XOJO

User Guide

Book 1
Fundamentals

2015 Release 4

Xojo, Inc.

Preface

Xojo User Guide

Book 1: Fundamentals

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.

editorial fashion only. In addition, terms suspected of being trademarks, registered trademarks, or service marks have been appropriately capitalized, although Xojo, Inc. cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark, registered trademark, or service mark. Xojo, Inc. is not associated with any product or vendor mentioned in this book.

Section 2

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.

Section 3

Table of Contents

Chapter 1

Introduction

Welcome to Xojo, the easiest way to create cross-platform desktop and web applications.

CONTENTS

Section 1

About Xojo

Xojo makes it easy to build powerful, multi-platform desktop, web and iOS applications quickly and easily. If you are new to programming, you will find Xojo's programming language easy to learn. If you are an experienced programmer, you will find the language to be powerful and robust. In either case, you will find you can accomplish quite a bit in a short period of time.

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

Xojo's programming language is a compiled, object-oriented programming language that is easy to understand. This means that it uses the same type of modern architecture that most programming languages (like C++, Objective-C, and Java) are using today. Object-oriented programming languages make it easier to write and debug because the code is written as individual objects that are similar to objects in the real world.

Xojo makes application development faster and easier than traditional languages by removing the need to learn how to access the programming interface for the operating system. This application programming interface (or API for short) consists of thousands of commands, none of which you ever need to learn to build applications in Xojo.

Installation and System Requirements

For latest Xojo System Requirements, refer to the Xojo Dev Center:

http://developer.xojo.com/system-requirements

Getting Started

Xojo ID Sign In

You should sign in to your Xojo account in order for Xojo to use your licenses.

Figure 1.1 Xojo ID Sign In Window

Enter your Xojo ID and password and click the Sign In button to sign in. Once you do this, any licenses associated with your account are applied.

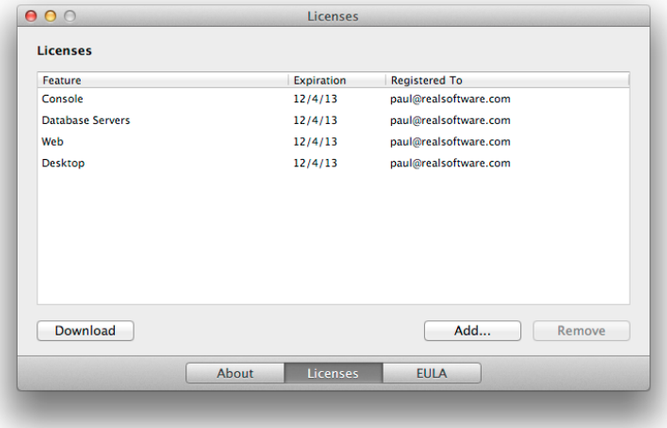
You can check your licenses on the About window in the Licenses tab. Your available license appear along with the Expiration date and the email to which it is registered.

Use the Download button to force Xojo to check for new licenses.

You can also use the Add button to manually add licenses files. The Remove button removes licenses.

On the Xojo web site you can sign in your account to view licenses, download license files and to purchase or renew licenses.

Figure 1.2 About Window Licenses Screen



http://www.xojo.com

QuickStart and Tutorials

If you are new to programming, you should begin by going through the QuickStart and then the Tutorial for either Desktop or

Web Applications. These guides will give you a good overview of Xojo and introduce you to the programming language.

Next, read the Xojo User Guide. This guide provides you with detailed information on Xojo, its programming language, user interface controls, framework and other features. The User Guide is in the process of being updated and migrated to the Xojo Dev Center. Be sure to check there for the latest content:

http://developer.xojo.com

Also refer to the Dev Center for the Reference Guide, providing specifics of every control, class and feature included in Xojo:

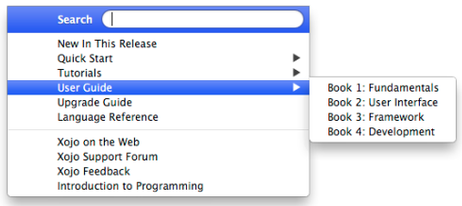
http://developer.xojo.com/reference-guide

Online Help

QuickStart, Tutorials and User Guide

The QuickStarts, Tutorials and Xojo User Guide are included in PDF format with your Xojo installation. These books are located in the Documentation folder in your Xojo installation and are also accessible from the Help menu within Xojo.

Figure 1.3 Xojo Help Menu



If you prefer to read these books using iBooks for iOS or iBooks for Mac, you can also find them in the iBooks Store (search for "Xojo") or on the Xojo Documentation Wiki.

Dev Center

The Xojo Dev Center is accessible directly from Xojo. To access the Dev Center, choose Help ↠ Dev Center from the menu.

Language Reference

The Language Reference is built into Xojo. To access the Language Reference, choose Help ↠ Language Reference (F1 on Windows and Linux or ⌘-? on OS X) or use the Help button on the toolbar.

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

Use the Arrows in the header area to move backward and forward through the items you have been browsing. The Home button takes you back to the page with the list of categories and subcategories.

When you are programming, context-sensitive help is also available. Select the item in your Code Editor for which you want help and right-click (Control-click on OS X). A contextual menu appears. Choose the "Help for ItemName" menu command and Xojo will open the Language Reference to the desired item. You can also hold down the Command key (Control key on Windows and Linux) while double-clicking an item to open the Language Reference for it.

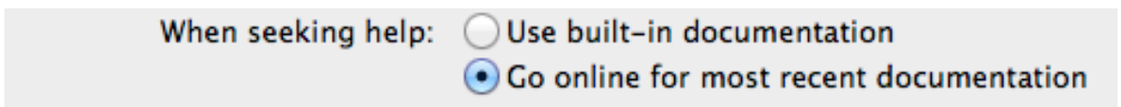
Normally when you access the Language Reference from within Xojo, the online version is accessed so that you get the most up-to-date documentation.

But there may be times when you want to use the Language Reference while not online. Each release of Xojo includes a built-in copy of the Language Reference.

To access the local Language Reference, choose Edit ↠ Options (or Xojo ↠ Preferences on OS X) and choose the General page. The section “When seeking help” allows you to specify whether to use the online documentation or the built-in documentation.

On OS X, you can also hold down Option while using accessing the Language Reference from the Help menu to temporarily reverse this setting.

Figure 1.4 Choosing Online or Built-In Language Reference



Using the Dev Center

In addition to the documentation accessible through Xojo itself, you can also access everything directly at the Xojo Dev Center:

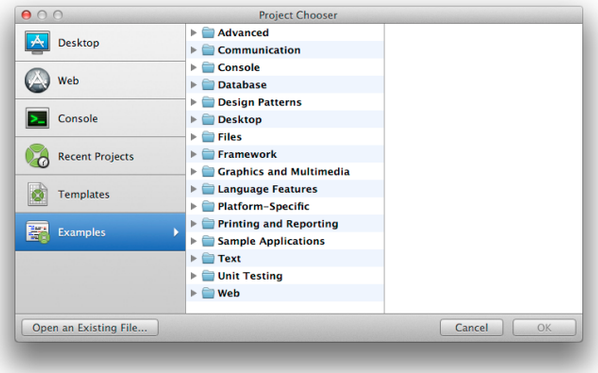
http://developer.xojo.com

Example Projects

Xojo includes a wide variety of example projects (over 350) that demonstrate many of its features and functions.

The Example Projects can be accessed using the Project Chooser window that appears when Xojo starts (refer to Chapter 2, Section 1 for more information on the Project Chooser window).

Figure 1.5 Project Chooser Showing Examples



The example projects are grouped into these folders:

- Advanced
- Communication
- Console
- Database
- Design Patterns
- Desktop
- Files
- Framework
- Graphics and Multimedia
- iOS
- Language Features
- Platform-Specific
- Printing and Reporting
- Sample Applications
- Text
- Unit Testing
- Web

When you choose an example project, Xojo opens a copy of it for you. You can modify and save this copy in any location you like.

Section 5

Feedback

Contact Information

Contact Method	Contact Info
Email	Customer Service/Sales: custserv@xojo.com Technical Support: productsupport@xojo.com
Web	https://www.xojo.com/support/
Mail	Xojo, Inc. PO Box 162181 Austin, TX 78716

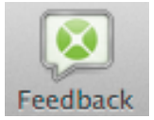
Reporting Bugs and Making Feature Requests

If you think you have found a bug in Xojo or have a feature request, please let us know about it. The best way to report bugs or make feature requests is with the Feedback (http://www.xojo.com/feedback/) application. Feedback is designed to gather all the necessary information that helps us track down bugs and implement feature requests. For each bug or feature request reported (called a **case**), you will receive a confirmation message via email with the case number.

When cases are updated, you get an email telling you about the change.

You can launch the Feedback application directly from within Xojo. Just choose Help ↠ Xojo Feedback or click the Feedback button on the toolbar. If you have already installed Feedback this will open the Feedback application. If you have not yet installed Feedback, your default web browser will open the Xojo Feedback page.

Figure 1.6 Feedback Button



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

Creating Good Feedback Cases

When creating a Feedback case, the best thing you can do is to provide detailed information.

Below are descriptions of the fields that you need to fill out to create a Feedback case:

Summary

Provide a descriptive summary of the issue. Your summary should describe the actual problem.

Examples:

Poor Summary	Good Summary
Xojo crashes	Xojo crashes on 64-bit Linux with missing 32-bit libraries
Listbox doesn't work	Listbox does not work when there are more than 64 columns

Product

Select the Xojo product related to this case.

Category

Some products have categories. Select the category most closely related to this case.

Case Type

Cases are most often either *Bug* or *Feature Request*.

Details

Provide a more detailed description for the case. In the case of a bug, provide specific details on how to reproduce the bug. Some important things to include:

- The problem that occurred.
- Steps to cause the problem to occur.
- What you expected to occur.

For Feature Requests, include information about what you need to accomplish. Try to limit specific implementation suggestions.

Workarounds

If you have found a way to work around the problem, please include it here. This is helpful for others that may have also run into the issue.

Attachments

The best way to demonstrate a bug is to include a sample project. Please take a moment to put together a sample project that demonstrates the problem and attach it to your case.

Include System Data

Feedback can collect information about your system, such as operating system, Xojo version, plugins and much more. Please allow Feedback to include this information.

Chapter 2

Overview

This chapter introduces the Xojo Integrated Development Environment (IDE).

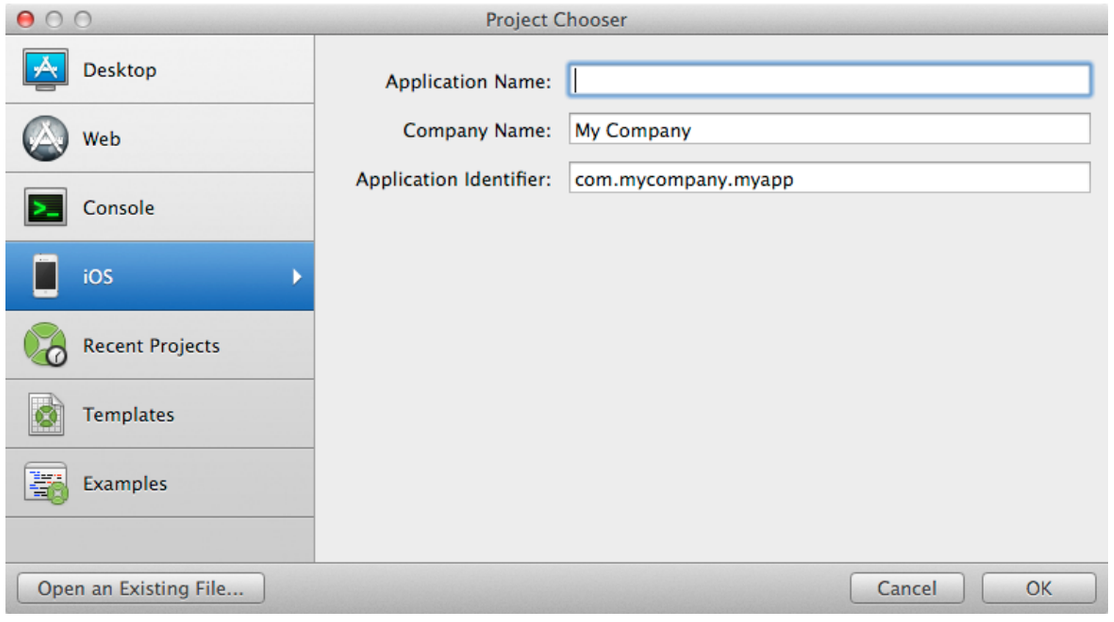
CONTENTS

Project Chooser

The Project Chooser

The Project Chooser window displays each time you start Xojo and when you select File ↠ New Project.

Figure 2.1 The Project Chooser



New Projects

The Project Chooser lets you quickly decide what project you want to work with. You can choose to create a new blank Desktop, Web, Console or iOS project.

Recent Projects

You can view the most recently used projects and choose to open one of them.

Templates

Templates are Xojo projects that are saved in the Templates folder alongside the Xojo application.

If you have a set of commonly used modules, classes or anything else, you can put them all in a project and then save it to the Templates folder. When you open the template, you get a new project with all your project items already included.

Xojo includes templates for creating Service Applications, CGI Applications and Event-Driven Console Applications.

Examples

The Examples section lets you view the example projects included with Xojo. The examples demonstrate how to use specific features and functions of Xojo.

When you choose an example, it is opened in a new project for you to run, edit or save where you like.

Open an Existing File

You can open an existing Xojo project by clicking the *Open an Existing File* button. A file selector dialog will appear allowing you to choose the file to open.

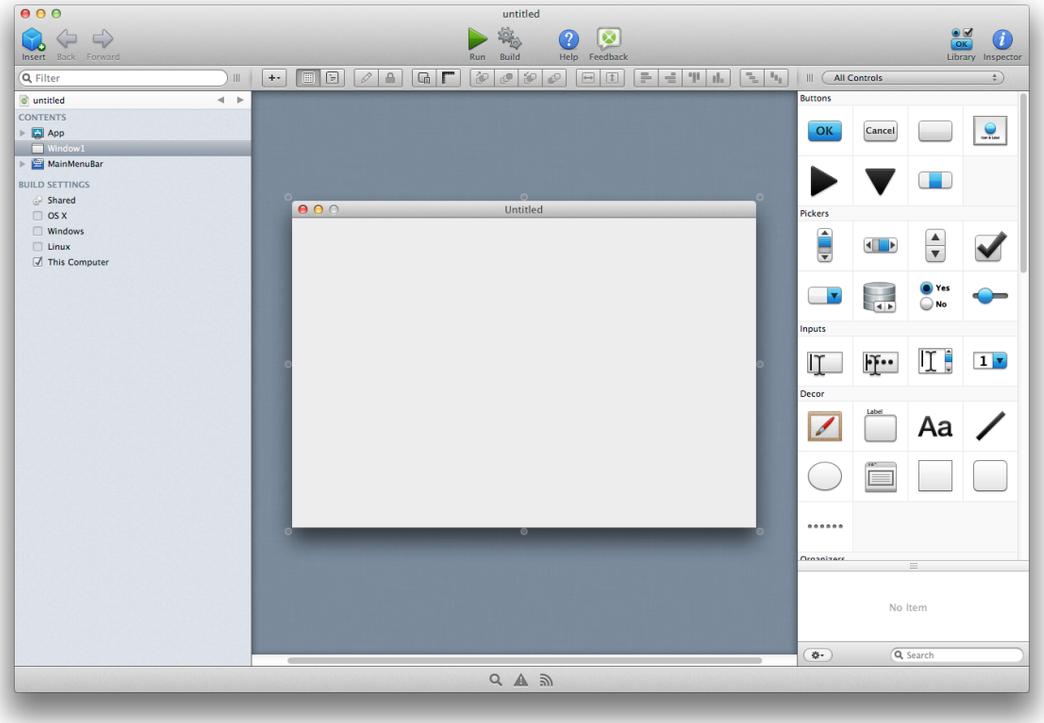
You can also drag a Xojo project onto the Xojo application icon (in the Dock or on the Desktop) or double-click a Xojo project to open it.

Workspace

IDE

Xojo is an Integrated Development Environment (IDE). This means that all its components (layout editor, code editor, compiler, and debugger) are all integrated into one package. In traditional programming languages, these items would each be a separate application.

Figure 2.2 The Xojo Workspace



When you open Xojo, by default all the items in your project are organized into a single window, called the Workspace. Within this window you can navigate among project items by clicking on their names. You can also choose open project items in their own tabs.

You have more than one project open at the same time; each will be shown in its own workspace.

Workspace

The main window that you see after choosing a project is called the Workspace. It is within this window that you will do all your Xojo development. You can open multiple workspace windows for a single project by using File ↠ New Workspace, which can be handy when working with multiple displays.

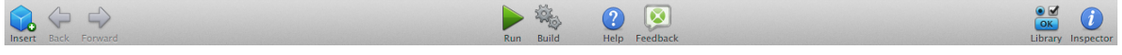
The Workspace consists of these areas:

- Toolbar
- Navigator
- Editor

- Library/Inspector
- Panels

These areas are described in the next sections.

Figure 2.3 Workspace Toolbar



Workspace Toolbar

The toolbar at the top of the main window is called the Workspace toolbar. It has these buttons:

- Insert
  Displays a menu of items that can be added to your project or to the selected project item.
- Back
  Used to navigate back to the last item that was displayed in the tab.
- Forward
  Moves forward through the navigation history for the tab.
- Run
  Used to run your project in “debug mode”.
- Build
  Creates a standalone build of your application for the specified platforms.
- Help
  Displays the Language Reference window.
- Feedback
  Opens the Feedback app.
- Library
  Toggles the display of the Library.
- Inspector
  Toggles the display of the Navigator.

You can hide the toolbar by selecting View ↠ Hide Toolbar.

Full Screen Mode

On OS X you can also click the “full screen” button in the window to put the workspace into full screen mode.

When in full screen mode, the window title bar and the main menu bar are hidden. To see the main menu bar, move the mouse cursor to the top of the screen; the menu bar will slide down. To exit full screen mode, click the full screen icon in the menu bar.

Navigator

What is the Navigator?

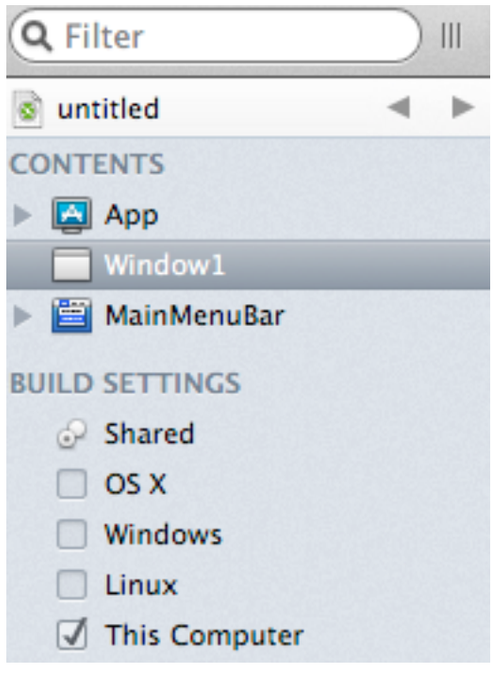
The Navigator is the area on the left sidebar that allows you to navigate through your project. The Navigator has these sections: Contents, Run, Profiles and Build Settings.

The **Contents** section contains the items in your project that were added using the Insert button or menu, such as windows, web pages, classes, menus and folders.

You can click on project items in the Contents section to view them or edit them.

When an item is selected, you can use the arrow keys to move the selection between items.

Figure 2.4 Navigator



The **Run** section is only visible in the Debugger tab when you are running your project.

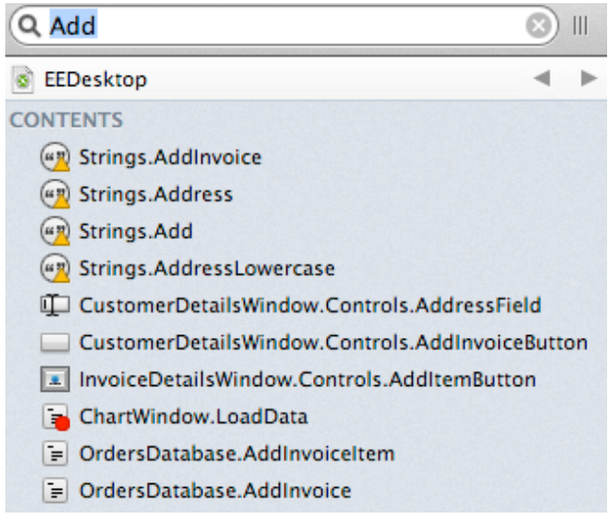
The **Profiles** section only appears when you have enabled the Profiler and have run your apps in debug mode to get profiler data.

The **Builds Settings** section contains all build-related information, including the build targets (and their settings) and active build steps used by Build Automation.

Filter

The Navigator has a Filter field at the top that can be used to filter what is displayed in the Contents section. Use the Filter to quickly show specific project items based on your criteria. For example, if you know you have a method that is called

Figure 2.5 Using the Filter



"AddInvoice", but you don't recall what class or window it is on, you can just type "Add" in the Filter field. The Navigator will then display any project item that have something containing "Add" in its name (e.g. a method, control name, constant or property).

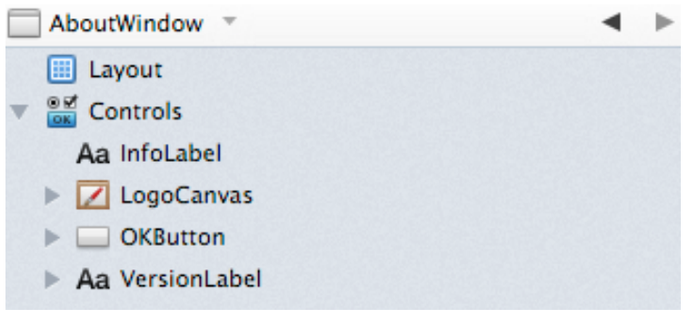
You can then click on the project item to see it or edit it.

Jump Bar

The Navigator displays items using a hierarchical list. The scope of what is displayed is controlled by the Jump Bar. By default, your entire project is in scope, so the Jump Bar displays your project name.

When you double-click on an item that is a parent, the Jump Bar changes to show the parent (this is referred to as "drilling into" the project item). Now only the items that are its children are displayed in the Navigator.

Figure 2.6 Using the Jump Bar



***Note**: The "Double click opens item in new tab" setting in Preferences alters the above behavior. When that setting is checked, double-click opens the item in a new tab, so use Option+⌘ when double-clicking on a project item to drill into it (on Windows and Linux, use Shift-Control).*

Click the name in the Jump Bar to see the entire history and to jump quickly to a specific item in the history.

The Jump Bar is incredibly powerful when used with tabs because it allows you to focus the tab on just the specific item you are working with.

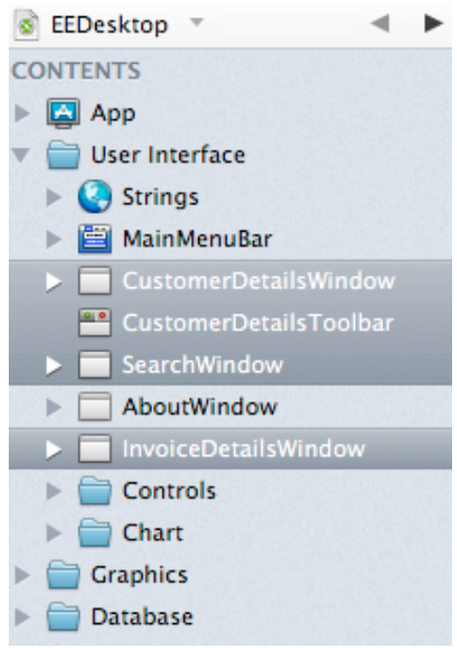
Contents

The Contents section shows the items in your project.

You can click on project items in the Contents section to view them or edit them.

When an item is selected, you can use the arrow keys to move the selection between items.

Figure 2.7 Navigator with Multiple Items Selected



You can do the usual things such as deleting, copying or pasting project items. You can also drag project items to reorganize them or to move them between project items.

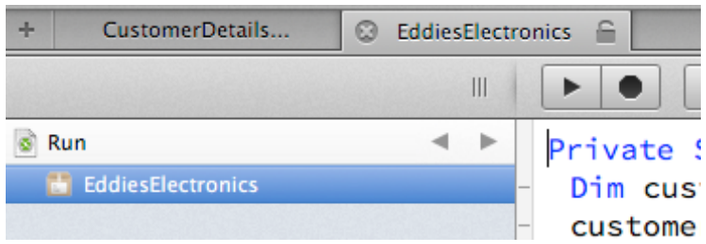
For example, you can drag a method from one class to another class to move it.

In addition, objects can be multi-selected. For example, you can select a Class and a Window, and drag them into a Folder.

Run

The Run section appears in a separate tab when you run your project, which causes the App name to appear and displays the Debugger.

Figure 2.8 Run Section Displayed in New Tab

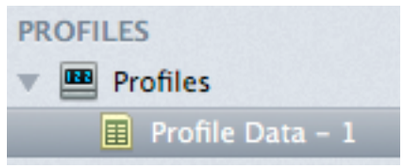


Should you navigate to another section of your project, clicking on this tab while a project is running displays the debugger.

Profiles

If you run your app with Profiling enabled, the Profiles section appears when the app quits. You can review the Profile results in this section.

Figure 2.9 Profiles Section



Build Settings

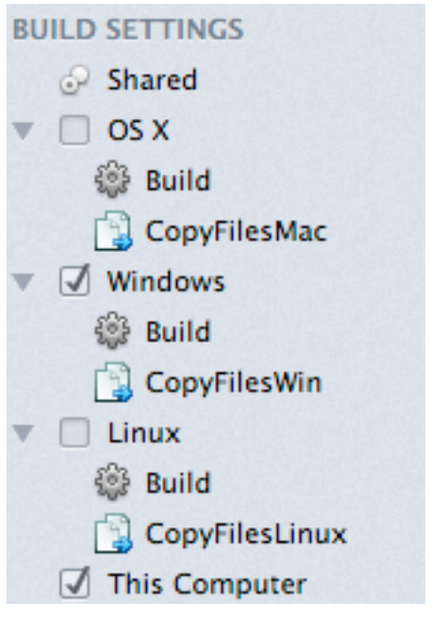
The Build Settings section is used to view and change the information needed to build and run your project.

The Build section also shows the various build OS targets available to you. “This Computer” is checked by default and contains the settings for the platform you are currently using.

Check the check box next to other targets to create a build for it the next time you build the project. Click on a build target name to change its settings using the Inspector.

The Build section also is where you manage your Build Steps. You add new Build Steps using the Insert button or menu. New Builds Steps can be added to the Contents area, but will not run when there. To activate a Build Step, drag it onto the appropriate OS target. Steps that are before the default “Build” step occur before the project is built and steps after it occur after the project is built.

Figure 2.10 Build Targets with Windows Build Selected and Displaying Build Steps



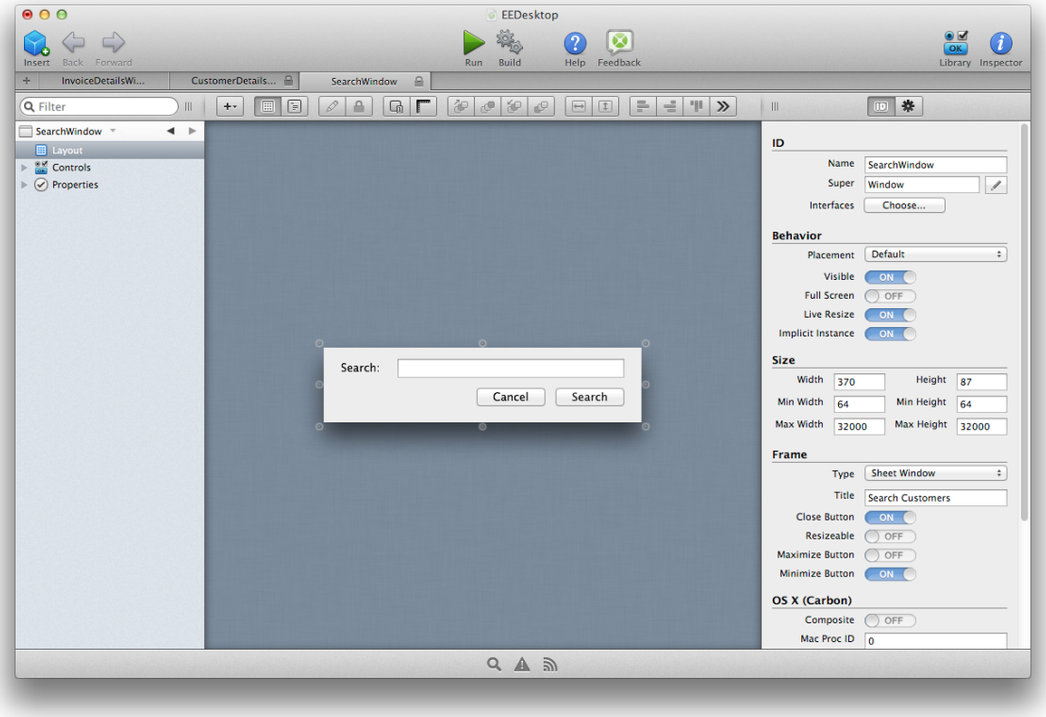
To disable a Build Step, drag it out of the Target and back into the Contents area.

Build Steps for pre-Xojo projects are automatically placed in the appropriate target when the project is opened.

Tabs

A tab is simply another view into your project. When you create a new tab, you get new Navigator and Editor areas.

Figure 2.11 Workspace with Multiple Tabs



You can navigate anywhere you want within a tab, just as you can when there are no tabs. Everything works exactly the same; you just now have multiple views into your project, each of which can show different information.

Tabs can be a great way to keep frequently used items available, particularly when used in conjunction with the Jump Bar.

To open a project item in a tab, you can use the contextual menu and select “Open in New Tab”. You can also use Option+⌘ when double-clicking on a project item to open it in a new tab (on Windows and Linux, use Shift-Control). There is also a preference to have project items open in a new tab when double-clicked.

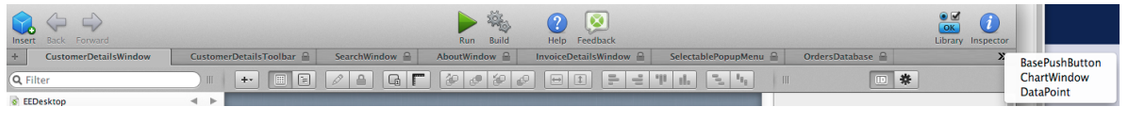
***Note**: The “Double click opens item in new tab” setting in Preferences alters the above behavior. When that setting is checked, double-click opens the item in a new tab. Using Option+⌘ when double-clicking on a project item allows you to drill into it (on Windows and Linux, use Shift-Control).*

Tabs can be locked or unlocked. A locked tab will not have its contents changed when you click on Filter or Search results, nor will it be changed when you use Go To Location. In those cases, a new tab (or the next available unlocked tab) are used.

Click the small “x” in the tab to close it. Hold down Option (on OS X) or Alt (on Windows or Linux) to close all tabs but the left-most one.

If you open more tabs than can be displayed in the tab bar, an “overflow” icon appears for you to click to get a drop-down list of the remain tabs. Select a tab from the list replaces the currently selected tab.

Figure 2.12 Tab Bar with Overflow Tabs



You can switch between tabs (wrapping around at the beginning or end) using ⌘+Shift+[or ⌘+Shift+].

Adding Project Items

Use the Insert button on the toolbar or the Insert menu to add new project items.

You can also add new project items by dragging a control directly from the Library to the Navigator. This creates a subclass of the control.

***Note**: Project items cannot be named "Xojo".*

Xojo shares the following project items for both desktop and web projects:

- Class
- Class Interface
- ContainerControl
- File Type Set
- Folder
- Module
- Database
- Build Steps
- Copy Files
- Script
- External Script

These project items are specific to desktop projects:

- Menu Bar
- Report
- Toolbar
- Window

These project items are specific to web projects:

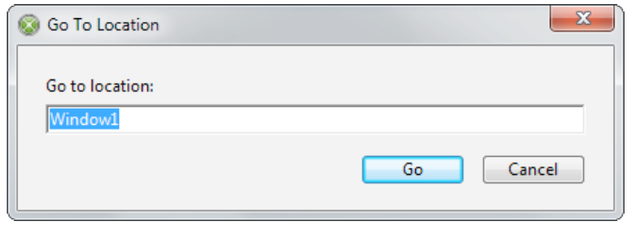
- Web Page
- Web Dialog
- Web Style

Go To Location

If you know its name, you can jump to a specific project item using the Go To Location feature.

Select Project ↠ Go To Location to display the Go

Figure 2.13 Go To Location Window



To Location window. Enter the name of the project item you want to jump to and press Return (or click Go). The Navigator will select the item.

Printing

You can print your source code using File ↠ Print from the menu.

When you have project items selected, only the selected items print.

If you do not have anything selected, the entire project prints.

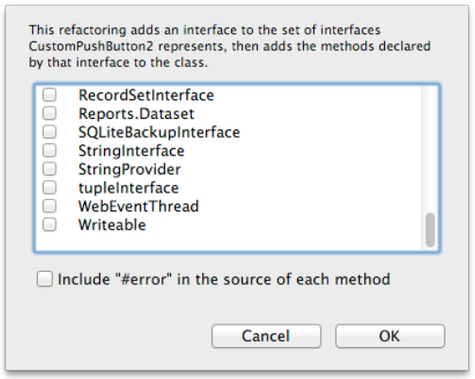
Working with Project Items in the Navigator

You can right-click (Control-Click on OS X) on any project item in the Navigator to display the contextual menu. From the contextual menu you have these options:

- Add to
  Use this command to add code items such as event handlers, methods, properties, etc. to the project item.
- Inspect
  Displays the Inspector for the project item.
- Cut/Copy
  Use to cut or copy the project item to the clipboard.
- Paste
  Pastes a project item in the clipboard into the Navigator, adding it to your project.
- Delete
  Deletes the project item. To put it in the clipboard, use Cut.
- Duplicate
  Creates a copy of the project item in the Navigator.
- Make External/Internal
  External items can be used to shared project items between your projects. Refer to the Development Guide, Chapter 5 for information on sharing project items.
- Encrypt/Decrypt
  Refer to Section 10, “Project Types, Formats and Templates” for information in encrypting and decrypting project items.
- Export
  Use to export a project item to a file. This is a useful way to share a project item with someone else.
- Print
  Prints all the source code for the project item.
- New Subclass
  Creates a new class in the Navigator that uses the project item as its superclass.

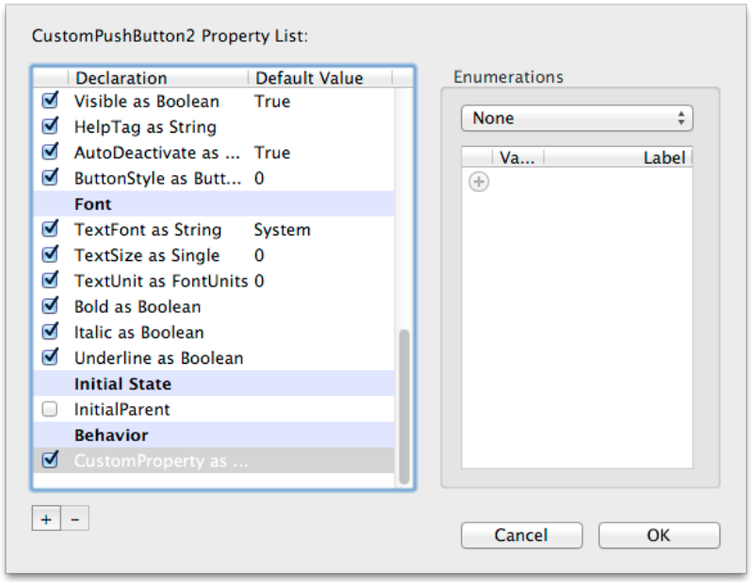
- Implement Interface
  Lets you select an interface that contains methods to implement for the class. When you select the interface (or interfaces) to add, the methods from the interfaces are added to the project item. Interfaces are covered in Chapter 5, Section 5.

Figure 2.14 Implement Interface Window



- Inspector Behavior
  Displays the Inspector Behavior list where you can customize the properties that display in the Inspector for classes and controls that you add to layouts. You can choose to display properties for your custom classes and subclasses and you can choose to hide properties that would normally be displayed. Add new properties using the “+” button on the left below the list of properties. Check or uncheck the check box next to the property name to determine if it is visible in the Inspector. You can also specify default values for any properties that are displayed. Drag properties in the list to reorder them or change their grouping in the Inspector. Use the Enumerations section to add a list of values that the user can select with a drop-down menu.

Figure 2.15 Inspector Behavior Window



- Edit Super Class
  Displays the parent (super) class for the currently selected class. This item only appears for subclasses.

Window and Web Page Layout Editors

Designing Your User Interface

The window and web page layout editors are the primary editors you use to design the user interface for your application.

Layout Area

The Layout Area displays as either a window or a web page, depending on the type of project. In either case, you add controls to the area by dragging them from the Library or the Navigator onto the Layout Area.

Toolbar

The Layout Editor has its own toolbar with the following features, in order from left to right:

Figure 2.16 Layout Editor Toolbar

Add

The Add button is used to add code-related items to the window or web page. This includes:

- Event Handler (refer to the Event Handler topic below)
- Menu Handler
- Method
- Note
- Property
- Computed Property
- Constant
- Delegate
- Enumeration
- Event Definition
- External Method
- Shared Computed Property
- Shared Method
- Shared Property
- Structure

View Layout

This button is grouped with View Code and is a toggle. When viewing a Layout, this button is selected.

When not viewing a layout, you can click this button to quickly switch back to the Layout Editor to see the last item you were working on.

View Code

This button is grouped with View Layout and is a toggle. When viewing a Layout, this button can be used to switch back to the code editor for the last item being edited.

Set Default Value

The Set Default Value button allows you to set the default value for various controls. Refer to the Default Values topic below.

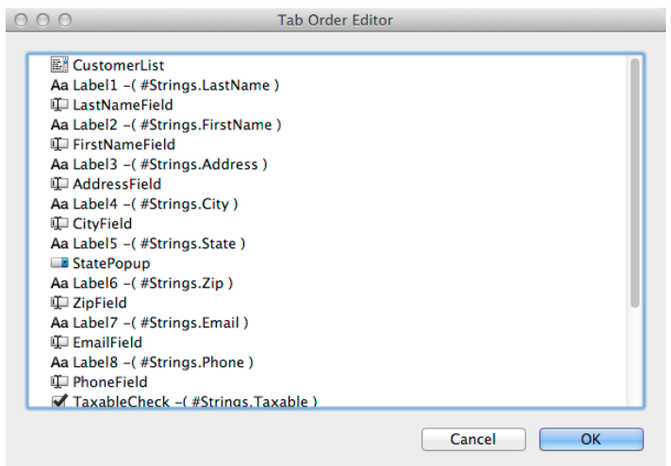
Lock Position

The Lock Position button is used to lock controls so that they cannot be moved. You can use this feature to prevent your user interface from being accidentally changed while editing it.

Show Tab Order

Show Tab Order displays each control in the Tab Order Editor. The controls on the layout appear in the editor list in tab order. You can drag controls around to change their tab order on the layout.

Figure 2.17 Tab Order Editor

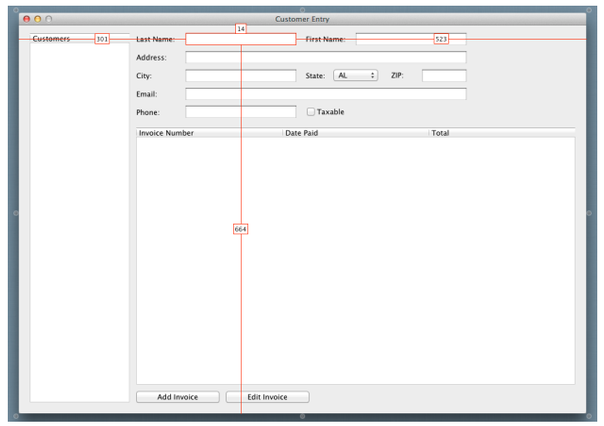


Show Measurements

The Show Measurements

button allows you to better visualize your layout. When you click enable Show Measurements view, you can move the mouse around your layout to see various measurements, such as how far a control is from the top of the window or web page.

Figure 2.18 Displaying Measurements



Select multiple controls to see various measurements, including distances between the controls.

Ordering

The Ordering buttons (Order Forward, Order Front, Order Backward and Order Back) are used to change the ordering of the controls on the layout.

Fill

The Fill Width and Fill Height buttons expand the selected control to fill the remaining space in its container.

Alignment

The Alignment buttons (Align Left, Align Right, Align Top and Align Bottom) are used to align controls on the layout with each other.

Spacing

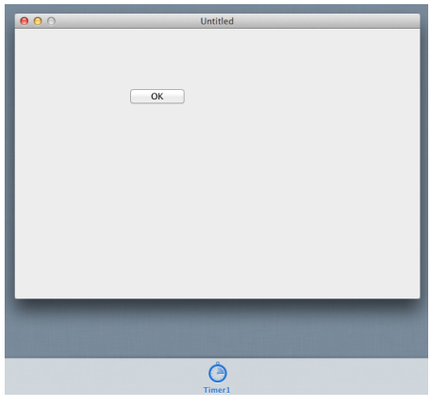
The Space Horizontally and Space Vertically buttons align the selected controls so that they are spaced equally apart.

Shelf

Some controls that are added to the layout are not actually part of the window or web page, such as a Timer.

When you add these types of controls to the layout, a Shelf is automatically displayed at the bottom of the Layout Editor and the control is added to it.

Figure 2.19 A Window with a Timer in the Shelf



In web applications, Web Dialogs also appear on the shelf.

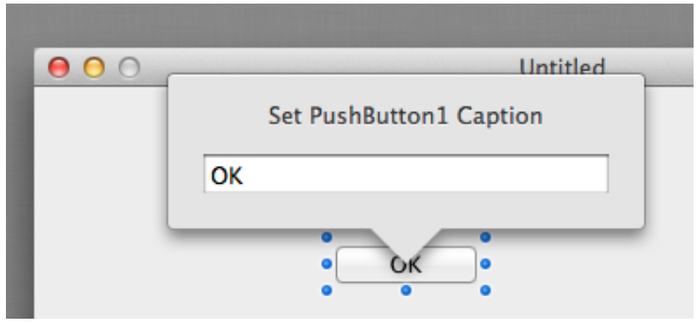
In desktop applications, a toolbar added to a window appears on the shelf.

Default Values

A control on the Layout Editor can have its default value specified by pressing *Return* while the control is selected, by clicking the **Pencil** rollover icon that appears when you move the mouse over a control, or by clicking the **Set Default Value** button on the Layout Editor Toolbar .

This opens a pop-out window to enter the default value. For example, with a PushButton, you can specify the Caption. To close the pop-out window, press *Return*, click outside the pop-out window or click the “Set Default Value” button on the Layout Editor toolbar.

Figure 2.20 Setting the Default Value for a PushButton



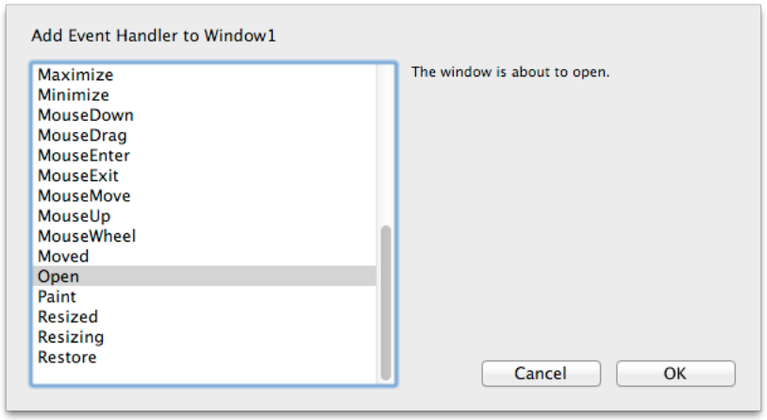
Event Handlers

To add an event handler to your control, you click the **Add** button and select **Event Handler**.

This opens the **Add Event Handler Dialog** which displays the events that are available for the control (or the window or web page if that is what was selected). You can click on each event to view the description of the event. Select one or more events and click **OK** to create corresponding Event Handlers. You can also press ⌘-A (Ctrl+A on Windows and Linux) to select all the event handlers to add at once.

Event handlers appear in the Navigator underneath the selected control. You can click on an event handler to see its code.

Figure 2.21 Add Event Handler Dialog

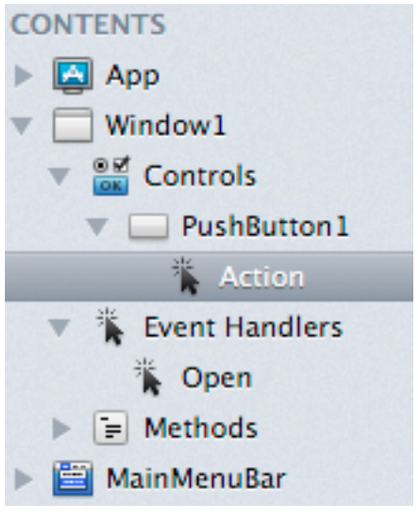


Alignment Guides

As you move controls around on a layout you will see additional alignment guides to help with positioning.

To disable the alignment guides, hold down Command while dragging (Control on Windows and Linux).

Figure 2.22 Navigator Showing Event Handlers



User Controls (Control Subclasses)

You can access user controls (subclass of built-in controls) from the Project Controls group of the Library.

Or you can simply drag the subclass from the Navigator to the Layout Area.

Working with Controls on the Layout

You can right-click (Control-Click on OS X) on any control on the layout to display the contextual menu. From the contextual menu you have these options:

- Add to
  Use this command to select “Add Event Handler” from the submenu to add event handlers to the control.

- Inspect
  Displays the Inspector for the control.

- Cut/Copy
  Use to cut or copy the control.

- Paste
  Pastes a control in the clipboard onto the layout.

- Delete
  Deletes the control. To put it in the clipboard, use Cut.

- Duplicate
  Creates a copy of the control on the layout. You can also create a copy of a control by clicking on it with the mouse button and then holding Command (Shift+Control on Windows and Linux) while dragging the mouse.

- Print
  Prints all the source code in the event handlers of the control.
- New Subclass
  Creates a new class in the Navigator that uses the control as its superclass.
- Select All
  Selects all the controls in the layout.
- Select
  This submenu contains all the controls on the layout. Use it to find an select a control by name or to select a control that it not visible in the layout due to layering.
- Lock/Unlock Position
  Locks the control at its position on the layout preventing it from being accidentally moved while editing the layout. To move the control, unlock it first.
- Edit Superclass
  If the control has a superclass, then this command takes you directly to the superclass so you can edit it.

Library and Inspector

Library

The Library is the list of all built-in controls that can be added to your layout. It also contains all the project items that you can add to your project.

To show the Library, click the Library button on the toolbar, select View ↠ Library from the menu or press the shortcut key (⌘-L on OS X, Ctrl-L on Windows and Linux). The Library displays on the right side of the workspace by default, but you can also set a preference to have it display as a floating palette.

The Library allows you to group the controls in a variety of ways. By default the controls are shown in large size, but you can also change the size, add group headers and change the sorting by using the small “gear” icon at the top of the Library. Display settings include:

Figure 2.23 Library Showing Web Controls

- Large Icons
- Large Icons and Labels
- Small Icons and Labels
- Large Icons and Descriptions
- Show Group Banners
- Sort Alphabetically

Hovering the mouse over any control displays its description in the Description Area at the bottom. If you do not need to see the description, you can resize the area to hide it.

You can also search the controls. Using the drop-down at the top of the Library you can choose to show only the controls from a specific group. Or you can use the Search field also at the top to quickly search for and show controls by name or type. For example, type “button” to see all the button controls.

You can drag controls from the Library onto the Layout Editor if it supports the control you have selected. You can also drag

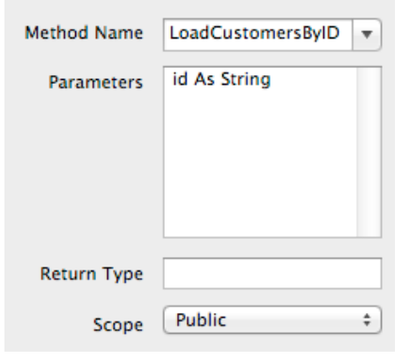
controls directly to the Navigator to immediately create a subclass. Lastly, you can also just double-click the control to add it to either the Navigator (as a subclass) or the Layout Editor, depending on which one has the focus.

Inspector

The Inspector displays information about the current selection. This could be properties of controls on a Layout Editor, project items in the Navigator, Build Settings or methods or properties when using the Code Editor.

The Inspector shares its space with the Library on the right side of the workspace. To show the Inspector, click the Inspector button on the toolbar, select View ↠ Inspector from the menu or press the shortcut key (⌘-I on OS X, Ctrl-I on Windows and Linux).

Figure 2.24
Inspector Showing Properties for a Method



What you see in the Inspector depends on what you have selected. For controls in the layout editor, you will see all the properties available for selected control. These properties are grouped by topic to make it easy to find what you are looking for.

The Inspector may more than one tab at the top depending on what is being viewed. More common items are contained in the "ID" tab and less commonly used (or advanced items) are in the "Gear" tab.

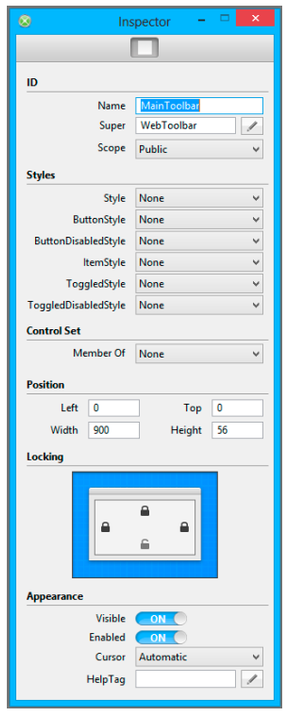
Figure 2.25
Inspector Showing Properties for a WebToolbar

Palettes

By default, the Library and Inspector display on the right side of the window and only one appears at a time. You can quickly toggle between the Library and Inspector using View ↠ Toggle Palettes in the menu.

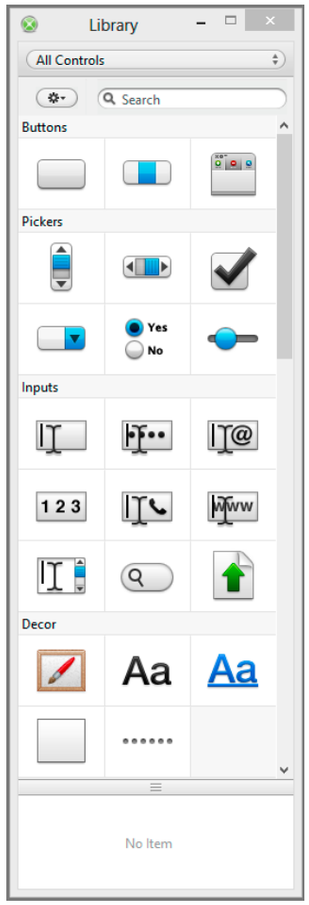
If you would prefer to position the Library and Inspector anywhere and have them both onscreen at the same time, you should use the preference to display them as Palettes.

Figure 2.26
Inspector as a Palette



In the General area of Preferences (Options on Windows and Linux) change “Show Library and Inspector” from “In the project window” to “As floating palettes”.

Figure 2.27
Library as a Palette

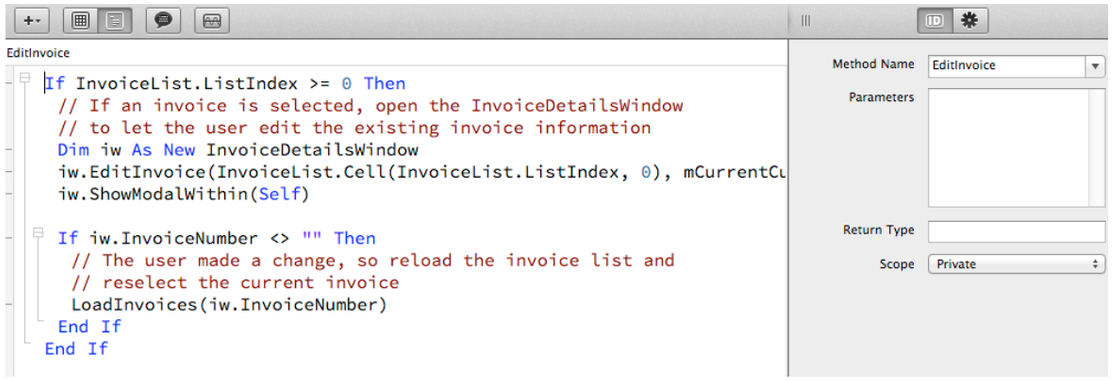


Section 6

Code Editor

To program an application, you write code using the Xojo programming language in the built-in Code Editor, which looks like this:

Figure 2.28 Code Editor



Code can be attached to nearly any item that is in your project, including windows, web pages, controls, classes, modules and more.

Toolbar

The Code Editor has its own toolbar with the following features, in order from left to right:

Figure 2.29 Code Editor Toolbar Buttons

Add

The Add button is used to add code-related items to the window or web page. This includes:

- Event Handler
- Menu Handler
- Method
- Note
- Property
- Computed Property
- Constant
- Delegate
- Enumeration
- Event Definition
- Shared Computed Property
- Shared Method

- Shared Property
- Structure
- Using Clause

View Layout

This button is grouped with View Code and is a toggle.

When viewing code, you can click this button to quickly switch back to the Layout Editor to see the last item you were working on.

View Code

This button is grouped with View Layout and is a toggle. When viewing code, this button is selected.

When not viewing code, you can click this button to quickly switch back to the Code Editor to see the last item you were editing.

Comment/Uncomment

Use this button to comment out the selected code in the code editor. If no code is selected, then the current line is commented out.

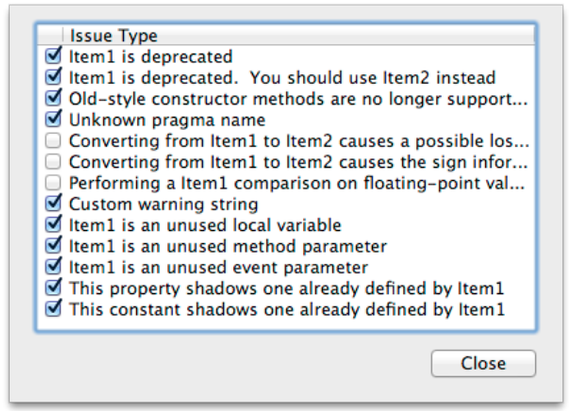
If the code is already commented, then this will uncomment the code.

Analyze Item

Use Analyze Item to validate the source code for the current project item. Analyze checks the project item for compile errors and warnings (such as unused variables).

You can control which warnings appear when analyzing a project by using the Analysis Warnings window. Select Project ↠ Analysis Warnings from the menu and then select the warnings you want to see.

Figure 2.30 Analysis Warnings Window



Editing Features

The Code Editor has several features designed to make it easy to write your code.

Syntax Highlighting

The Code Editor highlights your source code for you as you type. You can change the default colors for things such as keywords, string, integers, comments and more in Preferences.

Figure 2.32 Auto-Complete Offering "FolderItem" when typing "Fol"

```
Dim f As New FolderItem...
```

Auto Indentation

The Code Editor automatically indents your code as you type it. Code such as If/Then, While/Do loops, Select/Case and other commands are indented for you automatically.

Figure 2.31 Syntax Highlighting and Auto Indentation

```
Dim customerSearch As New SearchWindow
customerSearch.ShowModalWithin(Self)

If customerSearch.SearchText <> "" Then
  LoadCustomers(customerSearch.SearchText)
End If
```

Code that is indented is referred to as a code block. In the Code Editor, you can see that code blocks have a small "-" sign to their left. You can click this to collapse the code block (changing it to a "+'), hiding all the code within it. The code will still run and is part of your project, it is just not visible in the Code Editor.

Click the "+" to expand a code block.

Auto-Complete

As you are typing in the Code Editor you will often see three small ellipses that appear after the text you are typing. This indicates that the Code Editor has some auto-complete suggestions for you. When you see the ellipses, press the tab key to display a list of available commands that can be auto-completed.

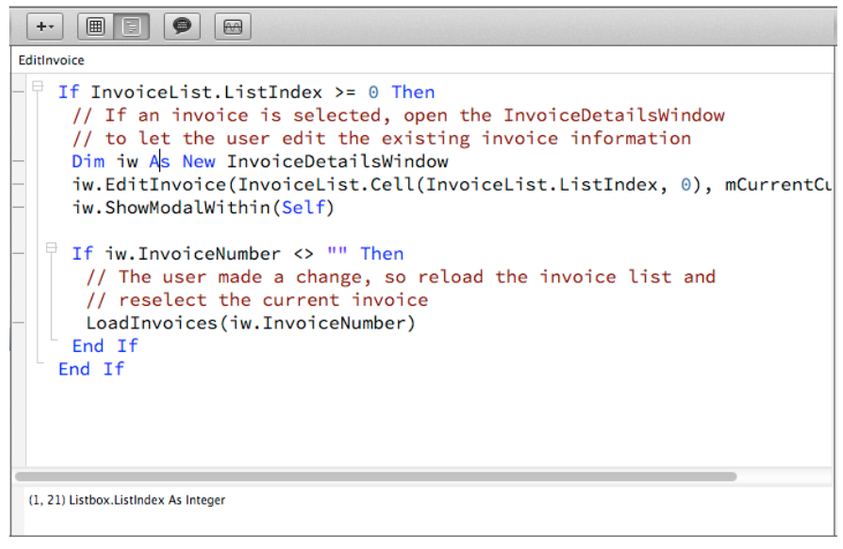
The auto-completion is smart and only tries to show you commands that are relevant based on the context of your code.

Syntax Help

There is a small area at the bottom of the code editor that is used to display syntax help.

The Syntax Help Area displays syntax information, method signatures, declarations and other

Figure 2.33 Syntax Help Area



information about the code that is under the mouse cursor.

Breakpoints and Bookmarks

The gutter on the left side of the Code Editor is used to set breakpoints and bookmarks. A breakpoint is a line of code that will stop execution when your app is run in the debugger. A bookmark is a quick way to jump to specific code. Next to each line of code, you will see a dash ("-") in the gutter indicating that this is a line that can have a breakpoint or bookmark. Click on the dash to toggle a breakpoint on or off. A breakpoint is indicated by a red circle. Option-click on OS X or Control-click on Windows/Linux to toggle a bookmark on or off. A bookmark is indicated by bookmark graphic. If you have both a breakpoint and a bookmark set for the line, the indicator appears as a red bookmark graphic.

The Project ↠ Breakpoint menu has menus to turn a breakpoint on or off, show all the breakpoints or remove all the breakpoints.

Similarly, Project ↠ Bookmarks has menus to turn a bookmark on or off, show all the bookmarks or remove all the bookmarks.

Contextual Menu

You can right-click (or control-click) on a variable, method, property, class or any item in your project to display the contextual menu, which offers several useful features:

- Cut/Copy/Paste/Delete
  Provides the usual cut, copy, paste and delete functionality.
- Select All
  Selects all the text in the code editor.
- Comment
  Comments (or uncomments) the selected code.
- Insert Color
  Opens the Color Selector window and allows you to choose a color, which is added to the code editor as a color literal.
- Go To
  Jumps to the definition of the selected item in the code editor. This works for variables, methods, classes and any other item you have created in your project.
  You can also quickly jump by double-clicking on a variable, method, class while holding down the command key (OS X) or control key (Windows and Linux).
- Switch To
  Switches to another event, method or property.
- Standardize Format
  Applies standard formatting rules to the selected text.
- Wrap in If/End If
  Wraps the selected code in an If Then / End If code block.
- Wrap in Do/Loop
  Wraps the selected code in a Do / Loop code block.

- Wrap in While/Wend
  Wraps the selected code in a While / Wend code block.

- Convert to Method
  Moves the selected code to its own method and switches you to the method.

- Convert to Constant
  Moves the selected to its own constant definition and switches you to the constant.

- Define Missing Method
  Creates a new method using the selected method as the method name. If you have supplied parameters, their types are automatically added to the method definition.

- Clean invisible ascii characters
  When cutting and pasting code, sometimes invisible ASCII characters can confuse the code editor. Use this command to clean up the selected text.

- Turn Break Point On/Off
  Turns the Break Point for the current line on or off.

- Help for / Open Language Reference
  Opens the Language Reference. If the command at the cursor is recognized as a Xojo command, then “Help for” appears, allowing you to jump directly to its entry in the Language Reference.

Toolbar

The Code Editor toolbar has these buttons:

- Add
  Allows you to add other source code elements, including: event handlers, menu handlers, methods, properties, notes, computed properties, constants, delegates, enumerations, event definitions, external methods, structures and shared methods, properties and computed properties.

- Comment
  Comments (or uncomments) the selected code.

- Check for Errors / Analyze Current Item
  Runs the Analyze Current Item command to check the current method for compilation errors and warnings.

Useful Keyboard Shortcuts

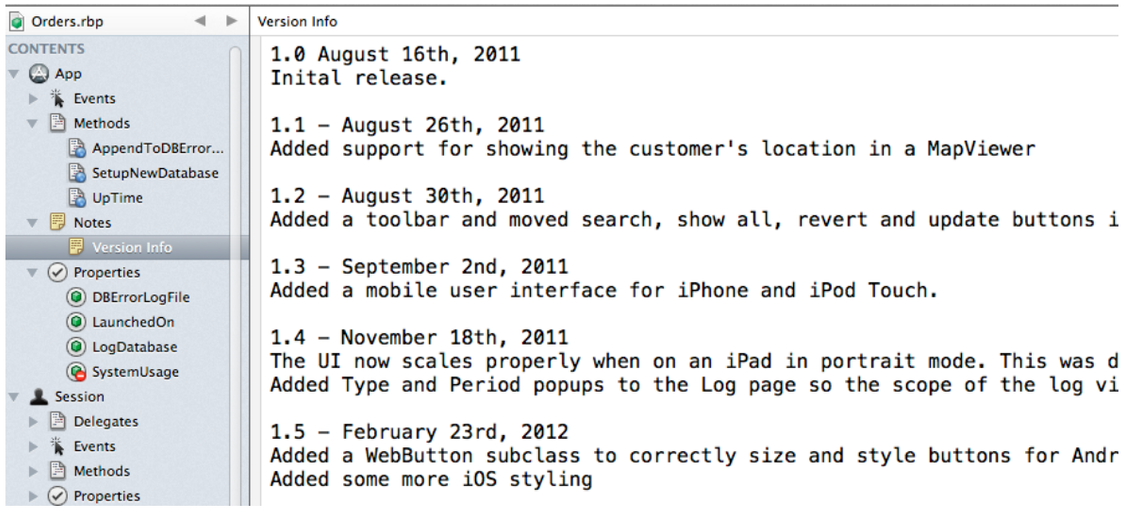
While working in the code editor, there are several keyboard shortcuts that will help make you more productive.

- Ctrl + Return (⌘-Return on OS X) completes the code block. For example, typing “If True” and pressing Ctrl + Return will automatically fill in the “Then” and the “End If” and place the cursor between them. This works for other code blocks such as Select, While, Do and For.

- Ctrl + Enter (Option-Enter on OS X) extends code to a new line by automatically adding the “_” character.

- Ctrl + \ (⌘-\) toggles the breakpoint on the line on or off.
- Ctrl + ' (⌘-') comments or uncomments the selected code lines or the line the cursor is on if no code is selected.

Figure 2.34 Notes Editor



- Ctrl (⌘ on OS X) + double-clicking on a method name takes you to the method in the Navigator.
- Tab is used to invoke the auto-complete. Press it when you see grayed-out text to auto-complete the text or when you see the ellipses to display the auto-complete items.

Notes Editor

The Notes Editor is a simple text editor where you can enter notes related to the project item. It is useful place to put technical documentation about your project, for example.

Section 7

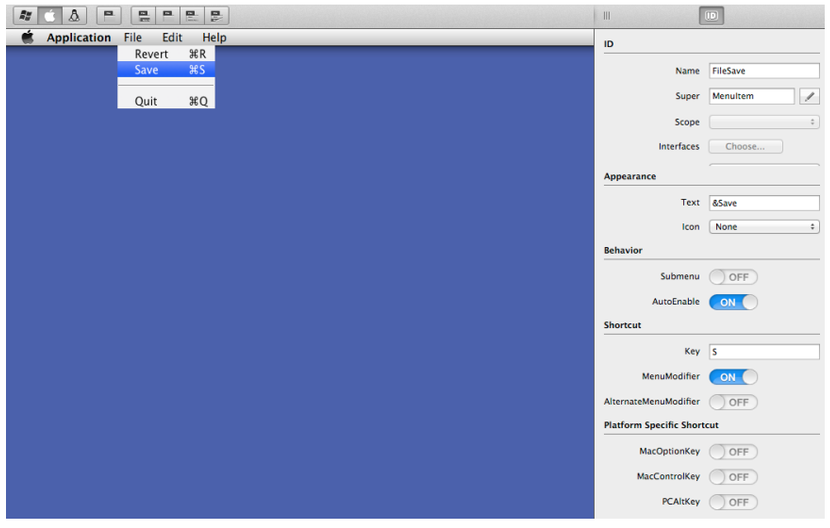
Other Editors

Although you will spend most of your time in the Layout and Code Editors, there are several other editors for menus, toolbars, reports, file type sets and containers.

Menu Editor

The Menu editor makes adding menu bars, menus, and menu items to your desktop projects easy. The standard Desktop Application template includes a menu bar that is used as the default menu bar for the entire application. You can accept the default or create other menu bars that are used for certain windows. In this case, you can assign a menu bar to a window.

Figure 2.35 Menu Editor Editing a Save Menu Item



The default menubar for a Desktop Application, MainMenuBar, includes File and Edit menus and the standard File and Edit menu items. The File menu has one menu item, Exit (on Windows) or Quit (on OS X and Linux). The properties of the Quit/Exit menu item are supplied, so that the menu item works automatically. You don't need to modify or add to the menu item's properties in order to enable it.

Similarly, the Edit menu is populated with Undo, Cut, Copy, Paste, Delete, and Select All menu items.

You can drag menus around to rearrange them or move them to entirely new menus. The toolbar has functions for adding new menus and menu items.

You can also use Cut/Copy/Paste to move and copy menu items to other areas of the menu.

Toolbar

The toolbar provides these commands:

- Platform Display

  The platform icons on the toolbar allow you to see how the menu looks on Windows, OS X and Linux.

- Create New Top-Level Menu

  Adds a top-level menu to the menu bar.

- Create New Menu Item

  Creates a new menu item under a selected top-level menu .

- Create Separator

  Creates a separator item below the selected menu item.

- Create Submenu

  Creates a submenu below the selected menu item.

- Convert Select Menu to a Top-Level Menu

  Use this command to convert the selected menu item to a top-level menu.

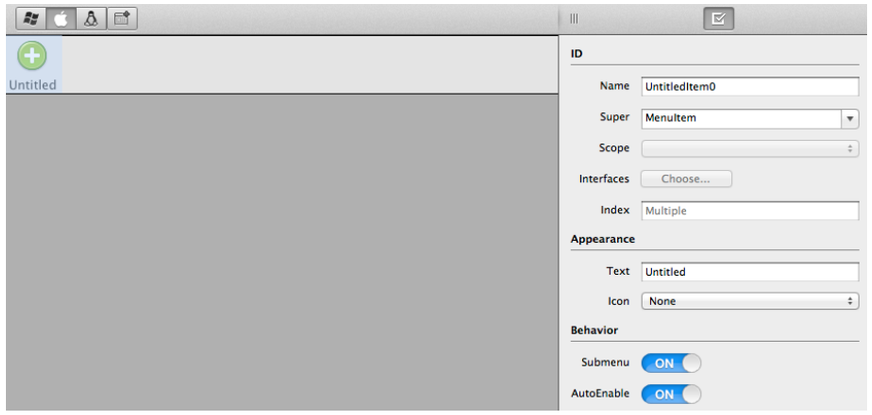
Inspector

The Inspector displays the properties for the selected menu or menu item.

Toolbar Editor

Also for desktop projects, the Toolbar Editor is used to design your toolbars for your desktop apps.

Figure 2.36 Toolbar Editor Editing a Button



Toolbar

The toolbar has these commands:

- Platform Display

  The platform icons on the toolbar allow you to see how the toolbar looks on Windows, OS X and Linux.

- Add

Adds new items to the toolbar.

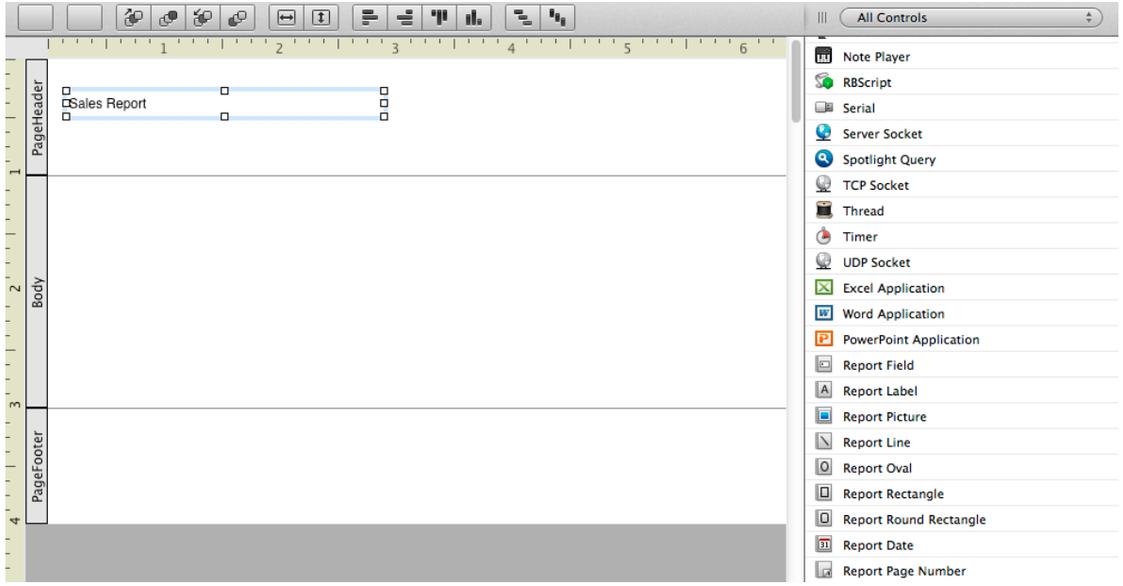
Inspector

The Inspector displays the properties for the selected toolbar item.

Report Layout Editor

The Report Layout Editor is used to design reports for desktop apps.

Figure 2.37 Report Editor



Toolbar

The toolbar has these commands:

- Add Group
  Adds a new group section around the body.

- Add Page Header/Footer
  Adds a new page header and footer to the report.

- Ordering

Specify the ordering of the controls on the report.

- Fill

Fill the report control to available space.

- Alignment

Adjust report control alignment.

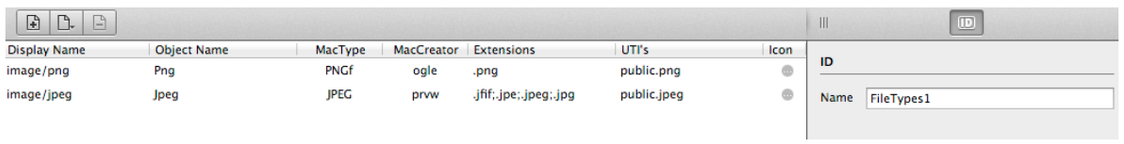
- Spacing

Adjust report control spacing.

Library and Inspector

The Library displays the report controls that can be added to the report layout. The Inspector displays the properties for the selected report control.

Figure 2.38 File Type Sets Editor



File Type Sets Editor

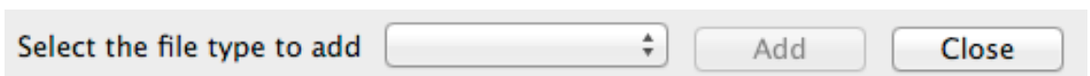
The File Type Sets Editor is used to define the file types that your desktop application supports.

Toolbar

The toolbar has these commands:

- Add Custom File Type
  Used to add a custom file type for your app. Double-click on a row to use the inline editor to add the values.

Figure 2.39 Prompt to Choose Additional Standard File Types



- Add Standard File Type
  This button displays a drop-down menu of commonly used standard file types. Click the “More” option in the menu to display a prompt at the bottom with a full list from which to choose.
- Remove File Type
  This button is used to remove the selected file type from the list.

ContainerControl Editor

ContainerControls are special project items that allow you to combine multiple controls into a single control to add to either a window or web page.

The ContainerControl Editor works the same as the Window Editor for desktop projects or the Web Page Layout Editor for web projects.

Panels

Overview

There are three buttons at the bottom of the workspace that open panels for Find, Errors and Messages.

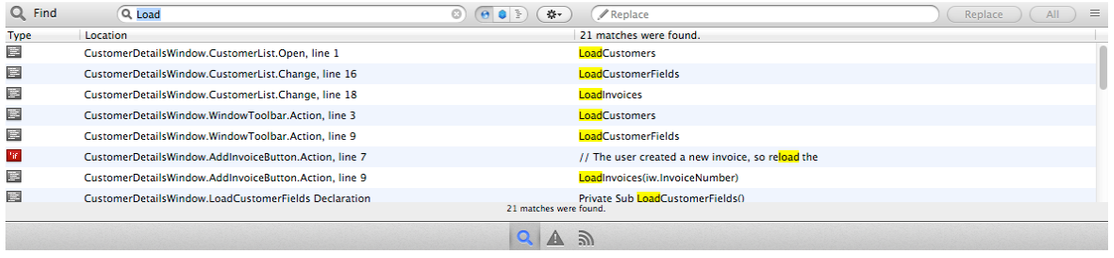
Figure 2.41 Panel Buttons

Find

The Find Panel is used to search (and optionally replace) text in your project. It searches instantly as you type text in the Find field. You can use the scope control to change the scope of the search and the “gear” button to change the matching criteria.

Figure 2.42 Scope Control

Figure 2.40 Find Panel



The scope choices are:

- Global (entire project)
- Current Project Item
- Current Method or Event (if applicable)

The matching criteria:

- Whole Word
  Only searches for your text as a whole word.
- Match Case
  Does a case-sensitive search.
- Use RegEx
  Use a Regular Expression to search.

Click once on a Find result to jump to where it is in your project. Find only searches the items displayed in the Navigator. If you have filtered the navigator using either the Filter or JumpBar, then Find only finds based on what is displayed.

Errors

The Errors Panel displays compiler errors and warnings. This panel appears automatically when you **Run** or **Build** if there are

compiler errors. Click on the issue to jump to where it is in your project.

Warnings are only displayed when you use the “Analyze Project” or “Analyze Current Item” commands.

You can choose to display errors or warning by type or by location using the selector at the top of the Error Panel.

Figure 2.43 Errors Displayed by Type

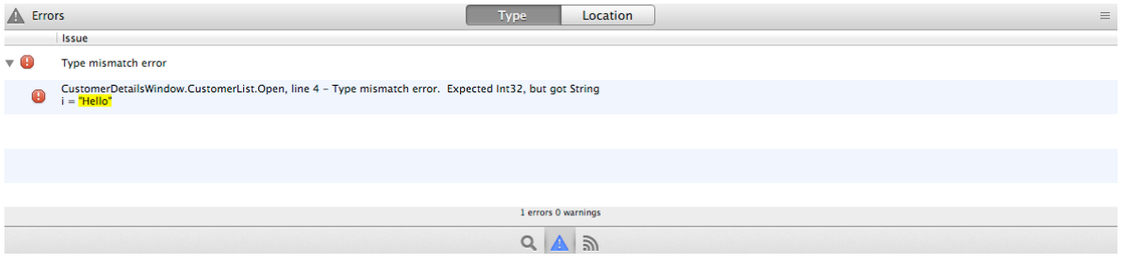
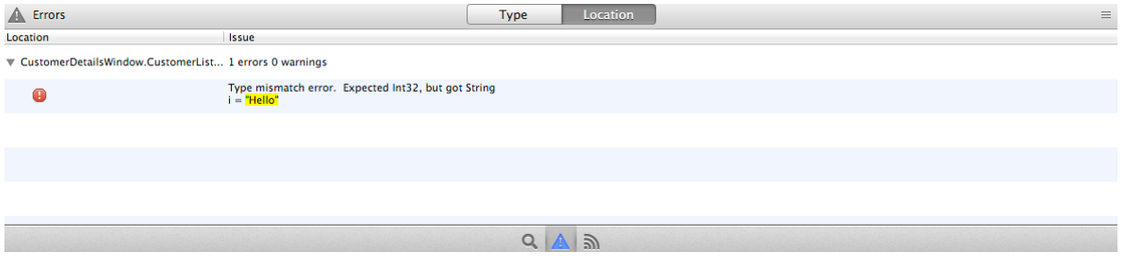


Figure 2.44 Errors Displayed by Location



You can choose which warnings appear by using the Analysis Warnings window (Project → Analysis Warnings).

Messages

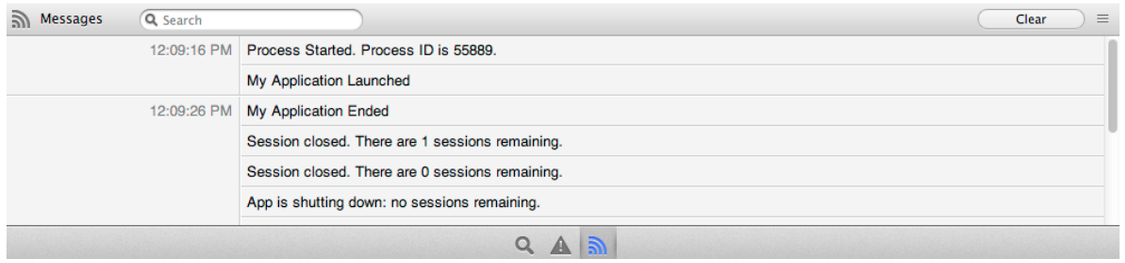
The Messages Panel is primarily used when running your project using the debugger.

When you run your project, messages for Application Launch and Application End are automatically created. Additional system messages may also be displayed here.

In addition, the output from the System.DebugLog method appears in the Messages Panel allowing your applications to generate their own logging messages.

You can search for messages using the search field. The Clear button clears all the messages in the list.

Figure 2.45 Messages Panel

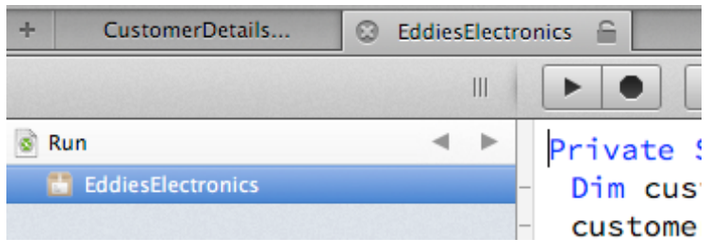


Running Your Applications

While working on your project, you are going to want to test it out. You can do this by clicking the Run button on the toolbar or selecting Project ↠ Run from the menu.

When you run your project like this, you are running it in **debug** mode. In this mode, Xojo is communicating in the background with your application. In the Workspace, a separate tab (containing the debugger) appears with the name of your application.

Figure 2.46 The Debugger Tab Showing an Application Running in Debug Mode



When this tab is selected, the Debugger displays. In this tab, the Navigator displays the Run section and the name of the app being debugged.

Debugger

The debugger can be used to watch how your application runs. It is divided into three main areas:

- Stack
  Displays the order that the code in your application was called.
- Variables
  Displays variables and their current values.
- Source Code
  Displays the source code that is currently running.

To see the Stack, Variables and currently running source code, you can set a breakpoint in your source code or you can click the pause button on the toolbar.

Toolbar

The debugger toolbar has these commands:

- Pause
  Pauses the app at the currently running line of code. The debugger will display the line of code highlighted in gray.

- Stop
  Stops the running app, closing the debugger.

Figure 2.47 Debugger Toolbar

- Step
  When Paused, steps through the code one line at a time.

- Step In
  When Paused, if the current line of code is a method then this action steps you into the first line of code in the method.

- Step Out
  When Paused, if you are within a method, this action runs the rest of the code in the method and pauses debugging at the line of code after the method call.

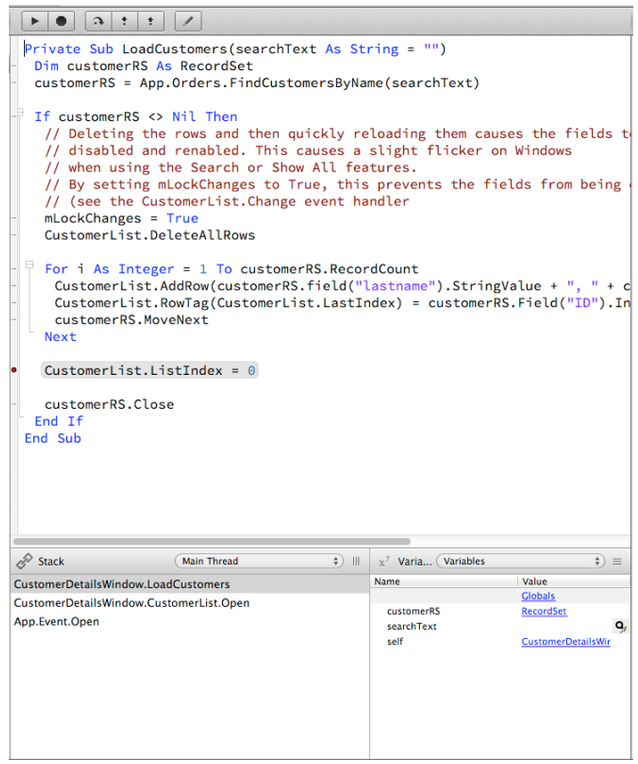
- View Source
  Takes you to the project item and displays the source code so that you can edit it. Changes you make to the code do not take affect until you run the project again.

Using the Debugger

You have three choices to Pause the debugger so you can review running code:

- Use the Pause button, which does not allow you to control where the code will pause.
- Set a breakpoint in your code.
- Use the Break command in your code.

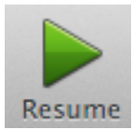
Figure 2.48 The Debugger



Once you are in the debugger, you can view the source code but not make any changes to it. To edit code, use the View Source button to find the code in the project to edit. Any changes you make do not take affect until you run the project again.

To resume running your application after it has paused, click the Resume button on the toolbar, which appears in place of the Run button.

Figure 2.49 Resume Button

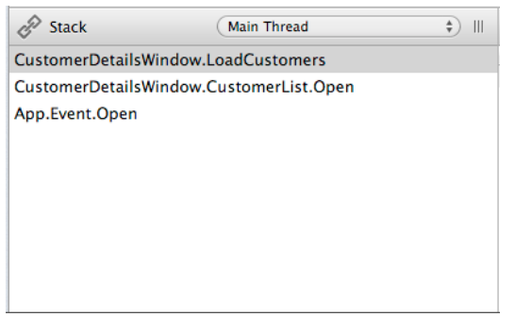


You can also completely stop the running application by clicking the Stop button on the toolbar. This does not take you to the debugger since the app has been terminated, but it can be useful if you need to quickly stop the app because of an infinite loop or some other programming error.

Stack

The Stack area (lower left corner) displays the name of the objects in the order they were called with the current object at the top of the list. You can click on other items in the list to view the code that previously ran.

Figure 2.50 Stack Area

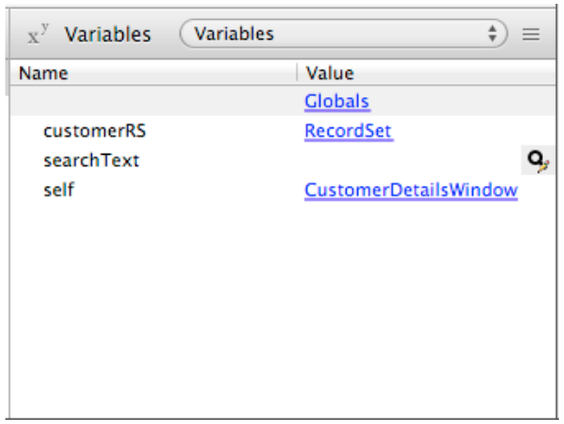


The PopupMenu in the Stack area usually just displays “Main Thread”. If you are specifically using Threading, you can use this popup to look at code that is running in other threads.

Variables

The Variables area (lower right corner) displays the variables that are local to the currently running method. The left column displays variable names and the right displays their values. You can click on values on the right side to display them in more detail. Some values (such as numbers and strings) can have their values edited so that you can affect the outcome of subsequent code.

Figure 2.51 Variables Area Showing for LoadCustomers Method



As you click to view details of variables, the PopupMenu in the Variables section changes to display what is being viewed. You can use this popup to go back to the prior variables.

In addition, the “Globals” item in the Variables section will let you review global values in the app, such as modules and the contents of the App object.

Setting Breakpoints

In the Code Editor, the gutter in the left margin shows horizontal lines for lines of code that can have their breakpoint set. To set the breakpoint, just click on the line. This causes a red dot to appear indicating that a breakpoint is set. To turn off the breakpoint, click on the red circle. You can also set breakpoints using Project ↠ Breakpoint ↠ Turn on (⌘+\ on OS X, Ctrl-\ on Windows and Linux).

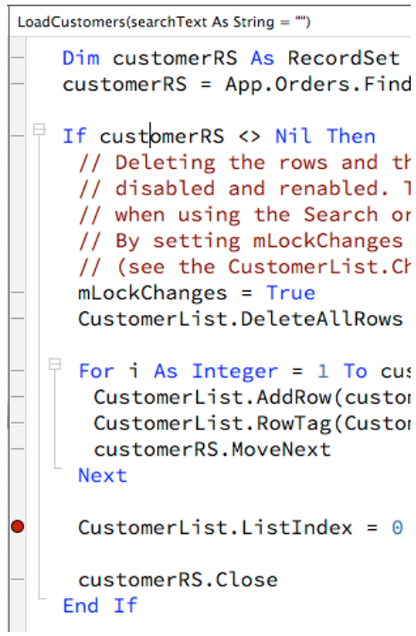
To disable all the breakpoints in your app, use Project ↠ Breakpoint ↠ Clear All. To view all the breakpoints in your project,

choose Project ↠ Breakpoint ↠ Show All, which displays the breakpoints in the Find panel.

You can also use the Break command to cause your application to pause. When your code reaches a Break command, the debugger pauses at that line. In a standalone app, the Break command is ignored.

You can use the Break command to conditionally pause your app and activate the debugger based on specific situations.

Figure 2.52 A Breakpoint Set in the LoadCustomers Method



For example, if your app is in a loop,
you may only want to break into the debugger when a certain value is reached:

```
For i As Integer = 1 To 100
  If i = 50 Then Break
Next
```

Project Types, Formats and Templates

Projects

Xojo projects are a document that contain all the items that make up your application. You can have several projects open at once and you can have several windows open per project.

When you choose to create a new project, you have these types from which to choose: Desktop, Web, Console or iOS.

Desktop

Desktop projects allow you to create applications with a graphical user interface that run on Windows, OS X and Linux desktop operating systems.

Web

Web projects allow you to create applications that run on the web. Users interact with these applications using a web browser.

Console

Console projects are used to create text-based applications that run from the command line, terminal or as a background application (service or daemon). Windows, OS X and Linux are supported.

iOS

iOS projects are used to create apps that run on iOS devices such as iPhones and iPads.

Project Formats

You can save your projects in a variety of formats, including the binary single-file format (xojo_binary_project), a text-based XML format (xojo_xml_project) and a text-based format (xojo_project) using separate files for project items.

***Note**: Xojo can continue to open and save existing projects in Real Studio project file formats.*

In Preferences you can change the format that is used by default.

Binary (xojo_binary_project)

This is a binary file format. Your project is stored in a single file (except for external items such as pictures) that is easy to distribute. This is the only file format that can be used without a license.

XML (xojo_xml_project)

The XML file format is simply an XML representation of the binary file format. It is also a single file, but it is entirely XML. Being

XML, this file format can be larger than the same file saved as Binary.

Text (xojo_project)

The Text file format saves your project as separate text files, one for each project item. Because of this, make sure that your save each Text project in its own folder and do not try to save multiple text projects into the same folder.

The Text file format is ideal for use with version control systems such as Subversion or Git because you can see exactly what files have changed and track the history of changes to files.

This format uses the following extensions for your project items:

- xojo_code
  Contains source code project items such as modules, classes and web pages.
- xojo_window
  Contains windows and container controls from desktop projects.
- xojo_menu
  Contains menus from desktop projects.
- xojo_toolbar
  Contains toolbars from desktop projects.
- xojo_report
  Contains reports from desktop projects.
- xojo_resources
  Contains binary information such as icons, encrypted items and other binary data.
- xojo_uistate
  Contains the UI layout settings such as window positions and sizes.

When used with a version control system, you should add all the files to your repository except for the xojo_uistate file.

When you delete (or rename) files or folders from a text project format, they are left on disk so that you can manually handle the change using your version control software.

External Items

Certain project items are always stored as external items and are not included in the project file. These are: pictures, movies, sounds, text files and other files added to the project.

It is also possible to include windows, classes, or modules in a project that are actually stored as external files. This feature allows more than one project to share the project item. When you modify an external project item and then save your project, your changes are written out to the external file on disk. When you open any other project that refers to the same shared file, that project will reflect your changes.

To convert a project item to an external item, select it in the Navigator and right-click (Control-click on OS X) to display the contextual menu for the item and choose Make External. This displays a Save File dialog box. Navigate to the directory in which you want to save the item, name it, and click Save. When you have successfully saved the item, it is shown in the Navigator with a shortcut badge and its name is in italics.

To add an item to a project as an external item, hold down the Alt key (Option+Command key on OS X). The File ↠ Import menu command changes to Import as External. Choose that command and select the item to be imported. You'll know you've done it correctly because the icon in the Navigator appears in italics with a shortcut badge.

If an external project item is set to read only (Windows or Linux) or locked (OS X), you can't modify that item within the project, though you can still view it. This provides a convenient way to protect external items (which may be shared by many projects) from accidental modification. Note that if an external file is changed on disk by something other than Xojo — such as a version control system — you'll need to close and re-open the project to reload that item.

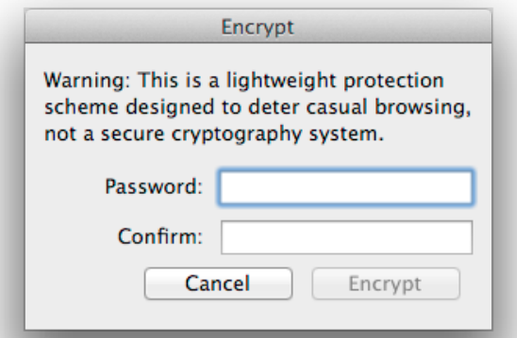
Encrypting Project Items

If you want to distribute a copy of project items for others to use but you do not want them to be able to view or edit your code, use the Encrypt command to protect the object prior to exporting it. An encrypted item displays in the Navigator with an icon containing a small key in its lower-right corner.

You can encrypt (protect) or decrypt (unprotect) a project item while it is in your project using the contextual menu "Encrypt ProjectItemName...". An encrypted item cannot be opened and no one can access any code (or layout) associated with it unless they know the password to decrypt it.

When encrypting an item, you supply a password that

Figure 2.53 Encrypt Project Item Window



can be used to decrypt it later. Do not forget the password as there is no other way to decrypt the item.

Importing Project Items

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

For code, open the method that you wish to import code into and drag the text clipping into the body of the method. You cannot use the File ↠ Import command to import text files into the body of a method.

To delete a file that has been imported to the project, highlight it in the Navigator and press the Delete key on the keyboard or choose Edit ↠ Delete. You can also delete a file in the Navigator by Control-clicking on the item and choosing Delete from the contextual menu.

Exporting Project Items

The code for methods, events, constants, properties, and so forth can be dragged to the desktop as text clippings or into a text or word processor (OS X only). You can also of course copy code to the Clipboard and paste it into a text editor or word processor.

You can export your source code to an XML file or a binary file format using the Export… menu command, which can be useful for sharing code.

Collect Project Items

Some of your project items may be external to the actual project files. These can be things such as graphics or even project items that are shared across multiple projects. For distribution, it can be helpful to collect all project items from their various locations next to a saved project.

External items remain as external items, but they are now grouped (pictures, data, scripts, etc.).

This process does not rename items on disk, so duplicate filenames will be reported as errors.

Collecting a project does not save the project. Typically you will want to do a “Save As” after collecting the project.

Choose File ↠ Collect Project Items to collect the current project.

Templates

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

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

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

Here are descriptions of the project templates that are included by default. Although these templates are built-in, but they can be overridden by providing templates that use these names and placing them in the Project Templates folder.

- CGIApplication: This is a specialized version of the Console Application template that is intended as a web application that interfaces with the Apache server. Its App class is derived from the CGIApplication class (in the project) that in turn is based on the ConsoleApplication class. CGI (Common Gateway Interface) is how a web application works with Apache.

  More detailed notes on the CGIApplication are found in the “Notes” section of the App class in the Template project. The HTTP module contains a custom class that manages HTTP requests. It contains properties and methods that you will use to build the interface to Apache.

- EmptyService: This is a service application template. Its App class is based on the ServiceApplication class. It also runs in the background with no user interface. Please see the Notes section for the ServiceApplication class in the Language Reference for more information.

- Event Driven Console: This is also a template for a Console Application. Its App class is also based on the ConsoleApplication class and it includes a custom class, MyApplication, that contains shell methods and properties for a Console Application. See the Notes section for the Console Application class in the Language Reference for information on how a console application works.

Changing Desktop, Web and Console Default Projects

When you create a new desktop, web or console project, a simple project is opened with just the basics to get you started.

You can use your own project files as a substitute for the simple projects used for desktop, web and console projects.

To do so, create projects (in Binary format) and add the project items you want to include by default. Save them with the following names in the Project Templates folder:

- Default Desktop Project
- Default Web Project

- Default Console Project

For example, if you create Default Desktop Project that has two windows and some standard modules and classes that you typically use in all your projects, they will be immediately available when you select Desktop from the Project Chooser.

Preferences / Options

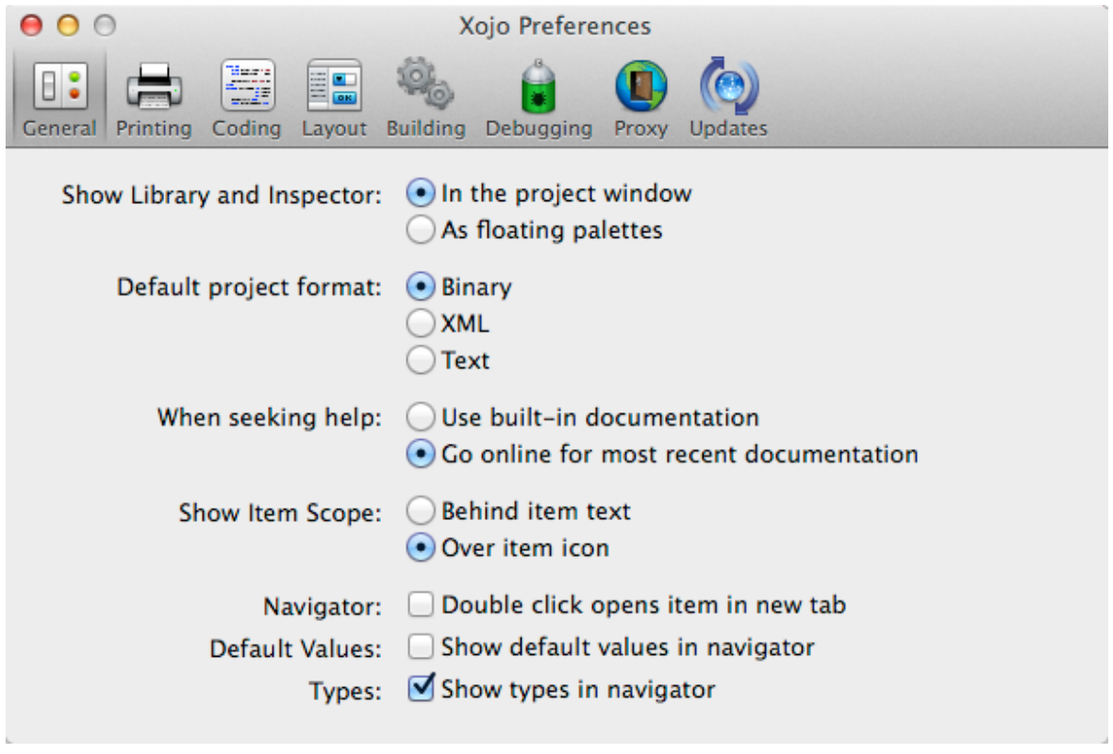
There are several preferences that you can use to alter default behavior. The preferences are grouped into these areas: General, Printing, Coding, Layout, Building and Debugging.

On Windows and Linux, the Preferences window is called Options.

General

The General area contains the most common preferences.

Figure 2.54 General Preferences



Show Library and Inspector

Specifies how you want to see the Library and Inspector. By default they appear on the right side of the Workspace, both sharing the same area. You can choose to display them as

floating palettes so that they are both visible on the screen at the same time and do not take up space in the Workspace.

Default project format

When you save your projects, Xojo uses the standard single-file binary format by default (xojo_binary_project). This format is easy to distribute, but does not work well with version control systems such as Subversion or Git.

Use this setting to change the default project format.

When seeking help

Xojo looks up commands using the Language Reference. By default, it uses the online Language Reference so that you get the most up-to-date information, but this does require an Internet connection.

If you would rather use the local copy of the Language Reference, change this setting.

Show Item Scope

This setting determines how the scope for properties and methods is displayed in the Navigator. Choosing “Behind item text” has the scope color displayed as the background color behind the item in the Navigator. Choosing “Over item icon” displays an overlay on the item in the Navigator indicating its scope.

Figure 2.56 Scope: Over item icon

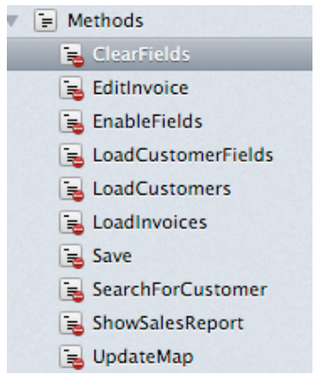
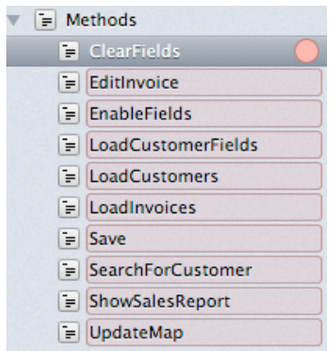


Figure 2.55 Scope: Behind item text



When closing last window

This setting is available on Windows. When “Keep Xojo running” is selected, a new icon appears in the Taskbar Notifications area (also known as the System Tray) and Xojo remains running when you close the last Workspace window (instead of quitting). You can right-click on the Notification icon to display a menu with “New Project”, “About Xojo” and “Exit”. “New Project” opens the Project Chooser window, “About Xojo” displays the About window and “Exit” quits Xojo completely.

Figure 2.57 When closing last window Preference

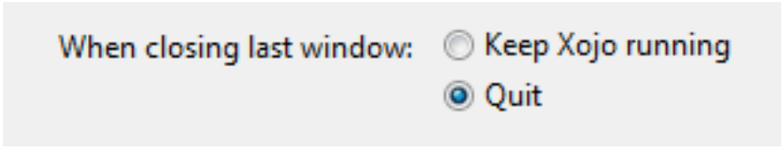


Figure 2.58 Xojo in the Taskbar Notification Area

Navigator

The “Double-click opens item in a new tab” setting controls the behavior of double-clicking project items in the Navigator. The default behavior is to “drill into” the project item using the Jump Bar.

Check the box to instead open the project item in its own tab when it is double-clicked.

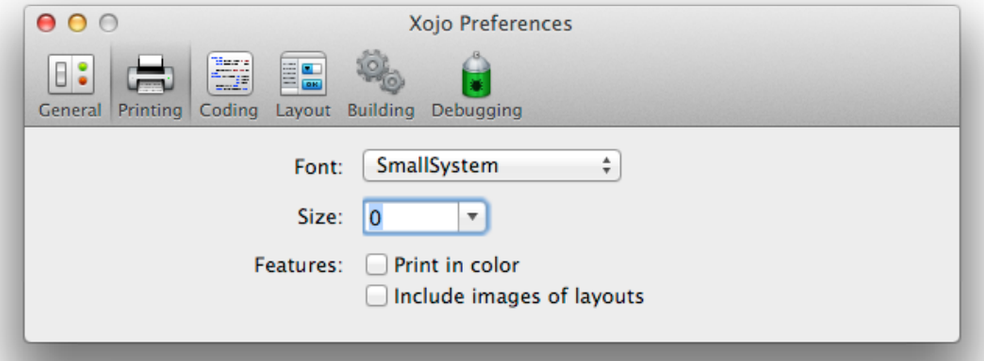
Check “Show default values in navigator” to display default values of properties in the Navigator.

When “Show types in navigator” is checked, the types of properties is displayed in the Navigator.

Printing

The Printing preferences control how your code looks when it is printed.

Figure 2.59 Printing Preferences



Font

Choose the font to use for the printed text.

Size

Choose the font size to use for the printed text.

Print in color

Check this box if you want the source code printed using the syntax colors specified in the Coding preferences.

Include images of layouts

Check this box if you want printouts to include images of your user interface layouts (windows and web pages).

Coding

The Coding preferences specify how the code is displayed in the Code Editor.

Font

Select the font to use in the Code Editor from the list of available fonts installed on your system.

Size

Specifies the font size to use in the Code Editor.

Syntax colors

You can customize the
colors of various parts of your source code. Select a particular syntax and then click the color box to choose a color for it.

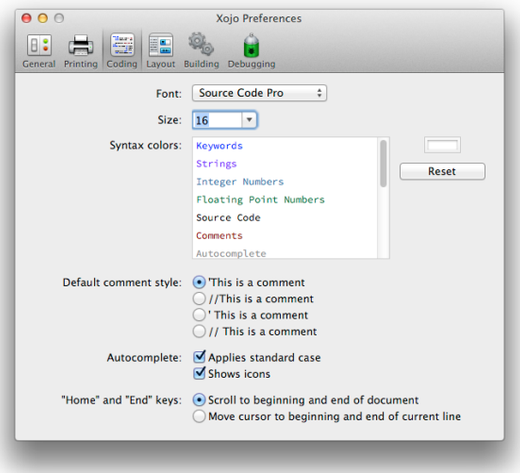
Default comment style

This setting determines the comment style that is used by the Comment button and function on the contextual menu.

Autocomplete

These two options allow you to control how Autocomplete works.
Apply standard case will use the correct case when autocompleting (MsgBox instead of msgbox, for example).

Figure 2.60 Coding Preferences



Show icons will show icons next to the values in the autocomplete list.

"Home" and "End" keys

Determine how you want the Home and End keys to work.

Layout

The Layout preferences adjust how the Layout Editors work.

Default control font

By default, Xojo uses System as the Font for all controls. This allows the platform to substitute its own default font at run-time. If you would rather use a specific font, you can choose it here.

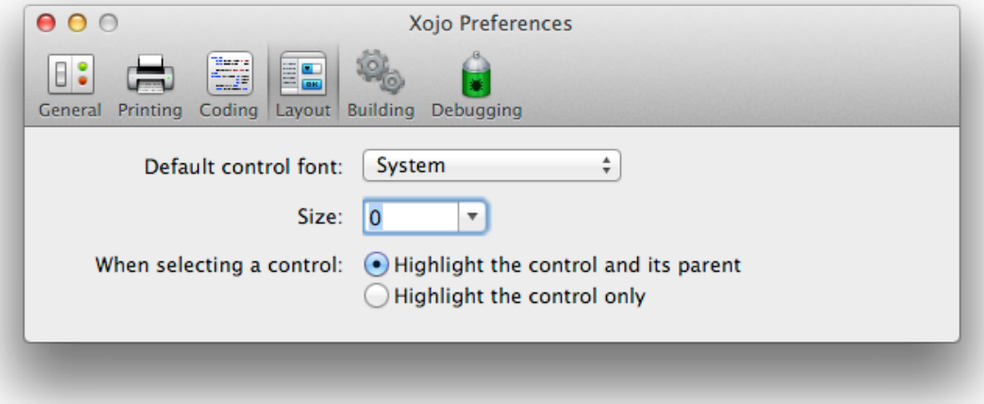
Size

By default, Xojo uses 0 as the Font Size for all controls. This allows the platform to substitute is own default font size at run-time. If you would rather use a specific font size, you can set it here.

When selecting a control

When you have nested controls (one control on top of another control), the Layout Editor highlights the parent control with a red outline. If you would rather not see this, you can turn it off here.

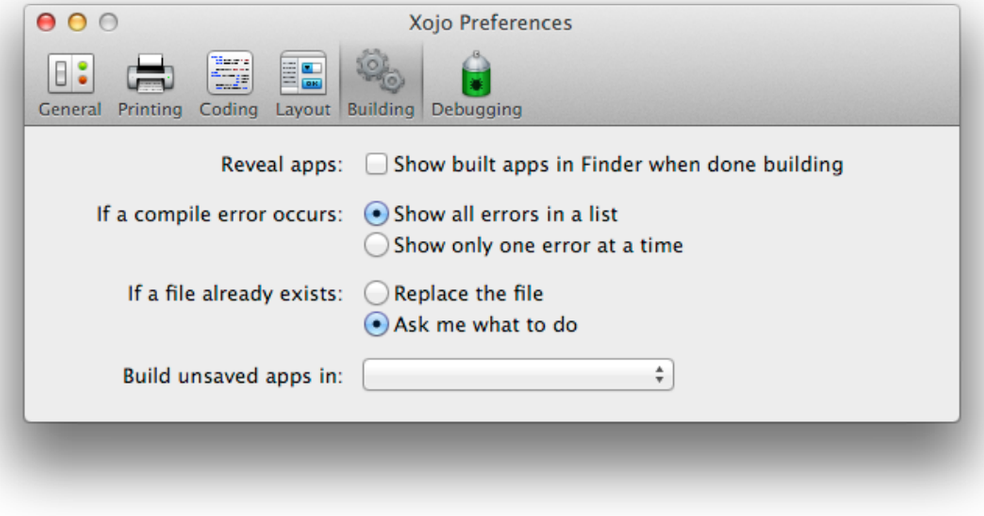
Figure 2.61 Layout Preferences



Building

The Building preferences control what Xojo does when building or running your applications.

Figure 2.62 Building Preferences



Reveal apps

After Building your application, Xojo can display the location in the built app using the Finder or Explorer.

If a file already exists

When Building your application, Xojo normally replaces any existing built applications (and related files) automatically. If you would rather be prompted when a file will be overwritten, you can change this here.

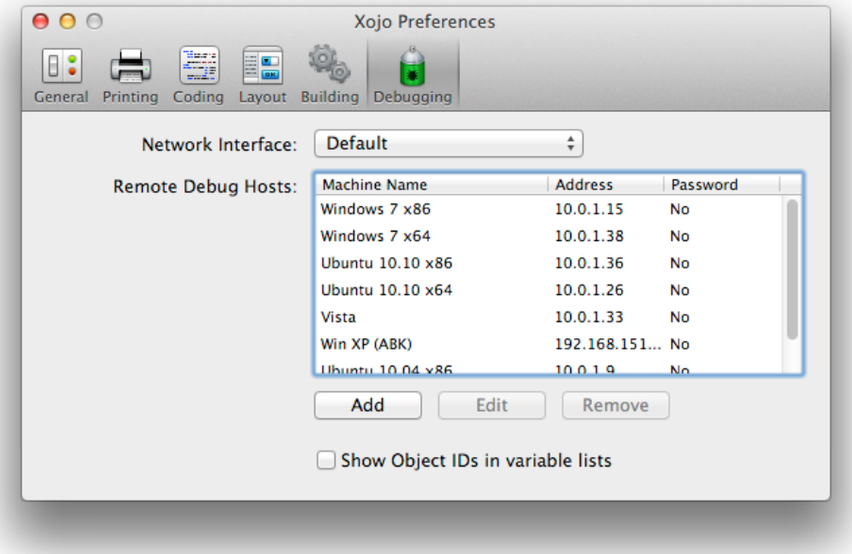
Build unsaved apps in

If you try to build an unsaved project, this location is used to store the built application.

Debugging

The debugging preferences are used to control remote debugging.

Figure 2.63 Debugging Preferences



Network Interface

Specify the networking interface to use to connect to the remote debugger.

Remote Debug Hosts

Here you can add, edit or remove remote debugging locations.

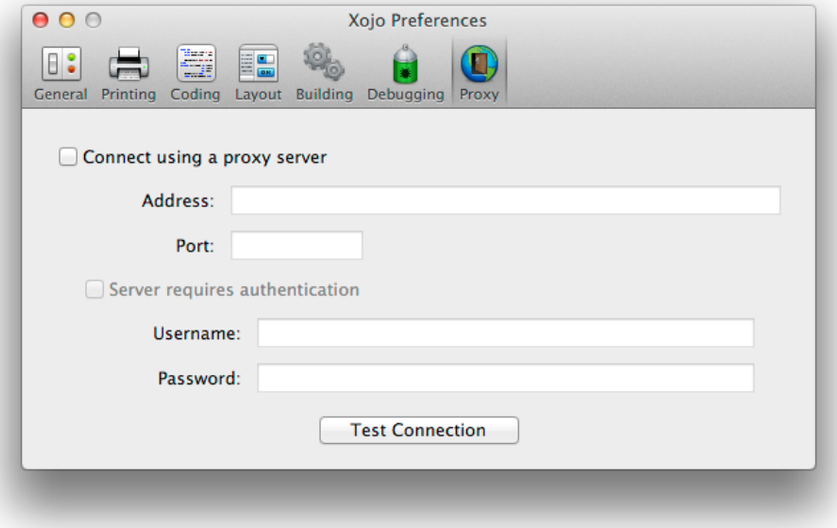
Show Object IDs in variable lists

Displays Object IDs in the debugger Variable area.

Proxy

The Proxy preferences are used to specify proxy information so that Xojo can connect to the server to validate your licenses. These settings are used when you “Sign In” to Xojo.

Figure 2.64 Proxy Preferences

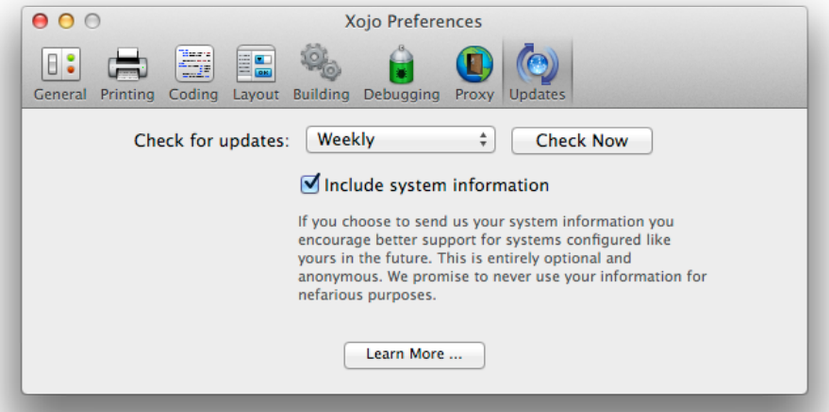


Select the “Connect using a proxy server” checkbox to enable use of the specified proxy server. Fill in the appropriate fields to specify the location of the proxy server and any information needed for authentication.

Updates

The Updates preferences allow you to choose how often Xojo checks for new versions. You can also specify whether you will allow your system information to be sent to Xojo servers.

Figure 2.65 Updates Preferences



Keyboard Shortcuts

General

Hold down Alt (Windows), Shift (on Linux) or Option (on OS X) to prevent loading the user interface state for the IDE. This prevents the loading of tabs and defaults Xojo to a standard window size.

Code Editor

While working in the code editor, there are several keyboard shortcuts that will help make you more productive.

- Ctrl + Return (⌘-Return on OS X) completes the code block. For example, typing “If True” and pressing Ctrl + Return will automatically fill in the “Then” and the “End If” and place the cursor between them. This works for other code blocks such as Select, While, Do and For.
- Ctrl + Enter (Option-Enter on OS X) extends code to a new line by automatically adding the “_” character.
- Ctrl + \ (⌘-\) toggles the breakpoint on the line on or off.
- Ctrl + ‘ (⌘-’) comments or uncomments the selected code lines or the line the cursor is on if no code is selected.
- Ctrl (⌘ on OS X) + double-clicking on a method name takes you to the method in the Navigator.
- Tab is used to invoke the auto-complete. Press it when you see grayed-out text to auto-complete the text or when you see the ellipses to display the auto-complete items.

Menus

These are all the keyboard shortcuts for menus used by Xojo.

File Menu		
Windows/ Linux Key	**OS X Key**	**Command**
	⌘-,	Open Preferences/Options
	⌘-Q	Quit/Exit Xojo
Ctrl+N	⌘-N	Open Project Chooser
Ctrl+Shift+N	⇧- ⌘- N	Open New Workspace
Ctrl+T	⌘- T	Open New Tab
Ctrl+O	⌘-O	Open Project File
Ctrl+W	⌘-W	Close Tab
Ctrl+Shift +W	⇧-⌘-W	Close Window
Ctrl+S	⌘-S	Save Project
Ctrl+Shift+S	⇧-⌘-S	Save As Project
Ctrl+P	⌘-P	Print
	⇧-⌘-P	Print Setup

Edit Menu		
Windows/ Linux Key	**OS X Key**	**Command**
Ctrl+Z	⌘-Z	Undo
Ctrl+Y	⇧-⌘-Z	Redo
Ctrl+X	⌘-X	Cut
Ctrl+C	⌘-C	Copy
Ctrl+V	⌘-V	Paste
DEL	DEL	Delete
Ctrl+D	⌘-D	Duplicate
Ctrl+A	⌘-A	Select All
Ctrl+Shift+A	⇧-⌘-A	Deselect All
Ctrl+’	⌘-’	Comment
↵ (return)	↵ (return)	Set Default Value Of
Ctrl+F	⌘-F	Find

Insert Menu		
Windows/ Linux Key	**OS X Key**	**Command**
Ctrl+Shift+E	⌥ -⌘-E	Event Handler
Ctrl+Shift+H		Menu Handler
Ctrl+Shift+M	⌥ -⌘-M	Method
	⌥ -⌘-N	Note
Ctrl+Shift+P	⌥ -⌘-P	Property
Ctrl+Shift+C	⌥ -⌘-C	Constant

Project Menu		
Windows/ Linux Key	**OS X Key**	**Command**
Ctrl+R	⌘-R	Run
Ctrl+ESC	⌘-.	Stop Debugging
Ctrl+Alt+R	⌥-⌘-R	Run Remotely
Ctrl+Shift+O	⇧-⌘-O	Step Over
Ctrl+Shift+I	⇧-⌘-I	Step Into
Ctrl+Shift+T	⇧-⌘-T	Step Out
Ctrl+\	⌘-\	Turn On/Off
Ctrl+K	⌘-K	Analyze Project
Ctrl+Shift+K	⇧-⌘-K	Analyze Item
Ctrl+B	⇧-⌘-B	Build Application
	Option-⌘-F	Go To Search
Ctrl+Shift+L	⇧-⌘-L	Go To Location

View Menu		
Windows/ Linux Key	**OS X Key**	**Command**
Ctrl+E	⌘-E	Go to Layout/Code Editor
Ctrl+L	⌘-L	Library
Ctrl+I	⌘-I	Inspector
Ctrl+Shift+F	⇧-⌘-F	Hide Find
	^-⌘-F	Enter Full Screen

Window Menu		
Windows/ Linux Key	**OS X Key**	**Command**
	⌘-M	Minimize
Ctrl+Tab	⌘-}	Next Tab
Ctrl+Shift+Tab	⌘-{	Previous Tab
	⌘-1..5	Select Window

Help Menu		
Windows/ Linux Key	**OS X Key**	**Command**
F1		Language Reference

Chapter 3

The Xojo Programming Language

This chapter covers the Xojo Programming Language, including language concepts such as data types, methods and commands.

CONTENTS

Section 1

Naming Rules

In the following sections, you will learn how to create variables, constants, methods, modules, classes and more. All of these items follow the same naming standards, which are as follows:

- Names must start with an ASCII letter (A-Z) or (a-z)
- The remainder of the name can contain any of the following:
    - alphanumeric ASCII characters (A-Z, a-z, 0-9)
    - underscore (_)
    - Any Unicode character with code point greater than 127
- Names can be any length
- Names are case-insensitive (**age** and **Age** are the same)

If you use an invalid character, you will get a compile error (usually reported as a Syntax Error) when you try to run the project. For names that are specified using the Inspector, you will get a prompt indicating that the name is invalid and the name will revert to its previous name.

Section 2

Variables and Constants

Remembering Information

A computer has two types of memory: temporary and permanent. Temporary memory is used to remember things for as long as the computer is on. As soon as the computer is turned off (or restarted), anything in temporary memory is lost. This is often referred to as RAM (Random Access Memory).

Permanent memory is remembered even after a computer is turned off. This type of memory is usually a hard disk, solid-state disk or flash memory. You save things to permanent memory using files, which are discussed in the Framework book.

When you are writing programs, you will often want to remember things. You use a concept called Variables to store information in the temporary memory. Variables are great for holding information such as counter values, field values and anything else you need while your program is running.

The term “variable” is used because the value that it contains can be changed by your program, so its value can vary.

Variables

A variable is the simplest way to store values. All variables are declared in your code using the Dim statement and each variable must be declared before it is used. When you declare a variable you tell it the type of data that it can contain.

A variable declaration consists of three parts: the Dim keyword, the variable name, and the data type. Here is an example variable declaration:

```
Dim age As Integer
```

These simple data types are available: **String**, **Integer**, **Double**, **Currency**, **Boolean**, and **Color**. Data Types are covered in the next section, Data Types and Storage.

Once you have declared a variable, you can assign it a value as follows:

```
Dim age As Integer
age = 42
```

Variables can be declared anywhere within a method, as long as the declaration precedes its first usage.

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

```
Dim i, j, k As Integer
```

If you want to declare variables of different types, you can also declare them in one Dim statement. For example, the following Dim statement is valid:

```
Dim Name, Address as String, ShoeSize As
Integer
```

When you create a variable with the Dim statement you can also assign it a value. You do so by following its data type with an equals sign and the value. For example:

```
Dim i As Integer = 1
Dim Name As String = "Igor", Address As
String = "15 Rue de Vallee"
```

You can also mix variables with and without initial values, as in:

```
Dim a As Integer, b As Integer = 15
```

In this case, the variable a is declared as an integer but is not assigned a value. If you examine the value of a, it will be zero. The variable b, on the other hand, gets the value of 15.

Notice also that the following is valid:

```
Dim a, b, c As Integer = 15
```

This statement declares three integer variables and assigns all of them the value of 15. Although this is valid, you should be careful about assigning values to more than one variable in a single Dim statement because you could easily lose track of the assignments.

Accessing a Variable

A variable can be accessed only within the method in which it was declared. When the method is finished, the memory that was used to store the variable's value becomes available for other uses. This means that another method in the application cannot access the variable value.

The term scope is used to describe where something, such as a variable, can be accessed. Variables and constants declared within methods have what is called a **local** scope because they are locally available only to the method.

If you declare a variable or constant inside of a code block (such as an If statement, Select statement or any looping statement), its scope is local to the code block and not the entire method.

For example, you can write:

```
If x > 5 Then
  Dim y As Integer
  y = 10
End If

// y goes out of scope here; the
variable name y can now be reused.
// It is redeclared as a string in
the following If statement
If x < 5 Then
  Dim y As String
  y = "hello"
End If
```

You can also assign variables a value by using another variable. Doing this does not affect the original variable.

When using the simple data types, the new variable gets a copy of the data in the original variable. Changing the new variable does not change the value in the original variable.

For information on how object variables work with class references, refer to the Classes chapter.

This example creates a string variable, assigns it a value and then assigns its value to another string variable:

```
Dim s As String = "hello"

Dim s2 As String
s2 = s // s2 now contains "hello"
MsgBox(s2)

// Changing s2 does not affect s
s2 = "world"
MsgBox(s2)

// s still contains "hello"
MsgBox(s)
```

Constants

A constant behaves similarly to a variable with two significant differences: you do not specify the type when declaring it and its value cannot be changed once it has been set.

Here is some example constant declarations:

```
Const pi = 3.14159
Const value = 256*10

Const kDefault = "DefaultName"
```

Constants can be assigned only a constant value or the result of a constant expression.

Section 3

Data Types and Storage

To make programming code execute faster and to provide powerful commands that save you time when programming, computers have to be able to make certain assumptions about the information you give them. For example, when you give a computer a piece of information, the computer needs to know if it is a number, a string of characters, a date, etc. If you give the computer the instruction “1 + 1”, it needs to know what kind of data each “1” is. Otherwise, it wouldn’t know whether you meant 1 plus 1 to be 2 or the string “11”. In this example, if you tell it that you mean numbers, it will add them. It will return the result of “2”. If you tell it that you mean characters, it will concatenate them and return the result of “11”. There are many data types that are available, but these are the data types that are by far the most common. They are **Text**, **String**, **Integer**, **Double**, **Currency**, **Boolean**, and **Color**.

Text

A Text is a series of characters. Sometimes referred to as a String (refer to next section), Text is the preferred data type for dealing with Text because it can more intelligently handle text in all types of languages.

The maximum length of Text is limited only by available memory.

You declare a Text like this:

```
Dim t As Text
```

You can also directly assign a value as part of the declaration:

```
Dim t As Text = "Hello, World!"
```

Basically, any type of information ca be stored as Text, simply by placing quotes around it.

String

A String is the older way of dealing with text. Essentially, a String just a series (or string) of characters (or bytes).

Because Strings deal primarily with bytes, they can be more difficult to use with some languages, so you should use the Text

data type in most cases. You can convert a String value to a Text value by calling String.ToText

```
Dim s As String = "Hello, World!"
Dim t As Text s.ToText
```

You declare a string like this:

```
Dim s As String
```

You can also directly assign a value as part of the declaration:

```
Dim s As String = "Hello, World!"
```

Strings that do not have a direct assignment have the value of the empty string by default.

Basically any kind of information can be stored as a string. “Jenni”, “3/17/98”, “45.90” are all examples of strings.

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

You can combine two strings with the addition symbol (+). For example, the statement “Big” + “Dog“ results in the string “BigDog”. That is really the extent of the “mathematics“ you can perform on strings. However, there are many built-in functions that make processing strings easy. For example, the **Lowercase** function takes a string and converts all the characters to lowercase. The **Trim** function trims off any leading and trailing whitespace characters.

Integer

An integer is a whole number. It cannot accept a decimal or fractional value.

***Note**: The Integer data type is a signed integer that uses the word length for the target platform. Currently this is four bytes (32-bits) on all supported platforms, giving you an integer range of -2,147,483,648 to 2,147,483,647.*

Because integers are numbers, you can perform mathematical calculations on them. Unlike strings, you do not put quotes around integer values in your code. This is how you declare an Integer:

```
Dim i As Integer
```

An Integer declared in this way gets a default value of 0. You can also declare an Integer and specify a default value:

```
Dim i As Integer = 100
```

Although you should normally just use the Integer type, there are also a variety of more specific integer types (both signed and unsigned Integer data types) that can be used in specific situations (typically only when communicating with an OS API). These include: Int8, Int16, Int32, Int64, UInt8 (Byte), UInt16, UInt32 and UInt64.

***Note**: The difference between signed an unsigned integers is that unsigned integers can only contain positive numbers (and 0).*

Double

A Double is a number that can contain a decimal value. In other languages, Double may be referred to as a double precision real number. Because Doubles are numbers, you can perform mathematical calculations on them. Doubles use 8 bytes of memory. The range of a double varies between ±1.79769313486231570814527423731704357e+308.

```
Dim value As Double
```

A Double declared in this way gets a default value of 0.0. You can also declare a Double and specify a default value:

```
Dim pi As Double = 3.1415926
```

Currency

A Currency is a (64-bit, 8-byte) fixed-point number format that holds 15 digits to the left of the decimal point and 4 digits to the right. It is scaled by 10,000 to give 4 digits to the right of the decimal point. It is always accurate to four decimal places. It is useful for calculations involving money and for calculations where accuracy is very important. It is compatible with the Currency data type offered in some versions of Visual Basic.

```
Dim value As Currency
```

A Currency declared in this way gets a default value of 0.0. You can also declare a Currency and specify a default value:

```
Dim value As Currency = 1.23
```

Boolean

A Boolean can take on the values of either True or False. Boolean values are False by default:

```
Dim b As Boolean = True
```

You can set a Boolean to True using the True value and back to False using the False value, or by any expression that returns a boolean value.

```
Dim b As Boolean = True
```

This example sets b to True or False depending on whether an integer value is greater than 5:

```
Dim i As Integer = 10
Dim b As Boolean
b = (i > 5) // b gets set to True
```

Color

A Color is a data type that stores the value of a color. A Color “value” actually consists of three numeric values that can be set using any of the three popular color models, Red-Green-Blue, Hue-Saturation-Value, or Cyan-Magenta-Yellow. Each RGB value is stored as a byte. That means that each number can take on 256 possible values.

Each color function also includes an optional Integer parameter that you can use to specify the level of transparency. Transparency is sometimes referred to as the alpha channel. It is also an Integer parameter that varies from 0 to 255. Zero means no transparency (completely opaque) and 255 is fully transparent. If you omit the transparency parameter entirely, it defaults to opaque.

A Color can also be set to a value using the &c prefix or via one of the three color functions (RGB, HSV, CMY) that correspond to the three color models.

Some example Color declarations:

```
// Different ways to assign Red
Dim red As Color
red = &cff0000
red = RGB(255, 0, 0)
red = Color.Red
red = &cff000088 // semi-transparent
```

Changing a Value From One Data Type to Another

There may be times when you need to change a value from one data type to another. This is usually because you want to use the value with something that is designed to work with a different data type. For example, you might want to include a number in the title of a window, but the title of a window is a string, not a number. Consequently, if you try to assign a number to the title of a window, an error message displays when you run your application. The error will tell you that the two data types are not compatible (they are different). Since the window title is a string, you will need to change the number into a string before you can assign it to the window title.

Fortunately, there is a built-in function called **Str** (which stands for String) that can change a number into a string. See the Str function in the Language Reference for more information. There is also a built-in function called **Val** (which is short for Value) that changes strings into numbers. See the Val function in the Language Reference for more information.

When working with Text, you typically use the ToText method of a non-Text type to convert it to a Text value. For example, an Integer can be converted to a Text value like this:

```
Dim i As Integer = 42
Dim t As Text = i.ToText
```

To create an Integer value from a Text value, you use the FromText method:

```
Dim i As Integer = 42
i = Integer.FromText("42")
```

Mathematical Operators

Performing mathematical calculations is a very common task in programming, and all of the common mathematical operations are supported.

Figure 3.1 Mathematical Operators

Operation Performed	Operator	Example
Addition	+	2+3=5
Subtraction	-	3-2=1
Multiplication	*	3*2=6
Floating Point Division	/	6/4=1.5
Integer Division	\	6\4=1
Modulo	Mod	6 Mod 3 = 0 6 Mod 4 = 2
Exponentiation	^	2^3=8

In addition, there are also many built-in mathematical functions that are covered in detail in the Language Reference.

Expressions support standard mathematical precedence. This means that equations surrounded by parentheses are handled first. The expression is evaluated beginning with the set of parentheses that is embedded inside the most outer sets of parentheses. Next, exponentiation is performed, followed by any multiplication or division from left to right. Finally any addition or subtraction is performed. In the “Sample Expressions” table, the three expressions return different results because of the placement of parentheses.

Operator Precedence

Operator Precedence determines the order in which operators execute when there is more than one operator in an expression. For example, the expression:

5+2/3

contains both the division and addition operators. It matters whether the division or the addition takes place first. You can force a particular precedence via parentheses. By default, division has precedence over addition, so this expression evaluates as 5 plus 2/3. You can force it to do the addition first by using parenthesis to override the default operator precedence:

Figure 3.2 Sample Math Expressions

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

(5+2)/3

If there is more than one operator in an expression, the precedence goes from left to right. For instance, multiplication is higher than floating-point division, which is higher than integer division, which is higher than modulus.

All operators are left-associative except for pairs (:) and exponentiation (^). Left association means that (foo or bar or baz) will evaluate like ((foo or bar) or baz) instead of (foo or (bar or baz)). Conversely, right association means that (foo:bar:baz) will evaluate like (foo: (bar: baz)) instead of ((foo:bar):baz).

Figure 3.3 Operator Precedence

Operator	Description
.	Dot operator
AddressOf	Delegate creation
IsA	Type checking
^	Exponentiation
-	Negation of the unary minus
Not	Logical Not
*, /, \, Mod	Multiplication and division arithmetic
-, +	Subtraction and addition
=, >, <, >=, <=, <>, Is	Comparisons
And	Bitwise and Logical And
Or, Xor	Bitwise and Logical Or
:	Pair creation

Boolean Expressions

Boolean expressions deal only with the values True and False. True and False are values that Xojo can manipulate, just as it does numbers and strings.

There are two types of Boolean expressions:

- Simple: Boolean expressions state something about a value in the program, generally using operators like = (equals), <> (not equals), <= (less than or equal), or >= (greater than or equal). Sometimes a property or something else in the program will actually just be a Boolean value. The toggle switches you see in the Inspector are good examples. Source1.Bold is a valid Boolean expression, since there is a Bold toggle in the Inspector when you click on one of the TextFields — so the expression Source1.Bold would evaluate to True when Source1's Bold toggle is set to ON; and

- Compound: Boolean expressions that are made out of other Boolean expressions (which can themselves be either simple or compound), using Boolean operators Not, And or Or.

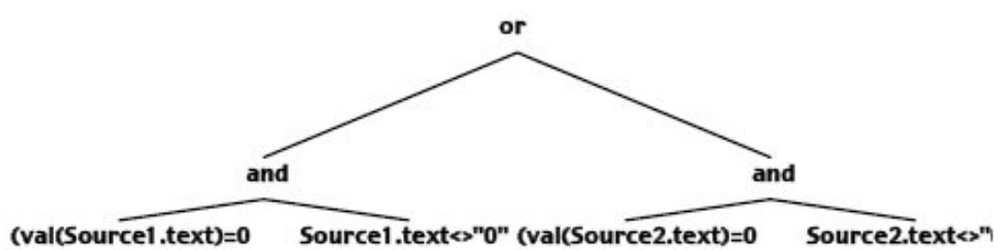
As with other types of expressions, you will often need to use parentheses in Boolean expressions to make it clear in what order operations should be carried out. Also, as with other types of expressions, Xojo works out the final value from the inside out.

An Example of a Compound Expression

Consider this expression:

```
(Val(Source1.Text) = 0 And
 Source1.Text <> "0") Or
(Val(Source2.Text) = 0 And
 Source2.Text <> "0")
```

Image 3.1 Boolean Syntax Tree



A compound expression such as this is evaluated according to the left-to-right rule. The two branches of the OR expression are evaluated separately prior to ORing them.

If a compound expression is joined by AND, then any sub-expression that evaluates to False ends the process of evaluating any subsequent sub-expressions within that compound expression. This is because one False sub-expression means that the entire compound AND expression must also be False.

An expression like this can be represented as a syntax tree. The syntax tree for this expression is shown in Image 3.1.

The expression is evaluated from the bottom of the tree up. Each level of the tree is evaluated left to right.

In this example, the first step is to evaluate the left branch of the OR expression. It is a compound expression of its own, as its branches use AND.

It starts with

```
Val(Source1.Text) = 0
```

If this expression returns True then it evaluates:

```
Source1.Text <> "0"
```

Lastly, if

```
Val(Source1.Text) = 0
```

is False, there is enough information to evaluate the “AND” so False is returned to the next higher level without evaluating the second expression.

Since the two compound expressions are joined by OR, it needs to evaluate the right compound expression at the base of the tree even if the left compound expression evaluates to False. If the left branch is False, but the right branch is True, then the complete compound expression is True.

The same procedure is followed in evaluating the right branch of the compound OR statement.

Variant

A Variant is a special data type that can contain any type of data. Variants are advanced features that should be used with caution because the values will automatically convert to other types as needed, which can often result in unintended behavior.

Refer to the Language Reference for more information on Variant.

Auto

The Auto type is a replacement for Variant. It can also contain any type of data, but it does not automatically convert to other types. You instead have to explicitly assign the value to a variable of the correct type in order to use it.

Because of this, the Auto type is much safer to use since you will not run into unintended behavior from type conversions.

For more information about Auto, refer to the Reference Guide.

Section 4

Collections of Data

Programs often have a lot of data to manage. Two techniques for managing data include arrays and dictionaries. Although they both store large collections of information, they do so in different ways.

Arrays

An array is a special type of variable that contains several values of the same data type. Each variable in the array is called an element of the array. To refer to a particular element, you use an index number.

The Dim statement also lets you create and type arrays. When you declare an array, you specify the number of elements it has. Later on, you can change the number of elements.

Creating Arrays

You declare an array by specifying the index of the last element of the array. The index that you specify in the Dim statement is actually one less than the number of elements in the array because Xojo arrays always have an element zero. Such an array is sometimes referred to as a zero-based array. If the first element of an array is element 1, then it would be called a one-based array. Since Xojo arrays are zero-based, the statement:

```
Dim names(10) As String
```

creates a string array with eleven elements.

The statement:

```
Dim names(0) As String
```

creates a string array with one element, element zero. You can read and write to this element just as with any element with a positive index.

In other words, you declare a variable as an array simply by adding the index of the last element to the Dim statement. The index of the last element must be either a number or a constant. You cannot use variables with this syntax.

If you don't know the size of the array you need at the time you declare it, you can declare it as a null array, i.e., an array with no elements. You do this by using an index of -1 in the Dim statement or leave empty parentheses. This means "an array of no elements." For example, the statements:

```
Dim firstNames(-1) As String
Dim lastNames() As String
```

creates the array with no elements. Later on, you can resize the array using the **Redim** statement or "grow" it element-by-element using the **Append** or **Array** methods.

Multidimensional Arrays

You can create multi-dimensional arrays. For example, a spreadsheet layout can be thought of as a two-dimensional array, rows by columns.

Each dimension is referred to by its own index. For example, the elements of an array with two dimensions are referred to by one index for the rows and the other for the columns. The first element in the upper-left corner is element 0,0.

You create a multi-dimensional array by specifying an index for each dimension. For example, the statement:

```
Dim names(2, 10) As String
```

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

You can use constants in Dim statements to set the size of an array. For example, the following declaration refers to a global constant, kLanguages, that is defined in a module:

```
Dim names(10, kLanguages) As String
```

In this example, kLanguages is the number of languages supported by the application and is used in numerous places in the application. When it changes, you only need to change its definition in the module.

Referring to Array Elements

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

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

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

Getting the Index of the Last Element

The Ubound function returns the index of the last element of a one-dimensional array. For example, the expression:

```
Ubound(names)
```

returns this value. The number of elements is one greater than this number, since the array has an element zero.

For example, the following example returns 5 in the variable i.

```
Dim i As Integer
Dim names(5) As String
i = Ubound(names)
```

You can also get the last element of an array using dot notation syntax. The following code is equivalent:

```
Dim i As Integer
Dim names(5) As String
i = names.Ubound
```

Initializing Arrays

After you have declared an array, you can assign initial values to the elements with the Array function as well as individual assignment statements, such as shown above. The Array function takes a list of values and assigns the values to the elements of the array, beginning with element zero. In other words, it provides the same functionality as separate assignment statements for each element of the array.

For example, the following statements initialize the array using separate assignment statements for each array element:

```
Dim names(2) As String
// Separate assignment statements
names(0) = "Fred"
names(1) = "Ginger"
names(2) = "Stanley"
```

The following statements use the Array function to accomplish the same thing. Note that you don’t have to declare the exact size of the array in the Dim statement.

```
Dim names() As String
names = Array("Fred", "Ginger",
"Stanley")
```

The Array function will add elements to the array as needed.

If you declare the array as a fixed size but don't specify as many values as elements, the Array function will start with element zero and use as many elements as are specified.

Array Assignment

If you have two arrays of compatible data types, you can assign one array to the other array. Simply use the assignment statement without the parentheses. Here is a simple example:

```
Dim names(2), copyNames(2) As String
names = Array("Fred", "Ginger",
"Stanley")
copyNames = names
```

The last statement assigns the values of all three elements of *names* to the first three elements of *copyNames*. If *copyNames* had fewer elements than *names*, then additional elements would first be added to *copyNames* and the assignment of all the elements of *names* to *copyNames* would be completed. For example, the following is valid:

```
Dim names(3), copyNames(2) As String
names(0) = "Fred"
names(1) = "Ginger"
names(2) = "Tommy"
names(3) = "Woody"
copyNames = names
```

After the code runs, the *copyNames* array has a fourth element for storing the value "Woody".

Resizing Arrays

Several methods in the language resize arrays.

- **Append**: The Append method adds an element to the array, increasing its size by one. You pass the value you want to add to the array when you call Append. For example, the following statement:

```
names.Append("Dave")
```

adds an element to the array *names* and sets the value of this element to the string, “Dave”. Append works on one-dimensional arrays only.

- **Insert**: The Insert method creates an additional element and inserts it in the place you specify. It takes two parameters, the index of the element to be inserted and the value of the new element. For example:

```
names.Insert(9, "Hal")
```

After this statement runs, the value of *names(9)* would be “Hal”. The old element(9) would be shifted up to element(10) and so forth. The size of the array would be increased by one, as with Append.

Insert also works only on one-dimensional arrays.

- **Remove**: The Remove method deletes the element whose index you specify. For example:

```
names.Remove(9)
```

removes the element with index 9, decreasing the size of the array by one and shifting the array elements after the removed element down by one. It also works on one-dimensional arrays only.

- **Redim**: The Redim statement resizes an existing array. You pass the new values of the array’s indexes but you don’t specify the data type, which is set by the initial Dim statement. For example, the statement:

```
ReDim names(100)
```

resizes the array *names* to 101 elements.

Redim works on both one- and multi-dimensional arrays. For multi-dimensional arrays, you can only resize the existing dimensions; you cannot add or reduce the number of dimensions themselves.

A difference between Dim and Redim is that Dim accepts only integers or constants, while Redim accepts any expression that returns an Integer. This includes, for example, a user-written function that returns an Integer value or a simple variable. Using this feature, you can dimension your arrays on-the-fly.

If you don’t know the size of the array you need at the time you declare it, you can declare it as a null array, i.e., an array with no elements, and use the Redim command to resize it later. If your program needs to load a list of names that the user enters, you

can wait to size the array until you know how many names the user has entered. You can write a function to figure out what that number is and use it with Redim or use the Append method to add the required elements to the array one-by-one.

Converting to and from an Array to Variables

Two functions enable you to take a take an array and break it up into separate variables and take a single string variable and convert it into an array.

- **Split**: The Split function takes a String variable and creates a one-dimensional array by dividing the string up into elements. It uses a delimiter—a character or character string that signals the end of one element and the start of the next element—to do this task. By default, the delimiter is a space, but you can specify another delimiter.

```
Dim names() As String
// resize using the Split method
Dim s As String
s = "Juliet,Taylor,Casting"
names = Split(s, ",")
```

The Split function takes the string as its first parameter and the delimiter as the second parameter. Here is an example that divides up the contents of a string into array elements. It specifies the comma as the delimiter. After this call, the resulting array, *names*, has three elements:

names(0) = "Juliet"
names(1) = "Taylor"
names(2) = "Casting"

- **Join**: The Join function takes a one-dimensional array and creates a String variable that contains all the elements in the array, separated by a delimiter character or character string. By default, the delimiter character is a space, but you can specify your own delimiter by passing the delimiter as the second parameter. The delimiter is passes as the second parameter.

For example, if you have an array that contains elements that store a person's first and last names and phone number, the Join function will create one String variable that contains all the information. Here is an example:

```
Dim names(2) As String
Dim s As String
names(0) = "Anthony"
names(1) = "Aardvark"
names(2) = "(406) 737-8946"

// "Anthony,Aardvark,(406) 737-8946"
s = Join(names, ",")
```

In this example, the call to the Join function specifies the name of the array and the comma as the delimiter. If the call were:

```
s = Join(names)
```

the result would have a space between each value: “Anthony Aardvark (406) 737-8946”.

Dictionaries

A dictionary is an object that is made up of a list of key-value pairs. That is, each value is paired with an identifying key. The interesting feature of Dictionaries is that both the key and the value are variants. This means that a dictionary can store a mixture of data types — and that the key doesn’t have to simply be an integer. You can look up a value in a dictionary by specifying either its key or its sequential position in the dictionary.

A dictionary is a great way to have fast, random lookups of information.

A dictionary is a class (described in Chapter 5) so you create one using the New command:

```
Dim d As New Dictionary
```

You add values to the dictionary by using the Value method. Note that the index can be anything, such as a string:

```
d.Value("FirstName") = "Bob"
```

You can look up a value using the index similarly:

```
MsgBox(d.Value("FirstName")) // Shows Bob
```

Pairs

The Pair class is similar to the Dictionary. It has two properties: Left and Right. Thus, each Pair instance consists of a key-value pair. As it the case with the Dictionary, the values in the pair are variants. You use the “:” operator to assign the Left and Right values when the Pair is declared. For example:

```
Dim p As Pair = "Telephone Number" :
"(406) 737-8946"
```

This assigns “Telephone Number” to the Left property of the pair and the value of the number to the Right property.

It also can store a linked list of pairs when it is passed a list of items, such as:

```
Dim p as Pair = 1 : 2 : 3 : 4 : 5
```

The first pair consists of the pair “1” (Left property) and the second pair object (Right property); the next pair consists of the “2” and the third pair, and so forth.

Section 5

Comparisons

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

Figure 3.4 Comparison Operators

Description	Symbol	Example	Result
Equality	=	5 = 5	True
Inequality	<>	5 <> 5	False
Greater Than	>	6 > 5	True
Less Than	<	6 < 5	False
Greater Than or Equal To	>=	6 >= 5	True
Less Than or Equal To	<=	6 <= 5	False

String and boolean values can also be used for comparisons. String comparisons are case insensitive and alphabetical. This means that “Jeannie” and “jeannie” are equal. But “Jason” is less than “Jeannie” because “Jason” falls alphabetically before “Jeannie”. If you need to make case sensitive or lexicographic comparisons, you use the **StrComp** function:

```
If StrComp("Jeannie", "jeannie", 0) <> 0 Then
  MsgBox("Strings do not match.")
End If
```

For more information, refer to the StrComp function in the Language Reference.

In addition, there is a floating point equals operator. It allows you to determine whether two floating point numbers are close enough to be considered equal. Use it to account for the imprecision of operations such as floating point division. The floating point equals operator is the **Equals** keyword. There is no symbol for it. It’s syntax is:

```
result = expression.Equals(NumValue, x)
```

Expression is the floating point value to be compared, NumValue is the value it is being compared to, and x is the number of units in the last position that denotes the acceptable range. Result is either True or False, depending on the result of the comparison.

For example, if you are comparing 10000 to a value and specify x=1, then the acceptable values are 10000.000000000002, 10000.0 and 9999.999999999998.

```
Dim d As Double = 10000
If d.Equals(compareValue, 1) Then
  // take action here..
End If
```

Logical Comparisons

You can test more than one comparison at a time using the And, Or, and Not operators. When passed boolean values, these operators determine whether the expression is true or false.

And Operator

Use this operator when you need to know if all comparisons evaluate to True. In the example below, if the variable x contains 10 then the expression evaluates to False:

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

This is because 10 is not both greater than 1 and less than 5.

Or Operator

Use this operator when you need to know if any of the comparisons evaluate to True. In the example below, if the variable x contains 10 then the expression evaluates to True:

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

This is because 10 is greater than 1. The fact that it is not less than 5 does not matter because you only care if one of the comparisons is True.

Xor Operator

Use this operator when you need to know whether any of the comparisons evaluate to True, but not all of them. In the example below, if the variable x contains 10 then the expression evaluates to True.

```
x > 1 Xor x < 5
```

This is because only one of the comparisons is True. However, if x contains 3 then the above expression returns False because both sub expressions are True.

Not Operator

Use the Not operator to reverse the value of a boolean variable. For example:

```
Not x < 0
```

tests whether x is equal to or greater than 0.

Figure 3.5 Boolean Evaluation Table (Truth Table)

Expr1	Expr2	Xor	Or	And
True	True	False	True	True
True	False	True	True	False
False	True	True	True	False
False	False	False	False	False

Bitwise Comparisons

You can also compare the individual bits of two integers to determine whether or not they are equal. This is done by re-expressing each integer as a binary number. Then you compare the bits in the pair of integers place-by-place. You get a new integer which is the result of each comparison.

Figure 3.6 Bitwise Comparison Table

Bit 1	Bit 2	Xor	Or	And
1	1	0	1	1
1	0	1	1	0
0	1	1	1	0
0	0	0	0	0

You can do this with the And, Or, and Xor operators. They are considered overloaded. This means that they can accept either booleans (as was shown above) or integers. If they are passed boolean expressions, they “know” that you want the logical comparisons that are described in the previous section. If they are passed integers, they “know” that you want to compare the bits that make up the two integers. In this case, they each return an integer made up of the results of all the bit comparisons. The following table summarizes the results for the comparison of each bit in the passed integers.

For example, consider the numbers 5 and 3. Written in binary they are:

101 (5)
011 (3)

To evaluate 5 Xor 3, you do the comparison on each bit. So for this example:

1 Xor 0 = 1
0 Xor 1 = 1
1 Xor 1 = 0
The new binary value is 110, which is 6.

So 5 Xor 3 = 6.

The Not operator is also overloaded. If you pass an integer to Not, it simply reverses each bit value.

Figure 3.7 Bitwise Not Value Changes

Bit 1	Not Bit 1
0	1
1	0

You can also do these bit comparisons using methods of the **Bitwise** class (refer to the Language Reference for more information).

Section 6

Branching

Making Decisions with Branching

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

If…Then…End If

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

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

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

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

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

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

```
If CheckBox1.Value Then
  // [Your code goes here]
End If
```

You can declare local variables using the Dim statement inside an If statement. However, such variables go out of scope after the End If statement. For example:

```
If error = -123 Then
  Dim a As String
  a = "Oops! An error occurred."
End If
MsgBox(a) // out of scope
```

If you need the variable after the End If statement, you should declare it local to the entire method, not within the If...End If statement.

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

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

```
If month = 1 Then
  MsgBox("It's January.")
Else
  MsgBox("It's not January.")
End If
```

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

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

```
If month = 1 Then
  MsgBox("It's January.")
ElseIf month < 4 Then
  MsgBox("It's still Winter.")
Else
  MsgBox("It's not Winter.")
End If
```

You could, of course, use an additional If...Then...End If statement inside the Else portion of the first If statement to perform another test. However, this adds another End If and needlessly

complicates your code. Instead, you can use as many ElseIf statements as you need.

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

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

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

If...Then...Else

An If statement can be written on one line, provided the code that follows the Then and (optionally) the Else statements can all be written on one line. When you use this syntax, you omit the End If statement. For example, the following statements are valid:

```
If error = 123 Then MsgBox("An error
occurred.")

If error = 123 Then MsgBox("An error
occurred.") Else MsgBox("Success")

If error = 103 Then Break
```

If Operator

For situations where you need to simply return a result based on a comparison, you can use the If operator.

```
If(condition, resultIfTrue, resultIfFalse)
```

The condition is evaluated and if it is True, the resultIfTrue is returned, otherwise resultIfFalse is return.

For example, this code outputs "Big":

```
Dim myInteger As Integer = 41
MsgBox(If(myInteger > 40, "Big", "Small"))
```

Select...Case

When you need to test a property or variable for one of many possible values and then take action based on that value, use a Select...Case statement

Consider the following example that uses If...ElseIf..End If to test a variable (dayNumber) and display the day of the week:

```
Dim dayName As String
If dayNumber = 2 Then
  dayName = "Monday"
ElseIf dayNumber = 3 Then
  dayName = "Tuesday"
ElseIf dayNumber = 4 Then
  dayName = "Wednesday"
ElseIf dayNumber = 5 Then
  dayName = "Thursday"
ElseIf dayNumber = 6 Then
  dayName = "Friday"
Else
  dayName = "the weekend."
End If
MsgBox("It’s " + dayName)
```

No two of these conditions can be True at the same time. While this method of writing the code works, it’s not that easy to read.

This next example uses a Select...Case statement to achieve the same result. It is far easier to read:

```
Dim dayName As String
Select Case dayNumber
Case 2
  dayName = "Monday"
Case 3
  dayName = "Tuesday"
Case 4
  dayName = "Wednesday"
Case 5
  dayName = "Thursday"
Case 6
  dayName = "Friday"
Else
  MsgBox "the weekend."
End Select
MsgBox("It’s " + dayName)
```

The Select...Case statement compares the variable or property passed in the first line to each value on the Case statements.

Once a match is found, the code between that case and the next is executed.

Select...Case statements can contain an Else statement to handle all other values not explicitly handled by a case.

You can create local variables using the Dim statement inside a Case statement. However, such variables go out of scope at the conclusion of the statement. For example:

```
Select Case dayNumber
Case 2
  Dim day As String
  day = "Tuesday"
Else
  MsgBox("It’s NOT Tuesday!")
End Select

MsgBox(day) // Error: day out of scope
```

The variable “day” should be declared prior to the Select...Case statement so that it is available after the End Select statement executes.

The Select…Case statement works with variables of any data type, including strings, integers, singles, doubles, booleans, and colors. For example, you can compare colors, as in the following example:

```
Dim c As Color
c = &cFF0000 // pure red

Select Case c
Case &c00FF00 // green
  MsgBox("Green")
Case &cFF0000 // red
  MsgBox("Red")
Case &c0000FF // blue
  MsgBox("Blue")
End Select
```

A Case statement can accept more than one value, with different values separated by commas. For example, the following is valid:

```
Dim c As Color
c = &cFF0000 // red

Select Case c
Case &c00FF00, &cFF0000 // green, red
  MsgBox("Green or Red")
Case &cFF0000 // red
  MsgBox(“Red”)
Case &c0000FF // blue
  MsgBox("Blue")
End Select
```

In the preceding example, the first Case statement is True, so its code executes. Although the color passed to Select…Case is Red, the code for the second case does not execute because it is not the first matching case.

The Select Case statement accepts an Else clause. The code in the Else clause executes only if none of the preceding cases match. The Else clause can be written as either “Else” or “Case Else”. In the following example, the Case Else clause executes because the color FFFF00 does not match any of the Case statements:

```
Dim c As Color
c = &cFF0000 // pure red

Select Case c
Case &c00FF00 // green
  MsgBox("Green")
Case &cFF0000 // red
  MsgBox("Red")
Case &c0000FF // blue
  MsgBox("Blue")
Case Else
  MsgBox("None of the above")
End Select
```

The Case statement can also accept a range of consecutive values using the “To” keyword. For example:

```
Dim i As Integer = 53
Select Case i
Case 1 To 25
  MsgBox("25 or less")
Case 26 To 50
  MsgBox("26 to 50")
Case 51 To 100
  MsgBox("51 to 100")
End Select
```

In this example, the third case, “51 to 100”, is true.

You can combine ranges with nonconsecutive values, by separating them with commas, such as:

```
Case 0, 26 to 50, 75, 100 to 200
```

You can write inequalities with the “Is” keyword and an inequality operator. The syntax is:

Is ineqalityOperator <value>

For example:

```
Dim i As Integer = 10
Select Case i
Case Is <= 10
  // this case selected
Case Is > 10
  // this case not selected
End Select
```

You can combine inequalities with values, as in:

```
Dim i As Integer = 75
Select Case i
Case 0, Is <= 10, 100
  // case not selected
Case Is > 10, Is < 99
  // case selected
End Select
```

You can even use functions that return a value of the specified data type in a Case statement. Here is a simple example:

```
Dim i As Integer = 4
Dim a As Integer = 2

Select Case i
Case CalcSquare(a)
  // case 1
Case a
  // case 2
Else
  // no match
End Select
```

Case a*a

The Select Case statement can also compare variables that are Objects. The following example uses a Select…Case statement to determine which button the user pressed in a MessageDialog box. The Select Case statement compares objects of type MessageDialogButton to determine which of three possible dialog buttons was pressed.

The function in the first Case statement is:

```
Function CalcSquare(a As Integer) As Integer
  Return a * a
End Function
```

In this example, the function squares the value passed to it, so the first Case statement matches.

In the case of a simple function like this, you can write the expression in the Case statement itself. That is, the following is an equivalent matching Case statement:

```
Dim d As New MessageDialog
Dim b As MessageDialogButton

d.Icon = MessageDialog.GraphicCaution
d.ActionButton.Caption = "Save"
d.CancelButton.Visible = True
d.AlternateActionButton.Visible = True
d.AlternateActionCaption = "Don’t Save"
d.Message = "Save changes before closing?"
d.Explanation = "If you don't save your
changes, you will lose your work."

b = d.ShowModal
Select Case b
Case d.ActionButton
  // user pressed Save
Case d.AlternateActionButton
  // user pressed Don’t Save
Case d.CancelButton
  // user pressed Cancel
End Select
```

You can also use the IsA operator to determine whether an object is of a particular class. The syntax is:

Case IsA ClassName

Here is a simple example. The code in a PushButton Action event handler in a window is:

```
Select Case Me
Case IsA PushButton
  MsgBox("I’m a PushButton.")
Case IsA TextField
  MsgBox("Nope!")
End Select
```

The term “Me” refers to the PushButton, so the first Case statement returns true.

Section 7

Looping

Executing Instructions Repeatedly with Loops

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

```
MsgBox("One")
MsgBox("Two")
MsgBox("Three")
```

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

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

The following types of loop structures are available:

- **While...Wend**: The loop runs until the condition specified in the While statement is satisfied.
- **Do...Loop**: The loop runs until the condition specified in the Do or Loop statements are satisfied.
- **For...Next**: The loop runs a specified number of times given in the For statement. A local counter variable controls the execution of the loop.
- **For...Each**: The loop runs repeatedly for each element in an array.

You can declare local variables inside a loop structure. When you define a local variable inside a loop structure, its scope is local to the structure itself, not the entire method. It goes out of scope after the condition of the loop is satisfied. If you need to use the variable after the loop executes, define it outside the loop, so that it is local to the method.

While...Wend

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

Consider the following example:

```
Dim i As Integer
While i < 10
  i = i + 1
Wend
```

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

Do...Loop

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

```
Do Until i = 10
  i = i + 1
Loop

Do
  i = i + 1
Loop Until i = 10
```

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

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

```
Do
  i = i + 1
Loop
```

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

Endless Loops

Make sure that the code inside your While and Do loops eventually causes the condition to be satisfied. Otherwise, you will end up with an endless loop that runs forever. Should you do this accidentally, you can click the Stop button in the Debugger. If this doesn't work, you can always "force-quit" your running application using Task Manager on Windows, the Force Quit window on OS X or the Tasks window in Linux.

Lengthy Loops

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

- If the user should wait until the loop is finished before doing anything else (e.g., if a user action might invalidate the results of the loop), you can signal that a lengthy operation is in progress by changing the mouse cursor to a "wait" cursor until the loop ends. See the section on the MouseCursor class in the Language Reference for more information:

```
Self.MouseCursor = System.Cursors.Wait
```

- If the user is permitted to do other tasks while the loop is running, you should consider using a thread. A thread runs your code concurrently with the main application (the one that handles user input and maintains the user interface), but as a background task, allowing user operations in the foreground. The Framework Guide has more information on Threads.

For...Next

While and Do loops are perfect when the number of times the loop should execute cannot be determined because it is based on a condition. A For loop is for cases in which you can determine the number of times to execute the loop. For example, suppose you want to add the numbers one through ten to a List Box. Since you know exactly how many times the code should

execute, a For loop is the right choice. For loops also differ from While and Do loops because For loops have a loop counter variable, a starting value for that variable and an ending value. The basic construction of a For loop is:

```
Dim Counter As Integer
For Counter = 0 To 100
  // [your code goes here]
Next
```

Notice that the Dim statement declares the counter as an Integer. Although an Integer is the most common way to define the counter variable, you can also declare it as a Single or Double.

In this example, the counter variable was declared in the usual way, via the Dim statement. Since counter variables are rarely needed outside the For loop, you can also declare the counter variable right inside the For statement. In other words, you can redo this example like this:

```
For Counter As Integer = 0 To 100
  // [your code goes here]
Next
```

Notice that the Dim statement has been removed from the example. If you declare the counter variable this way, you can use it only within the For loop. It goes out of scope after the For loop is finished. This is the recommended way to declare a counter variable. Of course, if you need to read or change the value of the counter variable outside the For loop, which is rarely necessary, you should use the Dim statement instead.

In the prior examples, the starting value and the ending value are specified as numbers. You can also use variables, as shown in this example:

```
Dim startingValue, endingValue As Integer
startingValue = 0
endingValue = 100

For counter As Integer = startingValue To
endingValue
  // [your code goes here]
Next
```

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

EndingValue, the counter will be incremented and the loop will end.

Look back at the example mentioned earlier. You want to add the numbers one through ten to a List Box. The following code accomplishes that:

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

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

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

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

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

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

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

Performance Considerations

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

Therefore, for performance reasons it is advisable to perform any evaluations before entering the loop. For example, consider a loop that needs to process all the fonts that are installed on the

user's computer. This number cannot be known in advance but there is a built-in function, FontCount, that you can use to obtain the total number of fonts. If you use it in the For statement to compute EndingValue (like so):

```
For i As Integer = 0 To FontCount-1
  .
  .
Next
```

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

```
Dim numFonts As Integer = FontCount-1
For i As Integer = 0 To numFonts
  .
  .
Next
```

However, the difference in speed may be of no practical value unless it is a very lengthy loop. On a typical computer the difference between these two loops is only small fractions of a second—not enough to lose sleep over.

Nested Loops

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

```
Dim count As Integer
For row As Integer = 0 To ListBox1.ListCount-1
  For column As Integer = 0 To
ListBox1.ColumnCount-1
    If ListBox1.Cell(row, column) = "hello" Then
      count = count + 1
    End If
  Next
Next
MsgBox(Str(count))
```

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

The For…Each statement

Another situation in which you want to loop through a group of values is array processing. Rather than looping through a set of statements for each value of a counter, the For…Each statement processes each element of an array that is passed to it.

Take a look at an example to sum the values of an array using a counter variable:

```
Dim values() As Double
values = Array(2.2, 1.1, 3.3, 4.4)

Dim sum As Double
Dim element As Double

For i As Integer = 0 To values.Ubound
  sum = sum + values(i)
Next
```

Here is the same example using For...Each:

```
Dim values() As Double
values = Array(2.2,1.1,3.3,4.4)

Dim sum As Double
Dim element As Double

For Each element In values
  sum = sum + element
Next
```

In the For...Each statement, instead of a counter variable, there is a variable that automatically gets the value of each element in the array. In the above example, the element variable gets the next value in the values array each time through the loop.

When you are working with arrays, For...Each allows you to write simpler code.

Since the array doesn’t necessarily have to be numbers, this statement enables you to process a group of objects of any type. They could be pictures, colors, documents, sets of database records, and so forth.

As is the case for the For...Next loop, you can declare the data type of the element variable inside the For...Each statement rather

than in a separate Dim statement. For example the previous example could be rewritten like this:

```
Dim values() As Double
values = Array(2.2, 1.1, 3.3, 4.4)

Dim sum As Double

For Each element As Double In values
  sum = sum + element
Next
```

Adding Loops Using the Code Editor

The Code Editor's contextual menu offers an especially convenient way of adding loops to your code. The last three items in the contextual menu wrap the selected lines of code inside a type of loop. To use these menu items, simply write the code that goes inside the loop, select the lines, and then choose the type of loop from the contextual menu. Your choices are If...End If, Do...Loop, and While...Wend.

The Code Editor can wrap the selected lines in the loop but it doesn't know what condition should terminate the loop. Therefore, it inserts the placeholder text _**condition**_ in the loop statement. This condition is selected for you so all you need to do is start typing to replace it with the condition you want.

Section 8

Methods

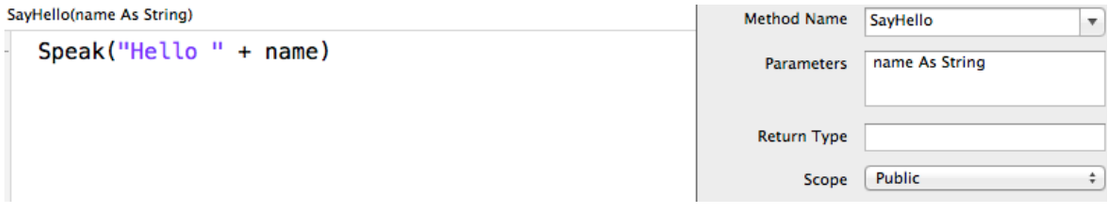
Methods are the building blocks of your application. The code you write most often exists in a method.

A method is one or more instructions that are performed to accomplish a specific task; an action of some sort. There are many built-in methods. For example, the Quit method causes your application to quit. Most classes have built-in methods. For example, the ListBox class has a method called AddRow for adding rows to it (as the name implies).

You can also create your own custom methods.

A method has four values that you set using the Inspector:

Figure 3.8 Editing a Method in the Code Editor



- **Method Name**
  The name of the method. Just like variables, methods are given names to describe them and the same rules apply: the name can be any length, but must start with a letter and can contain only alphanumeric values (a-z, A-Z, 0-9) or an underscore (_).

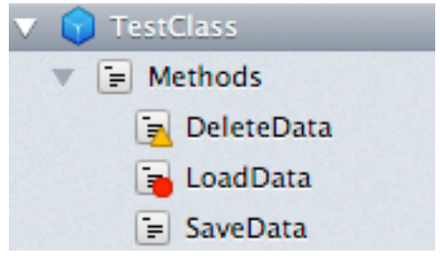
- **Parameters**
  Parameters are values that you pass to the method that it can then use.

- **Return Type**
  Methods that do not specify a return type are called **Subroutines**. Methods that specify a return type are called **Functions**.
  The Return Type can be any valid data type, including classes.

- **Scope**
  Scope indicates what can call the method. Values are Public, Protected and Private.

  - Public methods can be called by anyone with no restrictions.
  - Protected methods have

Figure 3.9 The Navigator Displaying Protected, Private and Public Methods on a Class



some restrictions, which vary depending on where the method is located (class or module).

- Private methods can only be called by the module or class that contains the method.

Passing Values to Methods

Some methods require one or more pieces of information to perform their function. These pieces of information are called parameters. Parameters are passed to a method placing them to the right of the method name. In the following example, the AddRow method of a ListBox called ListBox1 is being called. AddRow adds one row to the end of the ListBox and writes text in the new row. In order to do this, you need to pass the text to AddRow as a parameter. Here is an example:

```
ListBox1.AddRow("January")
```

When you pass parameters, you must pass values of the correct data type. AddRow, for example, requires a String. If you needed to add a number to the ListBox, you must convert it to a String. For example, if you need to add the number 12.7 to the ListBox, you can write:

```
ListBox1.AddRow("12.7")
```

Or, you can use the built-in Str function which converts a number passed to it into a string. The following also works:

```
ListBox1.AddRow(Str(12.7))
```

The Str function is itself a method that takes one parameter, a number, and returns a String. So, this expression takes the number you want to display, converts it to a string and then passes the string to the AddRow method.

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

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

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

```
Dim month As String
month = "January"
ListBox1.InsertRow(3, month)
```

Passing Arrays as Parameters

An array can be passed as a parameter in a call to a method or function. You can pass both one and multi-dimensional arrays. To specify that a parameter is a one-dimensional array, put empty parentheses after its name in the declaration. For example:

```
names() As String
```

can be used in the declaration when you want to pass an array of strings to the subroutine. Since you do not need to specify the number of elements in the array to be passed, you can pass a different number of elements at different places in your code. When you pass an array to the method or function, omit the parentheses in the array. For example:

```
PrintLabels(allNames)
```

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

You can pass multi-dimensional arrays without specifying the number of elements in each dimension, but you need to indicate the number of dimensions. Do this by placing one fewer commas in the parentheses than dimensions. For example, if *names* were a two-dimensional String array, you would declare the array in the following manner:

```
names(,) As String
```

When you pass a multi-dimensional array to a method or function, you can include the parentheses but not any commas:

```
PrintLabels(allNames())
```

Returning Values from Methods

Some methods return values. These methods are called Functions. When a method returns a value, the value is passed back from the method to the line of code that called the method. For example, the global method **Ticks** returns the number of ticks (a tick is 1/60th of a second) that have passed since you turned on your computer. You can assign the value returned by a method the same way you assign a value. In the example below, the value returned by Ticks is assigned to the variable *elapsed*:

```
elapsed = Ticks
```

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

```
carriageReturn = Chr(13)
```

In the example below, the numeric value returned by the Len function (which returns the number of characters in the string passed to it) is then passed to the Str function (which converts a numeric value to a string). The string returned by the Str function is then passed as a parameter to the InsertRow method of a ListBox:

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

Passing Parameters by Value and by Reference

By default, you pass values to a method by value. When you do so, the method receives a copy of the data that you pass allowing the method to modify it without affecting the original value.

Parameters passed by value are treated as local variables inside the method — just like variables that are created using the Dim statement. This means that you can modify the values of the parameters themselves rather than first assigning the parameter to a local variable. For example, if you pass a value in the parameter “x”, you can increment or decrement the value of x rather then assigning the value of x to a local variable that is created using Dim.

This example modifies the parameter and displays the result:

```
Sub SquareIt(d As Integer)
  d = d * d
  MsgBox(Str(d))
End Sub
```

When you write your own methods, you have the option of passing information by reference. When you pass information by reference, you actually pass a pointer to the object containing the information. The practical advantage of this technique is that the method can change the values of each parameter and replace the values of the parameters with the changed values. When you

pass parameters by value, you can't do this because the parameter only represents a copy of the data itself.

Note: All arrays and any object types are always passed by reference regardless of what is specified in the method declaration.

Very few built-in methods work like this. One of them is the ParseDate function. ParseDate takes a date written as text and returns a Date object. What is unusual about ParseDate is that the Date that is returned is in a parameter that is passed to it. The ParseDate function itself returns a Boolean value that indicates whether the function was successful or not.

A method or function that passes a parameter by reference uses the keyword ByRef in its syntax. In this case, the syntax for ParseDate is:

```
result = ParseDate(Text, ByRef Parsedate)
```

- Text contains the string that will be parsed into a Date
- ParsedDate contains the Date, provided the call to the function works
- Result is a Boolean value that will indicate whether or not the function call was successful.

To use ParseDate, declare a Date variable and a Boolean variable with Dim statements. Then pass the new Date variable along with the string to be parsed. After the call, test the result to determine whether the function was successful. For example:

```
Dim theDate As New Date
Dim success As Boolean
success = ParseDate("12/14/1991", theDate)
If success Then
  MsgBox(thedate.AbbreviatedDate)
Else
  MsgBox("Invalid Date format!")
End If
```

Optional Parameters and Default Values

There are two ways to make a parameter optional, both of which you do at the time you write the declaration: You can assign a value to it in the declaration or you can use the Optional keyword in the declaration.

The way to make the parameter's explicit status clear is to use the Optional keyword. This modifier precedes the parameter name and may be used with or without a default value. If the caller omits this parameter, it will receive the default value for its data type if no default value was passed in the declaration. For example:

```
MyMethod(a As Integer, b As Integer,
Optional c As Integer)
```

The parameter c is optional and if omitted from the method call gets the default value of an integer, which is 0.

If you want to specify a different default value, you assign it:

```
MyMethod(a As Integer, b As Integer,
Optional c As Integer = 5)
```

In this example, the parameter c is still optional but if it is omitted from the method call it gets the value of 5.

Lastly, you can omit the **Optional** keyword when supplying the default value:

```
MyMethod(a As Integer, b As Integer, c As
Integer = 5)
```

Identifying the Method Name in Code

For logging purposes, it can often be useful to know the name of the currently running method. Use the CurrentMethodName Constant in your code to get the name of the currently running method:

```
MsgBox(CurrentMethodName)
```

Setters

A value passed to a method is normally supplied in parenthesis following the method name, such as:

```
MyMethod(10)
```

Sometimes it is preferred to have the method call behave differently so that you assign the parameter like this:

```
MyMethod = 10
```

This is called a Setter because it looks like you are setting the MyMethod value to 10 rather than passing 10 as a parameter to MyMethod. You can enable this alternative syntax by using the Assigns keyword in the parameter declaration:

```
Sub MyMethod(Assigns value As Integer)
```

Events

Events are a special type of method commonly used with control classes. But they can also be implemented in your own non-control classes as well. Events can be called only by the class that declares the event. Generally speaking, events are used to provide a way for subclasses to provide additional functionality.

You don’t need to worry about Events right now. Both subclasses and events are described in more detail in the Classes chapter.

Chapter 4

Modules

Learn how modules can be used to organize your code.

CONTENTS

Section 1

About Modules

Understanding Modules

A module is a collection of project items, usually methods, properties and constants. But a module can also contain other project items such as classes or even other modules. In fact, modules can pretty much contain anything except Windows and Container Controls.

Modules are considered global and are accessible anywhere in your project.

However, you can control the scope of the items you add to modules.

When to Use a Module

In general, modules should used sparingly in an object-oriented language such as Xojo. In most cases, you will probably be better served by a class.

With that said, here are some common (and valid) uses for modules:

- Global constants, particularly for localization
- Grouping project items into namespaces
- Class extensions
- Global properties and methods, but these should be minimal

Modules are covered here first since they are easier to use and learn and transition nicely to classes (which are covered in the next chapter).

Adding a Module

You can add a new module to your project by clicking the Insert button on the toolbar and selecting Module (or by using the menu Insert ↠ Module or the contextual menu). The new module appears in the Navigator with a default name (the first module you add will be named **Module1**, for example). You can use the Inspector to rename the module to something more appropriate. For example, if the module contains financial functions, you might name it **Financial**.

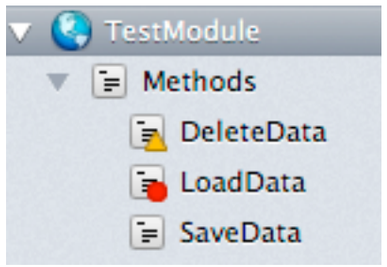
You edit modules with the Code Editor: simply click the module's name in the Navigator. Modules are identified by their special icon () in the Navigator.

Modules primarily contain properties, constants and methods.

Adding Properties to Modules

Properties are variables that belong to the entire module rather than a single method.

Figure 4.1 The Navigator Displaying Protected, Private and Global Methods on a Module



To add a property to a module, use the Add button on the Code Editor toolbar, Insert ↠ Property from the menu, the contextual menu or the keyboard shortcut (Option-Command-P on OS X or Ctrl+Shift+P on Windows and Linux).

You can set the property name, type, default value and scope using the fields in the Inspector.

***Note**: To quickly create a property, you can enter both its name and type on one line in the Name field like this: PropertyName As DataType. When you leave the field, the type will be set in the Type field.*

Adding Methods to Modules

To add a method to a module, use the Add button on the Code Editor toolbar, Insert ↠ Method from the menu, the contextual menu or the keyboard shortcut (Option-Command-M on OS X or Ctrl+Shift+M on Windows and Linux).

You can set the method name, parameters, return type and scope using the Inspector.

Adding Constants to Modules

As covered earlier in Chapter 3: The Xojo Programming Language, constants are similar to variables whose values cannot be changed while the program is running.

When you can add a constant to the module, it belongs to the module and not just a single method.

To add a constant to a module, use the Add button on the Code Editor toolbar, Insert ↠ Constant from the menu, the contextual menu or the keyboard shortcut (Option-Command-C on OS X or Ctrl+Shift+C on Windows and Linux).

Unfortunately, constants created in this manner cannot be assigned values from functions like they can when created using the Const keyword in a method. One way to simulate this behavior is to instead add a method to the module that returns the desired value.

For example, to create a method that returns a carriage return character (ASCII 13), you could add the following method to a module:

```
Function CR as String
  Return Chr(13)
End Function
```

Scope of a Module's Items

When you add an item to a module, you need to set its Scope using the Scope property in the Inspector. The Scope is also indicated in the Navigator. The Scope of an item determines its accessibility to other items in the project. There are three choices:

- **Global**: A Global item is available to code throughout the application. The Global scope is available only for items in modules. For example, you can use a global property to store a piece of information that needs to be available to several different windows and to the application as a whole even if no window is open (OS X only). When your code needs to access a global item you simply reference it by name from anywhere in the application. The Global scope is not available for items in nested modules.

- **Public**: A Public item is also available to code throughout the application. When you need to access a public item outside of the module, you use the "dot" notation and precede its name with the module's name. For example, if you declare a Public property, *MyPublicProperty*, in Module1, you call it with the syntax `Module1.MyPublicProperty` outside Module1. If you create a Public property in a nested module, you need to include the full path to the property when referring to it outside its module. For example, if you declare a Public property, myPublicProperty, in Module2 which is nested in Module1, you refer to it outside its module as `Module1.Module2.MyPublicProperty`.

- **Private**: A Private item is available only within the module. It is "invisible" to the rest of the application. When you need to access the Private item inside the module in which it was created, you simply reference it by name. If you create a Private item in a nested module, it cannot be accessed by higher-level modules. However, Private items in higher-level modules can be accessed from nested modules.

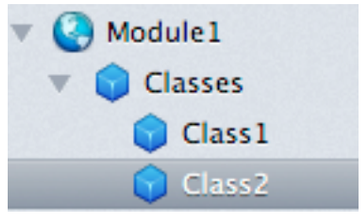
Grouping Project Items (Namespaces)

Adding Project Items to Modules

Modules can also contain other project items, including classes, class interfaces and even other modules. Modules cannot contain Windows, Web Pages, iOSViews or Containers.

The ability of modules to contain classes, class interfaces, and other modules represents the module namespace system. The classes, interfaces, and modules "live" in the module's namespace. A module namespace is in contrast to the **root** namespace, which is represented by the project itself. Classes, class interfaces, and modules that are listed in the Navigator are technically at the root namespace of the project.

Figure 4.2
Navigator Showing a Module Containing Classes



A class in a module "lives" inside the module. Outside the module, a module class is referenced by dot notation. Some examples:

```
Dim c As New Module1.Class1
c.MethodCall
```

In the above example, Class1 is contained within Module1.

A module in the Navigator has a disclosure widget (a small triangle) if it contains other project items.

The Using Statement

You can use the Using statement to avoid having to write the full namespace name each time you want to access an item within it. For example, you can write code like this to avoid having to use full dot notation to access classes within Module1:

```
Using Module1
Dim c As New Class1 // in Module1
c.MethodCall
```

Converting a Project Class to a Module Class

You can change a project class to a module class. Project classes appear in the Navigator and are available globally to the project.

To convert a Project class to a module class in the Navigator, drag the class onto the name of the module.

Notice a blue highlight box surround the module name. This indicates that the class will be added to the module.

When the drag is successful, the class appears indented in the module.

Nesting a Module in a Module

A module can contain other modules. A module nested in another module can have classes, class interfaces, methods, properties, constants, structs and enums just like the top-level module. The scope of the item determines how you refer to it outside its module. However, higher-level modules cannot "see" the items in nested modules.

You refer to items in a module by prefixing the module name.

The new module can contain other modules. The nesting of modules can continue indefinitely.

A method inside a class inside a module sees the module's members in the same way that a method directly inside the module would. The contained class has access to all the private module members, and can refer to the public module members without having to specify the module's name.

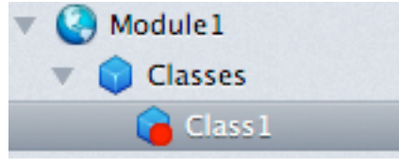
Scope of a Nested Module's Items

When you add an item to a nested module, you need to set its Scope attribute. The Scope of an item determines which other items in the project can access it. There are two possible values:

- Public: A Public item is also available to code throughout the application. If you create a Public item in a nested module, you need to include the full path to the item when referring to it. For example, if you declare a Public property, MyPublicProperty, in Module2 which is nested in Module1, you refer to it outside its own module as `Module1.Module2.MyPublicProperty`. If you declare a Public property in Module1, it can be accessed in Module2 as `Module1.MyPublicProperty`.

- Private: A Private item is available only within the module. It is "invisible" to the rest of the application. When code inside the module needs to access a Private method, property, or constant, you simply reference it by name. If you create a Private item in a nested module, it cannot be accessed by

Figure 4.3 A Module Containing a Private Class



higher-level modules. However, Private items in higher-level modules can be accessed from nested modules.

Section 3

Extension Methods

A class extension method is a method that can be called using syntax that indicates that it belongs to another object. For example, you can add a method that is called from any FolderItem object to save something in a particular format. After you add the class extension method to a module, you can call it as if it were built into the FolderItem class.

This feature is useful when you need to extend a class for which you do not have access to the source code. In particular, it allows you to extend the functionality of built-in framework classes.

To define a method as a class extension method, use the **Extends** keyword prior to the first parameter. The data type of the first parameter is the object type from which the method must be called. In other words, the use of the Extends keyword indicates that the parameter is to be used on the left side of the dot (“.”) operator in a calling statement. When you use the Extends keyword, you do not have to pass an object of that type to the method. You can add “normal” parameters to the parameter declaration that follow the Extends parameter.

The Extends keyword can be used only with global methods that reside in modules. As long as the module is in the project, the class extension method can be called from anywhere in the project.

To use the class extension method in another project, simply import the module into that project.

Writing a Class Extension Method

For example, you can add a method that can be called from a Text Area object. Suppose you want a method, **Clear**, that clears the text in the Text Area. Create a module in the project if it doesn’t already have one and add a **Clear** method to the module.

This is what the Parameter is:

```
Extends ta As TextArea
```

The code for this extension is simple:

```
ta.Text = ""
```

The Extends keyword indicates that the method can be called from any Text Area object that is in the project. Note also that the Scope for the class extension method must be Global.

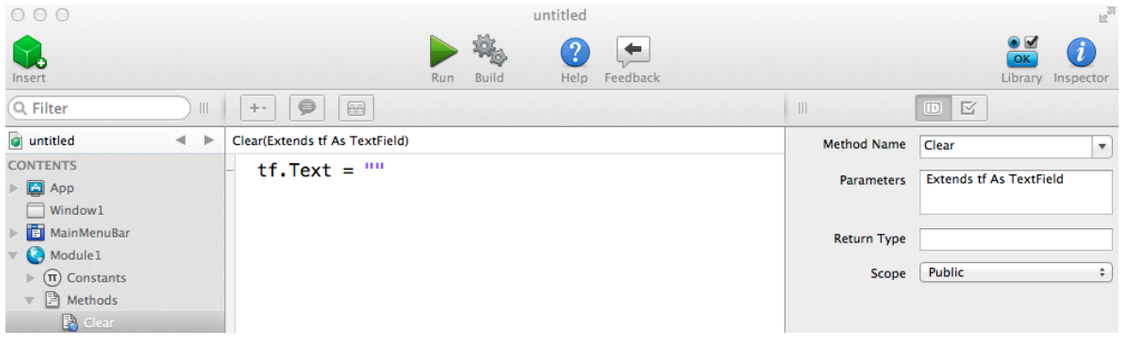
Calling a Class Extension Method

You can call this method from anywhere in the project. You can simply use the line somewhere on a Window:

```
TextArea1.Clear
```

This assumes that there's a Text Area named TextArea1.

Figure 4.4 Clear Extension Method



To use the class extension method in another project, all you need to do is copy the module that contains the definition of the class extension method to the other project.

Chapter 5

Classes

Classes are the fundamental building blocks of all Xojo applications. This chapter covers object-oriented design and how to effectively use classes in your projects.

CONTENTS

Section 1

About Classes

What is a Class?

In its simplest form, a class is a container of code much like a module. But unlike a module, a class provides better code reuse.

Classes are the fundamental building blocks of object-oriented programming.

The Benefits of Classes

Classes offer lots of benefits, including:

- **Reusable Code**
  When you add code to a PushButton control to customize its behavior, you can only use that code with that one PushButton. If you want to use the same code with another PushButton, you need to copy the code and then make changes to the code in case it refers to the original PushButton (since the new PushButton will have a different name than the original).

  Classes store the code once and refer to the object (like the PushButton) generically so that the same code can be reused any number of times without modification. If you create a class based on the PushButton control and then add your code to that class, any instances of that custom class will have that code.

- **Smaller Projects and Applications**
  Because classes allow you to store code once and use it over and over in a project, your project and the resulting application is smaller in size and may require less memory.

- **Easier Code Maintenance**
  Less code means less maintenance. If you have basically the same code in several places in your application, you have to keep that in mind when you make changes or fix bugs. By storing one copy of the code, you will spend less time tracking down all those places in your project where you are using the same code. Making a change to the code in a class automatically updates any places where the class is used.

- **Easier Debugging**
  The less code you have, the less code there is to debug.

- **More Control**
  Classes give you more control than you can get by adding

code to the event handlers of a control in a window. In fact, some classes can even manage menus. You can also use classes to create custom controls. And with classes, you have the option to create versions that don't allow access to the source code of the class, allowing you to create classes you can share or sell to others.

Using Classes in Your Projects

Before you can use a class in your project, it is important to understand the distinction between these three concepts: the class itself, the instance of the class and the reference to the class.

The Class

Think of the class as a template for a container of information, much like a module. And like a module, each class exists in your project only once. But unlike a module, a class can have multiple instances.

The Instance

Classes provide better code reuse because of a concept called instances. Unlike a module, which exists only once in your application, a class can have multiple instances. Each instance (also called an **object**) is a separate copy of the class and all its methods and properties.

For example, there are many built-in classes for user interface controls such as PushButton, WebButton, Label, WebLabel, TextField, WebTextField, ListBox and WebListBox.

By themselves, control classes are not all that useful; they are just abstract templates. But each time you add a control class to a window you get an instance of the class. Because each button is a separate instance, this is what allows you to have multiple buttons on a window, with each button being completely independent of each other.

These instances are what you interact with when writing code on the window and is what the user interacts with when they use your application.

For example, when you drag a TextArea from the Library to a window, you create a usable instance of the TextArea on the window. The new instance has all the properties and methods that were built into the TextArea class. You get all that for free — styled text, multiple lines, scroll bars, and all the rest of it. You customize the particular instance of the TextArea by modifying the values of the instance's properties.

When you add a control to a window (or web page), the Layout Editor creates the reference for you automatically (it is the name of the control).

When you write code, you create instances of classes using the New keyword like this:

```
Dim car As New Vehicle
```

The Reference

A reference is a variable or property that refers to an instance of a class.

In the above code example, the *car* variable is a reference to an instance of the Vehicle class.

You interact with properties and methods of the class using dot notation like this:

```
Dim car As New Vehicle
car.Brand = "Ford"
car.Model = "Focus"
```

Here the Brand and Model were defined as properties of the Vehicle class and they are given values for the specific instance.

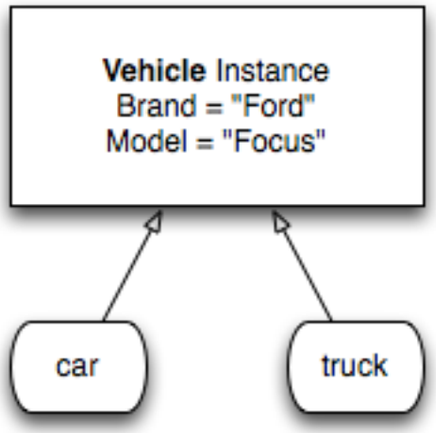
Keep in mind that this variable or property is pointing to a reference to the instance. This has important considerations when assigning references to another variable.

When you do an assignment from one reference variable to another, the second variable points to the same reference as the first variable. This in turn means it refers to the same instance of the class, not a copy of it. If you change a property of the class with either variable, then it is changed for both. An example might help:

```
Dim car As New Vehicle
car.Brand = "Ford"
car.Model = "Focus"

Dim truck As Vehicle
truck = car
// truck.Model = "Focus"

car.Model = "Mustang"
// truck.Model is now also "Mustang"

truck.Model = "F-150"
// car.Model is now also "F-150"
```

Looking at Figure 5.1, you can see that the variables for both car and truck point to the same instance of Vehicle. So change either one effectively changes both.

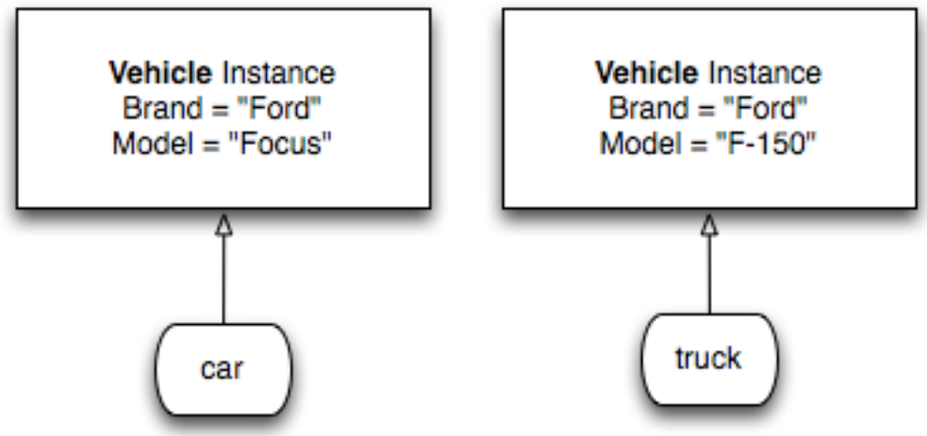
Figure 5.1 Two variables that point to the same instance



If you want to create a copy of a class, you need to instead create a new instance (using the New keyword) and then copy over its individual properties as shown below:

```
Dim car As New Vehicle
car.Brand = "Ford"
car.Model = "Focus"

Dim truck As New Vehicle
truck.Brand = car.Brand
trunk.Model = car.Model

// truck.Model is now also "Focus"
truck.Model = "F-150"

// car.Model remains "Focus"
```

When you do it this way, you get two separate instances. Changes to one do not affect the other.



Creating Classes

Adding a class to a project is easy. To add a new class, click the Insert button on the toolbar and choose Class or select Class from the Insert menu. This adds a new class to the Navigator with the default name (Class1 for the first class).

Use the Inspector to change the name of the class.

Like modules, classes primarily contain properties and methods.

Adding Properties to Classes

Properties are variables that belong to an entire class instance rather than just a single method.

To add a property to a class, use the Add button on the Code Editor toolbar, Insert ↠ Property from the menu, the contextual

menu or the keyboard shortcut (Option-Command-P on OS X or Ctrl+Shift+P on Windows and Linux).

You can set the property name, type, default value and scope using the Inspector.

***Note**: To quickly create a property, you can enter both its name and type on one line in the Name field like this: PropertyName As DataType. When you leave the field, the type will be set in the Type field.*

Properties added in this manner are sometimes called **Instance Properties** because they can only be used with an instance of the class.

You can also add properties that can be accessed through the class itself without using an instance. These are called Shared Properties.

Shared Properties

A shared property (sometimes called a **Class Property**) is like a "regular" property, except it belongs to the class, not an instance of the class. A shared property is global and can be accessed from anywhere its scope allows. In many ways, it works like a module property.

It is important to understand that shared methods are literally shared. If you change the value of a shared property, it essentially is changed everywhere.

Generally speaking, shared properties are an advanced feature that you only need in special cases.

Adding Methods to Classes

To add a method to a class, use the Add button on the Code Editor toolbar, Insert ↠ Method from the menu, the contextual menu or the keyboard shortcut (Option-Command-M on OS X or Ctrl+Shift+M on Windows and Linux).

You can set the method name, parameters, return type and scope using the Inspector.

***Note**: When typing the method name, the field will autocomplete with the names of any methods on its super classes.*

Methods added in this manner are called **Instance Methods** because they can only be used with an instance of the class.

You can also add methods that can be accessed through the class itself. These are called Shared Methods.

Shared Methods

A shared method (sometimes called a **Class Method**) is like a normal method, except it belongs to the class, not an instance of the class. A shared method is global and can be called from anywhere its scope allows. In many ways, it works like a module method.

It is important to understand that shared methods are literally shared. They do not know about an instance so can only access other shared methods or properties of the class.

Generally speaking, shared methods are an advanced feature that you only need in special cases. For example, if you are using an instance of a class to keep track of items (e.g., persons, merchandise, sales transactions, and so forth) you can use a shared property as a counter. Each time you create or destroy an instance of the class, you can increment the value of the shared property in its constructor and decrement it in its destructor. (For information about constructors and destructors, see the section Constructors and Destructors.) When you access it, it gives you the current number of instances of the class.

Object-Oriented Design Concepts

This section covers some of the important object-oriented design concepts that apply to classes: Encapsulation, Overloading, Inheritance and Polymorphism.

Encapsulation

Encapsulation is the process of hiding information that does not need to be exposed. With classes, you can control this using the scope of your properties and methods. In order from least visible to most visible, there are: Private, Protected and Public.

Private

Setting the scope to private means that the item is only accessibly from with the current class instance (or class for shared items). This makes it invisible to the rest of your code.

This is the most restrictive setting. It is good coding practice to get into the habit of setting your items to Private if they are not needed outside the class instance.

It is common practice to prefix private properties with “m”, such as:

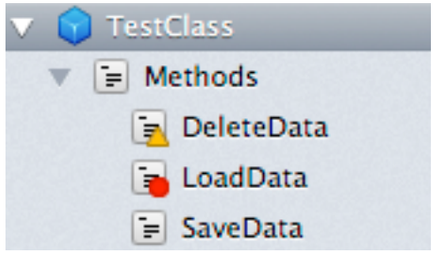
```
mUserName As String
```

Protected

A scope of protected means the item is accessible from the current class instance (or class for shared items) and its subclasses.

A Protected method, property, or constant is available only to other code within the class and subclasses based on the class. It is “invisible” to the rest of the application. When other code in the class needs to access a Protected method, property, or constant, you simply reference it by name. If you try to access a Protected method, property, or constant outside of the class, you will get an informative error message indicating that the item is out of scope when you try to run your project.

Figure 3.9 The Navigator Displaying Protected, Private and Public Methods on a Class



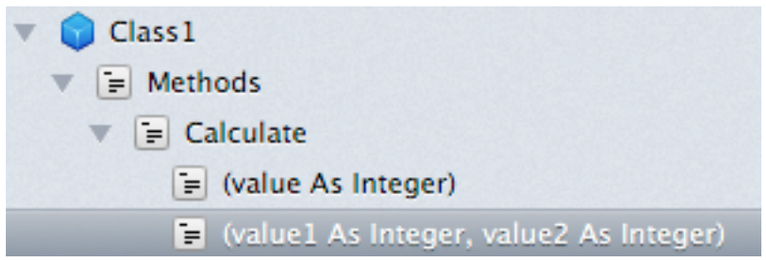
Public

A Public method, property, or constant is available to code throughout the application. You reference it using dot notation using the class reference and the name, such as `car.Model`.

Overloading

Overloading is the term for methods that have the same name, but have different method signatures (such as a different number of parameters, different data types for the parameters or belonging to a subclass).

Figure 5.2 Overloaded Calculate Method



***Note**: You cannot overload methods based on return values.*

To overload a method in your own classes, simply add the new method to the class and give it different parameters. The method appears in the Navigator with the overloaded parameters shown below the method so that you can tell them apart.

To overload methods in built-in framework classes, you first need to create a subclass using inheritance.

You can also overload operators using a variety of specially implemented functions. A good example of a built-in overloaded operator is the "+" operator. If its arguments are numbers, it computes the sum; if the arguments are strings, it concatenates the strings. The *Advanced Class Features* section in this chapter has an example of this.

Inheritance (Subclassing)

Inheritance a feature of object-oriented programming where you create a new class that is based on an existing class. The new class is called the subclass and the existing class is called the super class.

Why is this useful? You may find situations where you would like to have an object that is a slightly altered version of one of the built-in classes. For example, you might want a version of the TextArea control that disables the Cut and Copy items on the Edit menu, preventing the user from putting sensitive data on the Clipboard. You might want to create a List Box that, by default, has the months of the year in it. You can create your own versions of these built-in classes by creating subclasses that you add to your project.

There's an important difference between adding an instance of TextField to a window versus adding a subclass based on TextField to your project. In the latter case, you can customize the subclass based on TextField itself and use instances of the customized subclass in several places in the application. You can also save the customized subclass so that you can reuse it in other projects.

What is a Subclass?

A subclass is simply a class that has a **super** class. A super class is a class the subclass is based on, also sometimes called the **parent** class. Subclasses inherit all of their super's properties, methods, constants, and events. The subclass can then modify them. In fact, a subclass is identical to its super class until you start modifying it. After that, it's different from its super class only in the ways you make it different by adding properties, modifying events, and adding or modifying methods.

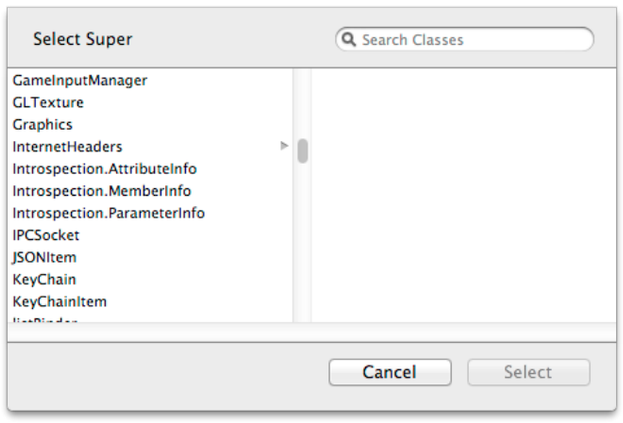
Creating a Subclass from an Existing Class

There are a several ways to create a subclass from an existing class in your project or from an existing framework class.

1. If want to subclass a built-in control class, you can simply drag the control from the Library to the Navigator.

2. You can select a class in the Navigator and use the contextual menu option "New Subclass" to create a new subclass based on the selected class.

Figure 5.3 Select Super Class Dialog

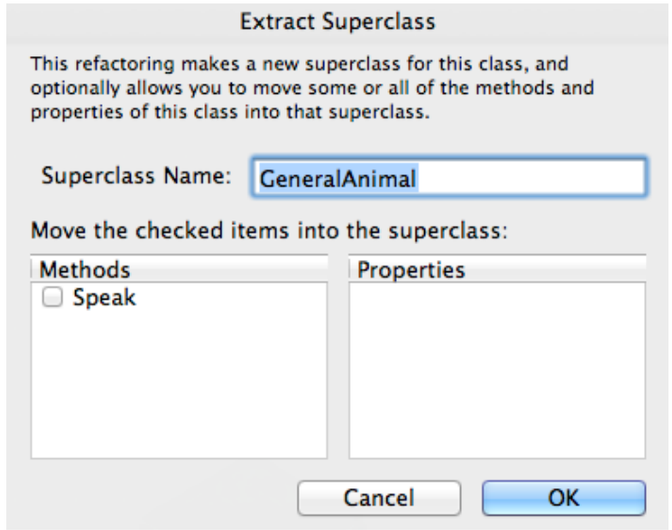


3. You can add a class to your project (using the Insert button on the toolbar or the Insert menu) and then manually change its Super property to the name of the class that it is based on using the Select Super Class Dialog.

Creating a Superclass from an Existing Class

If you have an existing class that you have determined should really be a subclass, you can create a super class from it. Use the contextual menu for the class and select "Extract Superclass". This opens a dialog (Figure 5.3) that allows you to specify the name of the super class and to choose the method and properties to move from the class to the super class.

Figure 5.4 Extract Superclass Dialog



Virtual Methods (Overriding)

Virtual methods provide a way for a subclass to override its own version of a method that its super class has. Ordinarily, a subclass inherits the methods belonging to its parent.

When a subclass has a method that has the same name (and parameters) as a method on the super class, the subclass

method is instead called. This is referred to as “overriding”. You can still call the method on the superclass by calling it specifically using the Super prefix with the method call:

```
Super.MethodName
```

Polymorphism

Polymorphism is the ability to have completely different data types (typically classes) behave in a uniform manner. You can do this using inheritance and overriding, inheritance and events or class interfaces (described later in this chapter).

A common example of this is the Animal, Cat and Dog relationship. If you want to have a generic animal object that knows how to speak “Meow!” or “Woof!” depending on whether it contains a Cat or Dog, you could implement it using inheritance and overloading.

First, create a new class called Animal. Add to it a Speak method that returns a String.

Now create a subclass of Animal, called Cat. Add to it a Speak method that returns a String. This means you are overriding the unimplemented Speak method on Animal. The code for this method is:

```
Return "Meow!"
```

Next, create a subclass of Animal, called Dog. Add to it a Speak method that returns a String with this code:

```
Return "Woof!"
```

To test this, create a button on a window. In the Action event of the button add code to create an array of Animals:

```
Dim animals() As Animal
animals.Append(New Cat)
animals.Append(New Dog)

For Each a As Animal In animals
  MsgBox(a.Speak)
Next
```

When you run this code, you will see “Meow!” and then “Woof!”.

Because Cat and Dog are both subclasses of Animal, they are allowed to be assigned to a variable (the animals array) with a type of Animal.

And when you call the Speak method of Animal (in the loop), because of polymorphism, your code calls the Speak method of the actual subclass type that was added to the array.

Later sections will also show you how you can accomplish the same thing using inheritance with events and with class interfaces.

Properties, Methods and Events

Properties

Properties are a trait that tells you something about an object.

You use them as a way for class instances to store values. You can add four types of properties to classes: properties, computed properties, shared properties and shared computed properties.

You can add properties to a class to store values that its super class doesn't store. For example, you might want to create a subclass of the TextField control that stores the last value the user entered. This would allow you to selectively reject the current entry and restore the last entry.

Properties

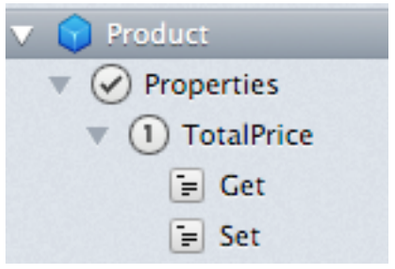
Properties (Instance Properties) were described in Section 1. They are properties that belong to a class instance.

Computed Properties

A computed property is a property that does not store its value but instead calculates it each time it is accessed.

A computed property has two parts to it: a getter and a setter. This displays in the Navigator as a Get and Set node below the property name. You add code to the Get node to get a value for the property. You add code to the Set node to set a value for the property. If you do not include code in Set then the property becomes a read-only property.

Figure 5.5 A Computed Property



You can also leave out code in the Get node to make the property write-only, but that is rarely useful.

An example of a computed property might be a TotalPrice property of a Product class.

You could implement TotalPrice with this code in Get:

```
Return Price * Quantity
```

If your computed property needs to retain a value, often you would create a matching private property to do so (usually prefixed with an “m”). This means you might do something like this:

```
Get:
Return mName

Set:
mName = Value
```

Shared Properties

As discussed in Section 1, Shared Properties (also called class properties) are properties that belong to the class rather than an instance.

Shared Computed Properties

Shared Computed Properties work exactly like you expect. They are shared versions of computed properties.

Methods

Methods provide functionality for your classes. They usually perform some action, such as loading data or calculating values.

Chapter 3, Section 8 discusses methods in more detail.

Methods

Methods (Instance Methods) were described in Section 1. They are methods that belong to a class instance.

Shared Methods

As discussed in Section 1, Shared Methods (also called class methods) are methods that belong to the class rather than an instance.

Events

Events are a type of method that is called by some action (or event) that has occurred. These events have nothing to do with the “events” in “event-driven programming”.

Many classes and controls have their own built-in events (which are covered in the User Interface and Framework books), but you can also create your own events using Event Definitions.

Event Definitions gives you a way to add your own Event Handlers to your classes. Event Definitions can be called only from the class itself, but they can be implemented only by its subclasses.

To add an event definition to a class, use the Add button on the Code Editor toolbar, Insert ↠ Event Definition from the menu or the contextual menu.

For example, if you have a Save method on a class you may want to give subclasses a way to do processing before and after the

save. One way you could do this is by overriding the Save method on the subclass like this:

```
Sub Save
  PreSave
  Super.Save
  PostSave
End Sub
```

This works but it relies on you defining and implementing everything properly. If you do this with events, then there is no room for error.

You would create two event definitions on the super class: PreSave and PostSave.

Figure 5.7 Event Definitions

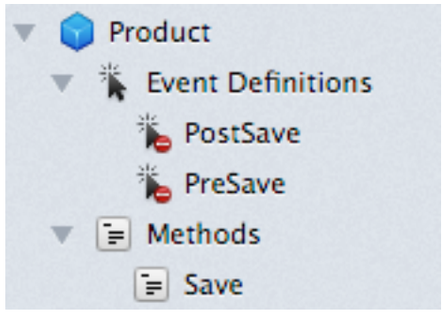
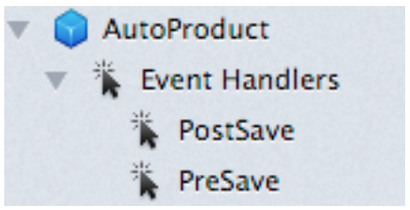


Figure 5.6 Subclass Showing Event Handlers



In the Save method on the super class, you then call PreSave at the beginning and PostSave at the end.

When you create a subclass, you will see that it has two Event Handlers you can add to it: PreSave and PostSave. Simply add those event handlers and implement them as needed.

The built-in control classes all contain a wide variety of event handlers (such as Open), all of which work in this manner.

Note that when you implement an event handler, it no longer appears in subclasses. If you want the event handler to still be available to additional subclasses, you need to create a new event definition in the subclass (matching the name and parameters) and then call it in the event definition you implemented.

Animal Example

Looking back at the Animal example, this is how you would do it using events.

Create a new class called Animal. Add to it a Speak method that returns a String with this code:

```
Return SpeakSound
```

Now add to the Animal class an Event Definition called SpeakSound and set its return value to String.

Next, create a subclass of Animal, called Cat. Click on the button "+" button on the toolbar and select "Add Event Handler". In the

dialog, you will see the SpeakSound event handler. Select and and press OK.

The code for this event handler is:

```
Return "Meow!"
```

Now create a subclass of Animal, called Dog and also add the SpeakSound event handler with this code:

```
Return "Woof!"
```

To test this, create a button on a window. In the Action event of the button add code to create an array of Animals:

```
Dim animals() As Animal
animals.Append(New Cat)
animals.Append(New Dog)

For Each a As Animal In animals
  MsgBox(a.Speak)
Next
```

When you run this code, you will see “Meow!” and then “Woof!”.

Because Cat and Dog are both subclasses of Animal, they are allowed to be assigned to a variable (the animals array) with a type of Animal.

When you call the Speak method of Animal (in the loop), it in turn calls the SpeakSound event handler of your subclass that was added to the array.

Section 4

Constructors and Destructors

When you create a new object, you will sometimes want to perform some sort of initialization on the object. The constructor is a mechanism for doing this. An object's constructor is the method that is executed automatically when an instance of the class is created.

Conversely, when an object is removed from memory, its destructor is called.

Constructors

You add a constructor to a class by adding a method to the class called Constructor. For convenience, "Constructor" is a choice in the Method Name popup menu in the Code Editor.

The constructor will be called automatically when an instance is created via the New operator. If your constructor accepts parameters, you must pass the required number of parameters and they must be of the correct data type.

Whenever you create a constructor for a subclass, the Code Editor automatically adds a call to the super class's constructor for you in the Code Editor and adds comments that explains what it is doing.

```
// Calling the overridden superclass constructor.
Super.Constructor
```

This is added because the constructor that you are writing overrides its Super class's constructor, but the new object may not be initialized correctly unless its Super class's constructor executes. The Super method is how you call a method of a Super class that is being overridden by a method of the subclass.

Here is a simple example of a constructor. Suppose you are managing a service that sells monthly subscriptions. You want a custom version of the Date class that automatically takes the value of the expiration date when an instance is instantiated. First, add a new class to the project and set its Super class to Date and rename it "ExpirationDate". Now you can add a method called "Constructor". The constructor should take one integer parameter, numMonths, which the number of months the customer has signed up for. So the constructor has only one line of code:

```
Self.Month = Self.Month + numMonths
```

When an instance of a “regular” Date object is created, it is initialized to the current date, so this line increments the current month number by the number of months that is passed in as a parameter.

When you need to get the expiration date for a customer who signs up for 6 months, you can get it like this:

```
Dim expDate As New ExpirationDate(6)
MsgBox(expDate.ShortDate)
```

The number months is passed in as a parameter and the new instance holds the expiration date instead of the current date.

Initializing Control Subclasses

If you need to initialize an instance of a control subclass, don’t use the constructor. Instead, put the code that does the initialization in the Open event handler for the control.

When you access Inspector properties in the Constructor, they will not have the default value specified in the Inspector and will instead have the default value for the data type. If you assign values to these public properties, the values will get overridden with what is specified in the Inspector.

Destructors

You can also create a destructor, although they are not often necessary. The destructor is called automatically when an instance of the parent class is deleted or goes out of scope — for example, when the user closes the window. Name the new method “Destructor”. Destructors take no parameters and do not return a value.

Destructors are called when the last reference to an object is removed, even if execution is in a destructor for another object. Note that this means you can cause a stack overflow if your destructor triggers other destructors in a deep recursion. However, such overflow will not happen as long as properties of the object are being cleaned up automatically. So, it is generally preferable to not set properties to Nil in your destructor, but instead allow them to be automatically cleaned up for you.

Memory Management

Memory is managed for you automatically using something called **reference counting**. A counter is used for each instance (object) that you create and it records references to the object. Each reference increments the counter; removing a reference to the object decrements the counter. You remove a reference, for example, by setting it to Nil, by letting it go out of scope such as by closing the window in which the instance is located. When the

reference counter reaches zero, the object is removed from memory immediately.

This means that instances of classes are removed from memory automatically when they are no longer used. Suppose you create a class based on a ListBox. You then create an instance of that class in a window. When the window is opened, the instance of the class is created in memory automatically. When the window is closed, the instance of the class is automatically removed from memory. If you store the reference to a class in a local variable, when the method or event handler is finished executing, the instance of the class is removed from memory. If you store a reference to an instance of a class in a property, the instance will be removed from memory when the object owning the property is removed from memory.

Section 5

Interfaces

A class interface is a construct that you can use to tie together classes that do not share a super class but have something in common in your application. Class interfaces are used to specify what an object does without specifying how it does it.

In order to understand class interfaces better, it's helpful to think of a class as consisting of two components, the (public) interface to the class and the implementation. The interface consists of the class's Public method calls and the implementation is the code that implements the methods. The interface says what the class does and the implementation says how it does it.

In the object hierarchy, a subclass inherits both the interface and the implementation from its super class. That is, it gets both the method calls and the specific implementation of the method.

Class interfaces enable you to separate the two constructs. If two or more classes need to do the same thing but do it in different ways, you use an interface instead of a super class.

A class interface operates as a "spec" that contains a list of methods that custom classes in your project use. It does not actually contain any code for the methods themselves.

The methods in the class interface are placeholders for methods that are actually contained in each custom class that "implements" the class interface. Also, a custom class can implement more than one class interface.

The term "implement" simply means that the class has methods of the same names and declarations that are found in the class interface. The class interface specifies the methods and their declarations but not the code.

Several class interfaces are built-in to the framework. You can implement any of these in your classes or add and implement your own. For example, the Readable and Writeable interfaces specify methods for reading and writing data. Each class that uses the Readable or Writeable interfaces supplies the implementations. For example, a class that reads data from the Serial port would use a different implementation than a class that reads data from a binary file.

When you specify that a class implements a class interface, the class must implement all the methods in the class interface and the method declarations must match. However, the classes are

free to implement the methods in different ways. For example, a method that changes the font in a Label would be implemented in a different way than a method that changes the font in a Canvas control that displays text via calls to the Graphics class.

The process involves three basic phases:

- Creating the class interface,
- Creating the classes that implement the class interface,
- Adding the classes to your project and calling the class interface methods in your program. Typically, that means writing generic code that tests whether a class implements a class interface and executing class interface methods where appropriate.

To add a new class interface, click the Insert button on the toolbar and choose Class Interface or select Class Interface from the Insert menu. This adds a new class interface to the Navigator with the default name (Interface1 for the first class interface).

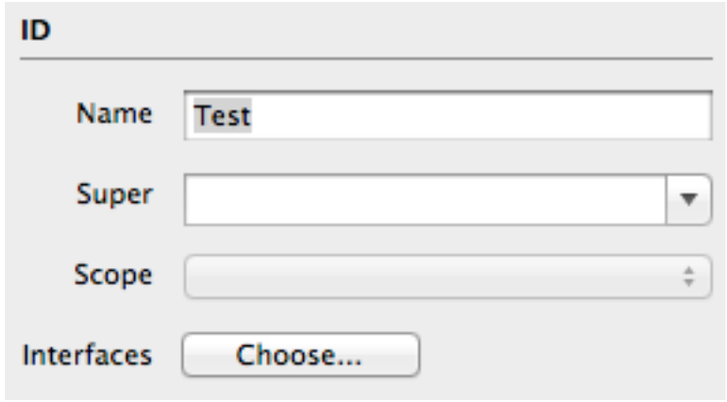
Use the Inspector to change the name of the class interface.

You can only add methods to class interfaces. To add a method to a class interface, use the Add button on the Code Editor toolbar, Insert ↠ Method from the menu, the contextual menu or the keyboard shortcut (Option-Command-M on OS X or Ctrl+Shift +M on Windows and Linux). You cannot specify any code in the class interface, so the Code Editor is disabled. Use the Inspector to specify method names and parameters.

Implementing the Interface Methods in a Class

To implement the methods of the class interfaces, you need to assign the class interface to a class. To do so, select the class in the Navigator and in the Inspector select the Choose button next to the Interfaces label. You can also use the "Implement Interface" option in the contextual menu of the method in the Navigator.

Figure 5.8 Choose an Interface button in the Inspector



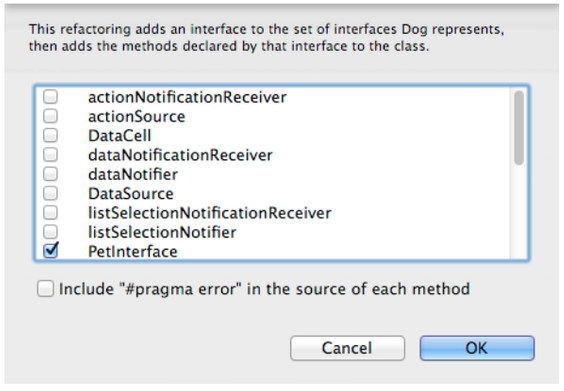
When you click Choose, the Choose Interfaces dialog displays showing the names of all the class interfaces available to you. Some of the interfaces are ones that are built-in and others are ones you created yourself. Select one or more interfaces to assign to the class.

You can also select the "Include #pragma error" in the source of each method" to force a compiler error to be generated with the message "Don't forget to implement this method!". Remove the

pragma after you have added code to implement the interface method.

Press OK to have the Code Editor automatically adds the methods of the interface to your class with a comment telling you the interface to which it belongs (along with the optional pragma).

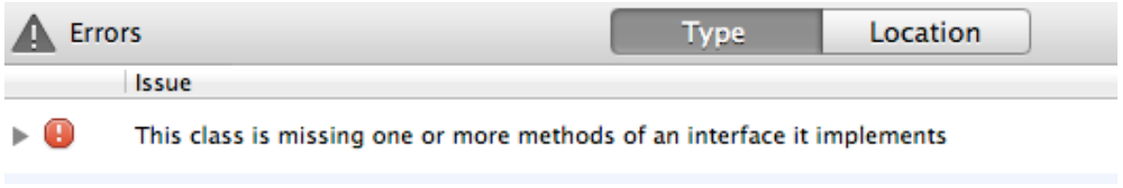
Figure 5.10 Choose Interfaces Dialog



If you modify or change the class interface after you have already applied it to a class, you will need to manually update the class.

If you attempt to compile a project that contains a class with an interface, but the class does not implement all the interface methods, you get a compile error: “This class is missing one or more methods of an interface it implements.”

Figure 5.9 A Compilation Error Due to a Missing Interface Method



Modifying and Deleting Interfaces

If you change your mind and want to delete or replace an interface, you do so the same way you added the interface. When the Interface dialog appears, uncheck the interfaces you no longer want.

***Note**: Any methods that were added to the class while the interface was implemented are not modified or deleted when you remove the interface that they belonged to. That is, if you add an interface via the Implement Interfaces dialog, the methods that are specified by that interface remain as part of the class even if you delete the interface itself. You must take care of any “clean up” activities.*

Specifying the Interface Being Implemented

In rare cases you may find that a class implements more than one method with the same name but from different class interfaces. You may have decided that a method implements two interfaces that both have a method that shares the same name and declaration. Normally, there is no way to tell them apart. However, there is a way to specify the class interface to which each method belongs. You can do so using the optional Interfaces field in the method declaration dialog box, available using the “Show Implements” contextual menu of the method.

To use it, suppose you have two class interfaces, Foo and Bar, that both contain a method named “wahoo” that has no parameters. You can have a class called (Baz) that implements both methods separately.

Add the wahoo method to Baz. Then right-click (control+click on OS X) on the method name and select “Show Implements”. This displays an additional field in the Inspector called “Implements”. Here you can specify the name of the interface and method that this method actually implements: Foo.wahoo.

Now add another method called wahoo and select “Show Implements” for it. Specify “Bar.wahoo”. Now you have told the class exactly which interface methods it implements.

Creating a new Class Interface from an Existing Class

If an existing class in your project contains methods that you want to extract as a class interface, you can generate the new class interface from the Navigator. To do so, select “Extract Interface” from the contextual menu of the method in the Navigator.

The new class interface is added to the project and the current class is made an implementor of the new interface.

Polymorphism

Polymorphism is the ability to have completely different data types (typically classes) behave in a uniform manner. This can be done using class interfaces.

Here is how to implement the Animal, Dog and Cat example using class interfaces.

First, create a new class interface (called Animal). Add to it a Speak method that returns a String.

Now create a new class (Cat) and select Animal as its interface. The Code Editor automatically adds the Speak method for you. Add this code to it:

```
Return "Meow!"
```

Next, add a new class (Dog) and select Animal as its interface. The Code Editor adds the Speak method where you can put this code:

```
Return "Woof!"
```

To test this, create a button on a window. In the Action event of the button add code to create an array of Animals:

```
Dim animals() As Animal
animals.Append(New Cat)
animals.Append(New Dog)

For Each a As Animal In animals
  MsgBox(a.Speak)
Next
```

When you run this code, you will see “Meow!” and then “Woof!”.

Because Cat and Dog both implement Animal, they are allowed to be assigned to a variable (the animals array) with a type of Animal.

And when you call the Speak method of Animal (in the loop), because of polymorphism, your code calls the Speak method of actual type that was added to the array.

Example Subclasses

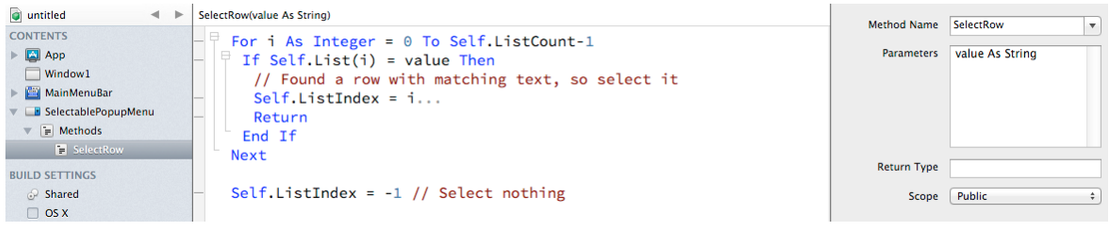
Example SelectablePopupMenu

As an example, you can create a PopupMenu subclass that adds a method that can be used to select an existing item in the PopupMenu.

Create a new class, called SelectablePopupMenu, and set its Super to PopupMenu. On this class, add a new public method and call it “SelectRow” with the parameter “value As String”.

This method will loop through all the rows in the PopupMenu and check to see if the text for a row matches the supplied value. If the text matches, it will select the row and return. If no matches are found, then nothing is selected.

Figure 5.11 SelectRow Method on SelectablePopupMenu class



This is the code for the SelectRow method:

```
For i As Integer = 0 To Self.ListCount-1
  If Self.List(i) = value Then
    // Found a row with matching
    // text, so select it
    Self.ListIndex = i
    Return
  End If
Next

// Select nothing
Self.ListIndex = -1
```

Example: MonthPopup

Here is another example. Suppose you want to create a PopupMenu that, by default, displays the names of the months of the year, with the current month selected. You create a new class, call it MonthPopup, and choose PopupMenu as its super class.

Now you can add an Open event handler to MonthPopup and add the month names, the heading, and code that selects the appropriate month in the list. In this case, the Open event handler is:

```
Self.HasHeading = True
Self.InitialValue = "Months"
Self.AddRow("January")
Self.AddRow("February")
Self.AddRow("March")
Self.AddRow("April")
Self.AddRow("May")
Self.AddRow("June")
Self.AddRow("July")
Self.AddRow("August")
Self.AddRow("September")
Self.AddRow("October")
Self.AddRow("November")
Self.AddRow("December")

// Select the current month
Dim d As New Date
Self.ListIndex = d.Month-1)
```

To add an instance of the MonthPopup class to a window, switch to the window's Window Editor and then drag MonthPopup from the Navigator to the Window Editor. When you run the application, the Open event handler will run and populate the PopupMenu with the months of the year, selecting the current month.

***Note**: Once created, custom control classes can be exported as self-contained objects that can be used in other projects. Just right-click (Control-click on OS X) on the custom control class in the Navigator and choose Export... from the contextual menu.*

Subclasses are Classes

Remember, subclasses are classes. They are called subclasses to differentiate them from the original classes and emphasize the point that they inherit the properties, events, and methods of their parent class. Because subclasses are classes, they can be the super class to other subclasses. For example, suppose you had a NumbersOnlyTextField subclass, but now you need a TextField that allows only numbers within a certain range. You could duplicate the NumbersOnlyTextField subclass and then modify its code. However, this would make your project larger and more difficult to maintain. If you found a bug in the code of the NumbersOnlyTextField, you would have to remember that you used that code in other places as well, track them down, and fix them. A more efficient way is to create a new subclass and choose the NumbersOnlyTextField as its super class. The new subclass (let's call it "NumberRangeTextField") would utilize all of the properties, events, and methods of its super class. However,

you can add code to the TextChange event handler that allows only numbers within a specific range.

Section 7

Advanced Class Features

This section covers more advanced features of classes. You will not need these features often, but they are definitely useful.

Assigning a Value to a Method

Sometimes it is helpful, from a syntax standpoint, to have a method that gets a value using an assignment operator.

Rather than doing this:

```
order.Quantity(2)
```

You would write:

```
order.Quantity = 2
```

which looks very much like a property. You can get the syntax behavior of a property using a method by using the Assigns keyword in the parameter list.

The declaration looks like this:

```
Sub Quantity(Assigns amount As
Integer)
```

You can also have more than one parameter, something that you can not do with a simple property:

```
Sub Quantity(productName As String,
Assigns amount As Integer)
```

This command would be used like this:

```
order.Quantity("Book") = 2
```

When you use this technique, it will look like you are setting a property, even though you are really dealing with a method. This can be handy when designing classes and can make your code easier to read.

Operator Overloading

You can easily overload methods on a class, but how do you overload an operator on a class? For example, what if you have two instances of a SalesOrder class and want to tell if one is greater than the other?

You do this by implementing one of the Operator overload methods (Figure 5.10) on the SalesOrder class, in this case Operator_Compare.

```
Sub Operator_Compare(rightOrder As SalesOrder)
  If Self.TotalAmount > rightOrder.TotalAmount Then
    Return 1
  ElseIf Self.TotalAmount < rightOrder.TotalAmount Then
    Return -1
  Else
    Return 0
  End If
End Sub
```

For more information about operator overloading for custom classes, see the section on each Operator_ keyword in the Language Reference.

Operator Lookup

Dot notation is used to access methods and properties of a class instance. If you try to access a method that does not exist, you get a compilation error.

Figure 5.12 Operator Overload Methods

Operator	Functions
+	Operator_Add Operator_AddRight
-	Operator_Subtract Operator_SubtractRight
*	Operator_Multiply Operator_MultiplyRight
/	Operator_Divide Operator_DivideRight
\	Operator_IntegerDivide Operator_IntegerDivideRight
Mod	Operator_Modulo Operator_ModuloRight
And	Operator_And Operator_AndRight
Or	Operator_Or Operator_OrRight
Not	Operator_Not
=, <, >, <=, >=	Operator_Compare
(lookup)	Operator_Lookup
(negation)	Operator_Negate
(convert)	Operator_Convert
Subscript	Operator_Subscript
Redim	Operator_Redim

Operator Lookup is a special class method that is called when anything following the “.” that is not an actual method or property of the class is used.

One use for this feature is to look up a value that may not be known until run-time. Suppose you are creating a Preferences class that will be used to save your application preferences. Normally, you would have to create a property for each preference you want to save. If you find you have a new preference value, you’ll need to go back to the class and add the property before you can use it.

But you could use operator lookup instead.

With operator lookup, you would use the method to check what was typed after the “.” and have that be the name of the preference. The value that is assigned can then be stored (perhaps using a Dictionary).

Your code to save a value might look like this:

```
Sub Operator_Lookup(name As String, Assigns value As
String)
  If mPreferenceDict = Nil Then
    mPreferenceDict = New Dictionary
  End If
  mPreferenceDict.Value(name) = value
End Sub
```

And the code to get a value might look like this:

```
Function Operator_Lookup(name As String) As String
  If mPreferenceDict = Nil Then Return ""

  If mPreferenceDict.HasKey(name) Then
    Return mPreferenceDict.Value(name)
  End If
End Sub
```

With this in place, you can now write code to save preference values even though you have never defined specific properties or methods for the preference. In the App class, add a property called Prefs as Preferences. In the App.Open event, initialize this property:

```
Prefs = New Preferences
```

Now, anywhere in your project you can write code like this to store a preference value:

```
App.Prefs.UserName = "Bob Roberts"
App.Prefs.UserEmail = "bob@gmail.com"
```

And code like this to get a preference value:

```
Dim userName As String
userName = App.Prefs.UserName
```

Attributes

Attributes are an advanced class feature that allow you to give a class a value that can be checked at runtime through the use of Introspection. This feature is discussed in the Advanced chapter of the Framework Guide.

Casting

An extremely powerful way of creating generic, reusable code is to take advantage of the ability to convert an instance of a class to an instance of its subclass.

The idea is best illustrated by an example. Since the objects you create are subclasses of base classes, you can always test to see whether an object is a member of a specific subclass. The **IsA** operator does this.

With the IsA operator, you test whether an object is of a specific subclass and, if it is, cast it as that type to do something specific with it. You cast an object by using the classname as a function that operates on the instance.

When you cast an instance, compile-time error-checking cannot guarantee that you are casting to a legal object type. The instance that you are casting has to be of the type that you specify. Casting just tells your code to treat the object as a instance of the class to which it is cast. It doesn't convert it from one class to another. If you cast incorrectly, you will get an IllegalCastException at run-time.

Here is an example that uses a For loop to cycle through all the controls in a window. It checks each control to see if it is a Label and if it is, it casts the control and displays its text:

```
Dim labelText As String
// Loop through controls in window
For i As Integer = 0 To Self.ControlCount-1
  If Self.Control(i) IsA Label Then
    // Cast the control to a Label
    // and gets its Text property
    labelText = Label(control(i)).Text
    MsgBox(labelText)
  End If
Next
```

As you can see, the code does not refer to any specific windows, Labels or anything else. Therefore, you can write this routine once and use it in any window in any project.

Static Variables

A Static variable is a global variable that has the scope of a local variable.

It is declared in a method and, unlike with a normal local variable, the value is retained the next time the method in which it is declared is called. This value is retained for the method in all instances of the class, whether they already exist or are created later.

You declare a Static variable by using the Static keyword in place of Dim:

```
Static myValue As Integer
```

Chapter 6

Application Structure

Your applications have an overall structure that determines how they work. This chapter explains the structure of desktop, web and console applications.

CONTENTS

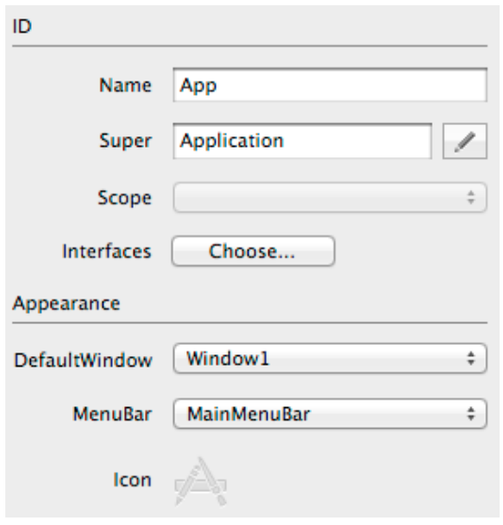
Desktop Applications

Desktop applications have a graphical user interface and are run directly on the user's computer.

The Application Class

The Application class is used to create a subclass that represents your application. When you create a new desktop project, a subclass based on the Application class is added to your project automatically. This is the App class in the Navigator. The Inspector indicates that its Super is Application.

Figure 6.1 Application Properties



On this App subclass is where you specify the default window that opens when your application starts, the default menu bar that is used by the application and the application icons.

If you wish, you can even subclass App. When you do this, your new subclass gets the properties for the default window, menu bar and icon.

Event Handlers

The Application class for a desktop application has these event handlers:

- **Open**: This event handler is called when your application first starts, either by running the application directly (in debug mode) or by running a built application.
- **Close**: The close event is called when the user quits the application.
- **NewDocument**: This event is called when the built application is run without supplying a document.
- **OpenDocument**: This event is called when one of the application's documents is double-clicked from the desktop causing the application to start. It is also called if you drop a document on the application.

- **EnableMenuItems**: This event is called when the user clicks in the menu bar but before any menu items are displayed. The EnableMenuItems event handler executes after the EnableMenuItems event handler of any classes with instances in the frontmost window and after the window's EnableMenuItems event handler. This is the event handler that should be used to enable menu items that should be enabled regardless of whether there is a window open or not (this possibility exists on OS X applications). Note that if a menu item should always be enabled, you should use its AutoEnable property instead of an EnableMenuItems event handler.

- **HandleAppleEvent**: Executes when an AppleEvent is received by the application.

- **Activate**: The application is being activated. This occurs when the application is opening and when it is being brought to the front.

- **Deactivate**: The application is being deactivated. This occurs when another application or a desktop window is being brought to the front or when the application quits.

- **UnhandledException**: Called when a runtime error occurs that is not caught by your code. This event gives you a "last chance" to catch runtime errors before they cause your application to quit.

Properties

- **DefaultWindow**: This is the window that is opened automatically when the application starts.

- **MenuBar**: This is the primary menu bar for the application. It is used as the default menu bar for newly created windows. This property is accessible by the App function (see next topic).

- **Icon**: This opens the Icon Editor and lets you specify the various size icons that are used by the application.

Scope of the App Class Properties, Methods and Constants

You can set the scope of properties, methods and constants of the App subclass using the same rules as for any class (see chapter 5).

The global App function gives you a reference to the App class in your project so that you can access its public properties, methods and constants. If you have subclassed App, the then App function gives you a reference to your subclass instead.

To access a public property somewhere in your project, you would write it like this:

```
value = App.MyProperty
```

Build Settings

The Build Settings section of the Navigator contains the build-specific settings for your application. You can check the box next to each target in order to build an application for that target.

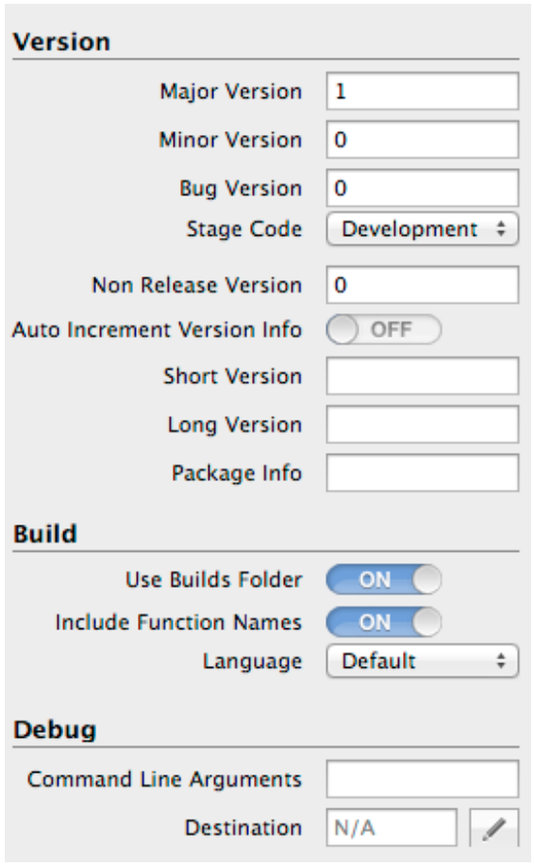
***Note**: Properties in this section that are accessible by the App function are noted by a "*" after their names.*

Shared

The Inspector for Shared settings contains these properties:

- **Major Version***: The Major version for your application. Version numbers are usually written as 1.2.3.4, where "1" is the major version.
- **Minor Version***: The Minor version for your application. Version numbers are usually written as 1.2.3.4, where "2" is the minor version.
- **Bug Version***: The Bug version for your application. Version numbers are usually written as 1.2.3.4, where "3" is the bug version.
- **Stage Code***: Used to indicate the type of application you are building (Development, Alpha, Beta, Final).

Figure 6.2 Shared Build Settings



- **Non Release Version***: The Non Release (build) version for your application. Version numbers are usually written as 1.2.3.4, where "4" is the non release version.

•**Auto Increment Version Info**: When ON, the Non Release Version is increased by one each time you do a Build (but not when you Run).

•**Short Version***: A short text description for your application. Usually this contains just the version number (such as 1.2.3.4) and is displayed by some operating systems in Get Info or Property windows for the application.

•**Long Version***: A longer text description for your application. Usually this contains the application name, copyright, version and other information. This is displayed by some operating systems in Get Info or Property windows for the application.

•**Package Info***: A text description for your application that may be displayed by some operating systems in Get Info or Property windows for the application.

- **Use Builds Folder**: When ON, a separate folder is placed alongside the project file. Each platform that is built (OS X,

Windows, Linux) also gets its own subfolder within this build folder.

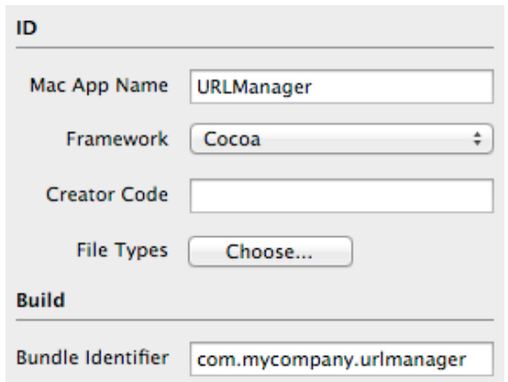
- **Include Function Names**: When ON, the actual names of your function calls are embedded in the built application. This is useful for debugging purposes and for getting stack traces.

- **Language**: Allows you to specify the language to use to resolve dynamic constants.

- **Command Line Arguments**: These are the command-line arguments that are passed to your application when you run it in Debug mode.

- **Destination**: Specifies the path where the application is located when you run it in Debug mode. If not specified, then the application is placed alongside your project file.

Figure 6.3 OS X Build Settings



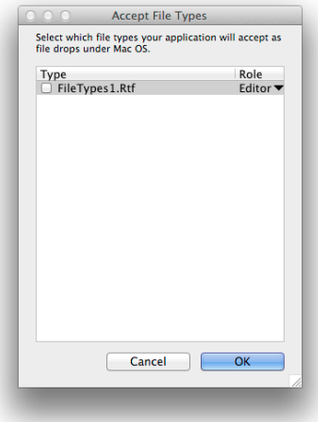
OS X

The OS X section specifies settings used when building the OS X application.

- **Mac App Name**: The actual file name for your OS X application. If not specified, the ".app" suffix is automatically added when the app is built.

- **File Types**: Opens the Accept File Types dialog used to specify the acceptable file types that the app supports. For each file type, you can specify its role as None, Viewer, Editor or Shell.

Figure 6.4 Accept File Types Window



  - None: Your app cannot handle the file type.

  - Viewer: Your app can open and read the file type, but cannot save it.

  - Editor: Your app can open, read, edit and write using the file type. This is what you should normally choose.

  - Shell: Not documented by Apple.

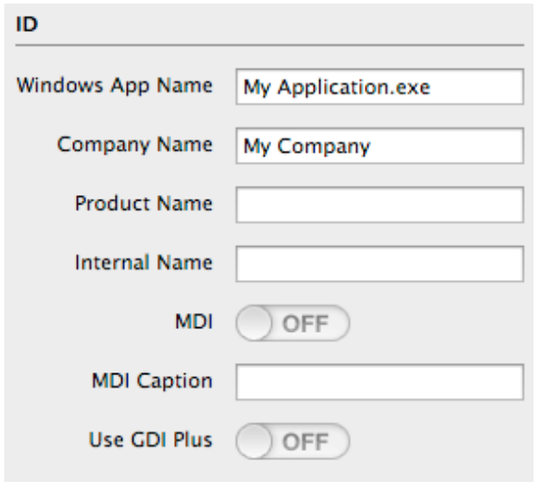
- **Bundle Identifier**: The bundle identifier is used by OS X as a unique descriptor for your application. It is usually specified as a reverse domain name, such as com.xojo.myapplication. This is added to the CFBundleIdentifier section of the application's plist.
  A bundle identifier is required for Cocoa applications.

Windows

The Windows section specifies settings used when building the Windows application.

- **Windows App Name**: The actual file name for the Windows application. If not specified, the ".exe" suffix is automatically added when the app is built.
- **Company Name**: On Windows 7, the "Company Name" appears in the Copyright section of the application properties in the Details tab.
- **Product Name**: The name of the product as installed in the Windows Start ↠ All Programs menu. This also appears in "Product name" on the application properties Details tab.
- **Internal Name**: This is useful when your product has a different internal name than its external name. This is not shown on the application properties Details tab.
- **MDI**: When ON, the application's windows will be enclosed in the "parent" MDI window. MDI stands for Multiple Document Interface. Although still supported by Windows, it is not commonly used.
- **MDI Caption**: If MDI is ON, you can enter the caption that appears in the title bar of the MDI window.
- **Use GDI Plus***: When ON, enables the newer GDI+ graphics subsystem that has support for transparency.

Figure 6.5 Windows Build Settings



Linux

The Linux section specifies settings used when building the Linux application.

- **Linux App Name**: The actual file name for the Linux application.

This Computer

The This Computer section shows you the appropriate section (OS X, Windows or Linux) for the platform you are currently using. For example, if you are running Xojo on OS X, then This Computer shows the OS X settings.

Deployment

Desktop applications are deployed by installing them on a computer and running them on the computer.

OS X

OS X applications are called "bundles". A bundle is a special folder that acts as an application. Because you can actually place

support files in the bundle, OS X applications are often deployed as a single application bundle that the user drags to their Applications folder.

You can also create installers on OS X, although they are not common. Apple includes the PackageMaker tool for creating installers with their development tools.

OS X applications you create can also be sold and distributed through the Mac App Store.

***Note**: When you compile an OS X application on Windows, the result application is created as a tar archive file. You can transfer this file to OS X and expand the archive in order to run it.*

Windows

Windows applications have a minimum of two parts: the main executable and its libraries (contained in a corresponding Libs folder). Both parts are required in order for the application to run. It is also possible that there is a Resources folder that may contain localization files (or other files that you chose to copy there using Build Automation).

***Note**: The Resources folder can be renamed to "AppName Resources", where AppName is the name of your app. This can be useful should you want to package several apps into a single folder.*

Generally you should use an installer to deploy Windows applications. With an installer you can create shortcuts in the Start menu or Desktop and copy the files to the appropriate locations. There are many products available for creating installers for Windows applications. Two that work well with Xojo applications are InnoSetup and Advanced Installer.

For simple needs, you can also just zip the Windows application and Lib folder for easy distribution.

Linux

Linux applications also have two parts: the main executable and its libraries (contained in a corresponding Libs folder). Both parts are required in order for the application to run.

How you deploy a Linux application varies depending on the Linux distributions you support. Different distributions have different formats for packaging.

For simple needs, you can also just zip the Linux application and Lib folder for easy distribution.

Section 2

Web Applications

Web applications have a graphical user interface that runs inside a web browser and a companion server application that runs on (or as) a web server.

The WebApplication Class

The WebApplication class is used to create a subclass that represents your web application. When you create a new web project, a subclass based on the WebApplication class is added to your project automatically. This is the App class in the Navigator. The Inspector indicates that its Super is WebApplication.

On this App subclass is where you specify the properties for the application such as the default web page that opens when your application starts, the default launch message that is used by the application, default disconnect message, HTML header and icon.

If you wish, you can even subclass App. When you do this, your new subclass gets the properties for the application.

Event Handlers

The WebApplication class for a web application has event handlers. They are:

- **Close**: The close event is called when the application quits.

- **HandleSpecialURL**:
  This event is called when the web application is accessed using the special URL, /special/value. In the Request parameter you can get information about the URL and return information back. This can be used to implement web services in your web applications.
  The URL “/api/value” is also supported and will result in this event handler being called.

- **Open**: This event handler is called when your application first starts.

- **UnhandledException**: Called when a runtime error occurs that is not caught by your code. This event gives you a “last chance” to catch runtime errors before they cause your application to quit.

Properties

- **DefaultWebPage**: This is the web page that is displayed automatically when the application starts.
- **LaunchMessage**: On the application loading screen, a progress bar, the application icon and the launch message all appear while the application is being loaded.
- **DisconnectMessage**: This message appears when the client has lost its connection to the server.
- **HTMLHeader**: Can be used to add static HTML code to each page.
- **Icon**: This opens the Icon Editor and lets you specify the various size icons that are used by the application. These icons are used by the loading screen and as a the browser favicon.

Scope of the App Class Properties, Methods and Constants

You can set the scope of properties, methods and constants of the App subclass using the same rules as for any class (see chapter 5).

You need a reference to your App class in order to access public properties, methods and constants. The global App function gives you a reference to the App class in your project. If you have subclassed App, the then App function gives you a reference to your subclass instead.

So to access a public property somewhere in your project, you would write it like this:

```
value = App.MyProperty
```

Web Application Design Considerations

Client/Server

All web applications uses a client/server design. This means that a web application has two parts: the client user interface and the application that runs on the server.

The client is the part that the user interacts with. For a web application, this is the user interface that runs in a web browser. Web applications support recent versions of modern web browsers such as Safari, Chrome, Firefox and Internet Explorer.

The application is where your code runs and it is typically run on a server. The client and the web application components can reside on the same computer, but they are usually different computers: the user's computer for the web browser/user interface and a server that is running the application.

Latency

When you are testing your web applications locally, both the client (web browser) and server (application) are running on your computer. This means that communication between them is quick. But when you deploy an application to a web server, there can be a longer delay as the web browser has to communicate across the Internet to the application on the server. This delay is called latency and has a lot of factors, including the speed of the user's Internet connection, general Internet congestion and the responsiveness of the server.

It is important when designing your application to keep latency in mind.

Multiple Users and Sessions

A web application typically has multiple users connected to it at one time. This is handled for you using a concept called Sessions. Session is a subclass of WebSession and is automatically added to all web projects. Each user that connects to your web application gets its own Session subclass that you can access using the Session global function.

Cookies

Cookies are a browser feature that allows you to save Session-specific data. Web applications have built-in support for Cookies as methods of the WebSession class.

Session

Events

AllowUnsupportedBrowser
Called when an unsupported browser connects to a web application.

Close
Called when the session is closing.

HashTagChanged
Called when a hashtag has changed.

JavaScriptError
Called when a JavaScript error occurs.

Open
Called when a session starts.

PrepareSession
Called when the browser first connects to the web application.

TimedOut
Called when the browser has been idle for the time specified in the TimeOut property.

UnhandledException
Used to catch otherwise unhandled exceptions.

Properties

ActiveConnectionCount
Returns the number of active connections, such as file uploads, downloads or requests.

Browser
Identifies the browser being used for this session. Types include:

- Safari
- Chrome
- Firefox
- Internet Explorer
- Opera
- ChromeOS
- SafariMobile
- Android
- Blackberry
- Unknown

ClientTime
Returns a date containing the time for the user. This can be different than the server time since the user can connect to the web application from anywhere.

ConfirmMessage
When closing a web application, the browser will display a warning for the user with this message.

CurrentPage
Allows you to get or set the current web page.

GMTOffset
Returns the GMTOffset for the time zone the user is in.

HashTag
Allows you to get or set a hashtag.

HeaderCount
Returns the number of HTTP headers sent when the session was created.

Identifier
Returns a unique session identifier string.

PageCount
Returns the number of open pages in the session.

Platform
Identifies the platform used by the browser. Types are:

- Macintosh
- Windows
- Linux
- Wii
- Playstation
- iPhone
- iPod touch
- iPad
- Blackberry
- WebOS
- Unknown

RemoteAddress
The client IP address.

RenderingEngine
Identifies the rendering engine used by the browser. Types are:

- WebKit
- Gecko
- Trident
- Presto
- Unknown

Secure
Gets the SSL status of the web page.

StatusMessage
This message is displayed in the status bar for supported browsers.

Timeout
The number of seconds that the browser can be idle before the TimedOut event handler is called. Set to 0 to disable the timeout.

Title
Gets or sets the browser title. This overrides the web page title.

URL
Returns the URL to the session.

Methods

Header, HeaderName
Used to access HTTP header information.

PageAtIndex, PageWithID, PageWithName
Used to look up web pages in the session.

Quit
Ends the session.

URLParameter, URLParameterCount, URLParameterExists, URLParameterName
Used to look up URL parameters (query strings).

Shared Methods

Available
Used to check if a session is available. A session may not be available for code that runs in WebApplication, for example.

Cookies

Cookies are a standard way for web browsers to retain information pertaining to a session. For example, if your web application has users log in using a UserID, you could save the UserID as a cookie. The next time they open the web app, you can look for the cookie and if it is available, pre-fill the UserID name for them.

This is how you set a cookie value:

```
Session.Cookies.Set("UserName", "BobRoberts")
```

And this is how you retrieve a cookie value:

```
Dim userName As String
userName = Session.Cookies.Value("UserName")
```

Hash Tags

Hash tags are a way for you to identify different areas of the web application. Web applications always show the same browser URL, even as you navigate to different web pages. You can use the Hash Tag as a way to identify different pages (or data) so that the user can return directly to it later.

Hash tags are written using the “#” character like this:

http://www.mywebsite.com/#details

The Session.HashTagChanged event is called when the hash tag is changed by the user. In this event, you can get the new value and choose to navigate to a different page or display different data.

Build Settings

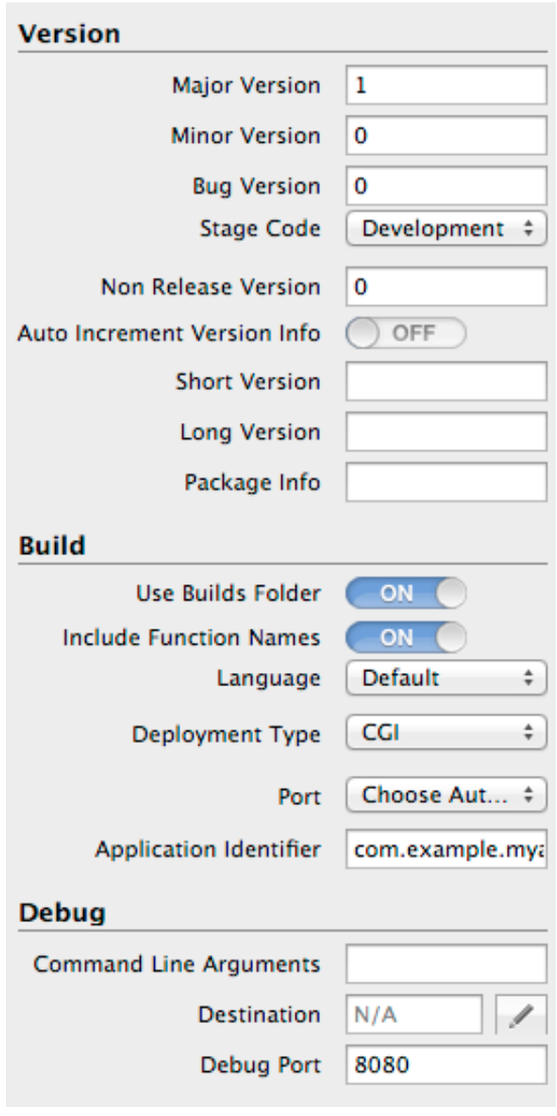
The Build Settings section of the Navigator contains the build-specific settings for your application. You can check the box next to each target in order to build an application for that target.

Shared

The Inspector for Shared settings contains these properties:

- **Major Version**: The Major version for your application. Version numbers are usually written as 1.2.3.4, where “1” is the major version.
- **Minor Version**: The Minor version for your application. Version numbers are usually written as 1.2.3.4, where “2” is the minor version.
- **Bug Version**: The Bug version for your application. Version numbers are usually written as 1.2.3.4, where “3” is the bug version.
- **Stage Code**: Used to indicate the type of application you are building (Development, Alpha, Beta, Final).
- **Non Release Version**: The Non Release (build) version for your application. Version numbers are usually written as 1.2.3.4, where “4” is the non release version.
- **Auto Increment Version Info**: When ON, the Non Release Version increases by one each time you do a Build.
- **Short Version**: A short text description for your application. Usually this contains just the version number (such as 1.2.3.4) and is displayed by some operating systems in Get Info or Property windows for the application.
- **Long Version**: A longer text description for your application. Usually this contains the application name, copyright, version and other information. This is displayed by some

Figure 6.6 Shared Build Settings



operating systems in Get Info or Property windows for the application.

- **Package Info**: A text description for your application that may be displayed by some operating systems in Get Info or Property windows for the application.
- **Use Builds Folder**: When ON, builds are placed in a separate folder alongside the project file. Each platform (OS X, Windows, Linux) also gets its own subfolder within this builds folder.
- **Include Function Names**: When ON, the actual names of your function calls are included in the built application. This is useful for debugging purposes and for getting stack traces.
- **Language**: Allows you to specify the language to use to resolve dynamic constants.
- **Deployment Type**: Web applications can be deployed as either CGI or stand-alone applications.
- **Port**: Select a specific port to use to connect to the web application or let the web application choose the port automatically.
- **Application Identifier**: Required for CGI applications, this is an unique identifier for the web application.
- **Command Line Arguments**: These are the command-line arguments that are passed to your application when running it in Debug mode.
- **Destination**: Specifies the path where the application is located when you run it in Debug mode. If not specified, then the application is placed alongside your project file.
- **Debug Port**: This is the port that is used when running your web applications in Debug mode.
  Note: If you ever have difficulty debugging your web applications, make sure the debug port is not in use by another application.

Xojo Cloud

The Xojo Cloud section is used to deploy your web app to your Xojo Cloud server. You can purchase a Xojo Cloud server from the Xojo Store.

Application Name: The name of your web application. Your web app will be deployed into a folder with this name.

Server: The Xojo Cloud server to which to deploy. You will need to have purchased a Xojo Cloud server and be logged in to your account in Xojo in order to see your servers.

OS X

The OS X section allows you to specify settings for the OS X applications.

- **Mac App Name**: The actual file name for your OS X application.

Windows

The Windows section contains build settings for your Windows applications.

- **Windows App Name**: The actual file name for the Windows application.
- **Company Name**: On Windows 7, the “Company Name” appears in the Copyright section of the application properties in the Details tab.
- **Product Name**: The name of the product as installed in the Windows Start ↠ All Programs menu. This also appears in “Product name” on the application properties Details tab.

Linux

The Linux section contains build settings for your Linux applications.

- **Linux App Name**: The actual file name for the Linux application.

Deployment

There are two ways to deploy your web applications: Standalone and CGI.

***Note**: You do not have to worry about the information in this section if you deploy using Xojo Cloud.*

Standalone

A Standalone web application is an application that you manually run on your server. You have to start the application (usually from the command line) and leave it running in order for people to access the web app. In addition, a standalone web application is accessed through a port, which you specify when building the app. Essentially, a standalone web application consists of both the web server and your web application.

A deployed standalone web app would be accessed with a URL such as this:

http://www.mywebsite.com:8080

You will usually want to run a standalone web application in the background, which you can do on OS X or Linux by running the standalone application as a daemon. On Windows you can run your standalone applications as a service to run them in the background.

CGI

A web application built to use CGI (Common Gateway Interface) uses an existing web server (usually Apache) as its web server. The web server then communicates to your web application using CGI. To facilitate this, a companion Perl script (supplied when you build your application) handles communication between the web server and your web application.

Some web browsers (notably Safari) continue to display a loading indicator even after the web page has finished loading. This is a result of the method used by the web server to communicate with your web application.

Because a CGI deployment uses your existing web server software, you do not have to specify a port when accessing your web application. A typical URL looks like this:

http://www.mywebsite.com/cgi-bin/mywebapp.cgi

Platforms

Your web application can be compiled for any of the supported desktop platforms. When it comes to web servers, Linux is the most commonly used operating system platform, followed by Windows and then OS X.

LINUX

In complete opposition of the situation on the desktop, the majority of web servers use some form of Linux.

To deploy your web application it needs to be hosted on a server. The two most common types of hosting are shared and VPS (Virtual Private Server). Shared hosting usually costs less, but is also often rather restricted. Most shared hosting providers work best with static web sites or pre-configured tools (such as WordPress) and do not allow general purpose applications to run on them. They are rarely a good choice for a Xojo web application.

Your best bet is to use a VPS (Virtual Private Server) to host your web applications. A VPS gives you your own server, usually running the Apache web server, with its own specs running inside of a Virtual Machine (VM). With a VPS you have complete control over the server and configure it to run Xojo web applications.

Refer to the documentation wiki for the latest information on configuring Apache to run web applications.

WINDOWS

Windows web servers primarily use IIS (Microsoft Internet Information Services). Windows servers are far less common than Linux servers. There are also fewer hosting companies offering Windows servers and they usually cost more. IIS can also be more difficult to configure than Apache.

OS X

OS X web servers typically use Apache and setup is mostly the same as it is for Linux. There are even fewer OS X servers than

Windows, but OS X servers can be simpler to configure. There are no known hosting companies that offer OS X hosting, but there are several that offer colocation services for your own Mac hardware.

Refer to the documentation wiki for the latest details on configuring Apache to run web applications.

Section 3

Console Applications

Unlike desktop and web applications, console applications are not event-based. Console applications have no graphical user interface and are designed to run from the terminal, command line or as a background process.

The ConsoleApplication Class

The ConsoleApplication class is used to create a subclass that represents your application. When you create a new console project, a subclass based on the ConsoleApplication class is added to your project automatically. This is the App class in the Navigator. The Inspector indicates that its Super is ConsoleApplication.

If you wish, you can subclass App.

Event Handlers

The Application class for a console application has event handlers. They are:

- **Run**: This event handler is called when your application first starts, either by running the application from Xojo or by running a built application. Your application quits when you exit the Run event.
- **UnhandledException**: Executes when a runtime error occurs that is not caught by your code. This event gives you a "last chance" to catch runtime errors before they cause your application to quit.

Scope of the App Class Properties, Methods and Constants

You can set the scope of properties, methods and constants of the App subclass using the same rules as for any class (see chapter 5).

You need a reference to your App class in order to access public properties, methods and constants. The global App function gives you a reference to the App class in your project. If you have subclassed App, the then App function gives you a reference to your subclass instead.

So to access a public property somewhere in your project, you would write it like this:

```
App.MyProperty
```

Background Applications

Console applications are often used to create background applications. On OS X and Linux, background applications are called daemons. On Windows, background applications are called services.

Daemon

To turn your console application into a daemon that can run in the background, simply call the Daemonize method of the ConsoleApplication class.

You can also use the Daemonize method on OS X, but Apple would rather you use launchd to start daemon processes.

ServiceApplication

To create a console application that can run as a service you need to use the ServiceApplication class. When you create your console application, change the Super of the App from ConsoleApplication to ServiceApplication. Alternatively you can choose ServiceApplication from the Templates in the Project Chooser.

A service application gives you additional events that help you manage things when the services starts, stops or is paused. Refer to the Language Reference for more information about these events.

Build Settings

The Build Settings section of the Navigator contains the build-specific settings for your application. You can check the box next to each target in order to build an application for that target.

Shared

The Inspector for Shared settings contains these properties:

- **Major Version**: The Major version for your application. Version numbers are usually written as 1.2.3.4, where “1” is the major version.
- **Minor Version**: The Minor version for your application. Version numbers are usually written as 1.2.3.4, where “2” is the minor version.
- **Bug Version**: The Bug version for your application. Version numbers are usually written as 1.2.3.4, where “3” is the bug version.
- **Stage Code**: Used to indicate the type of application you are building (Development, Alpha, Beta, Final).
- **Non Release Version**: The Non Release (build) version for your application. Version numbers are usually written as 1.2.3.4, where “4” is the non release version.

- **Auto Increment Version Info**: When ON, the Non Release Version increases by one each time you do a Build.
- **Short Version**: A short text description for your application. Usually this contains just the version number (such as 1.2.3.4) and is displayed by some operating systems in Get Info or Property windows for the application.
- **Long Version**: A longer text description for your application. Usually this contains the application name, copyright, version and other information. This is displayed by some operating systems in Get Info or Property windows for the application.
- **Package Info**: A text description for your application that may be displayed by some operating systems in Get Info or Property windows for the application.
- **Use Builds Folder**: When ON, a separate folder is created alongside the project file. Each platform (OS X, Windows, Linux) also gets its own subfolder within this builds folder.
- **Include Function Names**: When ON, the actual names of your function calls are included in the built application. This is useful for debugging purposes and for getting stack traces.
- **Command Line Arguments**: These are the command-line arguments that are passed to your application when running it from Xojo.
- **Destination**: Specifies the path where the running application is placed. Normally it is placed alongside your project, but you can choose a different location.

OS X

The OS X section allows you to specify settings for the OS X applications.

- **Mac App Name**: The actual file name for your OS X application.

Windows

The Windows section contains build settings for your Windows applications.

- **Windows App Name**: The actual file name for the Windows application.
- **Company Name**: On Windows 7, the “Company Name” appears in the Copyright section of the application properties in the Details tab.
- **Product Name**: The name of the product as installed in the Windows Start → All Programs menu. This also appears in “Product name” on the application properties Details tab.
- **Internal Name**: This is useful when your product has a different internal name than its external name. This is not shown on the application properties Details tab.

Linux

The Linux section contains build settings for your Linux applications.

- **Linux App Name**: The actual file name for the Linux application.

This Computer

The This Computer section shows you the appropriate section (OS X, Windows or Linux) for the platform you are currently using. For example, if you are running Xojo on OS X, then This Computer shows the OS X settings.

Deployment

Console applications are deployed by installing them on a computer and running them on the computer.

Console applications have two parts: the main executable and its libraries (contained in the Libs folder). Both parts are required in order for the application to run.

You can just zip the console application and its Lib folder for easy distribution.

Section 4

iOS Applications

iOS Applications runs on iOS devices such as iPhones and iPads.

The IOSApplication Class

The IOSApplication class is used to create a subclass that represents your iOS application. When you create a new iOS project, a subclass based on the IOSApplication class is added to your project automatically. This is the App class in the Navigator. The Inspector indicates that its Super is IOSApplication.

If you wish, you can subclass App.

Event Handlers

The Application class for an iOS application has event handlers. They are:

- **LowMemoryWarning**: Called when iOS tells your app that memory is running low. Your app should attempt to free memory be clearing any caches or data structures that may be using significant memory.
- **Open**: Called when your application first starts.
- **UnhandledException**: Called when a runtime error occurs that is not caught by your code. This event gives you a "last chance" to catch runtime errors before they cause your application to quit.

Scope of the App Class Properties, Methods and Constants

You can set the scope of properties, methods and constants of the App subclass using the same rules as for any class (see chapter 5).

You need a reference to your App class in order to access public properties, methods and constants. The global App function gives you a reference to the App class in your project. If you have subclassed App, the then App function gives you a reference to your subclass instead.

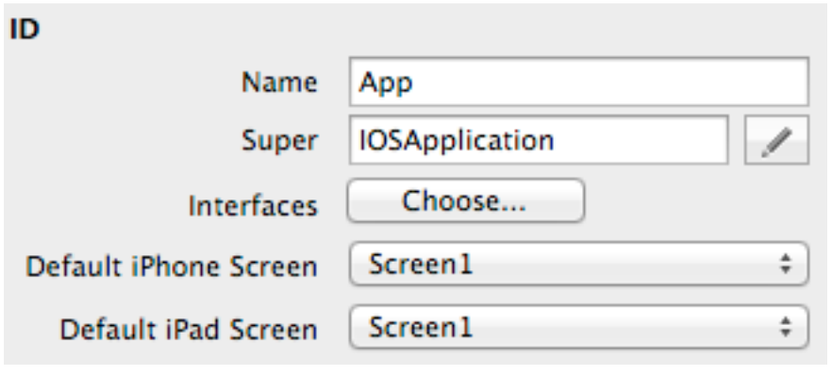
So to access a public property somewhere in your project, you would write it like this:

```
App.MyProperty
```

Screens

In the Inspector for the Application, you can specify the default screens to use when the app is run on an iPhone or an iPad. A

Figure 6.7 Application Properties in Inspector

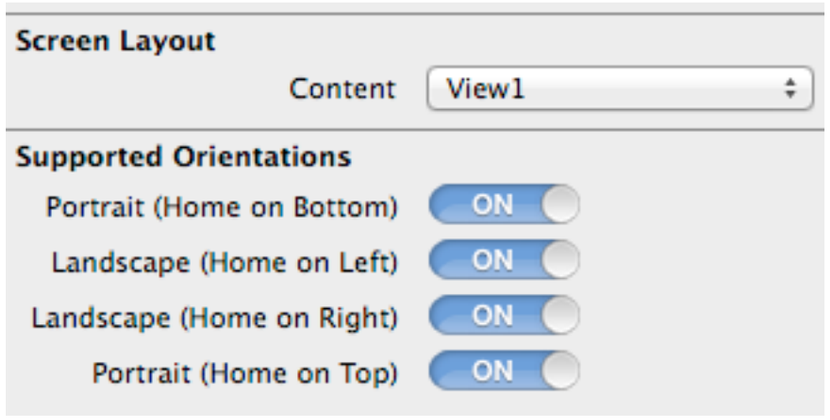


Screen is a project item that specifies the type of layout and supported orientations. Screen1 is added to your project by default. You can set either (but not both) of the default screens to None if you do not want a screen for that device. If you set Default iPhone Screen to None then your app will not run on an iPhone. If you set Default iPad Screen to None, then the app will run the iPhone screen on the iPad.

Viewing Screens

When you click on the Screen you are shown a preview of how it might look on an iPhone in portrait mode. You can change the preview by using the toolbar buttons for orientation and device type:

Figure 6.8 Screen Layout and Supported Orientations Properties



- Orientation: Clicking the button toggles between portrait and landscape.
- Device Type: Clicking Device Type displays a menu where you can choose to “View as iPhone” or “View as iPad”.

In the Inspector, you can specify the Screen Layout and the Supported Orientations.

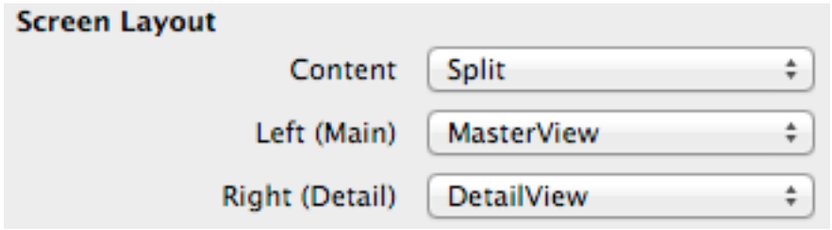
Screen Properties

The screen layout defaults to View1, but you can instead choose any View in the project. You can also choose Split or Tabs:

SPLIT

A Split is used on iPads in landscape mode to display a master area on the left and a detail area on the right, similar to how the Mail app works.

Figure 6.9 Screen Layout Properties for Split

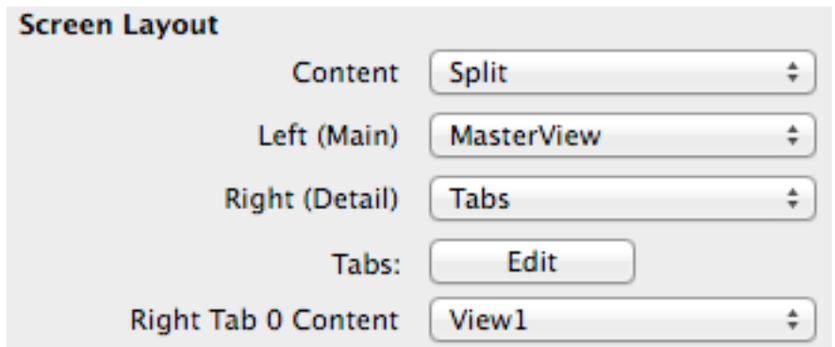


When you choose Split, two additional properties (Left - Master and Right - Detail) are shown in the Inspector. Use these properties to specify what to display in each section (Views or Tabs). If you choose Tabs, additional properties appear (see below).

TABS

Tabs are buttons on the bottom of the screen that can be used to display other Views or SplitViews. When you select Tabs, the Tabs and Tab 0 Content properties are displayed in the Inspector. The Tabs property has an Edit button that you can click to display the Tab Editor. Here you can manage the tabs to display, including their names, order and icons. Click on each tab on the Screen to set the view (or SplitView) that it should display.

Figure 6.10 Screen Layout Properties for Tabs



Supported Orientations

You can also specify the valid orientations for your screen. These are simply ON/OFF switches:

- Portrait (Home on Bottom)
- Landscape (Home on Left)
- Landscape (Home on Right)
- Portrait (Home on Top)

Not all settings will make sense for your app, so turn off the ones you do not need to support.

For example, if you turn off everything but “Portrait (Home on Bottom)”, your app will not rotate the screen when the device is rotated.

Build Settings

The Build Settings section of the Navigator contains the build-specific settings for your application. You can check the box next to each target in order to build an application for that target.

Shared

The Inspector for Shared settings contains these properties:

- **Major Version**: The Major version for your application. Version numbers are usually written as 1.2.3.4, where “1” is the major version.

- **Minor Version**: The Minor version for your application. Version numbers are usually written as 1.2.3.4, where “2” is the minor version.
- **Bug Version**: The Bug version for your application. Version numbers are usually written as 1.2.3.4, where “3” is the bug version.
- **Stage Code**: Used to indicate the type of application you are building (Development, Alpha, Beta, Final).
- **Non Release Version**: The Non Release (build) version for your application. Version numbers are usually written as 1.2.3.4, where “4” is the non release version.
- **Auto Increment Version Info**: When ON, the Non Release Version increases by one each time you do a Build.
- **Short Version**: A short text description for your application. Usually this contains just the version number (such as 1.2.3.4) and is displayed by some operating systems in Get Info or Property windows for the application. This is required when building for the App Store.
- **Long Version**: A longer text description for your application. Usually this contains the application name, copyright, version and other information. This is displayed by some operating systems in Get Info or Property windows for the application.
- **Package Info**: A text description for your application that may be displayed by some operating systems in Get Info or Property windows for the application.
- **Use Builds Folder**: When ON, a separate folder is created alongside the project file. Each platform (OS X, Windows, Linux) also gets its own subfolder within this builds folder.
- **Include Function Names**: When ON, the actual names of your function calls are included in the built application. This is useful for debugging purposes and for getting stack traces.
- **Command Line Arguments**: These are the command-line arguments that are passed to your application when running it from Xojo.
- **Destination**: Specifies the path where the running application is placed. Normally it is placed alongside your project, but you can choose a different location.

iOS

The iOS section allows you to specify settings for the iOS app.

- **iOS App Name**: The actual name for your iOS application. This is the name that appears below the icon in Springboard.
- **Bundle Identifier**: The bundle identifier is used by iOS as a unique descriptor for your application. It is usually specified as

a reverse domain name, such as com.xojo.myapplication. This must use your App ID for distributing in the App Store.

•

CODE SIGNING

- **Team**: Here you choose the Team that has Provisioning Profiles for deploying your app.
- **Entitlements**: Specifies a plist file containing entitlements for your app.
- **Build for App Store**: Switch this to ON to create an IPA (iOS App Archive) file to submit to the App Store using iTunes Connect.

IOS DEBUGGING

- **Simulator Device**: Select the type of Simulator device to start when debugging.

Deployment

Console applications are deployed by installing them on a computer and running them on the computer.

Console applications have two parts: the main executable and its libraries (contained in the Libs folder). Both parts are required in order for the application to run.

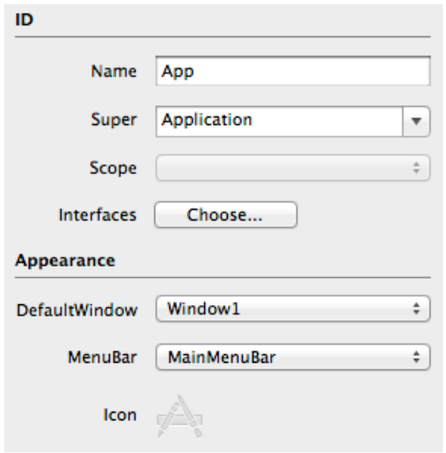
You can just zip the console application and its Lib folder for easy distribution.

Icon Editor

Icon Editor

The icon editor is used to add application icons. Web applications use the application icon as part of their loading screen.

Figure 6.11 The Icon Setting in the Inspector

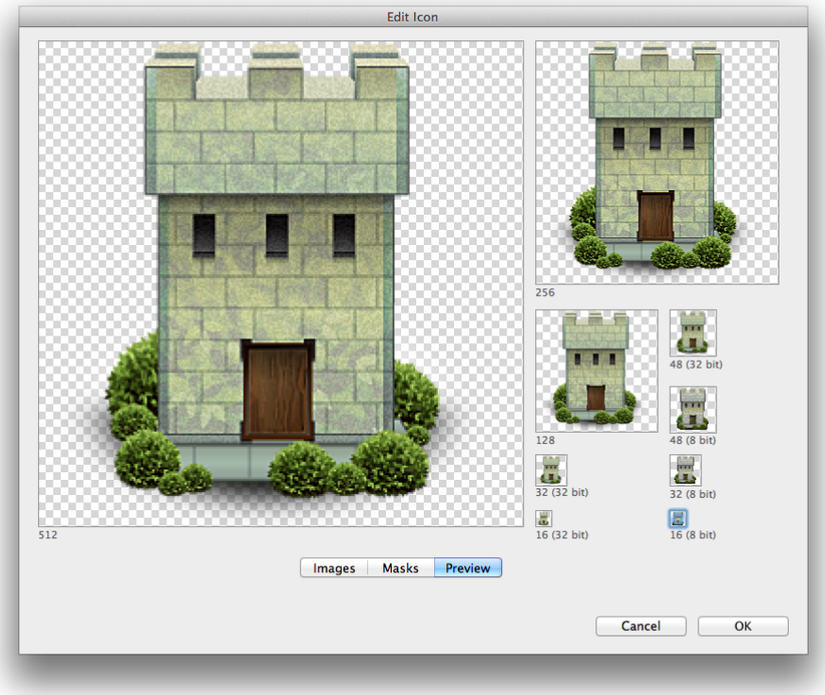


To open the Icon Editor, select the App object in the Navigator and then select the icon in the Inspector.

When you open the Icon Editor, a window displays with blank areas for icons of various sizes. You can drag icons files to these areas to set the icon for that size. You can also copy and paste icons between the sizes; they are scaled automatically for you.

Use the Images/Masks/Preview control to display the various components of the icon images. If you drag in an icon that has both the image and the mask (such as with most PNG files) then both the image and mask are added.

Figure 6.12 Icon Editor



For best results, be sure to add icons (and masks) for every available size. Different platforms use different sizes. If the masks are missing, the icons may not appear on some platforms.

You can also drag an ICNS file onto the window background (instead of an image area) to load and populate all the image sizes at one time.