Then
  mSavePicture.Save(file, Picture.SaveAsDefaultVector)
End If
```

To load a vector image, you use the OpenAsVectorPicture method of FolderItem. This code prompts the user to select a vector image and then displays it in an ImageWell:

```
Dim file As FolderItem
file = GetOpenFolderItem("")
If file <> Nil Then
  Dim p As Picture
  p = file.OpenAsVectorPicture
  If p <> Nil Then
    VectorWell.Image = p
  End If
End If
```

Movies and Sound

Your applications can display and play movies using several available controls: MoviePlayer (for desktop applications), WebYouTubeMovie and WebMoviePlayer (for web applications).

Playing Movies in Desktop Applications

To play a movie in your desktop application, simply drag a MoviePlayer control onto a window. You can then set the Movie property to the movie that you want to play. This can be done in the Inspector for movies that have been added to the project or it can be done at run-time by assigning a Movie to the Movie property.

On OS X, MoviePlayer uses QuickTime to play the movie. On Windows, you can choose to use QuickTime or Windows Media Player to play the movie by changing the PlayerType property.

***Note**: MoviePlayer is not supported on Linux.*

This code prompts the user to select a movie and then plays it in a MoviePlayer:

```
Dim f As FolderItem
f = GetOpenFolderItem("")

If f <> Nil Then
  MoviePlayer1.Movie = f.OpenAsMovie
  MoviePlayer1.Play
End If
```

Playing Movies in Web Applications

There are two controls that can be used to play movies in web applications: WebYouTubeMovie and WebMoviePlayer.

WebYouTubeMovie

WebYouTubeMovie is a simple control that can only play movies from YouTube. You simply specify the URL to the YouTube movie in the Inspector for the WebYouTubeMovie control.

WebMoviePlayer

WebMoviePlayer can play movies from a variety of sources. If available, it uses the browser's built-in HTML5 video capabilities.

If HTML5 is not supported, then it attempts to use Flash to play the movie.

To play a movie, you assign a URL to one of the URL properties: DesktopURL, MobileCellularURL and MobileWIFIURL. WebMoviePlayer will use the appropriate property depending on the device being used and its Internet connection speed.

There are several methods to control the movie, including: FastForward, FastForwardStop, FastRewind, FastRewindStop, GoToBeginning, GoToEnding, Mute, Play and Reset.

You should always call Reset after changing a movie URL at run-time.

Playing Sounds and Audio

With desktop applications, you have a several options for playing sounds.

Sound Class

Sound files that have been added to your project can be played simply by referencing their name and calling the Play method:

```
SoundName.Play
```

This works for most sound files such as WAV, AIFF, MP3, AAC, etc.

You can also load sounds at run-time using the FolderItem.OpenAsSound method and the Sound class:

```
Dim f As FolderItem
f = GetOpenFolderItem("")
If f <> Nil Then
  Dim s As Sound = f.OpenAsSound
  If s <> Nil Then
    s.Play
  End If
End If
```

The Sound class has properties to adjust the voluming and left/right panning of the sound. It also has methods to play, loop, stop, clone and check if a sound is playing.

Note Player

The NotePlayer class is used to play musical notes. On OS X, it uses QuickTime and on Windows it uses built-in MIDI functions.

***Note**: NotePlayer is not supported on Linux.*

This example plays “do-re-mi-fa-so-la-ti-do”:

```
NotePlayer1.Instrument = 1
// Notes for Do Re Mi Fa So La Ti Do
// (C, D, E, F, G, A, B, C)
Dim DoReMi(7) As Integer
DoReMi = Array(60, 62, 64, 65, 67,
69, 71, 60)

For Each note As Integer In DoReMi
  NotePlayer1.PlayNote(note, 100)
  // Pause to let note play
  App.SleepCurrentThread(500)
Next
```

MoviePlayer

You can also use a MoviePlayer to play sounds. Simply open your sound file as if it were a Movie and assign it to the Movie Property. Using the MoviePlayer to play sounds allows you to use the Movie controller to play and stop the sound.

You can make the MoviePlayer invisible if necessary.

Animation

Web applications can use the WebAnimator class to move, resize, rotate, scale and change the opacity of controls on a web page at run-time.

In order to animate controls, you add a WebAnimator control to your web page. This creates an instance that appears in the Shelf. You can then use this WebAnimator to animate any of the controls on the web page.

With WebAnimator, you can animate a control a single time (such as for moving it from one position to another) or you can queue up a series of animations (such as spinning a control before moving it to a new position).

***Note**: WebAnimator is not supported in desktop applications.*

Animator Actions

Move

The Move method moves the specified control from one location to another. This example in the Action event of a button moves it to coordinate 100, 100 over a period of half a second:

```
Animator1.Move(Me, 100, 100, 0.5)
```

Scale

The Scale method scales a control to a percentage of its original size.

This example reduces an image to 50% of its original size over a period of one second:

```
Animator1.Scale(ImageView1, 50, 50, 1)
```

Resize

The Resize method resizes a control to a specific pixel size.

This example resizes an image to be 100x100 over a period of one second:

```
Animator1.Resize(ImageView1, 100, 100, 1)
```

RotateX

RotateX rotates a control along the X-axis to the specified degrees.

This example flips an image upside down (180 degrees) in half a second:

```
Animator1.RotateX(ImageView1, 180, 0.5)
```

RotateY

RotateY rotates a control along the Y-axis to the specified degrees.

This example creates a mirror image of an image in half a second:

```
Animator1.RotateY(ImageView1, 180, 0.5)
```

RotateZ

RotateZ rotates a control around the Z-axis to the specified degrees:

This example spins an image to get it to be upside down in half a second:

```
Animator1.RotateZ(ImageView1, 180, 0.5)
```

SkewX, SkewY

The skew methods skews (or distorts) a control along the specified axis.

This example skews an image 20 degrees:

```
Animator1.SkewX(ImageView1, 20, 0.5)
```

Opacity

The Opacity method changes the opacity to the specified percentage (where 0 = completely transparent).

This example sets an image to half opacity making it look faded:

```
Animator1.Opacity(ImageView1, 50, 0.5)
```

Clipboard

The Clipboard class is used to get or add data to the system-wide clipboard. 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/raw data. Binary data is represented in string form and is marked with a type you specify so you can tell what the binary data represents.

Dealing with Pictures in the Clipboard

The clipboard can contain any picture that can be stored in the Picture class.

Getting a Picture from the Clipboard

Since the Clipboard can contain text, picture and binary data, you should ask it what type of data it contains before you attempt to use it. To check for a picture, you use the PictureAvailable method:

```
Dim c As New Clipboard
Dim clipPic As Picture
If c.PictureAvailable Then
  clipPic = c.Picture
End If
c.Close
```

Adding a Picture to the Clipboard

This example adds a picture to the clipboard:

```
Dim c As New Clipboard
c.Picture = ImageWell.Image
c.Close
```

Dealing with Raw Data in the Clipboard

Raw data refers to any data that is not text or is not a picture. This allows you to put any type of data you want in the clipboard. Only your own apps will be able to parse this data, however.

Getting Raw Data from the Clipboard

Use the RawData method to get raw data from the Clipboard. The raw data is in string format and you supply a type indicator used to identify the type of data being fetched.

This example gets RTF data from the Clipboard and stores it in a Text Area:

```
Dim c As New Clipboard
If c.RawDataAvailable("RTF") Then
  TextArea1.StyledText.RTFData =
c.RawData("RTF")
End If
c.Close
```

Adding Raw Data from the Clipboard

Use the AddRawData method to add raw data to the clipboard, specifying the type.

This example stores the RTF data from a styled Text Area in the Clipboard:

```
Dim c As New Clipboard
c.AddRawData(TextArea1.StyledText.RTFDat
a, "RTF")
c.Close
```

Chapter 4

Databases

Database support is often needed in applications. In this chapter you will learn about databases and how to connect to SQLite, PostgreSQL, MySQL, Oracle and ODBC.

CONTENTS

Database Concepts

The techniques described in the Files chapter are fine for dealing with your application data in most cases. But there will be times when you need something faster and more structured. That's where a database becomes useful.

Tables, Columns and Data

A database is a structured way of organizing data. This is done using a concept called "Tables". A table is a container for some common set of data. A table has one or more columns that define the data it may contain. Data is stored within the table as rows. Each row is a single set of data.

For example, you could have a table called Team that has these columns: ID, Name, Coach, City. Take a look at Figure 4.1 to see how this table looks with sample data.

Figure 4.1 Team Table with Columns and Sample Data

ID	Name	Coach	City
1	Seagulls	Mike	Albany
2	Pigeons	Mark	Springfield
3	Crows	Matt	Houston

***Note**: The ID column is required by most databases to ensure that each row in the table is uniquely identifiable.*

Relationships

A database usually consists of many tables. And these tables are often related in some way.

For example, to track the players for each team you would have another table, called Player, that is related to the Team table.

Figure 4.2 Player Table

ID	TeamID	Name	Position
1	1	Bob	1B
2	1	Tom	2B
3	2	Bill	1B
4	2	Tim	2B
5	3	Ben	1B
6	3	Ty	2B

Notice that the Player table has its own ID column as well. But in addition, it also has a TeamID column. This column defines the relationship between Player and Team. It clearly tells you the team to which the player belogs. For example, you can see that Bob and Tom both have TeamID = 1. Looking in the Team table, you can see that the team with ID = 1 is the Seagulls. So both Bob and Tom are on the Seagulls team.

You can look up the teams for the other players using the same technique.

The collection of your tables is called the “database schema” or “database design”.

Database Engines

A database design as described above is created using a “database engine”. This is the specific database product that is used to store your database design. The framework has built-in support for these database engines, which are usually just referred to as the database: SQLite, PostgreSQL, MySQL, Oracle and Microsoft SQL Server.

You are not limited to the built-in database support. You can also use many other databases, using a variety of 3rd party libraries, plugins and components.

There are generally two types of databases: embedded and server.

Embedded Databases

An embedded database is a database that is stored alongside your application, usually in a file or series of files. You do not need to install any other software in order to access the database; the necessary software is embedded in your application.

Embedded databases usually only allow a single user to access the data at one time.

SQLite is an example of an embedded database. This type of database is often used in applications such as Mail clients, web browsers, photo management software and anything else that needs to manage a lot of data but does not need to share it with others. You can also use an embedded database such as SQLite with web applications that have light to medium usage.

Server Databases

Server databases are more powerful databases that often have many advanced features, the most significant of which is multi-user access. A server database allows multiple users to access the database at one time.

Examples of server databases include: PostgreSQL, MySQL, Oracle and Microsoft SQL Server.

A server database is usually installed on its own dedicated server. Your application communicates with this server.

Database servers are used by applications that need to share data among multiple users such as a payroll system, billing system or other business applications. Database servers are used by large web sites such as Facebook and Twitter.

SQL (Structured Query Language)

Database engines all share a similar command structure, which is called SQL. It stands for Structured Query Language and is often pronounced as “sequel”. Unfortunately, SQL is not exactly the same between the various databases that you may use. This can make it challenging to switch between database engines (such as SQLite and PostgreSQL, for example) because the SQL used to send commands to the database will likely be different.

Regardless, there are several common SQL commands that are always available, even if their specific syntax changes slightly depending on the database you are using. They are: SELECT, INSERT, UPDATE, DELETE, COMMIT and ROLLBACK.

***Note**: SQL commands are usually written in uppercase.*

SELECT

The SELECT command is used to retrieve specific data from one or more tables. Using the Team example, this command gets the names of all the teams:

```
SELECT Name FROM Team;
```

INSERT

The INSERT command is used to add rows to a table. Using the Team example, this command adds the Seagulls team:

```
INSERT INTO Team (Name, Coach, City)
VALUES ('Seagulls', 'Mike', 'New
York');
```

Note that the ID column is not included here. This is because most databases will populate it for you automatically. The details for how that works vary by database, however.

UPDATE

The UPDATE command is used to modify existing rows in a table. This example changes the name of the Coach for the Seagulls:

```
UPDATE Team
 SET Coach = 'Ken'
WHERE Team = 'Seagulls';
```

DELETE

The DELETE command removes rows from a table. This example removes the Seagulls:

```
DELETE FROM Team WHERE Name = 'Seagulls';
```

COMMIT, ROLLBACK (Transactions)

Changes to a database are made in what is called a transaction. This is a block of processing that either all completes successfully or none completes successfully.

When you are using a transaction, changes made to a database are not made permanent until you commit. This serves two purposes. First, it ensures that data integrity is always maintained. If an error occurs partway through some changes to several tables, you do not want the data to be missing. A failure will revert everything back to its initial state, sort of like an Undo.

Second, for databases that can have multiple users, changes made in a transaction are not usually visible to other users until the transaction is marked as completed. This prevents people from seeing data before it is ready.

Starting a transaction varies depending on the database you are using. Some databases start a transaction for you automatically and some require you to use a specific command, such as:

```
BEGIN TRANSACTION;
```

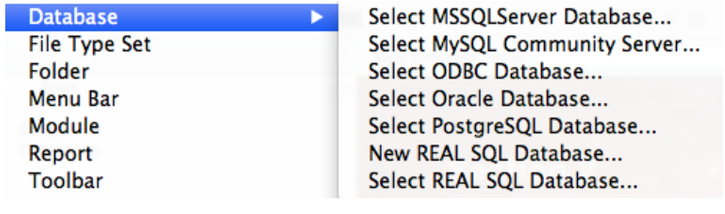
Regardless, to complete a transaction you use the COMMIT command. To cancel (or undo) a transaction you use the ROLLBACK command.

Simple Database Usage

For simple database usage, you can add a pre-existing database to your projects by using the Insert->Database command.

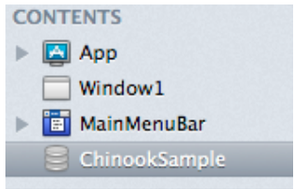
You can choose to add an existing Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL or SQLite database. In addition you can choose to create a new SQLite database.

Figure 4.3 Insert Database Menu



When a database has been added, it appears in the Navigator with its name.

Figure 4.4 SQLite Database in the Navigator

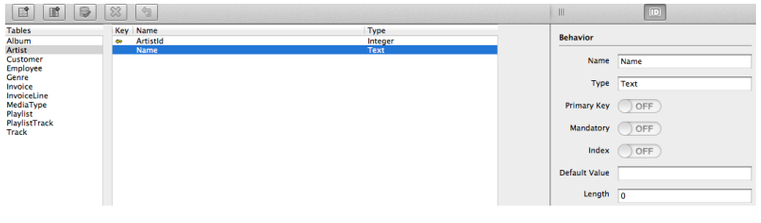


For SQLite, selecting the database in the Navigator displays the Database Editor.

Using the Database Editor

The Database Editor provides a way for you to edit the database schema for SQLite databases. In the left-most column, you see the names of tables in the databases. When a table is selected, the center area displays the columns in the table.

Figure 4.5 Data Editor with Artist Table and Name Column Selected



You can use the toolbar buttons to add and remove tables and columns.

DataControl Control

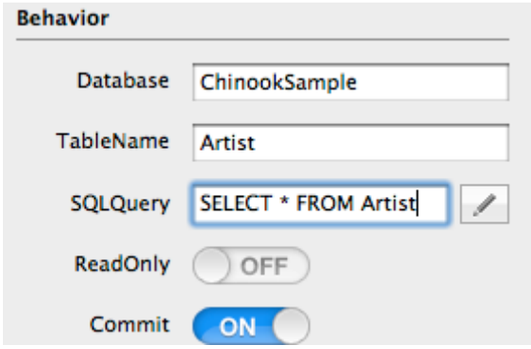
The DataControl control gives you a very simple and powerful way to create a data entry screen that works with a database table. The DataControl is a single object that consists of record navigation buttons (First, Previous, Next, and Last records) and a caption in the center.

With a DataControl, you can display data from a database without writing any code at all. You do this by specifying the database to use, the table to access and the SQL to run. Then

you tell other controls on the window to use the DataControl to display data or perform other actions.

For example, this is how you configure a DataControl to access the Artist information from the ChinookSample database:

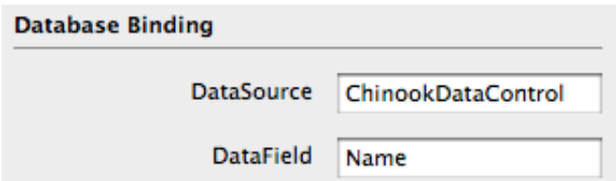
Figure 4.6 DataControl Configured to Access Artist Table



The controls that can display data from a DataControl are: TextField, ListBox, PopupMenu, ComboBox and Label.

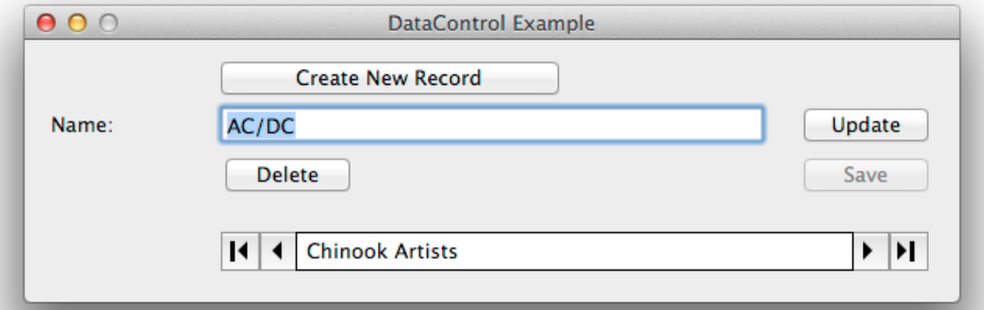
To have a TextField display the Artist Name, you specify the values for the DataSource and DataField properties in the Inspector. These settings are for a TextField called ArtistNameField:

Figure 4.7 DataControl Settings for a TextField



With a DataControl and TextField configured as above, running the project displays the first Artist in the TextField. You can use the navigation buttons on the DataControl to move through the artist names in the table.

Figure 4.8 Data Control in an App



Should you want to control the navigation manually, you can call these methods of the DataControl: MoveFirst, MovePrevious, MoveNext, MoveLast. You can also jump to a specific row using MoveTo.

In addition to displaying data, the DataControl can also manipulate the data in the table. You can add new rows, update existing rows and delete rows.

To do so, you call the appropriate methods on the DataControl class: NewRecord, Insert, Update, Delete

Typically you call these methods on buttons that perform the actions. For example, to delete the currently displayed row, put this code on a Delete button:

```
ChinookDataControl.Delete
ChinookDataControl.RunQuery
```

The RunQuery method repopulates the DataControl so that it will no longer have the deleted row. Generally you call RunQuery after anything that might change the data in the DataControl.

Similarly, this code allows you to edit the currently displayed name and update the row in the database:

```
ChinookDataControl.Update
ChinookDataControl.RunQuery
```

Adding a new row has a few extra steps. First, you'll need a button to create a new row. Then you can put this code in its Action event handler:

```
ArtistDBControl.NewRecord
ArtistNameField.Text = ""
```

This code creates a new row and clears the TextField so you can type the new artist name.

To save this to the database, you need a another button, perhaps called Save, with this code:

```
ArtistDBControl.Insert
ArtistDBControl.RunQuery
```

SQLite

About SQLite

SQLite is the built-in database engine. You access it using the SQLiteDatabase class.

SQLite is an open-source, public-domain embedded database that is used by all kinds of software. It is lightweight, fast and easy to use. It works great for both desktop and web applications.

Unlike most databases, SQLite does not have a strictly typed columns. You can put data of any type in any column, regardless of the type the column has defined.

You can learn more about SQLite by visiting their web site:

www.SQLite.org

Creating a Database

SQLite databases are single files that typically exist on the same computer as the running application. For desktop applications, the SQLite database is usually in the Application Data folder for the operating system. For web applications, the database often resides alongside the web application itself.

You use the SQLiteDatabase class to work with SQLite databases. This code creates a new SQLite database in the Application Data folder:

```
Dim dbFile As FolderItem
dbFile =
SpecialFolder.ApplicationData.Child("MyDatabase.sqlite")

Dim db As New SQLiteDatabase
db.DatabaseFile = dbFile
If db.CreateDatabaseFile Then
  // Use the database
End If
```

If the database already exists, then CreateDatabaseFile will connect to the existing database instead.

Creating a Table

The following SQL command creates the Team table used in prior examples:

```
CREATE TABLE Team (ID INTEGER, Name
TEXT, Coach TEXT, City TEXT, PRIMARY
KEY(ID));
```

With Xojo, SQL commands are sent to SQLite using the SQLExecute and SQLSelect methods of the SQLiteDatabase class. This code can be used to send the above SQL to SQLite from Xojo:

```
Dim dbFile As FolderItem
dbFile =
SpecialFolder.ApplicationData.Child("MyDatabase.sqlite")

Dim db As New SQLiteDatabase
db.DatabaseFile = dbFile
If db.CreateDatabaseFile Then
  // Create the table
  Dim sql As String
  sql = "CREATE TABLE Team (ID INTEGER, Name TEXT, Coach
TEXT, City TEXT, PRIMARY KEY(ID));"
  db.SQLExecute(sql)
  If db.Error Then
    MsgBox("Error: " + db.ErrorMessage)
  End If
End If
```

Connecting to a Database

If you are connecting to a database that already exists, you can instead just call the Connect method:

```
Dim dbFile As FolderItem
dbFile =
SpecialFolder.ApplicationData.Child("MyDatabase.sqlite")

If dbFile.Exists Then
  Dim db As New SQLiteDatabase
  db.DatabaseFile = dbFile
  If db.Connect Then
    // Use the database
  End If
End if
```

Retrieving, Adding, Changing and Deleting Data

For information on how to retrieve, add, change or delete data, refer to the Database Operations section later in this chapter.

Auto-Incrementing Primary Keys

With SQLite, if a table has a single column specified as the INTEGER primary key, then that column auto-increments when a row is added to the table. This column is said to map to the internal rowid column that is on all SQLite tables.

However, just because SQLite has an internal rowid column, you should not rely on it as your primary key. Rowid values can be changed behind the scenes by SQLite and this could possibly corrupt any relationships in your database. Always create a separate primary key for your tables.

When you INSERT data into a table with a primary key, you omit the primary key from the INSERT SQL:

```
INSERT INTO Team (Name)
VALUES ('Seagulls');
```

The above SQL is sent to SQLite using this Xojo code:

```
// db refers to a connected
// SQLiteDatabase
db.SQLExecute("INSERT INTO Team (Name))
VALUES ('Seagulls');"
```

After adding a row to the database, you can get the value of the last primary key value by calling the LastRowID method:

```
Dim lastValue As Integer
lastValue = db.LastRowID
```

Encryption

SQLite databases can be encrypted. An encrypted database cannot be viewed at all unless you know the encryption key.

Encrypting a Database

To encrypt a new database, specify a value for the EncryptionKey property before you call CreateDatabaseFile or before you call Connect.

```
Dim dbFile As FolderItem
dbFile =
SpecialFolder.ApplicationData.Child(
"MyDatabase.sqlite")

Dim db As New SQLiteDatabase
db.DatabaseFile = dbFile
db.EncryptionKey = "MySecretKey123!"
If db.CreateDatabaseFile Then
  // Use the database
End If
```

To encrypt an existing database, call the Encrypt method, supplying the encryption key as a parameter:

```
Dim dbFile As FolderItem
dbFile =
SpecialFolder.ApplicationData.Child(
"MyDatabase.sqlite")

Dim db As New SQLiteDatabase
db.DatabaseFile = dbFile
If db.CreateDatabaseFile Then
  // Encrypt the database
  db.Encrypt("MySecretKey123!")
End If
```

You can also use the Encrypt method to change the encryption key of an encrypted database.

Decrypting a Database

To decrypt an encrypted database, call the Decrypt method after you have connected to the database:

```
Dim dbFile As FolderItem
dbFile =
SpecialFolder.ApplicationData.Child(
"MyDatabase.sqlite")

Dim db As New SQLiteDatabase
db.DatabaseFile = dbFile
db.EncryptionKey = "MySecretKey123!"
If db.CreateDatabaseFile Then
  db.Decrypt // Decrypt the database
End If
```

Multiple User Support

SQLite is not technically a multiple user database. But by enabling a feature called Write-Ahead Logging (WAL), you can improve performance when multiple users are accessing the database.

This is most useful with web applications because they can easily have multiple users connected to the web app, each of which may be connecting to the database.

Large Objects

Large objects in a database are called BLOBs (Binary Large Objects). You can add large objects to a database using the

DatabaseRecord class, but there is a limitation on the amount of available memory.

If you need to store large objects in a database, but want to be able to read the data from the database sequentially, you use the SQLiteBlob class in conjunction with the CreateBlob and OpenBlob methods on the SQLiteDatabase class:

```
Dim blob As SQLiteBlob
blob = db.OpenBlob("Team", "Logo", 1, True)
If blob <> Nil Then
  // Read BLOB
  Dim data As String
  While Not blob.EOF
    // Read 1000 bytes at a time
    data = blob.Read(1000)
  Wend

  // Do something with the data
End If
```

Attaching Other SQLite Databases

Normally when you connect to a SQLite database, you are connecting to a single file. With SQLite it is possible to connect to multiple SQLite database files using one connection. You do this be “attaching” the additional databases.

The AttachDatabase method attaches the specified database file and lets you assign a prefix to use for all the tables in the attached database.

The DetachDatabase method is to remove the attached database.

```
dbFile = GetFolderItem("ExtraDB.sqlite")
If db.AttachDatabase(dbFile, "extra") Then
  Dim rs As RecordSet
  rs = db.SQLSelect("SELECT * FROM
extra.Table")
  // Process results...
End If
```

Determining the SQLite Version

It can sometimes be helpful to know exactly which version of SQLite is being used by your application. You can check this using the LibraryVersion property.

```
MsgBox(db.LibraryVersion)
```

PostgreSQL

About PostgreSQL

PostgreSQL is a free, powerful, cross-platform, open-source database server. To use it, you need to make sure the PostgreSQLPlugin.rbx plugin file is installed in the Plugins folder.

You can learn more about PostgreSQL at their web site:

www.PostgreSQL.org

Connecting to PostgreSQL

To connect to PostgreSQL, you need to have a PostgreSQL server installed on either your computer or an accessible server. You'll need to know several things about this installation, including:

- The Host IP address or name
- The Port being used (usually 5432)
- The name of the database on the server
- The username to use to connect to the server
- The password to use to connect to the server

With this information, you can connect to the database on the server using the PostgreSQLDatabase class:

```
Dim db As New PostgreSQLDatabase
db.Host = "192.168.1.172"
db.Port = 5432
db.DatabaseName = "BaseballLeague"
db.UserName = "broberts"
db.Password = "streborb"
If db.Connect Then
  // Use the database
Else
  // DB Connection error
  MsgBox(db.ErrorMessage)
End If
```

Secure Connections

You can also connect to a PostgreSQL database using SSL for a secure connection. You do this using the SSLMode property to specify the type of secure connection to use:

```
Dim db As New PostgreSQLDatabase
db.Host = "192.168.1.172"
db.SSLMode =
PostgreSQLDatabase.SSLRequire
db.Port = 5432
db.DatabaseName = "BaseballLeague"
db.UserName = "broberts"
db.Password = "streborb"
If db.Connect Then
  // Use the database
End If
```

Creating a Table

This SQL creates the Team table used in previous examples:

```
CREATE TABLE Team (ID INTEGER NOT
NULL PRIMARY KEY, Name TEXT, Coach
TEXT, City TEXT);
```

In place of the TEXT data type, which allows an unlimited length string, you might also use the VARCHAR data type which allows you to specify a maximum size for the string:

```
CREATE TABLE Team (ID INTEGER NOT
NULL, Name VARCHAR(100), Coach
VARCHAR(100), City VARCHAR(100));
```

Auto-Incrementing Primary Keys

PostgreSQL does not allow you to create a primary key that auto-increments. But the equivalent functionality is available by using **Sequences**.

A Sequence is a database object that manages unique values for use by primary keys. You use the sequence when you create new rows in a table.

This SQL declares a sequence for the Team table with values starting at 1:

```
CREATE SEQUENCE TeamSeq START 1;
```

You use the Sequence in INSERT SQL statements like this:

```
INSERT INTO Team (ID, Name)
VALUES (nextval('TeamSeq'), 'Seagulls');
```

The *nextval*, *lastval* and *currval* functions are used to access the next value in the sequence, the last value in the sequence and the current value of the sequence respectively.

Large Objects

Large Objects, or BLOBS, allow you to store non-traditional data in the database such as files, pictures and anything that is binary. Large Objects are stored independently of tables and are referenced using their own unique identifier. To work with BLOBs in PostgreSQL, you use the PostgreSQLLargeObject class.

This example saves a file to a LargeObject:

```
db.SQLExecute("BEGIN TRANSACTION")

// Create the Large Object and save its reference
Dim oid As Integer
oid = db.CreateLargeObject

// Open the newly created Large Object
Dim lo As PostgreSQLLargeObject
lo = db.OpenLargeObject(oid)

// Write the file to the Large Object
Dim bs As BinaryStream
bs.Open(inputFile)
Dim data As String
While Not bs.EOF
  data = bs.Read(1000)
  lo.Write(data)
Wend
bs.Close
lo.Close
db.SQLExecute("END TRANSACTION")
```

***Note**: PostgreSQL requires that all large object operations be performed inside of a transaction as shown in the above example.*

Notifications

Another feature of PostgreSQL is the Listen and Notify protocol. With Listen and Notify, you can have the PostgreSQL Server notifying you of events that you are listening for.

To listen for notifications, you call the Listen method with the name of the notification that you are listening for:

```
db.Listen("LogInNotification")
```

To actually ask the PostgreSQL Server if any of the notifications being listened for have arrived, you call the CheckForNotifications method. Typically you want to do this with a timer so that your application checks for notifications regularly:

```
db.CheckForNotifications
```

When actual notifications arrive, the ReceivedNotificationEvent is called with the name of the notification.

You send notifications using the Notify method like this:

```
db.Notify("LogInNotification")
```

MySQL

About MySQL

MySQL is a powerful, cross-platform, open-source database server.

To use it, you need to copy the MySQLCommunityServerPlugin.rbx plugin file into the Plugins folder. The plugin supports connecting to MySQL Community Edition from Windows, OS X and Linux.

You can learn more about MySQL at their web site:

www.MySQL.com

Licensing

MySQL's licensing is considerably more complex than licensing for other database servers. For more information about its licensing options, refer to their web site:

http://www.mysql.com/about/legal/licensing/index.html

Connecting to MySQL

To connect to MySQL, you need to have a MySQL server installed on either your computer or an accessible server. You'll need to know several things about this installation, including:

- The Host IP address or name
- The Port being used (usually 3306)
- The name of the database on the server
- The username to use to connect to the server
- The password to use to connect to the server

With this information, you can connect to the database on the server using the MySQLCommunityServer class:

```
Dim db As New MySQLCommunityServer
db.Host = "192.168.1.172"
db.Port = 3306
db.DatabaseName = "BaseballLeague"
db.UserName = "broberts"
db.Password = "streborb"
If db.Connect Then
  // Use the database
Else
  // Connection error
  MsgBox(db.ErrorMessage)
End If
```

Secure Connection

You can also connect to a MySQL database using SSL for a secure connection. You do this using the SSLMode property (and optionally other SSL properties) to specify the type of secure connection to use:

```
Dim db As New MySQLCommunityServer
db.Host = "192.168.1.172"
db.Port = 3306
db.DatabaseName = "BaseballLeague"
db.UserName = "broberts"
db.Password = "streborb"
db.SSLMode = True

Dim keyFile As FolderItem
keyFile = GetFolderItem("MySQLKeyFile")
db.SSLKey = keyFile

Dim certFile As FolderItem
certFile = GetFolderItem("MySQLCertificateFile")
db.SSLCertificate = certFile

Dim authFile As FolderItem
authFile = GetFolderItem("MySQLAuthFileFile")
db.SSLAuthority = authFile

Dim authPath As FolderItem
authPath = GetFolderItem("SSLCACertFile")
db.SSLAuthorityDirectory = authPath

Dim cipher As String
cipher = "DHE-RSA-AES256-SHA"
db.SSLCipher = cipher

If db.Connect Then
  // Use the database
End If
```

Creating a Table

This SQL creates the Team table used in previous examples:

```
CREATE TABLE Team (ID INTEGER NOT
NULL AUTO_INCREMENT PRIMARY KEY,
Name TEXT, Coach TEXT, City TEXT);
```

In place of the TEXT data type, which allows an unlimited length string, you might also use the VARCHAR data type which allows you to specify a maximum size for the string:

```
CREATE TABLE Team (ID INTEGER NOT
NULL AUTO_INCREMENT PRIMARY KEY,
Name VARCHAR(100), Coach
VARCHAR(100), City VARCHAR(100));
```

Auto-Incrementing Primary Keys

If a table has the AUTO_INCREMENT attribute assigned to a primary key, then that column auto-increments when a row is added to the table.

When you INSERT data into a table with a primary key, you omit the primary key from the INSERT SQL:

```
INSERT INTO Team (Name)
VALUES ('Seagulls');
```

After adding a row to the database, you can get the value of the last primary key value by calling the GetInsertID method:

```
Dim lastValue As Integer
lastValue = db.GetInsertID
```

Oracle

About Oracle Database

Oracle Database (often just referred to as Oracle) is a powerful database server that is commonly used in large companies. It works on Windows and Linux. Both free and commercial versions are available.

To use it, you need to copy the OraclePlugin.rbx plugin file into the Plugins folder. The plugin supports connecting to Oracle database from Windows, OS X and Linux.

For more information about Oracle Database, refer to their web site:

www.Oracle.com

Connecting to Oracle

To connect to Oracle, you need to have an Oracle server installed on either your computer or an accessible server. Each computer that is to connect to the Oracle Database needs to have the Oracle OCS 9i (or later) client installed.

You'll need to know several things about this installation, including:

- The name of the database on the server
- The username to use to connect to the server
- The password to use to connect to the server

With this information, you can connect to the database on the server using the OracleDatabase class:

```
Dim db As New OracleDatabase
db.DatabaseName = "BaseballLeague"
db.UserName = "broberts"
db.Password = "streborb"
If db.Connect Then
  // Use the database
End If
```

Unfortunately connecting to Oracle Database is rarely that simple. Oracle uses a TNSNAMES.ORA file to configure many of the connection settings. You will likely have to spend time

configuring that file before Xojo will be able to connect to the database server.

Alternatively, you may find that you can specify some settings directly in the DatabaseName property. This syntax connects to a local instance of Oracle Database XE:

```
db = New OracleDatabase

db.DatabaseName =
"(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=10.0.
1.14)(PORT=1521))(CONNECT_DATA=(SID=XE)))"
db.UserName = "SYSTEM"
db.Password = "dbexample"
db.Debug = 1

If db.Connect Then
  MsgBox("Connected to Oracle!")
Else
  MsgBox("Error connecting to Oracle: " +
db.ErrorMessage)
End If
```

Debugging the Connection

Since Oracle can be trickier to configure than some databases, it may help you to enable debug mode. Set the Debug property to True before you connect to log messages to the console:

```
db.Debug = 1
```

Creating a Table

This SQL creates the Team table used in previous examples:

```
CREATE TABLE Team (ID INTEGER NOT
NULL PRIMARY KEY, Name
VARCHAR2(100), Coach VARCHAR2(100),
City VARCHAR2(100));
```

Auto-Incrementing Primary Keys

Oracle does not allow you to create a primary key that auto-increments. But the equivalent functionality is available by using Sequences (similar to how PostgreSQL works).

A Sequence is a database object that manages unique values for use by primary keys. You use the sequence when you create new rows in a table.

This SQL declares a sequence for the Team table with values starting at 1:

```
CREATE SEQUENCE TeamSeq START 1;
```

You use the Sequence in INSERT SQL statements like this:

```
INSERT INTO Team (ID, Name)
VALUES (TeamSeq.nextval, 'Seagulls');
```

The *nextval* and *currval* functions are used to access the next value in the sequence and the current value of the sequence respectively.

Dual Table

In other databases, the FROM part of a SELECT statement is optional allowing you to write SQL like this to get the next sequence value:

```
SELECT TeamSeq.nextval;
```

Or SQL like this to perform a calculation:

```
SELECT 5*5;
```

Oracle does not support the optional FROM, but it does provide a "dummy" table, called DUAL, that you can use for this purpose:

```
SELECT TeamSeq.nextval FROM DUAL;
SELECT 5*5 FROM DUAL;
```

Microsoft SQL Server

About Microsoft SQL Server

Microsoft SQL Server (MSSQL) is a powerful database server that is commonly used in large companies that rely on Microsoft tools. It works on Windows and both free and commercial versions are available.

To use it, you need to copy the MSSQLServerPlugin.rbx plugin file into the Plugins folder. The plugin supports connecting to Microsoft SQL Server from Windows.

For more information about Microsoft SQL Server, refer to their web site:

www.microsoft.com/sqlserver

Connecting to Microsoft SQL Server

To connect to Microsoft SQL Server (MSSQL), you need to have a MSSQL server installed on either your computer or an accessible server. You'll need to know several things about this installation, including:

- The Host IP address or name
- The name of the database on the server
- The username to use to connect to the server
- The password to use to connect to the server

With this information, you can connect to the database on the server using the MSSQLServerDatabase class:

```
Dim db As New MSSQLServerDatabase
db.Host = "192.168.1.172"
db.DatabaseName = "BaseballLeague"
db.UserName = "broberts"
db.Password = "streborb"
If db.Connect Then
  // Use the database
End If
```

Creating a Table

This SQL creates the Team table used in previous examples:

```
CREATE TABLE Team (ID INT NOT NULL
IDENTITY PRIMARY KEY, Name TEXT,
Coach TEXT, City TEXT);
```

In place of the TEXT data type, which allows an unlimited length string, you might also use the VARCHAR data type which allows you to specify a maximum size for the string:

```
CREATE TABLE Team (ID INT NOT NULL
IDENTITY PRIMARY KEY, Name VAR-
CHAR(100), Coach VARCHAR(100), City
VARCHAR(100));
```

Auto-Incrementing Primary Keys

If a table has the IDENTITY attribute assigned to a primary key, then that column auto-increments when a row is added to the table.

When you INSERT data into a table with a primary key, you omit the primary key from the INSERT SQL:

```
INSERT INTO Team (Name)
VALUES ('Seagulls');
```

After adding a row to the database, you can get the value of the last primary key value by accessing the special column @@IDENTITY in a SELECT statement:

```
db.SQLExecute("SELECT @@IDENTITY")
```

Other Databases

In addition to the databases with built-in support described in the preceding sections, you can connect to just about any other database using a variety of methods.

ODBC (described in the next section) allows you to connect to any database for which you have an ODBC driver.

Other database you can use include:

- CubeSQL (http://www.sqlabs.com)
  A cross-platform database server based on SQLite.
- Valentina Database (http://www.valentina-db.com)
  A cross-platform columnar-based database (both embedded and server).
- OpenBase SQL (http://www.openbase.com)
  A cross-platform database originally created for OpenStep/NeXT.
- MonkeyBread Plugins (http://www.monkeybreadsoftware.de)
  MonkeyBread offers plugins to connect to databases using JDBC (Java Database Connectivity), a database connectivity standard similar to ODBC. Additionally, they have plugins to directly connect to a wide variety of databases.
- Studio Stable Database (http://www.studiostable.com)
  Studio Stable Database is a lightweight database server based on SQLite.

ODBC

About ODBC

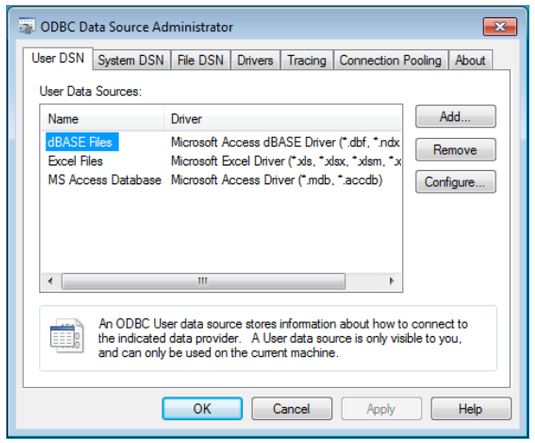
ODBC (Open Database Connectivity) is a database driver standard available on Windows, OS X and Linux. Using the ODBC plugin you can connect to any database for which you have an ODBC driver. ODBC drivers are available for almost any database.

To use ODBC, you need to copy the ODBCPlugin.rbx file into the Plugins folder.

Connecting to a Database using ODBC

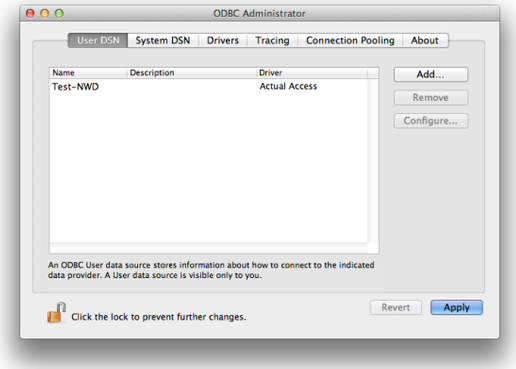
How you connect to an ODBC database depends on the database you are using. The first thing you need to do is configure the ODBC driver using the appropriate ODBC configuration tool for your operating system. In this tool you install the ODBC driver and enter the necessary credentials to connect to the database.

Figure 4.9 ODBC Data Source Administrator on Windows



This results in a DSN or Data Source Name. You can use the DSN to connect to the database in conjunction with the ODBCDatabase class in two ways. You can have your application prompt the user to choose a DSN using the operating system browser for selecting a DSN or you can supply the name of the DSN manually.

Figure 4.10 ODBC Administrator on OS X



This code supplies a blank DataSource property, which prompts the user for the DSN:

```
Dim db As New ODBCDatabase
db.DataSource = ""
If db.Connect Then
  // Use the database
Else
  // Connection error
  // or the user selected
  // Cancel from the ODBC browser
End If
```

Alternatively, you can provide a DSN name and supply credentials. This example uses the existing TestDSN to connect to the database:

```
Dim db As New ODBCDatabase
db.DataSource = "TestDSN"
db.UserName = "broberts"
db.Password = "streborb"
If db.Connect Then
  // Use the database
Else
  // Connection error
End If
```

Lastly, if you know the precise format used by the ODBC driver, you can create the DSN manually:

```
Dim db As New ODBCDatabase
db.DataSource = "DSN=TestDSN;UID=broberts;PWD=streborb"

If db.Connect Then
  // Use the database
Else
  // Connection error
End If
```

Sources for ODBC Drivers

Before you can connect to any database using ODBC, you will need to obtain an ODBC driver. Many database vendors make ODBC drivers available for free. Other sources include:

- Actual Technologies (http://www.actualtech.com)
- OpenLink (http://www.openlinksw.com)
- DataDirect (http://www.datadirect.com)

Database Operations

Once you have connected to a database (refer to the previous sections for the database you are using), the process of doing common database operations is nearly always the same regardless of the database engine you are using. Typically you need to do these actions:

- Retrieve data
- Add data
- Change data
- Remove data

Retrieving Data from a Database

RecordSet

The RecordSet class is used to retrieve data from a database. You supply the SQL SELECT command to get data from one or more tables and then use RecordSet to iterate through the results.

This example gets the name of all the teams and adds them to a ListBox:

```
Dim rs As RecordSet
rs = db.SQLSelect("SELECT * FROM Team")
If rs <> Nil Then
  While Not rs.EOF
    ListBox1.AddRow(rs.Field("Name").StringValue)
    rs.MoveNext
  Wend
  rs.Close
Else
  // Check if there was an error
  If db.Error Then
    MsgBox(db.ErrorMessage)
  End If
End If
```

The SQLSelect method returns a RecordSet. You should always check if the RecordSet is Nil before you attempt to use it. A RecordSet could be Nil because of a database error or something as simple as a typo in your SELECT statement. If it is Nil, you should check the database for an error, which is done in the Else clause.

The While loop iterates through the rows in the RecordSet until you reach the end (EOF stands for End Of File).

The Field method is used to get a particular column value for the current row in the RecordSet, in this case the Name column.

It is very important to call rs.MoveNext, which moves the current row of the RecordSet to the next row. If you forget to do this, it would cause an “infinite loop” because the RecordSet would also stay on the same row, adding it over and over to the ListBox ad infinitum (or until you run out of memory).

The Field method is used to get column values based on the column name. You can also use the IdxField method to get column values based on the position of the column in the SELECT statement (1-based).

SQL

Since you use SQLSelect to get RecordSets, your SQL mostly consists of SELECT statements.

You can just supply the SQL directly as a string (as done in the example above), which works fine for desktop applications. But you do not want to do that with web applications. Because of a hacking technique called “SQL Injection”, you instead want to make use of Prepared SQL Statements to make your SQL more secure.

The syntax for SELECT statements is generally like this:

```
SELECT column1, column2 FROM table
WHERE column = value
ORDER BY sortColumn
```

There are a lot of variations of SELECT statements and the syntax varies depending on the database. Be sure to refer to the documentation for the database you are using to learn about its SELECT statement.

Prepared SQL Statements and Database Binding

Each database class has its own PreparedSQLStatement class, but they all work the same way in general. A Prepared SQL Statement passes the SQL and its arguments separately to the database which then binds them together to create an SQL statement that cannot be affected by SQL Injection.

This is an example of how you would get the names of all the players on the Seagulls team without using Prepared SQL Statements:

```
Dim sql As String
sql = "SELECT * FROM Player WHERE Team = 'Seagulls'"

Dim rs As RecordSet
rs = db.SQLSelect(sql)
```

With a Prepared SQL Statement, you instead only supply a placeholder for the 'Seagulls' parameter (usually a "?", but it varies depending on the database you are using). The value for the parameter is supplied later:

```
Dim sql As String
sql = "SELECT * FROM Player WHERE Team = ?"

Dim ps As SQLitePreparedStatement
ps = db.Prepare(sql)
// Identify the type of the first parameter
ps.BindType(0, SQLitePreparedStatement.SQLITE_TEXT)
// Bind the first parameter to a value
ps.Bind(0, "Seagulls")

Dim rs As RecordSet
rs = ps.SQLSelect
```

As you can see, this code is a little longer because there are extra lines to set the type of each parameter (types are available as constants on the specific prepared statement class) and to bind the value to each parameter. But this is worthwhile because it is much safer than just using straight SQL.

You can simplify this somewhat by providing the values as part of the SQLSelect call like this:

```
Dim sql As String
sql = "SELECT * FROM Player WHERE Team = ?"

Dim ps As SQLitePreparedStatement
ps = db.Prepare(sql)
// Identify the type of the first parameter
ps.BindType(0, SQLitePreparedStatement.SQLITE_TEXT)

Dim rs As RecordSet
rs = ps.SQLSelect("Seagulls")
```

The database-specific SQLPreparedStatement classes are:

- SQLitePreparedStatement
- PostgreSQLPreparedStatement
- MySQLPreparedStatement
- OracleSQLPreparedStatement
- MSSQLServerPreparedStatement
- ODBCPreparedStatement

Adding Data to a Database

You can add new data to your database using two different methods. You can use the DatabaseRecord class to add new rows to the database. Or you can directly use the SQL INSERT statement.

DatabaseRecord

The DatabaseRecord class is used to create new rows in a specific table.

Use the various “Column” methods on the class to assign values to the columns. Then call the InsertRecord method of the database to insert the record into the specified table:

```
Dim row As New DatabaseRecord
row.Column("Name") = "Seagulls"
row.Column("Coach") = "Mike"
row.Column("City") = "Albandy"

db.InsertRecord("Team", row)
If db.Error Then
  MsgBox(db.ErrorMessage)
End If
```

This method works on any database and does not require changes should you change your application to use a different database engine.

SQL

You can also create the SQL for the insert statement manually, but you need to use the correct INSERT syntax for the database you are using.

Generally speaking, INSERT syntax looks like this:

```
INSERT INTO table (column1, column2)
VALUES (value1, value2);
```

You can build up this INSERT command using string concatenation:

```
Dim tableName = "Team"
Dim teamName = "Seagulls"
Dim coachName = "Mike"
Dim cityName = "Albany"
Dim insertSQL As String
insertSQL = "INSERT INTO " + tableName + "(" + _
  "Name, Coach, City) VALUES ('" + _
  teamName + "', '" + coachName + "'," + cityName + _
  ")"
```

But you should really use a PreparedSQLStatement for better results, more security and simpler code:

```
Dim tableName = "Team"
Dim teamName = "Seagulls"
Dim coachName = "Mike"
Dim cityName = "Albany"
Dim insertSQL As String
insertSQL = "INSERT INTO " + tableName + "(" + _
  "Name, Coach, City) VALUES (?, ?, ?)"
Dim ps As SQLitePreparedStatement
ps = db.Prepare(insertSQL)
ps.SQLExecute(teamName, coachName, cityName)
```

Changing Existing Data

Edit and Updating a RecordSet

As you progress through a RecordSet, you can choose to edit the current row and then save those changes back to the database. You do this by calling the Edit method of the RecordSet. This makes the current row editable so that you can change any of the column values.

After you have changed the values, you can update the database with the changes:

```
Dim rs As RecordSet
rs = db.SQLSelect("SELECT * FROM Team")
If rs <> Nil Then
  While Not rs.EOF
    rs.Edit
    If Not db.Error Then
      rs.Field("Name").StringValue = "Billy"
      rs.Update
    Else
      MsgBox(db.ErrorMessage)
    End If

    rs.MoveNext
  Wend
  rs.Close
Else
  // Check if there was an error
  If db.Error Then
    MsgBox(db.ErrorMessage)
  End If
End If
```

SQL

Of course, you can also use direct SQL for this. Like with other SQL, you need to create the UPDATE command to match the syntax required by your database. Generally, it looks like this:

```
UPDATE table
SET column1 = value1
WHERE column2 = value2;
```

Deleting Data

Deleting Using a RecordSet

You can delete the current row in a RecordSet by calling the DeleteRecord method:

```
rs.DeleteRecord
```

SQL

The SQL for deleting data is relatively simple:

```
DELETE FROM table WHERE column = value
```

You use SQLExecute to delete data:

```
Dim deleteSql As String
deleteSQL = "DELETE FROM Team WHERE Name = 'Seagulls'"
db.SQLExecute(deleteSQL)
If db.Error Then
  MsgBox(db.ErrorMessage)
End If
```

Error Handling

As you may have noticed in many of the examples, when dealing with databases proper error handling is essential.

Without error handling, you will have no way to know if your database commands completed successfully, causing you to have invalid data and possible application crashes.

Always check the Error property after every database command. If the Error property is True, then the ErrorMessage property contains a description of the error, which you should display or log.

Getting Information About the Database

At times it can be useful to get information about the database, such as the tables it contains and the columns and indexing on the tables. The Database class has three methods to return this information: TableSchema, FieldSchema and IndexSchema.

TableSchema

The TableSchema method of the database class returns a RecordSet with one column containing the names of all the tables in the database.

This example adds the table names to a ListBox:

```
Dim tables As RecordSet
tables = db.TableSchema
If tables <> Nil Then
  While Not tables.EOF
    ListBox1.AddRow(tables.IdxField(1).StringValue)
    tables.MoveNext
  Wend
  tables.Close
End If
```

FieldSchema

Similarly, FieldSchema returns a RecordSet with information for all the columns (fields) on the specified table. The results can vary depending on the database, but these columns are typically available: ColumnName, FieldType, IsPrimary, NotNull and Length.

This example displays the information for each column in the Team table in a ListBox:

```
Dim columns As RecordSet
columns = db.FieldSchema("Team")
If columns <> Nil Then
  While Not columns.EOF
    ListBox1.AddRow(columns.IdxField(1).StringValue,
columns.IdxField(2).StringValue,
columns.IdxField(3).StringValue,
IdxField(4).StringValue, IdxField(5).StringValue)
    columns.MoveNext
  Wend
  columns.Close
End If
```

***Note**: FieldType is an Integer that describes the data type of the column. Refer to the Data Types section below for a table that maps the integer to a data type.*

IndexSchema

IndexSchema returns the name of the indexes on a table. The RecordSet has one column, the name of the index.

Data Types

Each database has its own set of data types that it uses. These types are identified as an integer in the FieldSchema.FieldType column. These integer values are also used when defining your own data source for use with Reports. Use the following table to map the integer values to the appropriate data types:

FieldType	Value
Null	0
Byte	1
SmallInt	2
Integer	3
Char	4
Text/VarChar	5
Float	6
Double	7
Date	8
Time	9
TimeStamp	10
Currency	11
Boolean	12
Decimal	13
Binary	14
Long Text/Blob	15
Long VarBinary/Blob	16
String	18
Int64	19
Unknown	255

***Note**: Although this is a complete list of datatypes, not all database use every one.*

Chapter 5

Printing and Reports

In addition to being able to print almost anything, you can also design reports with the report designer.

CONTENTS

Printing

For desktop applications, printing is very similar to drawing to the screen. But instead of drawing into a Graphics object of a Canvas control, you draw to a Graphics object created specifically for printing. There are methods available to allow you to create new pages and get printer settings.

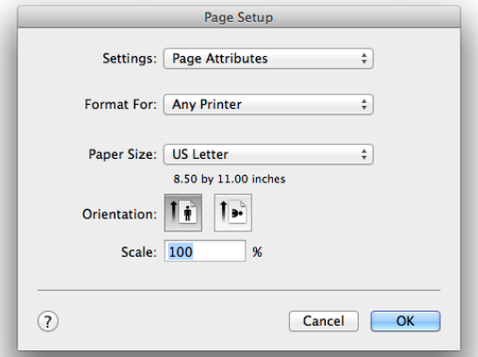
For web applications, printing is far more limited. Usually what works best is to simply create HTML, display it in an HTML Viewer control and then print that.

Desktop Printing

Printer Settings

Before printing anything, you usually want to give the user the ability to select the printer they want to use. This is done using the PrinterSetup class, which displays the Page Setup dialog for the operating system. In addition, this class returns the settings so that you can use them again later without prompting the user.

Figure 5.1 Page Setup Dialog on OS X



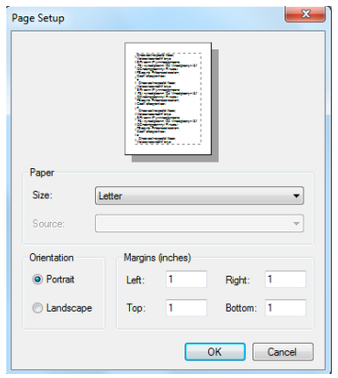

```
Dim ps As New PrinterSetup
ps.SetupString = mPrinterSettings

If ps.PageSetupDialog Then
  mPrinterSettings = ps.SetupString
End If
```

***Note**: mPrinterSettings is a String property on the Window containing this code.*

Since mPrinterSettings is a String, you can save it outside of your application (such as in a database or a file) for use the next time the user prints.

The PrinterSetup class has properties for printer settings such as landscape, page size and resolution. The Language Reference has more details on these properties.

Figure 5.2 Page Setup Dialog on Windows



***Note**: PrinterSetup is not supported in Linux applications.*

Printing Text and Graphics

To print text and graphics you simply draw to the Graphics object for the printer. To get this Graphics objects, you call either the OpenPrinterDialog or OpenPrinter global methods. The only difference between these two methods is that one displays the Print dialog and the other does not. Since the printing system on OS X has the automatic capability to print to PDF, you can use this method to easily generate PDF files.

Figure 5.3 Print Dialog on OS X

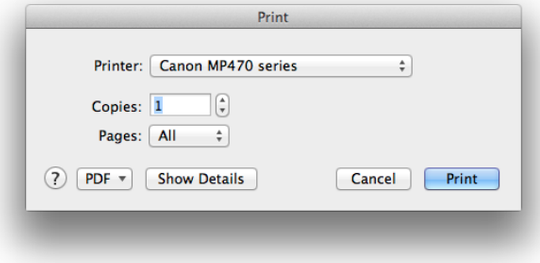
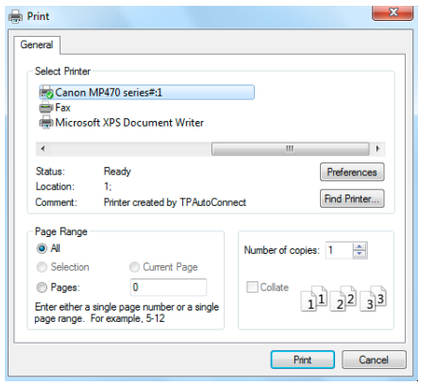


Figure 5.4 Print Dialog on Windows



Since you are drawing to a Graphics object, all the commands that you learned about in the Graphics and Multimedia chapter can be used.

But in addition, you also use the printing-specific method, NextPage, to control when the page is sent to the printer to print. This example prints “Hello” on page 1 and “World” on page 2:

```
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
```

You can also use any printer settings that were previously specified using the PrinterSetup class.

```
Dim ps As New PrinterSetup
ps.SetupString = mPrinterSettings

If ps.PageSetupDialog Then
  mPrinterSettings = ps.SetupString
End If

Dim page As Graphics
// Use Printer Settings
page = OpenPrinterDialog(ps)
If page <> Nil Then
  page.DrawString("Hello", 50, 50)
  page.NextPage
  page.DrawString("World", 50, 50)
  page.NextPage
End If
```

Each time you call NextPage, the graphics object (in this case *page*) is cleared so that you can immediately begin drawing the new page.

To print without displaying the Printer dialog, you just call the OpenPrinter method:

```
Dim page As Graphics
page = OpenPrinter
If page <> Nil Then
  page.DrawString("Hello", 50, 50)
  page.NextPage
  page.DrawString("World", 50, 50)
  page.NextPage
End If
```

Printing Styled Text

Because TextAreas are capable of displaying styled text and multiple font sizes, you will usually want to retain the styled text when you print. The StyledTextPrinter class is used for this purpose, using the DrawBlock method.

To print styled text, you first create a StyledTextPrinter object and then call the StyledTextPrinter method of the TextArea (specifying the graphics to use and the width of the text) to get an instance of a StyledTextPrinter that can be used for printing.

With this instance, you can call the DrawBlock method to draw the styled text to the page (specifying the starting coordinates and the height).

This example prints styled text in a TextArea:

```
Dim page As Graphics
page = OpenPrinterDialog
If page <> Nil Then
  Dim styledText As StyledTextPrinter

  // 72 pixels per inch * 7.5 inches
  Dim textWidth As Integer = 72 * 7.5

  styledText = TextArea1.StyledTextPrinter(page, width)

  // 72 pixels per inch * 9 inches
  Dim blockHeight As Integer = 72 * 9
  styledText.DrawBlock(0, 0, blockHeight)
End If
```

***Note**: In order to support styled printing, the Text Area must have both its Multiline and Styled properties selected.*

If the text to print is larger than what will fit in the specified block, then you can iterate through the text until it is all printed. You do this by checking the EOF property of the StyledTextPrinter class after each call to DrawBlock.

This example prints the contents of a Text Area into two columns with a quarter inch spacing between the columns:

```
Const ppi = 72
Dim page As Graphics
Dim stp As StyledTextPrinter
Dim colWidth, spaceBetweenColumns, pageHeight As Integer
Dim columnToPrint As Integer

// 18 pixels is 1/4 inch (72/4)
spaceBetweenColumns = ppi\4

// 7.5 inches minus 1/4 inch for space
//divided by 2  (72*7.5-18)/2
colWidth = (ppi*7.5 - spaceBetweenColumns) \ 2

// 10 inches * 72 pixels per inch
pageHeight = ppi * 10

page = OpenPrinterDialog()
If page <> Nil Then
  stp = TextArea1.StyledTextPrinter(page, ppi*7.5)
  stp.Width = colWidth
  columnToPrint = 1
  Do Until stp.EOF
    stp.DrawBlock((colWidth+SpaceBetweenColumns)*
(columnToPrint-1), 0, pageHeight)
    If columnToPrint = 2 Then //printing last column
      If Not stp.EOF Then // more text to print
        page.NextPage
        columnToPrint = 1
      End If
    Else // more columns to print on this page
      columnToPrint = columnToPrint + 1
    End If
  Loop
End If
```

Web Printing

Web printing is considerably simpler than desktop printing because it is much more restricted. Your web application runs in a web browser, so you are limited by what a web browser can print.

Web browsers generally do a good job of printing HTML, so in order to generate something to print, you want to first render it as HTML (either to a string or a file) and use an HTML Viewer control to display the HTML.

Once you have what you want displayed in an HTML Viewer, you can have a button that calls the Print method of the HTML Viewer to ask the browser to print its contents:

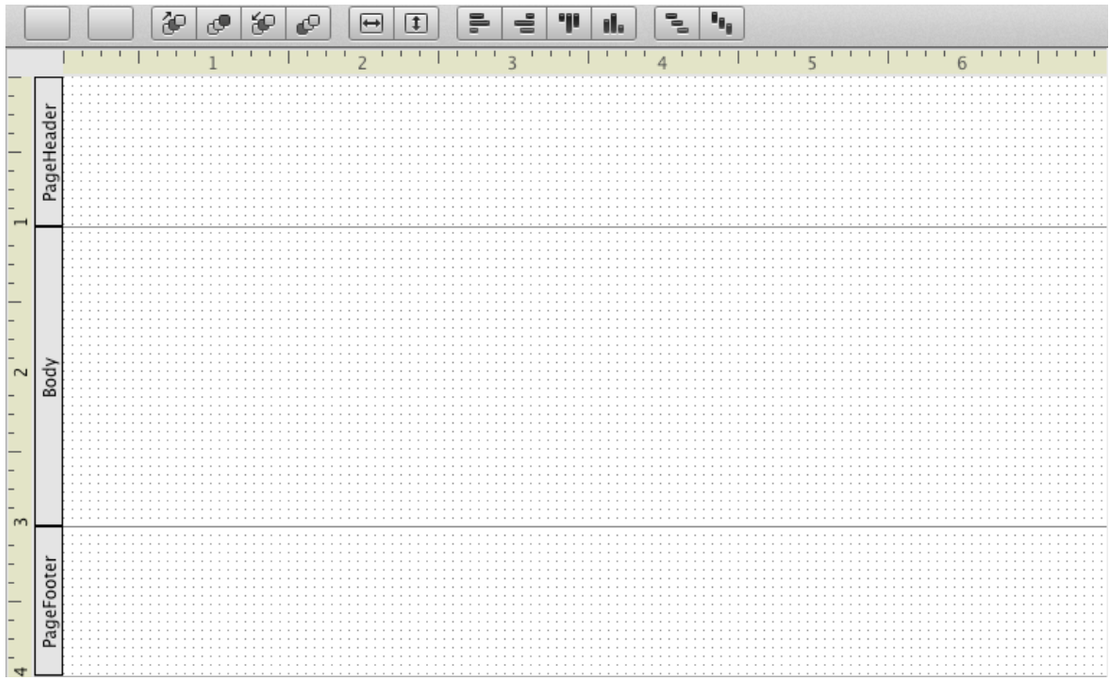
```
HTMLViewer1.Print
```

You can also call the Print method on an HTML Viewer in a desktop application.

Report Layout Editor

You can create reports for desktop applications using the Report Layout Editor. To create a report, use the Insert button or menu and select Report. This adds a report to your project and displays the Report Layout Editor.

Figure 5.5 The Report Layout Editor



You can specify the units for the ruler in Inches, Millimeters or Pixels and specify the width of the report page using the Inspector.

Much like the Window Layout Editor, you drag controls onto the Report Layout Editor to design you reports. The Reports uses a “banded” report design containing multiple bands where information can appear. By default, you see three bands: PageHeader, Body and PageFooter.

Whatever is in the PageHeader band appears at the top of every page, including the first page. Similarly, what is in the PageFooter band appears at the bottom of every page, including the first page.

The Body band is repeated for each line of data that is in the report. For example, if you have a report that is displaying a list of Teams, then you will get a separate Body band for each team.

In order to display data, a report has to have a data set, which is discussed in the next section of this chapter.

Report Layout Editor Controls

There are a variety of controls that you can use in your reports, including: Field, Label, Picture, Line, Oval, Rectangle, RoundRectangle, Date and Page Number.

Each control has two events that can be used for any processing: AfterPrinting and BeforePrinting.

Field

A Field on a report is used to display report data. The fields map to the data source you use with the report, by specifying a value in the DataField property. Generally, you use Fields within the Body band, but they work in any band.

Label

Like a Label on a Window or Web Page, a Label on a report displays text. Labels are used for things like report titles and column headings.

Picture

A Picture is used to display a picture on the report. The picture can be specified when you are designing the report or it can be specified at run-time using information in your data source.

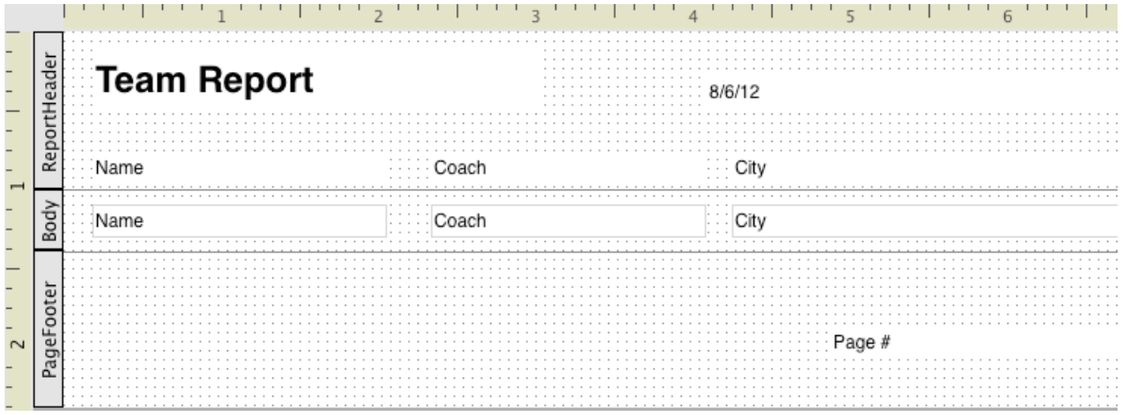
Line, Oval, Rectangle, RoundRectangle

Used to draw shapes on the report.

Date

Used to display the current date when the report is generated.

Figure 5.6 Team Report Layout



Page Number

Displays the page number on each page of the report.

If you wanted to display the Teams from the examples that have been previously used for XML, JSON and databases, you could do a layout similar to Figure 5.6.

Figure 5.7 Team and Player Report Layout

Grouping

Groups are used to display related data together. For example, in the Team and Player examples, there are two sources of data: teams and players. You could use Grouping on a report to display the teams with each player on the team listed below the team name and perhaps indented slightly.

To add a group to the report, you click the “Add New Group” button on the toolbar. This creates two new bands, called GroupHeader1 and GroupFooter1, where you can place additional information. The information in the Group Header appears once for each group. The information in the Group Footer also appears once for each group, but is displayed after the information in the Detail band.

Report Layout Editor Toolbar

The Report Layout Editor Toolbar contains buttons that allow you to alter the layout of the report. This includes:

Add New Group

Adds a new Group to the report layout. This adds both a Group Header and a Group Footer to the layout.

Add New Page Header/Footer

Used to add a Page Header/Footer to a report. A report can only have a single set of Page Header/Footer bands. Since it is possible to delete these bands, use this item to add them back.

Order Forward, Top, Backward, Back

Changes the layering of the controls on the report layout.

Fill Width, Height

Fills the size of the control to match its parent container.

Align Left, Right, Top, Button

Aligns selected report controls with each other.

Space Horizontally, Vertically

Spaces the selected report controls equally apart from each other.

Displaying Data and Printing Reports

Once you have designed your report, you have to provide data for it to display. This is called the data set. Databases are commonly used as data sets for reports, but you can also use any data you want by using the Reports.DataSet interface.

Using a Database as a Data Set

It is pretty simple to use a database as the data set. You simply get the data you want into a RecordSet. Then you can pass the RecordSet to the report for it to use as its data set.

```
// teamRecordSet comes from a database
// SELECT statement
Dim ps As New PrinterSetup

Dim rpt As New TeamReport
If rpt.Run(teamRecordSet, ps) Then
  If rpt.Document <> Nil Then
    // Save the document for
    // the Canvas to display
    mReportDocument = rpt.Document
  End If
End If
```

Using Text as a Data Set

To use anything else as a data set for a report, such a text file, you create a custom class that implements the Reports.DataSet interface. This interface specifies these methods: EOF As Boolean, Field(Integer) As Variant, Field(String) As Variant, NextRecord As Boolean, Run, Type(String) As Integer.

In these methods, you fetch the specific data you need from your actual data source (perhaps a text or XML file) and then return the result or do the expected action.

As a simple example, you can put the team names in a string array and then create a class to work with this array. You have to supply the code for each method in the interface.

To start with, create a class and call it TeamDataSet. Add the Reports.DataSet interface to it. Now add the array to contain the team names as a property of this class:

```
mTeams() As String
```

Create a Constructor method for the class and initialize the array there:

```
mTeams = Array("Seagulls", "Pigeons",
"Crows")
```

Now you can begin implementing the interface methods. Start with NextRecord as it is the simplest. The NextRecord method is used to move the current array position, but before that can be implemented, another property to track the current array position is needed:

```
mArrayPosition As Integer
```

Now you can add the code to move the array position to the NextRecord method:

```
mArrayPosition = mArrayPosition + 1
```

EOF is next. EOF should return True when there is no more data in the array and False if there is still data.

```
If mArrayPosition > mTeams.Ubound Then
  Return True
Else
  Return False
End If
```

This code checks if the current array position will be beyond the size of the array.

Moving along, the Type method is used to return the type of a specified field name. The array only has one field, the team name, so this method should check for the field name of “TeamName” and return its type, which is String. The type is

indicated by the same integer value that specifies types when working with databases. For String, this is 5.

***Note**: You can find the complete table of data type values in the "Getting Information About the Database" section of the Databases chapter.*

```
If fieldName = "TeamName" Then
  Return 5
Else
  Return 0
End If
```

The two Field methods are used to get the value for the specified field in the current array position. One Field method uses an integer field position and the other uses the field name. There is only one field, so that simplifies things.

For Field(String) As Variant:

```
If name = "TeamName" Then
  Return mTeams(mArrayPosition)
Else
  Return ""
End If
```

The name is matched to what is specified for the DataField property for each ReportField in the Report Layout.

For Field(Integer) As Variant, have it use the previous code:

```
If idx = 0 Then
  Return Field("TeamName")
Else
  Return ""
End If
```

Lastly, you need to implement the Run method. This method is called to start the process of getting the data. It is also simple because all you need to do for this example is initialize the array position:

```
mArrayPosition = 0
```

With this class in place, you can now use it with the Report.Run method to populate the report with your data:

```
Dim teamData As New TeamDataSet
Dim ps As New PrinterSetup
Dim rpt As New TeamReport
If rpt.Run(teamData, ps) Then
  If rpt.Document <> Nil Then
    // Save the document for
    // the Canvas to display
    mReportDocument = rpt.Document
  End If
End If
```

Displaying the Report

The Report Layout Editor lets you design your reports, but there is no way to preview them without running your application. Reports are generated as pictures, so you can display them by running the report with the data set and then displaying the generated report in a Canvas. Of course since the report is a Picture, you can do anything you can do with a Picture as well, such as save it directly to a FolderItem or database.

The above examples show how to run the report and have a comment that the document is saved for the Canvas to display.

In order to display the report, the Document for the report is saved to a property of the window: `mReportDocument As Reports.RBReportDocument`.

The RBReportDocument class has a property to tell you how many pages are in the report (PageCount) and methods that are used to display, print and save the report (Page, Print, Save).

To display the report, you get one page at a time using the Page method (which returns a Picture containing the page of the report). You then use a Canvas to display the picture.

For example, this code in the Paint event of a Canvas displays page 1 of a report:

```
If mReportDocument <> Nil Then
  g.DrawPicture(mReportDocument.Page(1), 0, 0)
End If
```

This code always only shows page 1 of the report.

For an actual report preview, you usually display one page at a time and have Back/Next buttons to move between pages. The Back/Next buttons could alter a mCurrentDisplayPage property that you would use here in the Paint event in place of the “1” for the page number:

```
If mReportDocument <> Nil Then

g.DrawPicture(mReportDocument.Page(mCurrentDisplayPage),
0, 0)
End If
```

The PageCount property tells you how many pages there are in total.

Printing the Report

Printing a report works similarly to printing in general. You use PrinterSetup and OpenPrinter (or OpenPrinterDialog) to get a Graphics object. You then call the Print method of the Report Document to print it:

```
// teamRecordSet comes from a database SELECT statement
Dim ps As New PrinterSetup

Dim rpt As New TeamReport
If ps.PageSetupDialog Then
  Dim g As Graphics
  g = OpenPrinterDialog(ps)
  If g <> Nil Then
    If rpt.Run(teamRecordSet, ps) Then
      If rpt.Document <> Nil Then
        // Print the report
        rpt.Document.Print(g)
      End If
    End If
  End If
End If
```

Saving the Report

To save your report, you have a couple options. If you are using OS X, you can let the user take advantage of its ability to print directly to a PDF file. The user can simply use the OS X print dialog to select to save the report to a PDF file rather than sending it to the printer.

Your other choice is to save the Pictures to disk rather than drawing them on screen.

```
Dim teamData As New TeamDataSet
Dim ps As New PrinterSetup
Dim rpt As New TeamReport
If rpt.Run(teamData, ps) Then
  If rpt.Document <> Nil Then
    Dim saveFolder As FolderItem
    saveFolder = SelectFolder
    If saveFolder <> Nil Then
      Dim saveFile As FolderItem
      For i As Integer = 1 To rpt.Document.PageCount
        saveFile = saveFolder.Child("TeamReport" +
Str(i) + ".png")
        rpt.Document.Page(i).Save(saveFile,
Picture.SaveAsPNG)
       Next
    End If
  End If
End If
```

Chapter 6

Devices and Networking

Your applications can communicate with physical devices and can communicate using many Internet networking standards. In this chapter you will learn how to talk to serial devices and how to communicate using the Internet.

CONTENTS

Serial Device Communications

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 are done with the Serial class. To communicate with a serial device you configure an instance of a Serial class, 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 ready to disconnect from the serial device.

Setting Up

The first step is to add a Serial class to your project. The easiest way to do this is to drag a Serial controller from the Library to your Window. You can also drag the Serial controller to the Navigator to create a subclass that you can instantiate in your code.

Before you can begin communicating with a serial device, you need to tell the Serial controller the port to use, the speed of the communication and other settings. If you've added the Serial controller to your window you can change these properties using the Inspector.

The specifics for how you configure these properties depends completely on the device you are using. You should consult the documentation for your device to determine the settings it supports.

Opening the Port

Once you have configured the Serial controller, you can open the serial port to initiate communications with the serial device. This is done by calling the Open method of class. This method returns True if the connection is opened and False if it is not. For example, suppose you have a Serial controller whose name is "SerialDevice". You can open the serial port with the following code:

```
If SerialDevice.Open Then
  MsgBox("Opened serial port.")
Else
  MsgBox("Could not open serial port.")
End if
```

Once you have successfully opened the serial port, it will be unavailable to all other applications (and to other Serial controllers as well) until it is closed.

To close the Serial port, call the Close method:

```
SerialDevice.Close
```

Reading Data

When the serial device sends data back to the Serial controller that is connected to it, 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, the DataAvailable event handler of the Serial controller is called.

In the DataAvailable event handler, you use the Read or ReadAll methods to get some or all of the data in the buffer. This data is returned as a String. 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 use the LookAhead method instead.

This example in the DataAvailable event appends incoming data to a TextArea:

```
TextArea1.AppendText(Me.ReadAll)
```

You can clear all data from the buffer without reading it by calling the Flush method.

Both the Read and ReadAll methods can 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:

```
TextArea1.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 Encodings in Chapter 2.

Writing Data

You can send data to the serial device at any time as long the serial port is open. You send data using the Write method. The data you wish to send must be a string.

The Write method is performed asynchronously. This means that your application does not wait for the Write method to finish before continuing with the next line of code. This behavior allows your application to remain responsive.

This example sends the text in a TextArea out to the Serial device:

```
SerialDevice.Write(TextArea1.Text)
```

If you would rather wait for all data to be sent before continuing with the next line of code, call the XmitWait method before calling Write.

```
SerialDevice.XmitWait
SerialDevice.Write(TextArea1.Text)
```

Changing the Serial Configuration

There may be times when you need to change the Serial controller 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 Poll method.

This updates all properties immediately and calls the DataAvailable event handler immediately if there is any data waiting in the buffer.

```
SerialDevice.Poll
```

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 controllers or other applications.

To close the Serial port, call the Close method using the same Serial controller that opened the port:

```
SerialDevice.Close
```

Communicating with USB and FireWire Devices

Generally speaking, USB and FireWire devices uses a completely different interface that often requires drivers in order to communicate with them.

With that said, some of these devices have a chip in them that makes them appear and behave as if they were a serial device. This chip is often an FTDI chip.

If your device is not recognized as a serial device, then you should investigate using the MonkeyBread Software plugin which has some USB support for specific types of devices.

USB Background

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 communicate with the library using Declare statements.

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.

Using Serial with Web Applications

Web applications can use the serial class to communicate with serial devices that are attached to the web server. The Serial class cannot be used to communicate with serial devices on the user's computer.

Internet Communications

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 the Internet works 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 (Transfer Control Protocol/Internet Protocol) for example, 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 a 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.

Limitations on Sockets in Web Applications

You cannot use a networking socket of any kind on a web page because the socket is really running on the server and would have no way to push its data back to the client browser to update the user interface. However, sockets that are a property of your App class will work.

TCP/IP Communications

Sometimes applications need to communicate with other applications on the same network. This can be accomplished using the TCPSocket class. TCPSocket can send and receive data using the TCP/IP Internet protocol.

TCPSocket is subclassed from the SocketCore class, which is the base class for several network classes.

TCPSocket 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.

Set Up

To get started with TCP/IP communication, you need to add a TCPSocket class to your project. The easiest way to do this is to drag a TCPSocket control from the Library to the window or the Navigator.

Before you can connect to another computer using a TCPSocket, 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 (80), email (25, 110, 143), FTP (20, 21), 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.

A TCPSocket has a Port property that can be assigned at design time or runtime but it must be assigned a value before you can connect to another computer. If you plan on initiating the connection, you must also set the Address property to the address (IP or resolvable name) of the computer to which you want to connect.

***Note**: OS X and Linux have built-in restrictions regarding port numbers. Ports below 1024 cannot be assigned by a user who is not running with "root" privileges. 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 OS X or Linux 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 on Creating a Server later in this chapter.*

Connecting 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 Connect method. Keep in mind that a connection is not necessarily established immediately after you call Connect. You have to wait for the application to which you are trying to connect to accept your connection. In addition, there could be general latency or other issues that take time to resolve.

```
TCPConnection.Connect
```

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. 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 Connected event handler is called. If a connection is not established, an error occurs and the Error event handler is called.

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

Your application can also listen for a connection request from another application. To do this, you use the TCPSocket Listen method:

```
TCPConnection.Listen
```

Once you put a TCPSocket into listen mode, it waits asynchronously for a connection request. Your application remains responsive while the socket is waiting and code continues to run.

When a connection is requested, the Connected event handler is called, which lets you know you have a connection.

Reading Data

When the application at the other end of the connection sends data back to the TCPSocket that it's connected to, the

DataAvailable event handler is called. 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.

In the DataAvailable event handler, you can use the TCPSocket Read or ReadAll methods to get some or all of the data (returned as a String) in the buffer. 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, use the LookAhead method.

This example in the DataAvailable event handler appends incoming data to a TextField:

```
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 specify which encoding was used. For more information on text encoding, see the section Encodings in Chapter 2.

Both the Read and ReadAll methods of the TCPSocket 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:

```
TextField1.AppendText(Me.ReadAll(Encodings.WindowsANSI))
```

Writing Data

You can send data to the application you are connected to at any time. You send data using the Write method. The data you wish to send must be a string. In this example, the text from a TextField is being sent via a TCPSocket control:

```
TCPConnection.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 sends the text in the TextField using the MacRoman encoding:

```
TCPConnection.Write(ConvertEncoding(TextField1.Text,
Encodings.MacRoman))
```

After all your data has been sent, the SendComplete event handler is called. As data is being sent, the SendProgress event handler is called so that you can tell how much data has been sent so far. Never assume how much data is sent between calls to the SendProgress event. It is normal to see this fluctuate.

***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 Error event handler is executed. Errors are represented by numbers. The LastErrorCode property contains 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.

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 Close method. Suppose you have a Socket named “TCPConnection” that has established a connection. To close the connection, you can use the following code:

```
TCPConnection.Close
```

Other Socket Information

Destroying a Socket

When you call the Connect or Listen methods your socket’s reference count is incremented.

This means that the socket does not have to be owned by the anything (such as 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 MySocket. In the action event for a Button you use the following code:

```
Dim s As New TCPSocket
s.Port = 7000
s.Address = "127.0.0.1"
s.Listen
```

Even after this event is completed, the socket stays active listening for connections. In fact, the socket stays active and alive until you specifically close it by either closing the connection locally or remotely.

Of course, the socket will also close when your application quits.

Security

The SSLSocket class is used to do secure communications via TCP/IP.

It works similarly to the TCPSocket class, but has additional properties for the connection type, managing certificate locations and for checking if a secure connection was made.

HTTP (web) Communications

HTTP (HyperText Transfer Protocol) is the protocol that web browsers use. To communicate using the HTTP protocol use the HTTPSocket class and the HTTPSecureSocket class. These classes contains methods and properties that enable you to do all types of HTTP communication such as retrieving a URL or posting a form.

HTTPSocket is subclassed from TCPSocket and HTTPSecureSocket is subclassed from TCPSecureSocket.

HTTPSocket

Some of the more common things you do with an HTTPSocket is to get the contents of a web page and post forms to web pages.

To use an HTTPSocket, you can drag a TCPSocket to your window and change its Super to HTTPSocket. Or you can add a subclass to your project.

Getting Web Page Contents

The Get method is used to get web page contents. It has a variety of ways it can be used. The simplest way to to use it is to synchronously download data to a string. This code downloads the contents of a web site and sets the timeout to 0 so that it waits indefinitely until the page is received.

```
Dim pageData As String
pageData =
HTTPSocket1.Get("http://www.wikipedia.org", 0)
```

You can also download directly to a FolderItem and you have the option of downloading asynchronously. When you download asynchronously, your code does not stop on the Get line. Instead, your code continues running and your app remains responsive. When the download is completed (and as data is being received), events on the HTTPSocket are called.

This syntax gets the same web page asynchronously:

```
HTTPSocket1.Get("http://www.wikipedia.org")
// Code continues
```

When the download has completed, the PageReceived event handler is called. There are also other events that get called as the page is being downloaded such as Connected, ReceiveProgress, HeadersReceived and DownloadComplete.

Posting to a Form

You can submit data to a web page by posting it. To do so, you create a Dictionary with the data for the form, assign this data to the HTTPSocket and then Post it to the web page.

Here is an example that sets information for forms with two fields, FirstName and LastName:

```
Dim formData As New Dictionary
form.Value("FirstName") = "Bob"
form.Value("LastName") = "Roberts"

Dim result As String
HTTPSocket1.SetFormData(form)
result = HTTPSocket1.Post("http://www.wikipedia.org", 0)
```

Like with Get, Post has a variety of ways it can be used. You can have its results go to a FolderItem and you can also process it asynchronously.

Security

The HTTPSecureSocket is used for secure HTTP communication. It works the same as the HTTPSocket class but uses SSL (Secure Socket Layer) for security.

SOAP

SOAP (Simple Object Access Protocol) is a communication protocol for exchanging data between applications using HTTP. These are often referred to as “web services”.

Use the SOAPMethod and SOAPResult classes to communicate to SOAP web services.

REST

REST (Representational State Transfer) is another way to provide web services. It is not a protocol, but more of an API (Application Programming Interface) based on HTTP that allows applications to exchange information.

Email

You can send and receive email using the SMTP (Standard Mail Transfer Protocol) and POP3 (Post Office Protocol) protocols.

Sending Email

To send email, you use the SMTPSocket class. To send secure email, use the SMTPSecureSocket class.

Before you can send an email, you need to create the email message. You do this using the EmailMessage class.

This example creates simple email message and then sends it using an SMTPSocket:

```
Dim email As New EmailMessage
email.FromAddress = "paul@xojo.com"
email.Subject = "Hello, world!"
email.BodyPlainText = "I love Xojo!"
email.AddRecipient("custserv@xojo.com")

SMTPSocket1.Address = "mail.myserver.com"
SMTPSocket1.Port = 25
SMTPSocket1.Messages.Append(email)
SMTPSocket1.SendMail
```

The EmailMessage class has even more properties and methods to set HTML message text, headers and CC recipients.

You can also add attachments to your email using the EmailAttachment class.

***Note**: You have to provide your own SMTP server to actually send the email. You can choose to use your ISP mail server or a 3rd party server such as Gmail. Most SMTP servers will require you to log on with a username and password.*

Receiving Email

To receive email, you use the POP3Socket class. To receive email from a secure server, use the POP3SecureSocket class.

The POP3Socket connects to your POP3 server and requests mail messages. As messages are received, the MessageReceived event handler is called with the email as an EmailMessage.

POP3Socket has many events, properties and methods to allow you to receive POP3 email.

***Note**: Many email services use IMAP instead of POP. If you need IMAP, consider looking into the MonkeyBread Software Plugin.*

Security

For secure sending and receiving of email, use the SMTPSecureSocket and POP3SecureSocket classes. They work the same as the classes described above, but use SSL (Secure Socket Layer) for secure communication.

UDP Communications

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.

Like the TCPSocket control, the UDPSocket control is derived from the SocketCore class.

In order to use a UDPSocket, it must be bound to a specific port on your machine. Once bound, the UDPSocket is ready for use. It immediately begins 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 is a built-in class with 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.xojo.com, or some other network address. It is a Datagram that has one intended receiver.

This example sends data to a specific machine:

```
Dim data As New Datagram
data.Address = "www.xojo.com"
data.Data = "Hello, World!"
data.Port = 9500

UDPSocket1.Write(data)
```

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 local 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 there is a property of the UDPSocket class that tells you the correct broadcast address.

This example broadcasts a message to all machines on your local network:

```
Dim data As String
data = "Hello, World!"

// 10.0.1.25 is the machine
// broadcasting the message
UDPSocket1.Write("10.0.1.25", data)
```

The third mode of operation for UDPSockets 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.

This example sends data to members of the multicast group:

```
Dim data As String
data = "Hello, World!"

If UDPSocket1.JoinMulticastGroup("224.0.0.0") Then
  UDPSocket1.Write("224.0.0.0", data)
End If
```

Creating a Server

Handling Multiple Connections

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.

***Note**: On 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.

***Note**: ServerSocket is only for use with TCPSockets. Since UDP is a connectionless protocol, 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.

Interprocess Communications

You use interprocess communication as a way for two applications on the same computer to communicate with one another.

IPCSocket

To do this, you use the IPCSocket class, which works very similarly to the TCPSocket class. Instead of specifying an IP address and a port, you specify a path to a “socket file”. Both applications must use the same path. The file is used for communication.

***Note**: The socket file is not removed after closing the connection. You should remove the file in your code or put the file in the system temporary folder where the OS can remove it as needed.*

Some practical examples of an IPCSocket include:

- Communication between two applications. For example, when you run your application in the debugger, Xojo and your application communicate using an IPCSocket.
- To create new tasks. If you have extremely complicated computations, you may want to have them run as their own console applications that are launched by your main application. These console applications can communicate with the main application using an IPCSocket.

Chapter 7

Concurrency

Timers and Threads can help make your applications more responsive.

CONTENTS

Timers

A timer is a class that runs code periodically on some sort of interval. They provide a great way to your application to be doing some processing while it might otherwise be idle, which can sometimes be a simple way to give the illusion of concurrency.

Timers are often used in conjunction with threads to provide UI updates for background processing.

Using a Timer

For desktop applications, use the Timer class to create a timer. Web applications use the WebTimer class. Both work similarly.

The code that you want to run periodically goes in the Action event. You specify the period using the Period property. It tells the timer how frequently the Action event is called.

The mode property is used in conjunction with this to specify if the timer is to run repeatedly (ModeMultiple), just once (ModeSingle) or is disabled (ModeOff).

The following code in a timer will increase a counter displayed in a Label:

```
Dim value As Integer
value = Val(CounterLabel.Text)
value = value + 1
CounterLabel.Text = Str(value)
```

Be sure that the timer is set to run repeatedly (Mode = ModeMultiple, or 2) and that the Period is set to a value like 1000 or so to cause it to update once per second.

Remember that the timer is not going to be called if your application is busy doing something else. So even though the above timer might be set to run every second, it might run much less often than that if your application happens to be doing any thing significant.

If you need more precise control of your processing, you will want to use threads.

Threads

Threads provide a way for you to run code in the background. They are particularly useful for long-running tasks that might otherwise make your application appear as if it is “frozen” because no user interface updates can occur.

Threads are cooperative, which means two things:

- They are relatively easy to use
- They only run on a single CPU core

***Note**: Pre-emptive threading, which has the ability to run your code on multiple CPU cores is very complex and not something that is supported by Xojo. If you need to run processes on multiple cores, you should consider using separate helper console applications in conjunction with IPCSockets for communication.*

Creating a Thread

To create a thread, first you need to add a Thread object to your project. You can do this by dragging a Thread from the Library onto a window, web page or to the Navigator.

The code that you want to run in the thread is placed in the Run event handler. Anything you access from the thread is considered part of the thread and also runs in the background.

To start a thread you call its Run method, which calls the code in the Run event handler.

Thread Priority

By default a thread has a priority of 5. This is the same priority as the main application thread, so if you leave your thread at 5 it will have the same amount of time allocated to it as the main thread.

For example, presume there are 100 “units” of thread time available. If both the main thread and your thread have a priority of 5 then the time unit split is calculated like this:

Total Priority = 5 (main) + 5 (your thread) = 10

Time Units (each thread) = (5/10) * 100 = 50

This means that the main thread runs 50 times and your thread runs 50 times.

But what if you want your thread to run more often because it is doing some heavy processing? In this case you would increase its priority. If you change your thread’s priority to 15 then the

time unit split is calculated the same, but results in more time units for your thread:

Total Priority = 5 (main) + 15 (your thread) = 20

Time Units (your thread) = (15/20) * 100 = 75

Time Units (main thread) = (5/20) * 100 = 25

This means your thread will get 75 of the 100 time units and the main thread will get only 25. So you thread is running 3 times more often than the main thread.

Thread Control

Threads can be slept, suspended, resumed and killed. When you sleep a thread, you specify the amount of time (in milliseconds) for the thread to sleep. It will automatically wake itself when the time has elapsed. If you suspend a thread, it stays suspended until you specifically resume it. Finally you can kill a thread, which terminates it.

Each of these actions changes the state of the thread. You can check the thread state any time using the State property. A thread can be Running (0), Waiting (1), Suspended (2), Sleeping (3) or NotRunning (4).

Communicating with the User Interface

Because of operating system restrictions, threads can not directly access or manipulate any user interface element. If you have a thread that need to update user interface in some way, such as updating a Progress Bar, you should instead use a Timer as the intermediary.

Rather than having your thread update a Progress Bar directly, have a Timer periodically get the progress value from the thread and then have the Timer update the Progress Bar.

Here is an example of the Run event handler of a thread that was added to a window. The thread simply loops through an array (with a pause in the middle). The position in the array is stored as a property of the window (ArrayPosition) as is the maximum value of the array (ArraySize):

```
Dim arrayValues() As Integer
arrayValues = Array(1, 2, 3, 4, 5, 6, 7, 9, 10)
ArraySize = arrayValues.Ubound
For i As Integer = 0 To ArraySize
  ArrayPosition = i
  App.SleepCurrentThread(1000) // Pause for 1 second
Next
```

A separate timer can check the value for ArrayPosition and ArraySize and use them to update the Progress Bar. This code is in the Action event handler of a Timer on the window:

```
If CountThread.State = Thread.NotRunning
Then
  Me.Enabled = False
  MsgBox("Finished!")
Else
  ThreadProgress.Value = ArrayPosition
  ThreadProgress.Maximum = ArraySize
End If
```

In the Action event handler of a button on the window, this code starts the thread and the timer:

```
If CountThread.State = Thread.NotRunning
Then
  CountThread.Run
  CountTimer.Period = 500
  CountTimer.Mode = Timer.ModeMultiple
  CountTimer.Enabled = True
End If
```

As the thread runs, it updates the property on the window. The thread periodically updates the Progress Bar on the window with the value of the property.

This two-step process prevents the thread from directly updating the user interface.

Using the Task class

Included with Xojo is an example of a Task class that can be used to do this as well (Desktop/UpdatingUIFromThread/UIThreadingWithTask). Task is a Thread subclass that has an UpdateUI event handler and method. Use it in place of a Thread when you want your thread to be able to update the user interface. In the Task's Run event handler, you call the UpdateUI method when you want to update any UI. Then in the UpdateUI event handler, you can have code that directly accesses the UI.

When you call UpdateUI, you can pass either a list of values (using a series of Pairs) or you can pass a dictionary of values.

Regardless, in the UpdateUI event handler, you get a dictionary of the values. You can then check the values in the dictionary to determine what to update in the UI.

Multiprocessing

There are times when you may want to take advantage of multiple CPU cores, something that you cannot do with Threads. With a multi-core CPU, your computer can literally do multiple things at one time, which is called multiprocessing. With a little careful planning, your Xojo apps can use multiprocessing for significant performance improvements.

You may be thinking, isn't this what threads are for? Sometimes, although not with Xojo. As noted in the previous section, Threads in Xojo use co-operative threading, which is simpler and more efficient but means that all the threads run on a single CPU core.

Console Apps

With Xojo, separate processes using console apps is the way to take advantage of multiple CPU cores.

The technique is relatively simple. You create a console app that does the processing you need and returns a result. You then have your main (GUI) app start one or more of these console apps and supply them with the data they need to process. Each console app goes off on its own (in its own memory space and using its own CPU core -- if one is available) to do the processing. When one finishes, it returns the result, which your app can then use.

Communication Techniques

There are two common ways for your main app to communicate with the console apps: Shell or IPCSocket.

Using a Shell is simplest. You create a Shell instance for each console app and launch the console app from the Shell. You can get the return result using the Shell methods and properties.

For a specific example of this technique, refer to the example included with Xojo:

- Examples/Console/Multiprocessing/WordCounter
- Examples/Console/Multiprocessing/WordCounterGUI

An IPCSocket is more useful when the console apps need to continuously communicate with the main app, but it is also a much more advanced technique. IPCSockets are discussed in Chapter 6: Devices and Networking.

Chapter 8

Debugging

This chapter covers techniques to help you create quality applications. You'll learn how to use the debugger, handle exceptions, remote debug your cross-platform applications and use the profiler to improve the performance of your code.

CONTENTS

About 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, the Debugger makes it easy to track down those nasty bugs and squash them like a, well, bug.

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. The debugger can help you find these types of bugs 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 keyword, 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.

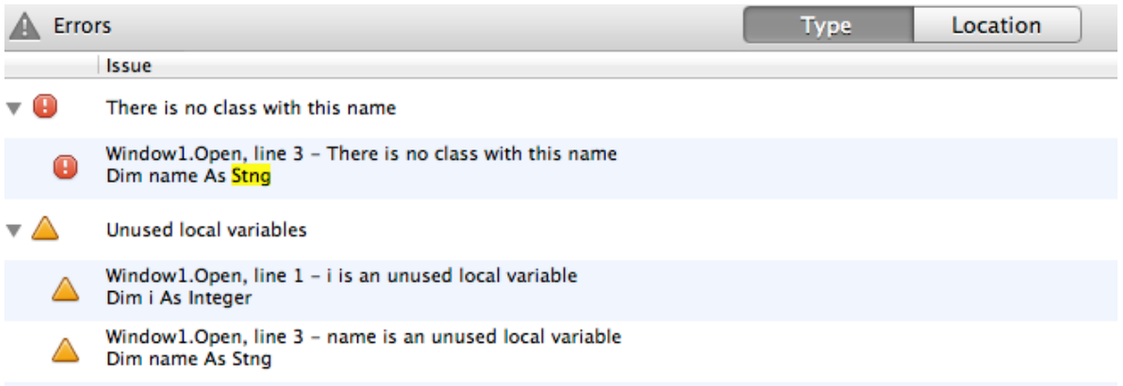
Analyzing the Project

Since a project cannot be compiled if it contains syntax errors, you have 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 issues but does not build the project. Some issues that it identifies are syntax errors, unused local variables and parameters, and type conversion issues.

Errors are indicated by a stop sign icon and will prevent you from being able to run or build your project. Warnings are indicated

by a yellow warning triangle and do not prevent you from building or running your project.

Figure 8.1 Analyze Project Results



If errors or warnings are found they are listed in the Errors panel at the bottom of the Workspace. Expand each row that is displayed to see the individual issues.

You can click on each issue and you will display the appropriate editor (usually the Code Editor) with the issue highlighted.

You should fix any errors that are reported. And you should evaluate the warnings to see if you think they need to be fixed. Not all warnings need to be fixed.

The Type and Location buttons in the Issues pane allow you to change how to view the Issues. The Type button groups the issues by the type of the issue. So all the "Unused local variables" issues would appear together.

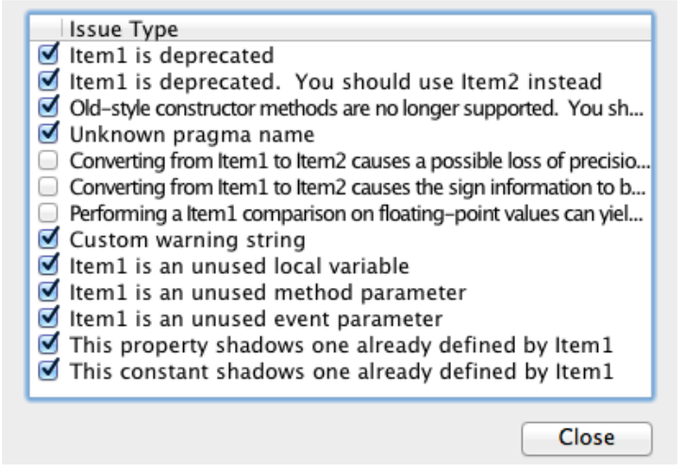
The Location button groups the issues by the object in which they occur. So all the issues for Window1 would appear together, for example.

Analyze Item

The Analyze Item command, available from the Project menu or on the Code Editor toolbar, works the same as Analyze Project but instead of analyzing the entire project, it only analyzes the code in the Code Editor.

Filtering Types of Warnings

Figure 8.2 Specifying Warnings to Display



You can control the types of warnings that the Errors panel displays. Choose Warnings from the Project menu to display a dialog box that lists all the types of warnings that can be found. Only the selected warnings are reported when you analyze. You can unselect any warnings that you do not wish to know about.

Using the Debugger

When you run your project from within Xojo, you are using the debugger. If there are no errors after compiling and building your application, it launches and the Workspace switches to the debugger tab. This is referred to as running the project in "debug mode".

While you are using your application you will not normally see the debugger, but there are several ways it can be activated.

Activating the Debugger

Manual Activation

You can manually activate the debugger by clicking the Pause button on the Debugger toolbar while your application is running. If your code is currently running, then this displays the method that is running and highlights the next line of code that will execute.

If your application was idle, then this drops you into the Event Loop where you can view global objects and their variables (such as App, Runtime and modules).

Application Error

Should your application raise an unhandled exception while you are running in debug mode, the debugger becomes active. The source code where the error occurred is displayed at the line that caused the error.

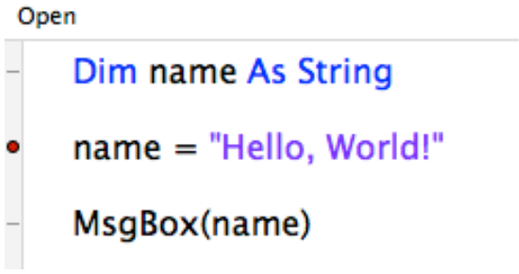
From here you are able to view variable values and review the call stack (see below).

Breakpoint

In your code you can set breakpoints. A breakpoint is an indicator that tells the debugger to activate itself when the line of code is reached.

For example, you might want to set a breakpoint at the start of a method if you want to carefully review the code as it executes.

Figure 8.3 Setting a Breakpoint



To set a breakpoint, you click on the "dashes" that appear in the gutter of the Code Editor. Each

dash indicates a line of code that can have a breakpoint set. You can also set a breakpoint using the Project → Breakpoint → Turn On menu (⌘+\ on OS X, Ctrl-\ on Windows and Linux). The same command turns off a previously set breakpoint on the line.

If you want to turn off all breakpoints throughout your project, use the Project → Breakpoint → Clear All menu.

To see all breakpoints in your project, use Project → Breakpoint → Show All menu to display the breakpoints in the Find panel.

Conditional Breakpoint

There will be times where you may want to stop at a breakpoint, but only if a specific condition occurs. For example, you could be in a long loop and you only want to stop at the 75th element.

You set up a condition breakpoint in your source code using a combination of a conditional If statement and the Break command. This example will stop at the breakpoint when the loop counter reaches 75:

```
For i As Integer = 1 To 100
  If i = 75 Then Break
Next
```

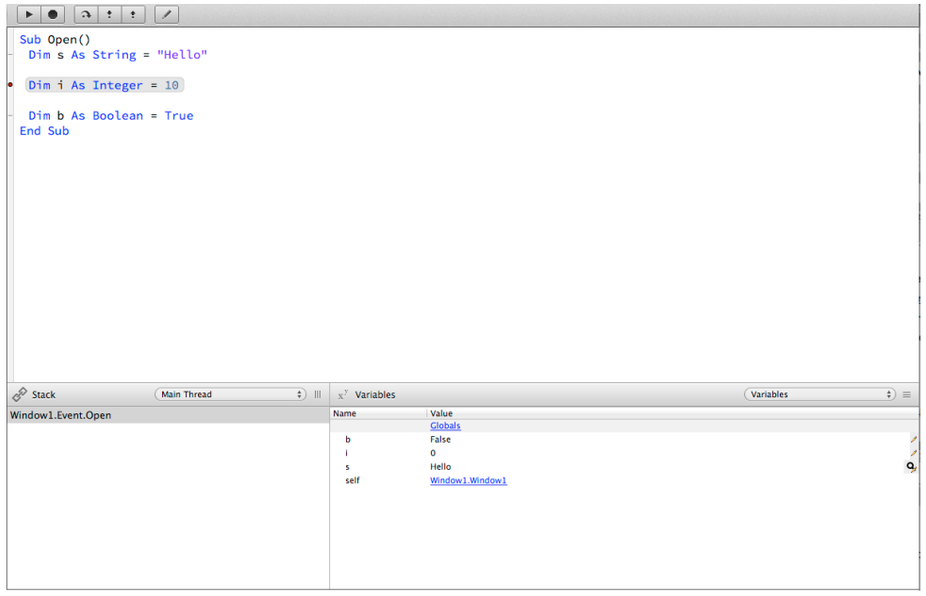
The Break command is called when *i* reaches 75 and activates the debugger.

***Note**: The Break command does not do anything in a built application. It is only useful when running your app in Debug Mode.*

The Debugger Screen

Once you are in the debugger, you can control it using the toolbar commands: Pause/Resume, Stop, Step, Step In, Step Out and Edit Code.

Figure 8.4 The Debugger



- **Pause/Resume**: Use Pause to pause a running application and activate the debugger. If you are in the debugger, the Resume button tells your application to continue running where it left off. You can also click the Run button on the main toolbar to Resume, select Project → Resume from the menu or you can use the ⌘+R (Ctrl-R on Windows and Linux) shortcut.

- **Stop**: The Stop button immediately stops the running application. The application is quit immediately and no further code is run. You can also use the shortcut Shift+⌘+R (Shift-Ctrl-R on Windows and Linux)

- **Step**: The Step button is used to run the code one line at a time. Each time you click Step, the highlighted code is executed and you remain in the debugger. If you click Step while on a method call, the method is called and you move to the next line of code.
  You will mostly use Step while debugging. In addition to clicking the button, you can use Project ↠ Step ↠ Step Over menu command or the shortcut Shift+⌘+O (Shift-Ctrl-O on Windows and Linux).

- **Step In**: The Step In button works like the Step button except when you reach a method call. Instead of calling the method and moving to the next line of code, Step In moves you to the first line of code in the method.
  In addition to clicking the button, you can use Project ↠ Step ↠ Step Into menu command or the shortcut Shift+⌘+I (Shift-Ctrl-I on Windows and Linux).

- **Step Out**: If you are in a method, clicking Step Out, runs the rest of the code in the method and then stops when the method returns.
  In addition to clicking the button, you can use Project ↠ Step ↠ Step Into menu command or the shortcut Shift+⌘+T (Shift-Ctrl-T on Windows and Linux).

- **Edit Code**: The Edit Code button allows you to jump to the Code Editor for the current method that is in the debugger. Here you can edit the code (which you can not do in the code display of the debugger). However, changes that you make to code in the Code Editor are not reflected in your application until the next time you run.

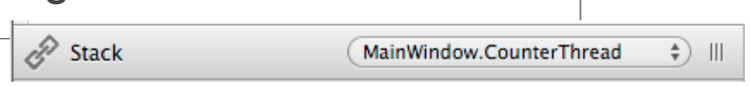
Watching Variables

The watch pane of the debugger allows you to view the values of variables. By default it shows you the currently declared variables for the method that is executing.

You can change the values of booleans, integers, doubles and strings by clicking on the value (or selecting the pencil icon on the far right), entering a new value and pressing Return.

Figure 8.5 Viewing Variables in the Debugger

Figure 8.6 The Thread Selector



***Note**: For booleans, anything entered*

besides True or true is treated as False.

With strings, the pencil icon on the right changes to show a magnifying glass. Clicking this displays the string editor which lets you view the string as text or as binary. It also allows you to make changes to the string using a larger editing area.

For integers, you can change the display format by right-clicking on the value and selecting View As Decimal, Hex, Binary or Octal.

For doubles, you can change the display format by right-clicking on the value and selecting View As Scientific, Decimal or Rounded.

If the variable type is a class, then you can click on the type to display the property values.

At the top of the variables pane is a Popup Menu that allows you to navigate the variable hierarchy. You can use it to quickly jump back to the method variables after having displayed the string editor or class instance details.

On some objects (such as Windows and RecordSets), there is a special link at the top called “Contents”. Clicking this displays information about the object, such as the controls on a window or the fields in a RecordSet.

Viewing Source Code

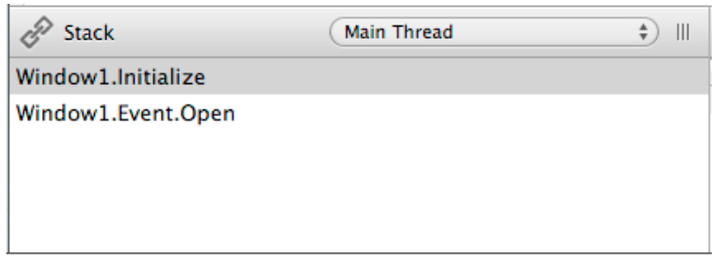
The source code for the current method (or event) displays in the debugger code viewer. The next line to execute is highlighted. In the code viewer, you can set additional breakpoints and watch the highlight move as you step through code.

If you want to edit the code, click the Edit Code button on the toolbar. Changes made to code do not take effect until the next time you run your project.

Viewing the Call Stack

The call stack displays the calling hierarchy of your methods. If your main window Open event calls a method named Initialize then the call stack shows Initialize at the top (since it is the current method) with Window1.Open underneath it.

Figure 8.7 Viewing the Call Stack in the Debugger



The call stack is very helpful for viewing your code path and for figuring out the methods that call other methods.

Viewing Threads

Each thread in your application is tracked separately in the debugger.

Use the selector in the Stack section to see and show the threads that are currently running.

Exception Handling

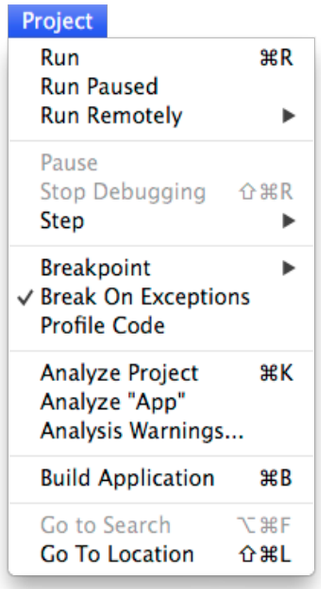
The debugger can help you verify that your code is working as you expect and it can help you find errors. Once you've found the source of errors, you want to make sure you handle them properly.

Exceptions are a type of error that occur when something unexpected happens. These errors will crash your application if you do not handle them in some way. The act of causing an exception to occur is called raising an exception.

All exception are subclasses of the RunTimeException class.

When an exception is encountered in your code, you can choose to have the Debugger displayed at the line causing the exception. To do this, select Project ↠ Break On Exceptions in the menu so that it has a checkmark next to it.

Figure 8.8 Break On Exceptions in Project Menu



NilObject Exceptions

The most common type of exception that occurs is a NilObjectException. These errors occur when you attempt to use an object but don't have an instance of it.

The simplest example of this is forgetting to use New to get an instance before you try to use a property or call a method:

```
Dim d As Date
d.Month = 8 // NilObjectException
```

Another place to check for this is in the return value of methods that return an instance. Some methods return Nil in certain situations, such as GetOpenFolderItem which returns Nil if the user clicks Cancel:

```
Dim f As FolderItem
f = GetOpenFolderItem("")
If f.Exists Then
  // Cancel causes NilObjectException
  // Do something
End If
```

To prevent these types of errors, you should always check if the value is Nil before you attempt to access it:

```
Dim f As FolderItem
f = GetOpenFolderItem("")
If f <> Nil Then
  If f.Exists Then
    // Do something
  End If
End If
```

Try Catch

Sometimes you may find that an exception is a normal part of processing. What you want to do in these situations is catch the exception so that you can deal with it appropriately. The Try...Catch command is used for this purpose.

For example, loading an XML file can raise an XMLException if the file is not valid XML. You can check for this by using a Try..Catch block:

```
Dim xmlFile As FolderItem
xmlFile = GetOpenFolderItem("")
If xmlFile <> Nil Then
  Try
    Dim xml As New XmlDocument
    xml.LoadXml(xmlFile)
  Catch e As XmlException
    MsgBox("Not an XML file!")
    Return
  End Try
End If
// Process the XML
```

If no XMLException is raised then the code in the Try section runs. If an XMLException is raised, then the code in the Catch section is run.

Exception

The Exception command is a simplified version of Try..Catch. Rather than focusing on a specific block of code, Exception catches errors for the entire method.

The Exception command goes at the end of the method and is called if an exception is generated anywhere in the method. The preceding example could be written like this using the Exception command:

```
Dim xmlFile As FolderItem
xmlFile = GetOpenFolderItem("")
If xmlFile <> Nil Then
  Dim xml As New XmlDocument
  xml.LoadXml(xmlFile)
End If
// Process the XML

Exception e As XmlException
  MsgBox("Not an XML file!")
  Return
```

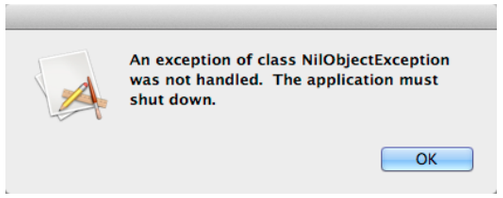
Although this is simpler, it is not as obvious what is occurring and it is also far less flexible.

App.UnhandledException

There is a special event handler in the App object of your project that is called if an unhandled exception is not caught.

The event has one parameter, *error*, which is the exception itself. You can put code in this event handler to display a message to the user, capture log information or anything else you want. Return True to hide the default unhandled exception error dialog and attempt to allow you application to continue (which is usually not recommended).

Figure 8.9 Unhandled Exception Dialog



This code displays a friendlier message to the user and then quits the application:

```
Dim msg As String = "An error occurred.
Please notify the author."
MsgBox(msg)

Quit
Return True
```

You can also display the runtime stack so it can be used to help pinpoint the location of the problem so you can fix it:

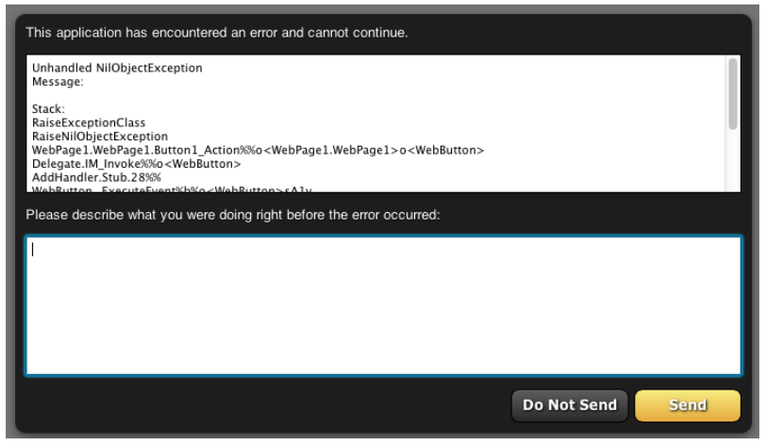
```
Dim msg As String = "An error occurred.
Please notify the author.  Stack: "
MsgBox(msg + Join(error.Stack,
EndOfLine))

Quit
Return True
```

WebApplication.UnhandledException

Web apps also have a special event handler in the App object of your project that is called if an unhandled exception is not caught.

The event has one parameter, *error*, which is the exception itself.

Figure 8.10 Web Application Unhandled Exception Dialog



The default error dialog displays information about the error and provides a field for users to add additional information. This information is written to errors.log alongside the web application. You can also get the information in the WebSession.JavaScriptError event handler so that you can log it yourself.

Remember, if this event handler has been called and you return False (the default) it means your entire web application is going to stop running.

Return True to hide the default error dialog and prevent the web app from quitting.

Creating Your Own Exceptions

Since RuntimeException is a class like any other, you can subclass it to create your own exceptions. This allows you to raise your own exceptions that you have defined.

If you create a RuntimeException subclass called *InvalidXMLFileFormatException* then you can raise it using this syntax:

```
If incorrectXmlFileFormat Then
  Raise New InvalidXMLFileFormatExcep-
tion
```

Remote Debugging

When you run your project it runs on the same platform you are using. So if you develop on OS X, then clicking Run will run your project on OS X.

Since Xojo is a cross-platform development tool, you are likely going to want to also run your projects on other platforms for debugging purposes. You can do this using the Remote Debugger, which has two version: Desktop and Console.

The desktop version is used to remotely debug desktop applications. The console version is used to remotely debug console and web applications.

Remote Debugger

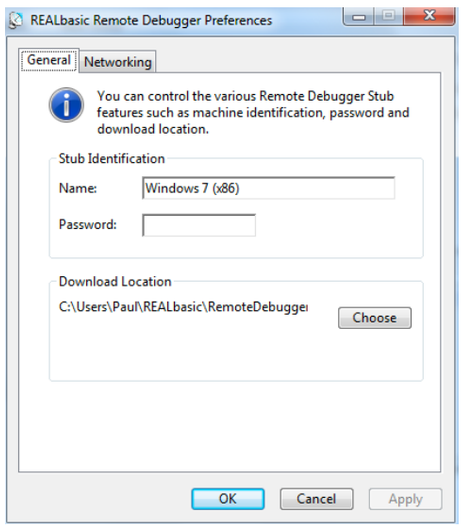
The Remote Debugger is a small program that lets you run your project on a different target platform. Xojo communicates with it so that when you run your project, it sends it to the target platform rather than running it locally. Normally when you run your project from within Xojo (Project → Run), it runs the debug build locally. If you have the Remote Debugger configured, you can instead have Xojo send and run your debug build on a remote computer, which is incredibly useful for testing cross-platform applications. This remote build still communicates with Xojo, so you can access the debugger and test just as if you were running it locally.

Remote Debugger Desktop Configuration

On the remote machine, you need to run the Remote Debugger Desktop. The Remote Debugger is included with your Xojo installation. There is a separate version for each platform (Windows, OS X, Linux). Copy the version you need to the platform you are using.

The Remote Debugger Desktop has an Options window that lets you configure some settings. In the General tab, you want to give the machine a name and select the Download location. You typically do not need to change the settings in the Networking tab.

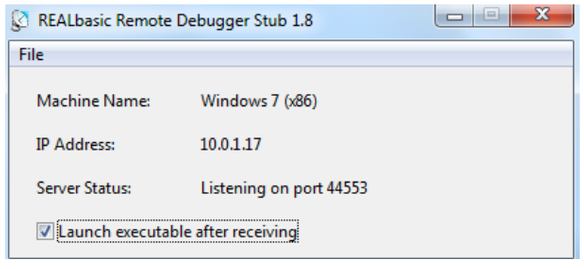
Figure 8.11 Remote Debugger Stub Preferences



After you have set the Options, click OK and leave the Remote Debugger Desktop running.

If you are using a firewall on the remote machine, you need to make sure that port 44553 is open for UDP and TCP connections.

Figure 8.12 Remote Debugger Stub



Remote Debugger Console

The Console version can be used to remote debug console applications, standalone web applications and CGI web applications on remote machines that do not have a desktop interface. Currently it can only be used across a local network (or VPN).

If you are trying to debug a CGI app, set the Remote Debugger to NOT launch automatically and set the path to point to where your CGI apps must be installed in order for them to work with the web server. The web server will launch the app when you visit the appropriate URL in your browser.

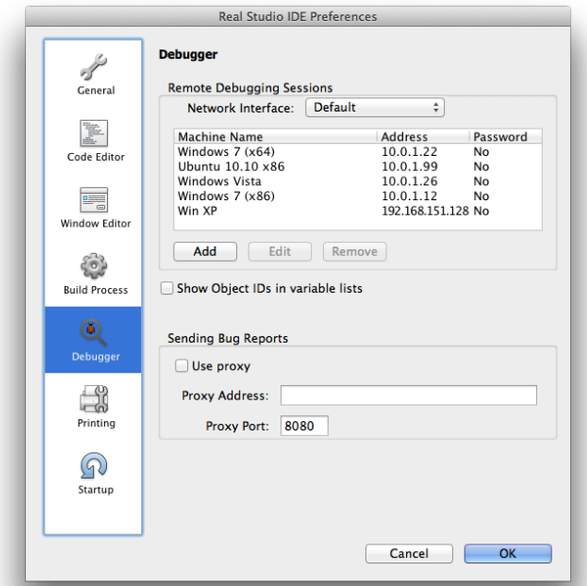
The Console Remote Debugger runs from Terminal or the command line. The first time you launch it, you are prompted for the settings:

- Machine Name
- Download Directory
- IP Address: Specifies the IP address to listen on (must match an IP address available on the computer).
- Maximum Connections: Sets the maximum number of connections.
- Auto Launch: Have the remote debugger automatically launch the app being debugged.
- Public: Indicates if the remote debugger is publicly visible.
- Password: Specifies a connection password.

These settings are saved in the RDS.config file in the same folder as the Console Remote Debugger.

You can also provide these an other options via the command line. Use the “--help” argument to get a list of available command line options.

Figure 8.13 Debugger Preferences

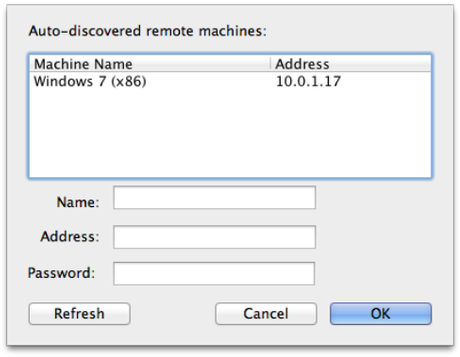


Development Machine Configuration

On the development machine, you need to

configure Xojo so that it can see the Remote Debugger. In the Preferences, select Debugger. There you'll see a list of configured remote machines.

Figure 8.14 Adding a Remote Machine



Click Add to add your remote machine. Depending on your network configuration, the remote machine may appear as an "auto-discovered remote machine". If it does, you can just click its name and then OK to add it. If it does not appear, you can enter the IP address (specified on the Remote Debugger Stub on the remote machine) and give it a name.

If you are using a firewall on your development machine, you need to make sure that port 44553 is open for UDP and TCP connections and port 13897 is open for TCP connections.

***Note**: Virtual machine software such as VMware Fusion, Parallels Desktop, VMware Desktop and Microsoft Virtual PC all work great with remote debugging.*

Remote Debugging

Now you can try running a project remotely. On the development machine, create a new desktop project.

To run the project remotely, instead of selecting Project ↠ Run (or clicking the Run button on the toolbar), choose Project ↠ Run Remotely and click your remote machine name. Your project is compiled and linked as usual, but you will now see an additional step where this debug build is sent over to the Remote Debugger Stub on the remote machine.

When the Remote Debugger Stub has received the debug build, it runs it. Interact with it on the remote machine. Any breakpoints you have set will jump to the debugger on the development machine.

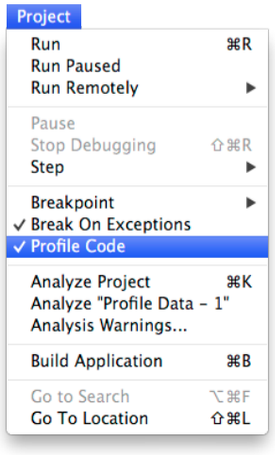
Profiling for Performance

The Profiler is used to track the length of time each method in your application takes to run. This information allows you to focus on performance optimizing the parts of your application that might be considered slow.

Using the Profiler

You can enable or disable the Profiler using the Project ↠ Profile Code menu. A check mark appears next to the menu when profiling is enabled.

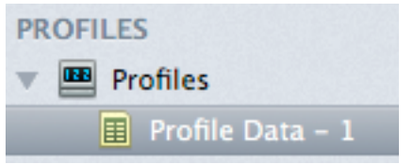
Figure 8.15 Enabling the Profiler



Now you run and use your project as you normally would. The profiler silently gathers information about the methods that are called and how long each one takes to execute. Be aware that your application runs slower than usual when profiling is enabled due to the overhead of tracking and timing everything to get the profiler data.

When you quit your application, the Profiles section appears in the Navigator with an entry for the profile data.

Figure 8.17 Profiles Section in Navigator



This section is a list of all the methods that were called while you were using your application. It shows the number of times each method was called and how much time was spent in the method. You can save the profiler data to a text file using the contextual menu. Simply right-click anywhere in the summary and choose “Save As...”.

You can expand methods to see the other methods that they called.

Figure 8.16 Profiler Data

Name	Called	Total (milliseconds)	Average (milliseconds)
▼ Main Thread App.Open	1	185.0000	185.0000
BasePushButton.Open	2	0.0000	0.0000
▶ CustomerDetailsWindow.CustomerList.Open	1	57.0000	57.0000
OrdersDatabase.SetupNewDatabase	1	2.0000	2.0000
SelectablePopupMenu.Open	1	0.0000	0.0000
▼ Main Thread CustomerDetailsWindow.CustomerList....	1	0.0000	0.0000
CustomerDetailsWindow.EnableFields	1	0.0000	0.0000
▶ CustomerDetailsWindow.LoadCustomerFields	1	9.0000	9.0000
▶ CustomerDetailsWindow.LoadInvoices	1	1.0000	1.0000

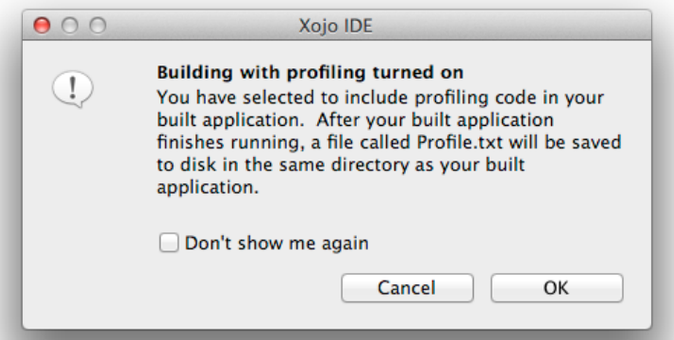
As you are optimizing your application, you can compare the current Profiler Data with previous Profiler Data to see if your changes are improving performance.

***Note**: Profiler data is not saved along with your project. use the Save As function described earlier if you wish to retain the profile data.*

***Note**: Profiler data is only collected if you application quits normally. If it quits because of a crash (or you press the Stop button in the debugger), no profiler data is collected. For web apps you can simply close all the browser windows/tabs and wait for the IDE to detect that the app has terminated or you can add a button or action that calls the Quit method.*

***Note**: Profiler data cannot be collected for OS X apps that have been sandboxed.*

Figure 8.18 Building with Profiler Dialog



Using the Profiler with Built Applications

You can choose to create a standalone build of your application with embedded profiling code that you can send to your users. To do this, just build your application with Profile Code selected in the settings. A dialog will appear to remind you that profiling code will be embedded in the application.

With this profiling code embedded, your application will log profiling data as it is being used. When it quits, the profile data is saved to a file called Profile.txt in the same folder as the application.

Using the Profiler with the Remote Debugger

The Profiler does not work with the Remote Debugger. If you need to use the Profiler to gather information about your app running on another platform, just install Xojo on the platform, copy your project over to it and run it from there with profiling enabled.

Chapter 9

Building Your Applications

This chapter covers strategies for building your desktop and web applications and shows you how you can automate your builds.

CONTENTS

Compilation Constants

Before you build your application, you choose the platform or platforms on which it will run. You build applications for Windows, Linux, and OS X as desktop, console or web. While programming, you can selectively enable or disable segments of your code that are valid only for particular platforms by using compilation constants.

Compilation Constants

The compilation constants tell you information about the platform on which the application is running. When used with conditional compilation you can specify blocks of source code to include or exclude for specific platforms.

These are the compilation constants:

- DebugBuild: True when the app is running in Debug mode
- TargetWin32: True when the app is running on Windows. Because Xojo creates 32-bit apps, this is True regardless of whether Windows itself is 32-bit or 64-bit.
- TargetMacOS, TargetCocoa, TargetCarbon: True for the various OS X targets.
- TargetLinux: True when the app is running on Linux. Because Xojo creates 32-bit apps, this is True regardless of whether Linux itself is 32-bit or 64-bit.
- TargetX86: Currently this is always True.
- TargetHasGUI: True for desktop and web applications. False for console applications.
- TargetWeb: True for web applications.

These constants are automatically set to either True or False depending on the platform being compiled. You can use conditional compilation commands to use specific code like this:

```
// Saving Preferences
#If TargetWin32 Then
  // Use Registry
#ElseIf TargetMacOS Then
  // Use plist
#ElseIf TargetLinux Then
  // Use XML
#EndIf
```

Versions

You can also check the version of Xojo being used with the XojoVersion and XojoVersionString constants.

These two constants can be used to block out code that is not compatible with older (or newer versions) of Xojo.

***Note**: You may also need to use the older RBVersion and RBVersionString to block out code that could be used with older versions.*

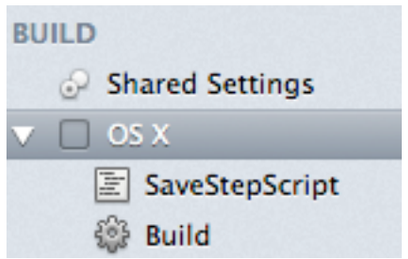
Build Automation

The Build Automation feature is used to automatically run specific tasks when you run or build your project. These tasks are called Build Steps. There are three build steps available: Copy Files, Script and External Script.

To add a Build Step to your project, select the Insert menu using the toolbar of main menu and then select the Build Step submenu.

Build Steps added to the CONTENTS area of the Navigator are inactive by default.

To activate a Build step, you need to move it to a build target. You do this by dragging the Build Step from the CONTENTS area to the BUILD area of the Navigator. Drop the Build Step onto one of the targets (OS X, Windows or Linux) and the Build Step becomes active when running or building for that target.



Once you add a Build Step to the target, the target can be expanded to see the build step. When you expand it you will see the build step and an item simply called Build. The Build item allows you to specify whether the Build Step occurs before the project is built or after. Drag the Build Step before the Build item to have it be processed before the Build is done and drag it after the Build item to have it be built after the Build is done.

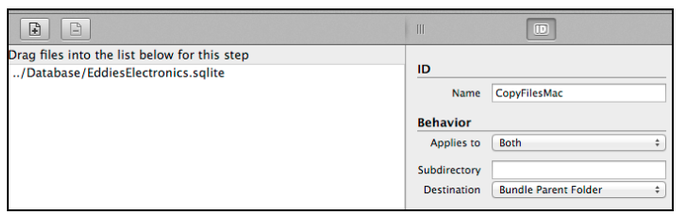
Each Build Step has an “Applies to” setting that allows you to specify: Both, Debug or Release.

Debug means the Build Step is processed only when doing a Debug Run. Release means the Build Step is processed only when doing a standalone release Build. And Both means the Build Step is always processed.

Copy Files

The Copy Files step allows you to copy a file, files or a folder to a specified location during the build process. For each of these locations you can also specify a subfolder for the files. The locations are:

Figure 9.1 Copy Files Step Editor



App Parent Folder, Resources Folder, Frameworks Folder, Bundle Folder Parent and Contents Folder.

App Parent Folder: On Windows and Linux, the specified files are copied alongside the application executable. On OS X, the files are copied next to the executable in the Application Bundle (Contents->Mac OS).

Resources Folder: The specified files are copied to a Resources folder. On Windows and Linux the Resources folder is created alongside the application itself. On OS X, the Resources folder is contained within the Application Bundle (Contents->Resources).

Frameworks Folder: On Windows and Linux, the files are copied to the Libs folder for the application. On OS X, the files are copied to the Frameworks folder in the Application Bundle (Contents->Frameworks).

Bundle Parent Folder: On Windows and Linux, the files are copied to the same folder containing the executable (same as App Parent Folder). On OS X, the files are copied to the folder containing the Application Bundle itself.

Contents Folder: On Windows and Linux, the file are copied to the same folder containing the executable (same as App Parent Folder). On OS X, the files are copied to the Application Bundle (Contents).

Script and External Script

Scripting is a powerful way to automated your application builds. Scripts can contain a wide variety of commands to control the build process. All the commands are listed in the Language Reference, but some useful ones include: DoCommand and DoShellCommand and Speak.

DoCommand is used to run built-in IDE commands, perhaps the most useful is to save your project before it is run:

```
DoCommand("SaveFile")
```

To add such a script to your build process, add a Script Build Step using the Insert menu or Insert button.

The script is added to the Navigator and the Script Editor appears. In the Script Editor, enter the code:

```
DoCommand("SaveFile")
```

Now you can drag the script to a target in the BUILD section of the Navigator so that it runs. If you want the script to run before the build is started, then drag it before the “Build” item. If you want it to run after, then drag it after the “Build” item.

The Speak command uses the built-in voice to speak the supplied text. You can use this to let you know when a task has finished.

```
Speak("Build Complete.")
```

DoShellCommand allows you to run a shell command. You can run any shell command that is available in the Terminal or Command Shell of the operating system. A shell command might be used for more robust file management, to manipulate permissions, run commands to digitally code sign your application or anything else that you want.

This example runs the codesign command on OS X:

```
DoShellCommand("codesign MyApplication.app")
```

An external script works the same except the script is stored in an external text file that you select. You would use an External Script if you have a script that you share across multiple projects.

IDE Scripting

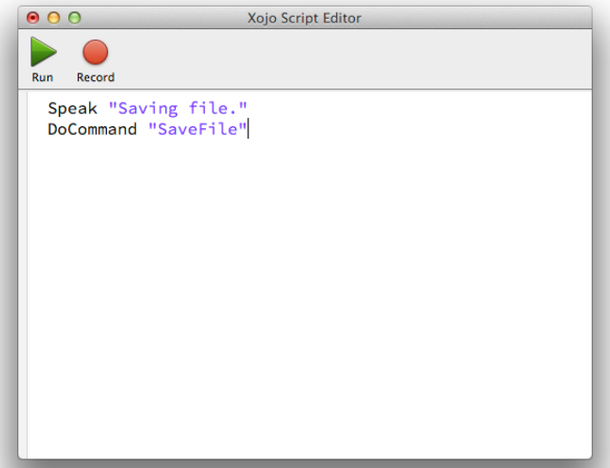
IDE Scripting is the ability to create scripts that allow you to manipulate the Xojo user interface (IDE).

Creating an IDE Script

To create an IDE script, select New IDE Script from the File menu. This opens the XojoScript Editor. You can open multiple IDE Script editors, each with their own scripts.

In this editor you write code using the XojoScript scripting language, which consists of Xojo programming language commands (such as If..Then or For..Next). You can also use special IDE Scripting commands such as DoCommand, DoShellCommand and more.

Figure 9.2 XojoScript Editor



All the IDE Scripting commands are listed in the next section.

This simple script saves the current project:

```
Speak("Saving file.")
DoCommand("SaveFile")
```

To run the script, click the Run button on the Xojo Script Editor toolbar.

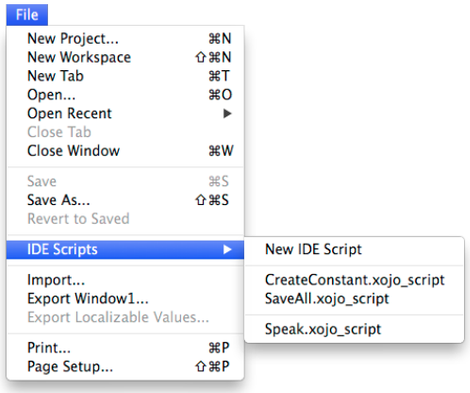
Saving and Loading IDE Scripts

IDE Scripts are saved as text files and are not included within the project.

To save a script, use File ↠ Save or Save As.

To load an existing script into the Script Editor, use File ↠ Open.

Figure 9.3 Saved Scripts in the IDE Scripts menu



You can also add scripts to the IDE Scripts submenu. Clicking scripts in this menu immediately runs them; they do not open in the Script Editor. To have a script appear in this submenu, add the saved script to the Scripts folder alongside the Xojo app or add them to a Scripts folder you create alongside your project file.

Recording a Script

On the XojoScript Editor toolbar is a Record button. When pressed, any commands in the IDE that have a corresponding script command are recorded when they are used.

To try this, click the Record button (the text changes to Stop Recording) and then select File → Open to open a project.

You'll see the the command DoCommand "OpenFile" appear in the XojoScript Editor.

Click Run to run your project (not the script) and you'll see command appear in the editor:

```
DoCommand "RunApp"
```

As you use the IDE, any actions that have corresponding script commands will appear in the XojoScript Editor as they are recorded. This is a great way to determine the commands you can use to automate something.

Controlling Xojo from the Command Line

Although Xojo cannot be run from the command line, it can be controlled using a separate command-line app that communicates with Xojo using IPC sockets and the "XojoIDE" path.

The IDECommunicator example project (Examples/Advanced/IDE Scripting/IDE Communication) is a fully functional app that can be used control Xojo from the command line by sending IDE Scripts to Xojo for it to run.

Xojo must already be running in order to the communication to work.

To use it, build the app for you platform and then run it from the command line with the "-h" option to see how it works. This command runs it on OS X and Linux (# indicates your shell/terminal prompt):

```
# ./IDECommunicator -h
```

IDE Scripting Commands

When creating IDE Scripts, in addition to using XojoScript commands, there are several additional commands you can use to control the IDE and modify the current project. The commands are grouped into these categories:

- Notification
- Building
- Project
- Project Items
- String
- Input/Output
- Miscellaneous

Notification

The Notification commands are used to get the user's attention by displaying a message or making a sound.

Beep

The Beep command plays the system beep sound.

Syntax
Beep

```
Beep
```

Print

Displays the specified text in a simple dialog box with an OK button.

Syntax
Print(text As String)

```
Print("Hello")
```

Speak

The Speak command speaks the specified text.

Syntax
Speak(text As String)

```
Speak("Hello")
```

ShowDialog

The ShowDialog command displays a dialog box on the screen. You can specify the message and determine the buttons and their text.

Syntax
ShowDialog(Message As String, Explanation As String, DefaultButtonCaption As String, CancelButtonCaption As String, AltButtonCaption As String, Icon As Integer) As String

If you pass in an empty string to CancelButtonCaption or AltButtonCaption, then the button does not appear. Returns a String containing the caption of the button that was pressed.

Values for Icon are:

-1: No icon

0: Note icon (App Icon on OS X)

1: Caution icon (On OS X, App Icon is superimposed)

2: Stop Icon (App Icon on OS X)

3: Question Icon (App Icon on OS X)

```
Dim result As String
result = ShowDialog("Hello!", _
  "Is it me you're looking for?", _
  "Yes", "No", "", 1)
```

ShowURL

Displays the specified URL in the default web browser.

```
ShowURL("http://www.xojo.com")
```

Building

BuildApp

Starts building the current project.

Syntax
BuildApp(buildType As Integer[, reveal As Boolean]) As String

The buildType can be one of these values:

Value	Built Target	32/64 Bit	Architecture
3	Windows	32-bit	Intel
4	Linux	32-bit	Intel
6	OS X console	32-bit	Intel
7	OS X GUI	32-bit	Intel
11	iOS Device	32-bit	Intel
12	Xojo Cloud	32-bit	Intel
14	iOS Device	64-bit	Intel
15	iOS Device Universal	32 and 64-bit	ARM
16	OS X (all)	64-bit	Intel
17	Linux	64-bit	Intel
18	Linux	32-bit	ARM
19	Windows	64-bit	Intel

If reveal is True, the built app is displayed using the OS file manager.

Returns a String containing the Shell path of the built app. BuildApp cannot be used in an IDE Script that is called by Build Automation.

```
Dim appPath As String
appPath = BuildApp(7) // Cocoa build
Print("Built: " + appPath)
```

BuildLanguage

Used to get or set the build language. This is the “Language” property on the Shared Build Settings.

Syntax
BuildLanguage = newStringValue

newStringValue = BuildLanguage

```
If BuildLanguage = "Default" Then
  BuildLanguage = "English"
End If
```

BuildLinux

Gets or sets whether to build the project for Linux.

Syntax
BuildLinux = newBooleanValue

newBooleanValue = BuildLinux

```
BuildLinux = True
DoCommand("BuildApp")
```

BuildMacCocoa, BuildCurrentPlatform, BuildMacMachOx86, BuildRegion, BuildWin32

Gets or sets whether to build a specific target.

BuildWebProtocol

Gets or sets the type of web app to build.

Syntax
integerValue = BuildWebProtocol

BuildWebProtocol = integerValue

The values can be:

0 = CGI

1 = Standalone

```
BuildWebProtocol = 0 // CGI
```

BuildWebPort

Gets or sets the web port for built apps.

Syntax
integerValue = BuildWebPort

BuildWebPort = integerValue

Use a value of -1 to choose the port automatically.

```
BuildWebPort = 8080
```

BuildWebDebugPort

Gets or sets the web port for running debug apps.

Syntax
integerValue = BuildWebDebugPort

BuildWebDebugPort = integerValue

```
BuildWebDebugPort = 8181
```

CurrentBuildAppName

Returns the name of the app that was built. This can only be used in a Build Automation IDE Script that runs after the Build Step.

Syntax
stringValue = CurrentBuildAppName

```
Print(CurrentBuildAppName)
```

CurrentBuildLocation

Returns the shell path to the app that was built. This can only be used in a Build Automation IDE Script that runs after the Build Step.

Syntax
stringValue = CurrentBuildLocation

```
Print(CurrentBuildLocation)
```

CurrentBuildTarget

Returns an integer that specifies the type of app that was built. This can only be used in a Build Automation IDE Script that runs after the Build Step.

Syntax
integerValue = CurrentBuildTarget

The return value is one of these values:

3 = Win32

4 = Linux

6 = OS X Carbon

7 = OS X Cocoa

```
Print(Str(CurrentBuildTarget))
```

Project

Location

Used to get or set a location in the project. This can be a project item or a property, method, event (etc.) or a project item. Separate each item with a period. When you set a location, the appropriate project item is selected in the Navigator.

Syntax
stringValue = Location

Location = stringValue

```
Location = "Window1"
```

NewConsoleProject

Creates a new console project and opens a new workspace window.

Syntax
NewConsoleProject

NewGUIProject

Creates a new desktop project and opens a new workspace window.

Syntax
NewGUIProject

NewWebProject

Creates a new web project and opens a new workspace window.

Syntax
NewWebProject

ProjectShellPath

Returns the shell path for the project currently being edited.

Syntax
stringValue = ProjectShellPath

If the project has not yet been saved, then it returns the empty string.

```
Print(ProjectShellPath)
```

QuitIDE

Quits the IDE, either ignoring any unsaved changes or saving changes without prompting.

Syntax
QuitIDE(saveChanges As Boolean)

```
Quit(True)
```

SelectWindow

Selects the IDE workspace window with the specified title or index, bringing it to the front.

Syntax
SelectWindow(windowTitle As String)

SelectWindow(index As Integer)

WindowCount

Returns the number of open workspace windows in the IDE.

Syntax
integerValue = WindowCount

WindowTitle

Returns the name of a workspace window using its index (0-based).

Syntax
stringValue = WindowTitle(index As Integer)

This example loops through and saves all the open workspace windows:

```
Dim saved As String
Dim comma As String
Dim i As Integer
For i = 0 To WindowCount-1
  SelectWindow(i)
  saved = saved + comma + WindowTitle(i)
  DoCommand("SaveFile")
  comma = ", "
Next
Print("Saved: " + saved)
```

SelectProjectItem

Selects the project item specified by the path.

Syntax
SelectProjectItem(itemPath As String) As Boolean

Returns True if the item was selected, False if not (usually because it does not exist). Specify the path using dot notation, with each level separated by a period.

To select a method, property or other item within a project item, use the Location method.

Sublocations

Returns all the locations within the given base location as a tab-delimited String (ChrB(9)).

Syntax
SubLocation(baseLocation As String)

```
// Display name of each item in a folder
Dim itemList As String
itemList = Sublocations("Folder1")

Dim items() As String
items = itemList.Split(ChrB(9))

Dim name As String
For Each name In items
  Print("Item: " + name)
Next
```

TypeOfCurrentLocation

Returns a string that is the type of the current location.

Syntax
stringValue = TypeOfCurrentLocation

```
Print(TypeOfCurrentLocation)
```

Project Items

ChangeDeclaration

Changes the declaration of the current property or method.

Syntax
ChangeDeclaration(name As String, parameters As String, returnType As String, scope As Integer, implements As String)

The value for *scope* may be one of the following:

0 = Public

1 = Protected

2 = Private

Generally this is used to rename a property or method after it has been added to a project item. In the case of a property, the parameters value is used for the default value of the property.

```
If SelectProjectItem("Window1") Then
  DoCommand("NewProperty")
  ChangeDeclaration("UserName", "Bob Rob-
erts", "String", 2, "")
End If
```

ConstantValue

Gets or sets the value of a project item constant.

Syntax
ConstantValue(name As String) As String

```
If SelectProjectItem("App") Then
  // kBeta must already exist
  ConstantValue("kBeta") = "True"
End If
```

DecryptItem, EncryptItem

Decrypts or encrypts the selected project item using the specified password. Returns True if it succeeded.

Syntax
DecryptItem(password As String) As Boolean

EncryptItem(password As String) As Boolean

```
// Encrypt the specified project items
Dim items() As String
items = Array("Class1", "Class2")

Dim name As String
For Each name In items
  If SelectProjectItem(name) Then
    If Not EncryptItem("pa55w0rd") Then
      Print("Error encrypting " + name)
      Exit For
    End If
  End If
Next
```

ProjectItem

Returns the name of the project item that is selected in the Navigator (of the current tab, if more than one tab is open).

Syntax
stringValue = ProjectItem

```
Print(ProjectItem)
```

PropertyValue

Gets or sets the value of a project item property. This only works for properties of project items that are part of the Xojo framework (such as Window or App). You can only modify properties that are part of the framework (such as Window1.Title), not properties that you have added.

Syntax
PropertyValue(propName As String) As String

You can specify just the property name which will use the currently selected project item. Or you can specify a project item, followed by the property name using dot notation. When referring to the App object, always use the name “App” even if you have renamed it in your project.

The property value is always returned as a string, even if the property itself is not a string.

```
// Set the ShortVersion to match the
// version numbers
Dim version As String
version = PropertyValue("App.MajorVersion") +
"." + _
PropertyValue("App.MinorVersion") + "." + _
PropertyValue("App.BugVersion") + " (" + _
PropertyValue("App.NonReleaseVersion") + ")"

PropertyValue("App.ShortVersion") = version
```

Text

Gets or sets the entire text of the current Code Editor.

Syntax
stringValue = Text

Text = stringValue

```
// Add a comment header to top of method
Dim header As String
header = "// Copyright 2013 Acme, Inc."

Text = header + EndOfLine + EndOfLine + Text
```

String

Clipboard

Used to get or set the text contents of the OS clipboard.

Syntax
stringValue = Clipboard

Clipboard = stringValue

This example creates a new Note for the selected project item and pastes in the contents of the Clipboard into the Note:

```
DoCommand("NewNote")
Text = Clipboard
```

EndOfLine

Returns the default line ending for the OS running the IDE. This is also the line separator used for code editor text that is returned by Text and SelText.

Syntax
stringValue = EndOfLine

```
DoCommand("NewNote")
Text = "Line1" + EndOfLine + "Line2"
```

SelLength

Gets or sets the length of characters in the current selection of the code editor.

SelStart

Gets or sets the offset of the selection (or insertion) point in the code editor.

SelText

Gets or sets the selected text in the code editor. After assign a string to SelText, the selection will be empty and positioned immediately after the inserted text.

Syntax
stringValue = SelText

SelText = stringValue

This example takes the selected text and adds an inline constant containing the text to the top of the method and replaces the selected text with the name of the constant:

```
Dim constantText As String
constantText = SelText

Dim newConstant As String
newConstant = "Const kValue = """ + con-
stantText + """"

SelStart = SelStart-1
SelLength = SelLength + 2
SelText = "kValue"

Text = newConstant + EndOfLine +
EndOfLine + Text
```

Input/Output

OpenFile

Attempts to open a project file using the specified path, which can be either a native or shell path.

Syntax

OpenFile(filePath As String)

```
OpenFile("c:\projects\IDEScriptTest.xojo
_binary_project")
```

Command and Scripts

DoCommand

Executes the specified command. Refer to the next section for a list of available commands.

Syntax
DoCommand(cmdName As String)

```
DoCommand("SaveFile")
```

DoShellCommand

Runs a shell command or shell script.

Syntax
DoShellCommand(command As String, timeOut As Integer = 3000, ByRef resultCode As Integer) As String

The Shell environment is configured with these variables:

IDE_PATH: Path to the folder containing the IDE

PROJECT_PATH: Path to the folder containing the current project

PROJECT_FILE: Path to the actual project file

The command is run synchronously and returns the output as the result. The timeout is in milliseconds.

```
Dim command As String
command = "codesign -f --deep -s ""Devel-
oper ID Application: YourName ""
""YourXojoApp.app """
DoShellCommand(command)
```

RunScript

Runs the passed script, found in the Scripts folder, either next to the IDE or next to the frontmost project file.

Syntax
RunScript(scriptName As String)

```
RunScript("CommentScript")
```

Commands used by DoCommand

The following commands may be used as parameters to the DoCommand method.

Project Navigation and Management

- OpenFile: Displays the Open Project File dialog.
- SaveFile: Saves the current project with no prompt.
- SaveFileAs: Displays the File Save As dialog.
- Import: Displays the import file dialog.
- CloseWindow: Closes the current workspace window.
- RunApp: Runs the current project in debug mode.
- RunPaused: Runs the current project, but does not launch the debug app.
- BuildApp: Builds the current project.
- Print: Displays the print dialog.
- PageSetup: Displays the page setup dialog.
- GoBack: Go backward in tab history.
- GoForward: Go forward in tab history.
- Help: Shows the Language Reference window.
- Library: Toggles the display of the Library.
- Inspector: Toggles the display of the Inspector.
- Find: Toggles display of Find pane.
- ToggleErrors: Not implemented.
- ToggleMessages: Not implemented.
- NewTab: Adds a new tab to the current workspace window.

Project Items

- NewClass: Add a new class to the current project.
- CopyFilesStep: Adds a new Copy Files Step to the current project.
- RunIDEScriptStep: Adds a new Script Step to the current project.
- NewRunExternalScriptStep: Displays file selector dialog to add a new External Script Step to the current project.
- NewInterface: Adds a new interface to the current project.
- NewContainerControl: Adds a new container control to the current project.
- NewFileTypes: Adds a new file type set to the current project.
- NewFolder: Adds a new folder to the current project.

- NewMenuBar: Adds a new menu bar to the current project.
- NewModule: Adds a new module to the current project.
- NewReport: Adds a new report to the current project.
- NewToolbar: Adds a new toolbar to the current project.
- NewWindow: Adds a new window to the current project.
- AddWebPage: Adds a new web page to the current project.
- AddWebDialog: Adds a new web dialog to the current project.
- AddWebStyle: Adds a new web style to the current project.

Project Items Editing

- AddEventImplementation: Displays the Add Event Handler dialog.
- NewMethod: Adds a new method to the selected project item.
- NewProperty: Adds a new property to the selected project item.
- NewNote: Adds a new note to the selected project item.
- NewMenuHandler: Adds a new menu handler to the selected project item.
- NewComputedProperty: Adds a new computed property to the selected project item.
- NewConstant: Adds a new constant to the selected project item.
- NewDelegate: Adds a new delegate to the selected project item.
- NewEnum: Adds a new enumeration to the selected project item.
- NewEvent: Adds a new event definition to the selected project item.
- NewExternalMethod: Adds a new external method to the selected project item.
- NewSharedComputedProperty: Adds a new shared computed property to the selected project item.
- NewSharedMethod: Adds a new shared method to the selected project item.
- NewSharedProperty: Adds a new shared property to the selected project item.
- NewStructure: Adds a new structure to the selected project item.

Editing

- Comment: Add the comment prefix to the selected text in the code editor (or the current line if no text is selected).

- CheckItemErrors: Equivalent to Project → Analyze Item.
- CheckProjectErrors: Equivalent to Project → Analyze Project.
- SelectAll: Not implemented. Use SelStart and SelLength instead.
- Copy: Copies the selected item in the Navigator to the clipboard.
- Paste: Pastes the text in the clipboard to the active code editor.
- Cut: Not implemented.
- Undo: Not implemented.
- DeleteSelection: Not implemented.

Layout Editor

- AddFromLibrary: Not implemented.
- EditModeCode: Switch to Code Editor.
- GoToLastEvent: Go to last edited code.
- EditModeView: Switch to Layout Editor.
- StartInlineEditing: Open Default Property popout window.
- LockPositions: Toggle lock for selected control.
- ShowHideTabOrder: Displays Tab Order Editor dialog.
- ToggleMeasurements: Toggle measurements.
- OrderForward: Order selected controls forward.
- OrderToFront: Bring selected controls to front.
- OrderBackward: Order select controls backward.
- OrderToBack: Send selected controls to the back.
- FillWidth: Fill width on the selected control.
- FillHeight: Fill height on the selected control.
- AlignLeft: Aligns the selected controls to the left.
- AlignRight: Aligns the selected controls to the right.
- AlignTop: Aligns the selected controls to the top.
- AlignBottom: Aligns the selected controls to the bottom.
- AlignSpaceHorizontally: Align and space the selected controls horizontally.
- AlignSpaceVertically: Align and space the selected controls vertically.

Menu Editor

- AddMenu: Adds a top-level menu to the menu bar.
- AddMenuItem: Adds a menu item to the selected menu.

- AddMenuSeparator: Adds a separator to the selected menu.
- AddSubmenu: Adds a submenu to the selected menu.
- ConvertToMenu: Converts a submenu item to a top-level menu.
- ViewAsWin32: Changes to the Windows menu view.
- ViewAsOSX: Changes to the OS X menu view.
- ViewAsLinux: Changes to the Linux menu view.

File Types Set Editor

- NewFileType: Adds a new file type to the editor.
- Clear: Removes the selected file type from the editor.
- AddCommonFileType$Pdf: Adds PDF file type.
- AddCommonFileType$Rtf: Adds RTF file type.
- AddCommonFileType$Mp3: Adds MP3 file type.
- AddCommonFileType$Jpeg: Adds Jpeg file type.
- AddCommonFileType$Png: Adds PNG file type.
- AddCommonFileType$Any: Adds “any” file type.
- AddCommonFileType$Text: Adds text file type.
- AddCommonFileType$Mpeg: Adds Mpeg file type.
- AddCommonFileType$Quicktime: Adds QuickTime file type.
- AddCommonFileType$More: Adds “more” file type.

Report Layout Editor

- AddPageSection: Add page header/footer section to report.
- AddGroupSection: Add group header/footer section to report.

Copy File Steps Editor

- AddFileToCopyFilesStep: Displays file dialog to select a file to add to the editor.
- RemoveFileFromCopyStep: Removes the selected file from the editor.

Chapter 10

Advanced Features

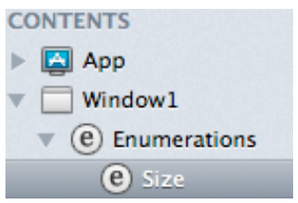
These advanced topics allow you to take your development skills to new levels!

CONTENTS

Enumerations

An enum or enumeration is a group of constants that you can refer to by name.

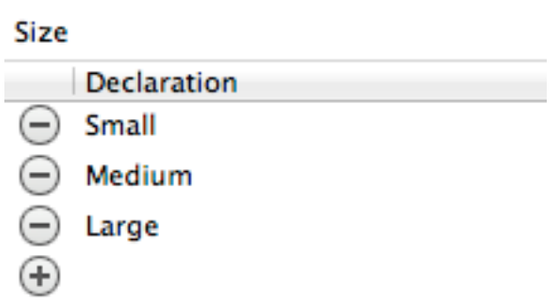
Figure 10.1 Enumeration in Navigator



When you create an enumeration, you are creating a new data type. This type can be added to a module, class, window or any other object in your project. Add an enumeration by selecting Enumeration from the Insert menu or Insert button. This displays the enumeration in the Navigator and the Enumeration Editor.

In the Enumeration Editor, you use the "+" icon to add a new enumeration value to the group. Use the "-" button to remove an enumeration from the group.

Figure 10.2 Enumeration Editor

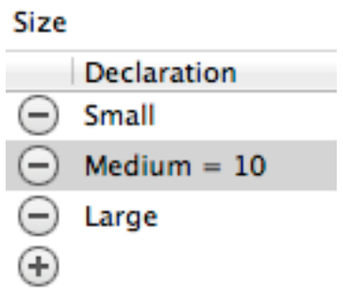


When you add an enumeration, you also give it a name. The name is used to refer to the enumeration. You can edit the name by clicking on it once to select it and a second time to edit it.

***Note**: The enumeration name cannot be left blank.*

Although enumeration items are integers behind the scenes, you don't normally worry about the integer values. However, should you need to assign a specific integer value to an enumeration item, you can do so by assigning it as part of the name.

Figure 10.3 Enumeration with an Assigned Integer Value



To use an enumeration in code, you refer to the container of the enumeration, the enumeration name and then the enumeration item name:

```
Dim pictureSize As Window.Size
pictureSize = Window1.Size.Large
```

If you find you need the actual integer value stored by the enumeration, then you have to cast it to an integer:

```
Dim pictureSize As Integer
pictureSize = Integer(Window1.Size.Large)
```

Enumerations are always an Integer type and default to “Integer”. You can change the type to other Integer types (such as UInt64) should you need to use larger values for the enumeration.

AddHandler

The AddHandler command is used to have a method on one object handle the processing of an event on another.

This is often used to allow you to instantiate a class directly and implement its event without having to create a separate subclass.

For example, if you want to implement the Action event of a Timer, you need to subclass it first. This can be done by dragging a Timer onto your window or web page or by adding a Timer subclass to your project and using that instead.

Alternatively, you can declare the timer in your code and then use AddHandler to have the Action event handled by a method on your window.

For example:

```
Dim t As New Timer
AddHandler t.Action, AddressOf MyTimerMethod
t.Mode = Timer.ModeMultiple
t.Period = 500
t.Enabled = True
```

In order for this to work, you must have a method in the window called MyTimerMethod and it must have a parameter of the timer itself:

```
Sub MyTimerMethod(t As Timer)
  // Your code goes here
End Sub
```

***Note**: If the event you are handling has additional parameters, include them after the initial parameter for the class.*

AddHandler works the same way for threads:

```
Dim t As New Thread
AddHandler t.Run, AddressOf MyThreadMethod
t.Run
```

XojoScript

XojoScript allows your applications to run Xojo source code within a running application. This feature is a great way to allow your users to expand the functionality of your application by providing their own scripts.

Scripts take full advantage of the Xojo language, but provide only limited access to its framework. The Scripting Language topic in the Language Reference has full details on what is supported.

You run scripts using the XojoScript class, which provides ways for you to assign source code, run it, supply input, get output, handle errors, interface with your application and more.

Running Code

To get started, you need an instance of the XojoScript class. You can drag a XojoScript control to a Window, WebPage or Container. Or you can create an instance in your code.

The next thing to do is assign a script to it. This is done by assigning the source code (as a string) to the Source property. In this example, Script is the name of a XojoScript object that has been added to a window:

```
Script.Source = "Print ""Hello, World! """
```

The Print command outputs the supplied text to the Print event of the XojoScript instance.

***Note**: You can access the Print event by subclassing XojoScript, by dragging a XojoScript control onto your layout or by using AddHandler.*

This code in the Print event displays a message with the supplied text:

```
MsgBox(msg)
```

Lastly, you can run the script by calling the Run method:

```
Script.Source = "Print ""Hello, World!"""
Script.Run
```

Handling Errors

There are two events used to handle errors: CompileError and RuntimeError.

CompilerError

The CompilerError event is called if there is a problem compiling the script because of invalid syntax or some other problem. This event provides you with the location of the error (using the XojoScriptLocation class), the error (as a XojoScript.Errors enum and extra errorInfo as a Dictionary).

It is important to handle this event if you are allowing the user to supply scripts.

CompilerWarning

The CompilerWarning event handler is called when there is a warning compiling the script. This could indicate unused local variables, deprecations and other warnings.

This event provides you with the location of the warning (using the XojoScriptLocation class), the warning (as a Warnings enum and extra warningInfo as a Dictionary).

It is important to handle this event if you are allowing the user to supply scripts.

RuntimeError

The RuntimeError event is called if a runtime exception occurs while the script is running.

The event provides the error as a RunTimeException which you can use to determine the type of exception that occurred.

It is important to handle this event if you are allowing the user to supply their own scripts.

This code can display the run time exception that occurred:

```
Dim dialog As New MessageDialog
dialog.Title = "Unhandled Exception"
dialog.Message = "Unhandled Exception"
dialog.Explanation = "A " +
Introspection.GetType( error ).Name + " was
not caught."
If error.Message <> "" Then
  dialog.Explanation = dialog.Explanation + "
" + error.Message
End If

Call dialog.ShowModalWithin( Self )
```

XojoScriptLocation

This class has properties to provide you with information about an error or warning. You can use this information to let the user know where the problem is and even to highlight the text in the script.

These are the properties:

- Character
  The starting character of the part of the script with an error or warning. Use in conjunction with EndCharacter to highlight the text in the script.
- Column
  The starting column of the script error or warning.
- EndCharacter
  The ending character of the part of the script with an error or warning.
- EndColumn
  The ending column of the script error or warning.
- EndLine
  The ending line number of the script error or warning.
- Line
  The starting line number of the script error or warning.

Enumerations

Use the XojoScript.Errors and the XojoScript.Warnings enumerations to check for specific errors or warnings.

Interfacing with your Application

Scripts run in a sandbox, which means they cannot directly interact with any of your application code beyond the specified events.

This can often be too restrictive, so you have the option of making some of your application code available by using the Context property.

The Context is an object that you have instantiated. Your script is allowed to call the methods on this object as if they were global methods defined in the script.

The typical way to do this is to have a special class that knows how to interface with your application. For example, this class could have a Save method that calls the appropriate code in your application to save something.

Normally your script would not be able to trigger a save in your application, but by using the Context as the go-between it can.

You assign the context like this:

```
Dim controller As New MyAppController
Dim script As New XojoScript
script.Context = controller
```

Then in the script, you can call the Save method as if it where any global method:

```
Dim controller As New MyAppController
Dim script As New XojoScript
script.Context = controller
script.Source = "Save"
script.Run
```

Advanced Features

You can optimize code before running it by calling the Precompile method and supplying it a value from the XojoScript.OptimizationLevels enum: High, Low, None. Using higher optimization levels makes the script run faster, but it will take the script longer to compile. Unless your scripts are especially math intensive, there is little need to change the optimization level.

If Precompile returns True, then you can run the script using the Run method.

You can check the state of a script by comparing the State property to the States enumeration with values: Aborted, Complete, Ready, Running.

Declare and MemoryBlock

Declare

The declare statement is used to create a reference to a method in an external shared library. These can be operating system libraries or other shared libraries. On Windows these are referred to as DLLs (Dynamic Linked Libraries), on OS X are called dylibs (Dynamic Libraries) and on Linux are referred to as so (Shared Object).

To create a declaration to an external method, you use the Declare keyword (usually prefixed with Soft so that the lookup only happens at run-time), its type, name, library and parameters.

For example, this code declares the GetDoubleClickTime function in the User32 library on Windows:

```
Declare Function GetDoubleClickTime Lib
"User32.DLL" () As Integer
```

You would call this method just as you would any method:

```
Dim dcTime As Integer
dcTime = GetDoubleClickTime
```

***Note**: In order to create a declare statement, you need access to the API (Application Programming Interface) documentation for the shared library you wish to use. The API defines the available methods and parameters.*

Creating a declare to a Cocoa shared library is similar. This declares access to the Center method that can center a Cocoa window on the screen:

```
Declare Sub Center Lib "Cocoa" Selector
"center" (windowRef As Integer)
```

***Note**: The Selector keyword is only used by OS X and is used to uniquely identify the Cocoa method.*

You can use it in the Open event of a Window to center it on the screen:

```
Center(Self.Handle)
```

MemoryBlock

MemoryBlocks are used whenever you need a container for any arbitrary binary data.

A MemoryBlock object allocates a sequence of bytes in memory and manipulates those bytes directly. A MemoryBlock can be passed in place of a Ptr when used in a Declare call.

For times when you need to directly manage a block of data in its raw (byte) form, you will need to use a MemoryBlock.

When reviewing the API for creating Declare statements, you will find that some parameters require a pointer to a chuck of data. You can set up a memory block as the location for this data.

A memory block can be initialized with a String value. The byte contents of the string are simply assigned as the raw data for the memory block.

```
Dim s as String = "Hello!"
Dim mb as MemoryBlock
mb = s
MsgBox(mb.StringValue(0, mb.Size))
```

For more information about MemoryBlock, refer to the Language Reference.

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. A structure can provide a convenient alternative to a MemoryBlock as they are also often used to group together information for external function calls.

The values of structures have specific sizes giving the structure its own specific size.

Structures are added to project items such as modules, classes and windows. To add a structure, select the Add button on the toolbar and select Structure from the menu (or use the Insert menu in the main menu bar). This displays the Structure Editor where you specify the fields in the structure and their sizes.

Figure 10.4 Structure Editor

Use the “+” button to add a new field. Fields are added by entering their name followed by the type in this form:

```
Name As Type
```

So to enter an Integer you would do:

```
Age As Integer
```

Note that when you enter a field, it shows the size. This is the amount of bytes that the field uses in the structure. Integers use 4 bytes, for example.

Strings are a special case because you have to specify the exact size of the string in bytes. Unlike normal strings, a string in a structure always takes up the specified size in the structure and you cannot exceed it. Also, strings in structures do not contain encoding information. The syntax is:

```
Name As String*size
```

So to have a String with a size of 50:

```
Name As String*50
```

When you are creating a structure to pass to an external method, be sure that your sizes precisely match the sizes specified by the API of the method.

You can reference a structure using dot notation:

```
Dim cust As CustomerStruct
cust.Name = "Bob Roberts"
cust.Age = 60

MsgBox(cust.Name)
```

Usage

Structures are useful for organizing data, but they are not object-oriented and are limited in many ways (such as with string sizes). For your project's internal data management, you should almost always use a class over a structure.

However, they are small and very memory efficient. In addition to being useful when used with external methods, they may also be useful in situations where memory must be managed carefully.

Weak References and Memory Management

A weak reference allows you to retain a reference to an object without increasing its reference count.

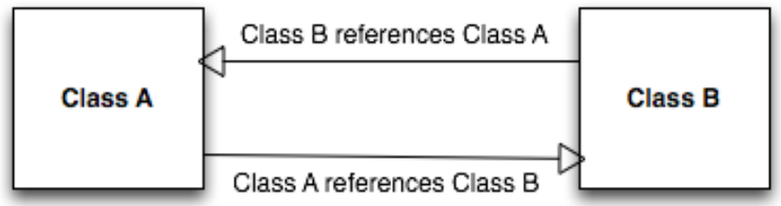
Reference Counting

When you create a new instance of a class, an internal reference counter for the class is increased. When the class goes out of scope, the reference is decreased. When the reference reaches 0, the class instance is removed from memory.

This type of memory management is called reference counting. It is simple, fast and easy to understand. But there are times when it is not sufficient and can lead to a memory leak. A memory leak is when memory that was set aside for a class is never released. This causes you application to continually use additional memory. If your application runs for a long time, you could eventually run out of memory.

The most common reason that this occurs is because of circular references. If class A has a reference to class B and class B has a reference to class A, then you have a potential memory leak because neither reference is able to get down to 0.

Figure 10.5 Circular Reference



Weak References allow you to get a reference to a class instance without changing the internal reference counter.

Weak References

To get a weak reference to an object, you use the WeakRef class:

```
Dim ref As WeakRef
ref = New WeakRef(object)
```

The Value property of the WeakRef returns Nil if the object is no longer available. If it is still available, it returns the object (which you will likely want to cast to the appropriate type).

For example, this code declares an instance of a FolderItem that will go out of scope and have its reference counter decreased. A weak reference is assigned to the FolderItem instance. While the instance is available, its name is displayed. When the instance is removed from memory, the weak reference value become Nil, so you now know there are no more references to it:

```
Dim ref As WeakRef
If True Then
  Dim f As New FolderItem
  f.Name = "TestFile.txt"
  ref = New WeakRef(f)
  If ref.Value <> Nil Then
     // This displays
    MsgBox(FolderItem(ref.Value).Name)
  Else
    MsgBox("f is Nil.")
  End If
End If

If ref.Value <> Nil Then
  MsgBox(FolderItem(ref.Value).Name)
Else
  MsgBox("f is Nil.") // This displays
End If
```

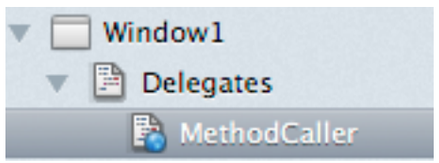
Delegates

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.

A Delegate can be declared in either a module or a class. You use the Insert→ Delegate menu command or the Add button on the toolbar in the Code Editor to create a Delegate entry, which appears under the containing object.

Figure 10.6 A Delegate called MethodCaller

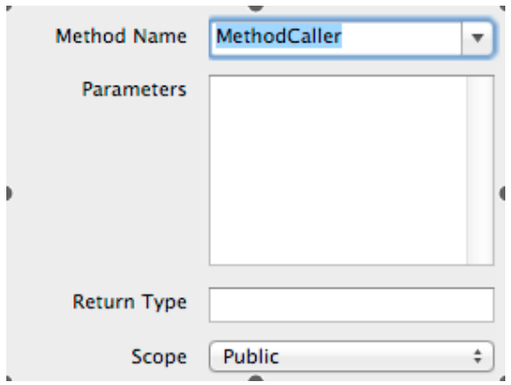


A delegate must have a name and can have optional parameters and a return type.

To use a delegate, you have to point it to an address for a method. You can do this using the AddressOf or WeakAddressOf commands. Only methods that match the parameters and return type (the signature) of the delegate may be assigned to the delegate.

When the delegate contains the address of a method, you can call the method using the Invoke method of the delegate.

Figure 10.7 Delegate Declaration



To try out a quick example, create a delegate on a window and call it MethodCaller.

Also on the window, create two methods: TestMethod and AnotherMethod. Each with this code:

```
MsgBox(CurrentMethodName)
```

Since these two methods have the same signature they can be assigned to a delegate and called using the Invoke method. This

code in the Action event handler of a Button calls a different method depending the state of a CheckBox:

```
Dim callMethod As MethodCaller

If MethodCheck.Value Then
  callMethod = AddressOf TestMethod
Else
  callMethod = AddressOf AnotherMethod
End If

callMethod.Invoke
```

You can also create a delegate using an external function that returns a pointer. That code would look like this:

```
Dim fp As Ptr = ExternalFunctionThatReturnsAPointer
Dim callMethod As New MethodCaller(fp)
```

Introspection

Introspection is a way for you to get information about your application structure at runtime.

For example, you could use it to get a list of the methods of a class instance, check for a specific method and then call that method. Introspection is a module containing the various methods that can act on class instances. For example, to get the type (name) of a class instance you can write:

```
Dim f As New FolderItem
Dim oType As Introspection.TypeInfo
oType = Introspection.GetType(f)
MsgBox(oType.FullName)
```

The above example is not really useful because you already know the type since your code declared it. But if you are writing a more generic method that does not know much about the code that is calling it, introspection starts to become much more useful.

In fact, introspection is most useful when you are writing generic code that is designed to be used in a wide variety of places. This type of code is often called framework code.

Some examples of when introspection can be useful:

- A unit testing library could use introspection to run methods that end in "test".
- A database framework could use introspection to get the type of properties on a class in order to create database columns for them.
- A serialization library could use introspection to query a class so that its information can be saved and restored.

A more simple, but still interesting use is in the App.UnhandledException event handler. Here, you usually want to display the type of exception that occurred. You would normally do this using a long Select Case statement for each type of exception. The problem with doing it this way, is that it makes your application larger because all the related code for the exception is pulled into your application, even if you do not need

it. It also leads to more complicated code and requires you to remember every exception name and update the code when new exceptions are added.

Instead you can use introspection to get the type of the RuntimeException parameter of the UnhandledException event handler. This code displays the type of the runtime error followed by the stack trace:

```
Dim excType As Introspection.TypeInfo
excType = Introspection.GetType(error)

Dim excName As String
excName = excType.FullName

Dim stack As String
stack = Join(error.Stack, EndOfLine)

Dim errMsg As String
errMsg =  excName + EndOfLine + EndOfLine + stack

MsgBox(errMsg)

Return True
```

Attributes

Attributes are compile-time properties. They can be added to project items and code items such as method and properties. An attribute consists of its Name and its Value. The Name is required, but the value is optional.

Attributes are added using the advanced tab of the Inspector. Attributes can be added to classes, modules, windows, containers, interfaces and toolbars.

A subclass inherits attributes from its super class. These inherited values can be overridden by the subclass by simply redefining them.

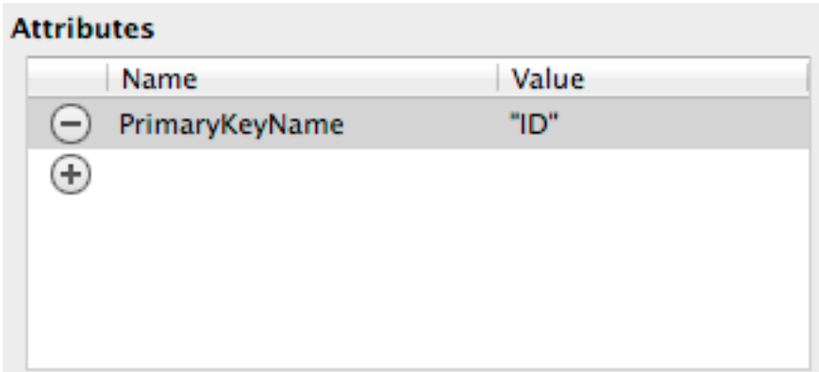
Creating an Attribute

You create an attribute using the Attribute Editor in the advanced tab of the Inspector for the item.

Use the “+” or “-” buttons to add or remove attributes. Specify the Name and Value for the attribute in the list.

There are two ways to specify the value. If it is a literal value, such as “ID”, then the value must be enclosed in quotes.

Figure 10.8 Attribute Editor in the Inspector



If it is a constant, you can just use the name of the constant, for example kID.

If you forget the quotes for a literal value, you will get a compiler error if the constant is not found.

Accessing an Attribute

Attributes are accessed in your code using Introspection. The AttributeInfo class is used to fetch the Name-Value attribute pairs for a particular object.

This code gets the attribute values of the default window and displays them in a List Box:

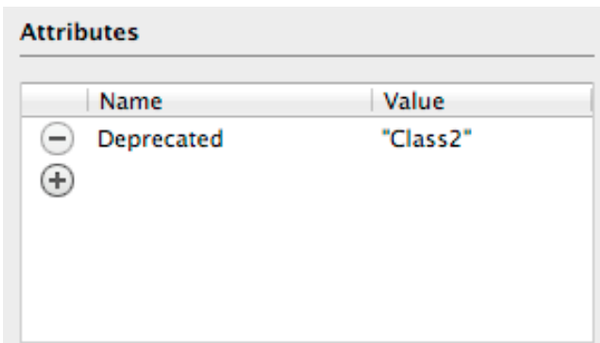
```
Dim myAttributes() As Introspection.AttributeInfo =
Introspection.GetType(Window1).GetAttributes

For i As Integer = 0 To UBound(myAttributes)
  ListBox1.AddRow(myAttributes(i).Name)
  If myAttributes(i).Value.IsNull Then
    ListBox1.Cell(ListBox1.LastIndex, 1) = "No Value"
  Else
    ListBox1.Cell(ListBox1.LastIndex, 1) =
Str(myAttributes(i).Value)
  End If
Next
```

Using Deprecated and Hidden Attributes

The **Deprecated** and **Hidden** attributes are reserved and perform special actions when used on a project item (such as a class or module) or a method, property (or constant, etc.) that you can add to a project item.

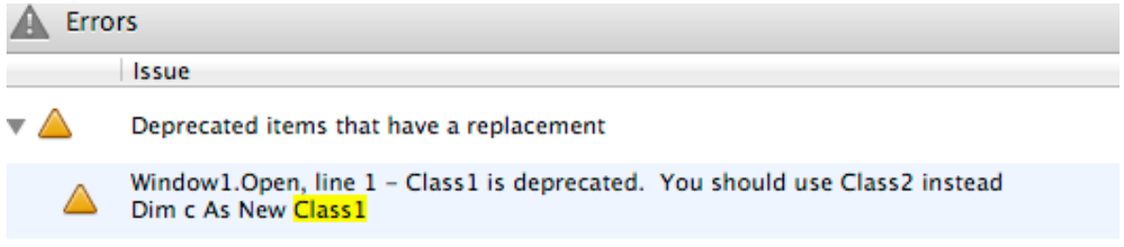
The Deprecated attribute allows you to indicate that an item "has a replacement". Set the attribute value to the name of the class/method/property that should be used instead (be sure to enclose the name with quotes).

Figure 10.9 Using the Deprecated Attribute on Class1



A deprecated item appears in the Errors Pane when you use Analyze Project.

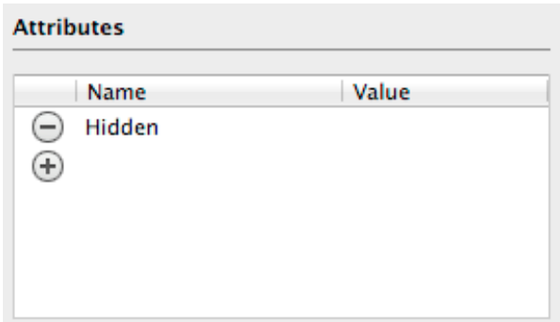
Figure 10.10 A Deprecated Item Appearing in the Errors Pane



The Hidden attribute hides the specified item from introspection, the debugger and auto-complete. You do not need to specify a value.

Both of these attributes may be useful on frameworks that are used by other developers.

Figure 10.11 Using the Hidden Attribute on a Method



Interface Aggregation

Interface aggregation is the ability to group several interfaces together into a “super interface.”

Consider these interfaces:

```
Interface Test
  Sub Foo()
End Interface

Interface Awesome
  Sub Bar()
End Interface

Interface Sweet
  Aggregates Test, Awesome
  Sub Blah()
End Interface
```

Sweet is an aggregate of the Test and Awesome interfaces.

In this example, there are three interfaces that contain a single method each. If a class were to implement the Test interface, then it would be required to implement just the method Foo. However, if the class were to implement the Sweet interface, then it would have to implement Foo, Bar and Blah. That’s because Sweet aggregates both the Test and Awesome interfaces.

Basically, when a class implements an interface, it must implement all of the methods from the interface, as well as methods from any aggregated interfaces. When a class implements an interface, then calling IsA on the class will return True for that interface as well as any aggregated interfaces. From the example above, a class implementing Sweet, then IsA Sweet, IsA Test and IsA Awesome are all True (since it implements all three of the interfaces).

This allows you the ability to combine interfaces in creative ways. Although interfaces don’t necessarily relate to one another, there are times when you need something that satisfies the IsA command for multiple different interfaces. Or, when you want to merge functionality together for interfaces.

For instance, say that you want to have an item which can be iterated over. A common interface might be:

```
Interface Iterator
  Function Current() as Variant
  Function MoveNext() as Boolean
  Sub Reset()
End Interface
```

This allows you to iterate over the object in a generic fashion. However, it could very well be that creating this iterator will cause circular references, or some sort of memory management issues. In that case, you might want to use a disposable interface, like this:

```
Interface Disposable
  Sub Dispose()
End Interface
```

In the example, you could have the Iterator interface aggregate the Disposable interface. This would guarantee that anything which can be iterated over can also be disposed of afterwards.

You might be wondering “so why not just put the Dispose method right onto the Iterator interface?” That's a perfectly legitimate way to accomplish the same goal — however, Iterator and Disposable aren’t the same conceptually. So it doesn’t make sense to force the two interfaces together in such a fashion. It doesn’t hurt, but it just doesn’t help either. Next is an example that would hurt.

Say you have a method that is going to generically take a parameter which represents something that can read and write to or from a source. In this case, you want something that’s both Readable and Writeable. You could just declare the method like this:

```
Sub DoSomethingAwesome( foo as Readable )
  If foo IsA Writeable Then
    // Do reading and writing
  End If
End Sub
```

This code will compile and will work, but it’s also not safe. The user could pass in something which is Readable, but not Writeable and it will compile. Instead, the user could easily do this instead:

```
Interface ReadableAndWriteable
  Aggregates Readable, Writeable
End Interface

Sub DoSomethingAwesome( foo as ReadableAndWriteable )
End Sub
```

Now there will be a compile error if the item doesn’t implement both interfaces. That's a much better solution to the problem.

Chapter 11

Xojo Framework

This chapter has a brief overview of the new Xojo framework.

Overview

The new Xojo Framework is an update and reorganization of the framework you use to create apps with Xojo. The full Xojo Framework is available with iOS projects and a smaller portion of it is available for desktop, web and console projects.

This framework is new and will continue to have more features added to it over time and more it will also be made available to desktop, web and console projects.

For more information about the new Xojo Framework, please visit the online documentation.

Online Documentation

Xojo Framework namespace

Only the Xojo.Core namespace is available for all projects types. The rest of the Xojo framework is only available for iOS projects.

- Xojo
  - Core (available for iOS, desktop, web and console projects)
  - Crypto
  - Data
  - Introspection
  - IO
  - Math
  - Net
  - System
  - Threading