

Get started with programming
the fast-growing language

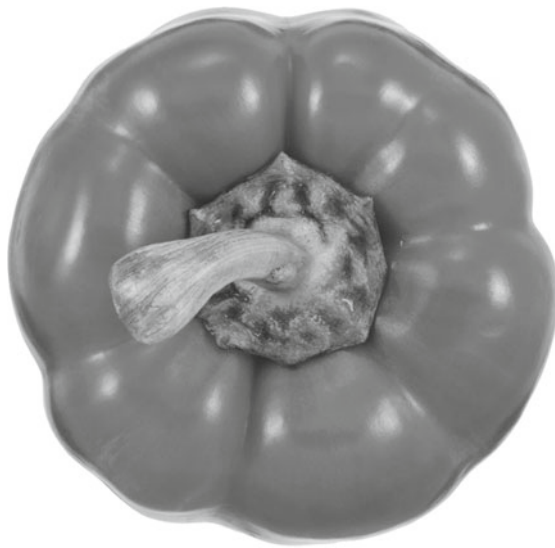


Beginning Objective-C

James Dovey | Ash Furrow

Apress®

Beginning Objective-C



James Dovey
Ash Furrow

Apress®

Beginning Objective-C

Copyright © 2012 by James Dovey and Ash Furrow

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN 978-1-4302-4368-7

ISBN 978-1-4302-4369-4 (eBook)

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

President and Publisher: Paul Manning

Lead Editor: Steve Anglin

Developmental Editor: Douglas Pundick

Technical Reviewer: Felipe Laso

Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan

Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie,

Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing,

Matt Wade, Tom Welsh

Coordinating Editor: Katie Sullivan

Copy Editor: Mary Behr

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

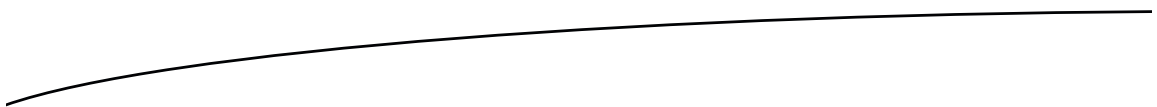
Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use.

eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales-eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/



For the bookends of the process: Clay Andres, who started the ball rolling three years (!) ago, and J'aime Ohm, on whose birthday this is released.

—James Dovey



Contents at a Glance

About the Authors.....	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
■ Chapter 1: Getting Started with Objective-C.....	1
■ Chapter 2: Object-Oriented Programming	23
■ Chapter 3: Foundational APIs	43
■ Chapter 4: Objective-C Language Features	75
■ Chapter 5: Using the Filesystem.....	107
■ Chapter 6: Networking: Connections, Data, and the Cloud	159
■ Chapter 7: User Interfaces: The Application Kit	189
■ Chapter 8: Data Management with Core Data	225
■ Chapter 9: Writing an Application	269
■ Chapter 10: Après Code: Distributing Your Application	353
Index.....	371



Contents

About the Authors.....	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
■ Chapter 1: Getting Started with Objective-C.....	1
Xcode	2
Creating Your First Project.....	4
The Application Template.....	5
Hello Interface Builder	7
User Interface Controls	7
Interface Bindings	11
Running the App	15
Language Fundamentals	16
Types and Variables	17
Pointers	18
Functions and Declarations	18
Scope.....	19
Conditions.....	20

Loops	21
Objective-C Additions	22
Summary	22
■ Chapter 2: Object-Oriented Programming	23
Objects: Classes and Instances	23
Encapsulation	24
Inheritance	24
Objects in Objective-C	26
Message-Passing and Dynamism	26
Writing Objective-C	28
Allocation and Initialization	28
Sending Messages	30
Memory Management	31
Class Interfaces	34
Methods	35
Properties	36
Protocols	38
Implementation	38
Summary	41
■ Chapter 3: Foundational APIs	43
Strings	43
Mutable Strings	46
Numbers	48
Numeric Object Literals	49
Data Objects	50
Collections	51
Arrays	51
Sets	57
Dictionaries	59
Rolling Your Own	61

Reflection and Type Introspection	64
Threading and Grand Central Dispatch	68
Run Loops.....	70
Coders and Decoders	71
Property Lists	73
Summary.....	74
■ Chapter 4: Objective-C Language Features	75
Strong and Weak References	75
Autorelease Pools.....	78
Exceptions.....	81
Synchronization.....	84
In-Depth: Messaging	87
Message Orientation.....	87
Sending Messages	88
Proxies and Message Forwarding	89
Blocks.....	93
Lexical Closures.....	95
Grand Central Dispatch.....	100
Summary.....	105
■ Chapter 5: Using the Filesystem.....	107
Files, Folders, and URLs	107
URLs	108
Creating and Using URLs	109
Managing Folders and Locations.....	122
Accessing File Contents	126
Random-Access Files	127
Streaming File Contents	129
Filesystem Change Coordination.....	137
File Presenters.....	137
Trying It Out	138

Searching with Spotlight	147
The Metadata API	148
Files in the Cloud	152
Summary	157
■ Chapter 6: Networking: Connections, Data, and the Cloud	159
Basic Principles	160
Network Latency	161
Asynchronicity	161
Sockets, Ports, Streams, and Datagrams	162
The Cocoa URL Loading System	164
Using NSURLConnection	166
Network Streams	173
Network Data	176
Reading and Writing JSON	176
Working with XML	177
Network Service Location	183
Service Resolution	184
Publishing a Service	186
Summary	187
■ Chapter 7: User Interfaces: The Application Kit	189
Coding Practices: Model-View-Controller	189
Windows, Panels, and Views	190
Controls	193
Buttons	194
Text Input	195
Interface Builder	196
User Interface Creation	199
Layout and Animation	206
Animating	208
Layout and Render Flow	210

Drawing Your Interface	211
Cocoa Graphics Primitives	213
Video Playback	219
Defining Documents	219
The User Interface	220
Document Code	221
Tying It Together	223
Summary	224
■ Chapter 8: Data Management with Core Data	225
Introducing Core Data	226
Components of an Object Model	228
Whose Fault Is It Anyway?	229
Creating an Object Model	230
A Better Model	232
Relationships and Abstract Entities	232
Custom Classes	234
Transient Properties	236
Validation	238
Firing It Up	241
Persistent Store Options	243
Multithreading and Core Data	244
Confinement	244
Private Queueing	246
Main-Thread Queueing	246
Hierarchical Contexts	246
Implementing Thread-Safe Contexts	247
Populating Your Store	250
Address Book Data	250
The User Interface	256
Sort Ordering	258
Laying It Out	259

Adding and Removing People.....	262
Viewing Addresses	263
A More Complex Cell View	266
Summary.....	267
■ Chapter 9: Writing an Application	269
Enabling iCloud	269
Enabling the App Sandbox.....	270
Core Data and iCloud.....	271
Sharing Your Data.....	276
Creating an XPC Service	277
Remote Access Protocols	280
Initializing The Connection.....	282
Implementing the Browser	284
Service Discovery	287
Vending Your Data	289
Becoming a Vendor	290
Providing Data	292
Server-Side Networking	297
Data Encoding	302
Encoding Other Data.....	303
Encoding Commands.....	306
Clients and Commands	309
Incoming Command Data	310
Sending Responses	314
Command Processing.....	315
Accessing Remote Address Books	317
Reaching Out	317
Implementing the Remote Address Book	321

Displaying Remote Address Books.....	335
The Browser UI	335
Viewing Remote Address Books	341
Summary.....	351
■ Chapter 10: Après Code: Distributing Your Application	353
Whither iOS?	354
Distributing Your Application	356
Developer Certificate Utility	357
Setting Up The Application.....	362
The App Store	363
Developer ID Distribution.....	368
Summary.....	369
Index.....	371

About the Authors



Jim Dovey has been writing software exclusively for the Macintosh (and later iOS) for 12 years now. A British expat, he works at Kobo in Toronto, Canada, where until recently he was the lead architect on the company's iOS applications, but these days he works as a liaison with the publishing industry and various standards committees and in the office carries a big stick labeled "Implement ePub 3" (no really; it looks kind of like Mallett's Mallet — Google that). Under the nom-de-hackuerre (is that a thing? can we make it a thing please?) he's the creator of many open source projects, including AQGridView, the original grid view control for iOS; AQXMLParser, the best event-based XML parser for the iPhone; and the original third-party development kit for the Apple TV. He also worked on Outpost, the original Basecamp client for iPhone and created an Apple TV-based digital signage system. This is his first book, but he hopes to churn out many more in the future.



Ash Furrow has been writing iOS application since the days of iOS 2. While completing his undergraduate degree, he worked on iOS applications for provincial elections and taught iOS development at the University of New Brunswick. He has also developed several of his own applications (for sale on the App Store) and contributes to open source projects. In 2011, Ash moved to Toronto to work with 500px to create their now wildly popular iOS application.

Currently, Ash works at 500px as the lead developer of the iOS team. He also tweets, blogs, and photographs.

About the Technical Reviewer



Felipe Laso Marsetti is a self-taught software developer specializing in iOS development. He is currently employed as a Systems Engineer at Lextech Global Services. Despite having worked with many languages throughout his life, nothing makes him happier than working on projects for iPhone and iPad. Felipe has over two years of professional iOS experience. He likes to write on his blog at <http://iFe.li>, create iOS tutorials and articles as a member of www.raywenderlich.com, and work as a technical reviewer for Objective-C and iOS related books. You can find Felipe on Twitter as @Airjordan12345, on Facebook under his name, or on App.net as @iFeli. When he's not working or programming, Felipe loves to read and learn new languages and technologies, watch sports, cook, or play the guitar and violin.

Acknowledgments

None of this would have come to pass without a chance meeting with Jeff LaMarche at Macworld 2009, who subsequently introduced me to Clay Andres of Apress at WDC that year. The authors and editors of the Apress family have all been a great help and inspiration, especially Felipe Laso Marsetti, whose assistance has been invaluable in ensuring the navigability of the mine of information within these pages; and editors Katie Sullivan, Douglas Pundick, and Steve Anglin, who should particularly be rewarded for putting up with my Douglas-Adams-like approach to deadlines over the last year.

—James Dovey

I've had a lot of help, both in the content I wrote for this book and with getting to a position where I had enough experience to write it. No one gets where they are on their own; everyone has help along their way. There are simply too many friends, teachers, and mentors to thank. I ran ideas and passages past two friends in particular who have always been invaluable in helping me perfect my writing; thank you to Jason Brennan and Paddy O'Brien for their discerning eyes.

My wife was absolutely supportive during my work on this book. She helped me keep working through late nights and weekends, and I couldn't have done this without her.

— Ash Furrow

Getting Started with Objective-C

The Objective-C programming language has a long history, and while it has languished in the fringes as a niche language for much of that time, the introduction of the iPhone has catapulted it to fame (or infamy): in January 2012, Objective-C was announced as the winner of the TIOBE Programming Language Award for 2011. This award goes to the language that sees the greatest increase in usage over the previous twelve months; in the case of Objective-C, it leaped from eighth place to fifth on the index during 2011. You can see its sudden, sharp climb in Figure 1-1.

The Objective-C programming language was created in the early 1980s by Brad Cox and Tom Love at their company StepStone. It was designed to bring the object-oriented programming approach of the Smalltalk language (created at Xerox PARC in the 1970s) to the existing world of software systems implemented using the C programming language. In 1988, Steve Jobs (yes, that Steve Jobs) licensed the Objective-C language and runtime from StepStone for use in the NeXT operating system. NeXT also implemented Objective-C compiler support in GCC, and developed the FoundationKit and ApplicationKit frameworks, which formed the underpinnings of the NeXTstep operating system's programming environment. While NeXT computers didn't take the world by storm, the development environment it built using Objective-C was widely lauded in the software industry; the OS eventually developed into the OpenStep standard, used by both NeXT and Sun Microsystems in the mid-1990s.

In 1997, Apple, in search of a solid base for a new next-generation operating system, purchased NeXT. The NeXTstep OS was then used as the basis for Mac OS X, which saw its first commercial release in early 2001; while libraries for compatibility with the old Mac OS line of systems were included, AppKit and Foundation (by then known by the marketing name Cocoa) formed the core of the new programming environment on OS X. NeXT's programming tools, Project Builder and Interface Builder, were included for free with every copy of Mac OS X, but it was with the release of the iPhone SDK in 2008 that Objective-C began to really take off as programmers rushed to write software for this exciting new device.

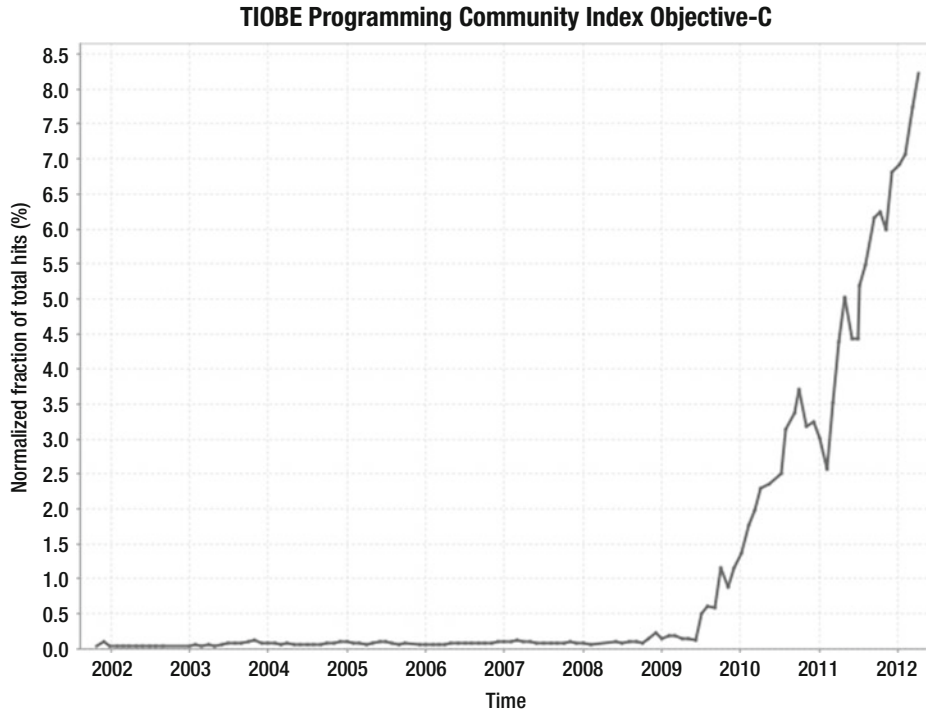


Figure 1-1. TPCI Objective-C Usage Trend, January 2002–January 2012

In this chapter you will learn how to use the Xcode programming environment to create a simple Mac application, including work on the UI and user interaction. After that you'll look at some of the details of the Objective-C language itself: the keywords, structure, and format of Objective-C programs, and the capabilities provided by the language itself.

Xcode

Programming for the Mac and iPhone is done primarily using Apple's free toolset, which chiefly revolves around the Xcode integrated development environment (IDE). Historically, Xcode shipped with all copies of OS X on disc or was available for download via the Apple Developer Connection web site. In these days of the App Store, however, Xcode is primarily obtained through it. Fire up the App Store application on your Mac, type "Xcode" into the search field, and hit Enter. You'll find yourself presented with the item you see in Figure 1-2.



Figure 1-2. The latest version of Xcode is freely available from the Mac App Store

Click to download it, and (admittedly some time later) you'll have a copy of Xcode in your Applications folder ready to use.

Xcode comes with a lot more than just its namesake IDE application. It also contains many useful debugging and profiling utilities, and provides optional downloads for command-line versions of the GCC and LLVM compiler suites. Among the available tools you will find are the following:

- **Instruments:** An application for generating detailed runtime profiling information for your applications—probably the most useful tool in your arsenal for a Mac or iOS developer.
- **Dashcode:** An HTML and JavaScript editor designed to help you to easily construct Dashboard widgets and Safari plug-ins.
- **Quartz Composer:** An application that enables the creation of complex graphical transformations, filters, and animations using a no-code patch-bay assembly technique.
- **OpenGL Apps:** A full suite of apps are provided to work with OpenGL (and OpenGL ES on iOS). Here you'll find profilers, performance monitors, shader builders, and an OpenGL driver monitor.

- *Network Link Conditioner*: A dream come true for network-based software engineers, this handy little tool lets you simulate a host of different network profiles. It comes with defaults for the most commonly encountered environments, and you can create your own, specifying bandwidth, percentage of dropped packets, latency, and DNS latency. Want to debug how your iOS app handles when it's right on the very edge of a Wi-Fi network? That becomes nice and easy with this little tool.

Those are a few of our favorites, but it's by no means an exhaustive list. As you will see later in the book, the technology underlying a lot of the Xcode tools is if anything even more impressive.

Creating Your First Project

Upon launching Xcode for the first time, you will find yourself presented with the application's Welcome screen. The following steps will guide you through the creation of the new project.

1. Click the button marked "Create a new Xcode project." You will be asked which type of project you would like to create.
2. From the Mac OS X section, select Application, then the Cocoa Application icon in the main pane.
3. Click Next to be presented with some options to define your project. Enter the details shown in Figure 1-3, then click Next again and choose where to save your project.

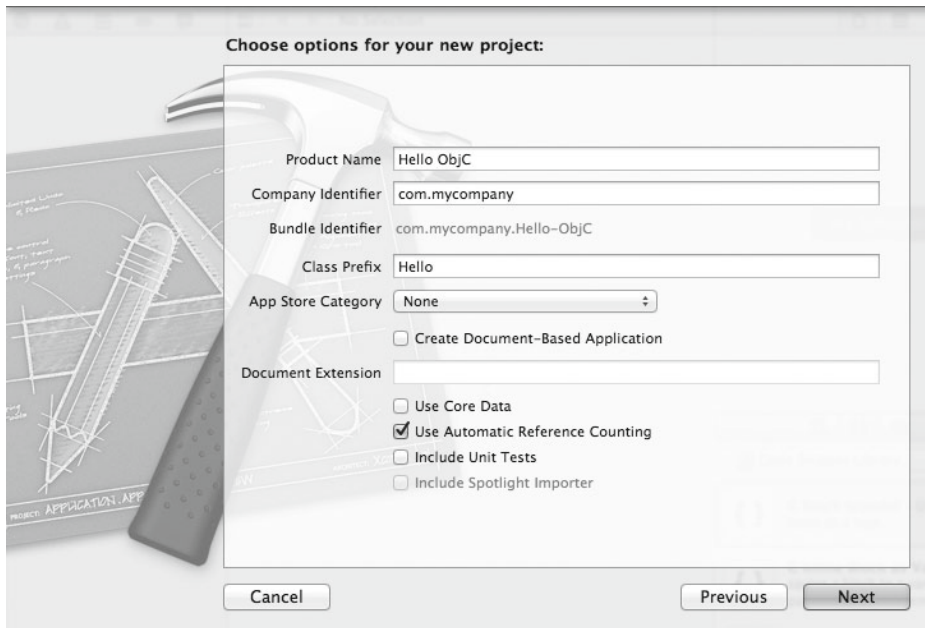


Figure 1-3. The options for your first project

Let's go through the layout of Xcode and the new project. On the left of the window you can see the Navigator, shown in Figure 1-4. This is where you can browse your project's source code files, resources, libraries, and output. The Navigator will also let you browse your project's class hierarchy, search and replace across your entire project, and browse build logs.

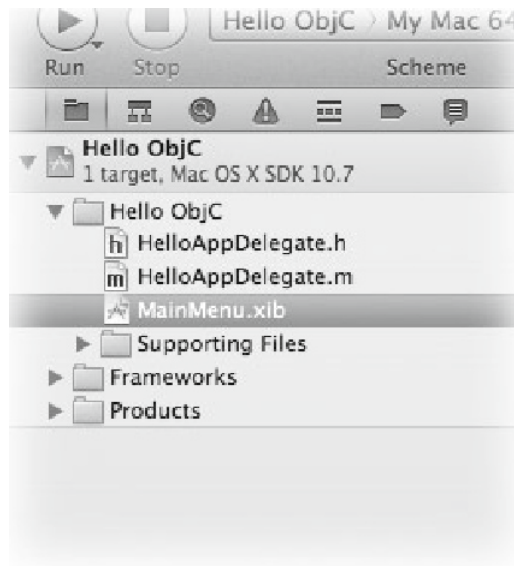


Figure 1-4. The Xcode Navigator pane

In the center pane of the Xcode window is the editor. Here's where you'll work with your code and your user interface resources.

On the right hand side is the Utilities pane. The upper part is context-sensitive and displays different choices of tabs depending upon the content currently focused in the editor pane. Below this is a palette from which you can drag user interface elements, new files based on templates, code snippets, and media. You can add your own templates and snippets here, too.

The Application Template

The Cocoa Application template generated a lot of information for you already. In fact, you already have a fully-functional application here. In the Navigator, switch to the browser tab (the leftmost option) and look inside the Hello ObjC folder. Here you'll see your primary source files and the user interface definition (a .xib file). Also in here is a Supporting Files folder; it contains the application's main.m file, which is responsible for kicking off the application itself, and the prefix header, which is included automatically into every file you add to the project. You'll also see Hello ObjC-Info.plist, which contains metadata about your application, and InfoPlist.strings, which holds localized versions of the data in the .plist file. You usually won't need to change these directly, as the Info.plist is most commonly edited through the target editor, to which you will be introduced in a later chapter.

The one item here that you might want to change is `Credits.rtf`. The contents of this file will be displayed within the application's About dialog; and as it's an `.rtf` file, you can style this as you like. The contents will be placed in a scrollable multi-line text field on the About dialog.

Below this is the Frameworks folder. It contains a list of all the frameworks and dynamic libraries upon which your application relies. Note that this is not an automatically-managed list: you need to add frameworks and libraries to the project yourself as you need them. Lastly, the Products folder contains a reference to the compiled application. Right now its name is likely in red, since it hasn't yet been built.

Click once on `HelloAppDelegate.h` to open it in the editor pane. Right now it looks a little bare, as seen in Listing 1-1. The code declares the structure and interface of a *class*, in this case named `HelloAppDelegate`. It tells the system that it implements all required methods defined in a *protocol* called `NSApplicationDelegate`, and that it has one property called `window`. You'll look into the details of this syntax in the next chapter, but for now just take it on trust that this works as expected.

Listing 1-1. HelloAppDelegate.h

```
#import <Cocoa/Cocoa.h>

@interface HelloAppDelegate : NSObject <NSApplicationDelegate>

@property (assign) IBOutlet NSWindow *window;

@end
```

Next is the implementation file, seen in Listing 1-2. This is similarly terse right now: in between some delimiters declaring the implementation of the `HelloAppDelegate` class all you can see is a directive named `@synthesize`, which seems to refer to the `window` property you saw a moment ago. This is, in fact, exactly the case: this directive tells the Objective-C compiler to synthesize getters and setters for the `window` property, saving you the need to write them yourself. It also specifies that the *instance member variable* used to store the property should be called `_window`; the compiler will create that member variable for you, too, again saving on the need to write it out explicitly.

Listing 1-2. HelloAppDelegate.m

```
#import "HelloAppDelegate.h"

@implementation HelloAppDelegate

@synthesize window = _window;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    // Insert code here to initialize your application
}

@end
```

Hello Interface Builder

If you select the `MainMenu.xib` file, the editor changes into Interface Builder mode, so named because the task of building user interfaces was until recently the domain of a separate (though integrated) application titled, appropriately enough, Interface Builder. You can see what this looks like in Figure 1-5.

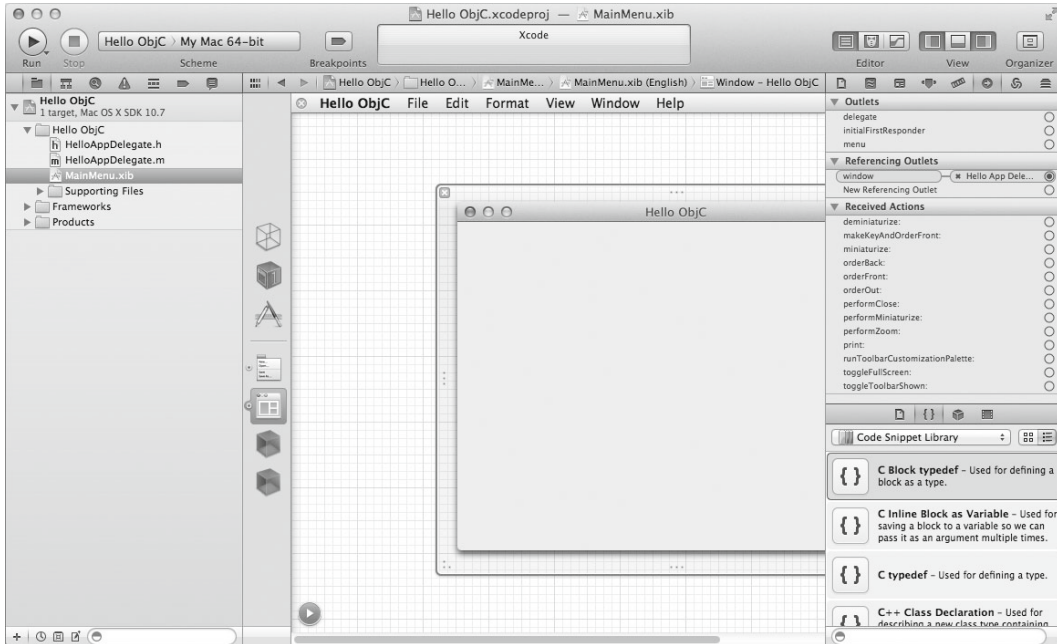


Figure 1-5. Interface Builder

In here you can see the application's menu, and down the left side of the editor is the document outline. All the objects in the interface document are listed here: at the top are the “inferred” objects, which are present in all (or almost all) `.xib` documents. Below the divider are objects explicitly added to the `.nib` file. The second item in this list is the application's window. Select that to make it appear in the editor.

Now that it's selected, the upper part of the Utilities pane on the right side of Xcode's window gains a lot more tabs. Click through these to see what they present; hovering the mouse over a tab selector will show a tooltip informing you of that tab's name.

Now, thanks to a lot of behind-the-scenes cleverness in the Interface Builder, you can build a nice application with user input and dynamically-updating feedback. What's more, you'll add only three lines of code to the project to do so!

User Interface Controls

First of all, you want to have somewhere for the user to type. Fetch your controls from the object palette in the lower half of the utilities pane; you can see all the items you'll use in Figure 1-6.

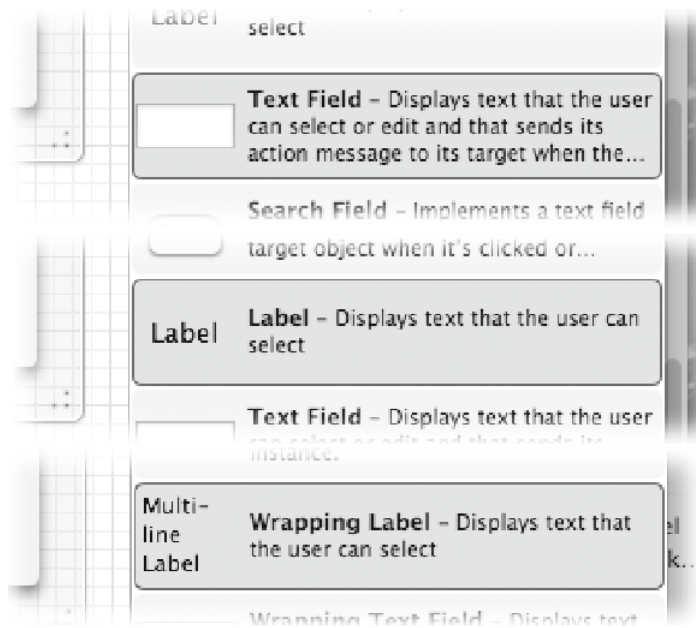


Figure 1-6. *The Text Field, Label, and Multi-Line Label controls*

1. In the lower part of the Utilities pane, select the second tab from the right (the box icon) to switch to the user interface Object Palette.
2. Pull down the Object Library pop-up menu and select Controls to limit the contents of the palette to just the standard controls for the moment (see Figure 1-6).
3. The first item you need is a text field. Scroll down a little way to find it.
4. Now drag that row from the palette straight out and onto the window in the editor. You'll notice that it changes into a real text field as it does so.
5. Move it up towards the top-right of the window's content area and blue lines will appear, helping snap the field into place. Position it there, at the top-right, as in Figure 1-7.

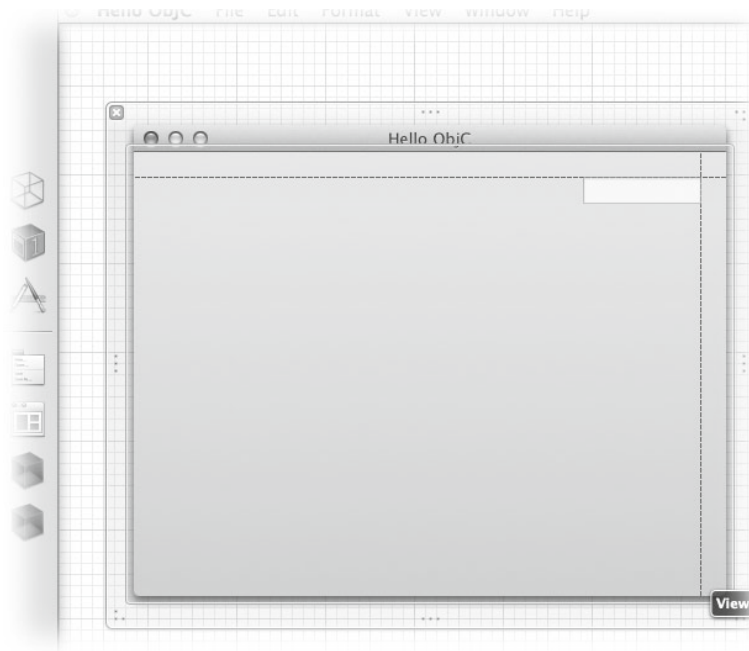


Figure 1-7. *Placing the text field*

6. Next you'll look for a label (a non-editable text field with no special background).
7. Drag this up to the top left, but notice that, while it can click into the top-left corner happily enough, guides also appear that cause it to align with the bottom edge of the text field you've already placed, or with the baseline of the text within that text field. This latter is the one you want to use: drop the label there, as shown in Figure 1-8.

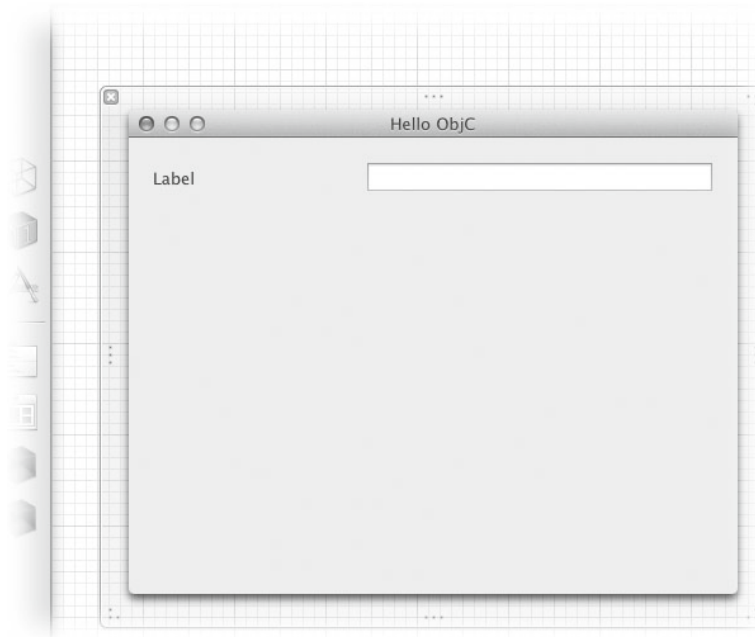


Figure 1-8. *Positioning the label*

8. To edit the label's text, double-click it. Type "Your Name:" and press Enter to store the change. Now clip the label's size to that of its text by pressing ⌘+=.
9. Select the text field and move the mouse cursor over the leftmost edge of the text field until the cursor changes to resize mode (a pair of arrows pointing both left and right). Click and drag the edge of the field over towards the label, and stop when the blue guide appears.
10. Lastly, look for the Wrapping Label control in the Object Library, and drag it into the center of the window, a little below the text field. More guides will appear to help snap it into the horizontal center of the window and to keep it well positioned below the text field itself. We suggest moving it a little further down so it has a nice amount of space around it.
11. Drag its edges out to meet the guides near the left and right edges of the window; this means the text can grow nicely. Now click on the handle in the center of the window's bottom edge and drag that up a little, shrinking the window so there's not quite so much empty space there.

Interface Bindings

If you're coming to Objective-C from another language, you might be used to the idea of handling your UI by hooking up variables referencing the various UI elements for manipulation. In Cocoa, however, that isn't always necessary. Instead, there is a system called *key-value coding* (KVC), which allows observation of a given value contained in a given object, which is referenced by a key. The key is either a method name or a member variable name—most commonly a method. The property declaration you saw earlier actually generates code that conforms precisely to that required by KVC, so that's how you'll be referencing and storing your values.

We will cover KVC in more depth in a later chapter, but for now you'll take advantage of a technology built on top of it: bindings. The essence of the idea is that certain properties of a user interface element can be *bound* to a value specified using KVC. This means that when one changes, the other does, too: editing a text field will change the value to which it's bound, and vice versa. Many properties of UI elements can be bound in this manner, but here you'll focus on arguably the most important one: the element's *value*.

In the case of a text field, the element's value is a string. So, first of all, you must create a string property somewhere to which you'll bind your interface. To do so, open up `HelloAppDelegate.h` and enter a new line under the existing property (see the line in bold in Listing 1-3).

Listing 1-3. The `userName` Property

```
@interface HelloAppDelegate : NSObject <UIApplicationDelegate>

@property (assign) IBOutlet NSWindow *window;
@property (copy) NSString * userName;

@end
```

This tells the world at large that `HelloAppDelegate` has a property called `userName` and that it is a string. It also states that the string is *copied* rather than *referenced* when set. Don't worry if you're not sure what that means yet: you soon will. For now, just accept it as a Good Thing.

This only declares the property, however. To actually implement it requires one more step. Open `HelloAppDelegate.m` and enter the highlighted line in Listing 1-4.

Listing 1-4. Synthesizing the `userName` Property

```
@implementation HelloAppDelegate

@synthesize window = _window;
@synthesize userName;

@end
```

Here you have asked the compiler to synthesize the implementation for you. Note that, unlike the `window` property, you have opted not to provide a name for the property's backing member variable; by convention, this means that the member variable's name matches the property's name exactly.

The next steps both happen in the Interface Builder: click `MainMenu.xib` to open it once more.

SYNTHESIZED VARIABLE NAMING

There are a number of different approaches to the naming of properties and their corresponding instance variables. Each developer no doubt has their own preference: we like to let the compiler handle the instance variables itself.

The following are the two most commonly seen approaches:

- *No name specified*: The compiler uses the exact same name for the creation of the backing variable.
- *An underscored name*: This matches Apple's internal naming scheme for instance variables. We and many other programmers follow this scheme, although Apple has, at times, recommended against it due to a potential clash with any instance variable names they might add to a class in the future.

A number of people argue that you should always explicitly supply a variable name when synthesizing properties, but we take the opposite approach as we believe it encourages the use of the accessor methods rather than directly accessing the underlying variable. This becomes especially important when using atomically accessed properties: the accessors are locked and synchronized, so nothing can read a variable mid-modification from a secondary thread. Accessing the instance variable directly has no such guarantees, however.

Binding User Input

1. Select the text field.
2. In the Utilities pane you'll see some of its attributes appear; the fourth tab contains the Attributes inspector where you can adjust the field's attributes: its font, colors, and some behavior. The fifth is the Size inspector where you can adjust the field's size and its placement, as well as its behavior when resizing its containing view (in this case, the window). The sixth is the Connections inspector, which you will see later in the book. Following that is the Bindings inspector, which is what you'll use to hook up the field's value.
3. At the top of the Bindings inspector is a pop-open row titled "Value." Open it to see a lot of options.
4. At the top is the "Bind to:" pop-up menu. In here you can see references to the application itself, the file's owner (the object that handles the interface definition in this .xib file at runtime), the global font manager and user defaults, and your app's delegate object, Hello App Delegate.

DELEGATES

The concept of a *delegate* is not peculiar to Objective-C, but due to the language's dynamic nature it is one of the core techniques used by the system libraries. A *delegate object* is an object that conforms to some predefined protocol—a list of methods it agrees to implement—by which another object can request that it undertake some actions or make some decisions on the other's behalf. For instance, a text field's delegate might check the text being entered and tell the text field to reject certain characters.

Delegation is a very powerful tool and is the reason why Objective-C applications rarely tend to subclass classes such as the `Application` class: instead, a delegate object is created to make the important decisions and leaves the `Application` instance alone to handle the guts of making the app “go.”

5. Select Hello App Delegate from the pop-up menu. In the Model Key Path field enter `self.userName`.
6. Check the “Continuously Updates Value” checkbox. The result should look like that in Figure 1-9.

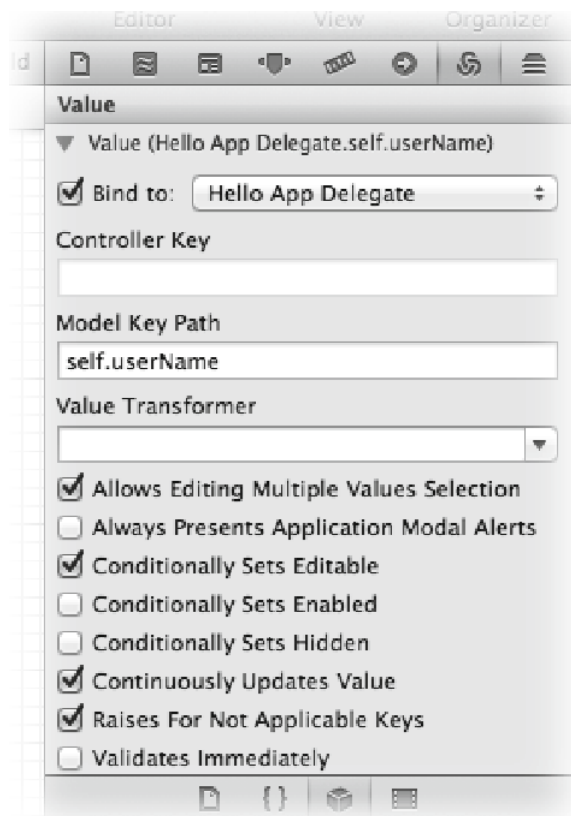


Figure 1-9. Binding the text field's value

This field's value is now bound to the property created earlier; as the user types in the field, the property's value will be updated to match.

The next step is to make some output from that value.

1. Select the Multiline Label and open the Bindings inspector once more. Here you'll not just set the value, however: you'll provide a pattern, similar to a format string, which will be augmented by a bound value.
2. Open the "Display Pattern Value1" item; it looks quite similar to Figure 1-9, with the addition of a Display Pattern value. By default this field contains `%{value1}@`, which is the way in which the Value1 binding created here will be applied to the label. You're going to bind to the same property here that you did before.
3. Select Hello App Delegate from the pop-up menu, and type `self.userName` into the "Model Key Path" field.
4. Now edit the Display Pattern field slightly, so it reads `'Hello, %{value1}@!'`. This will cause the field to display "Hello, *user!*" for a given value of user. Your input should leave the inspector looking similar to Figure 1-10.

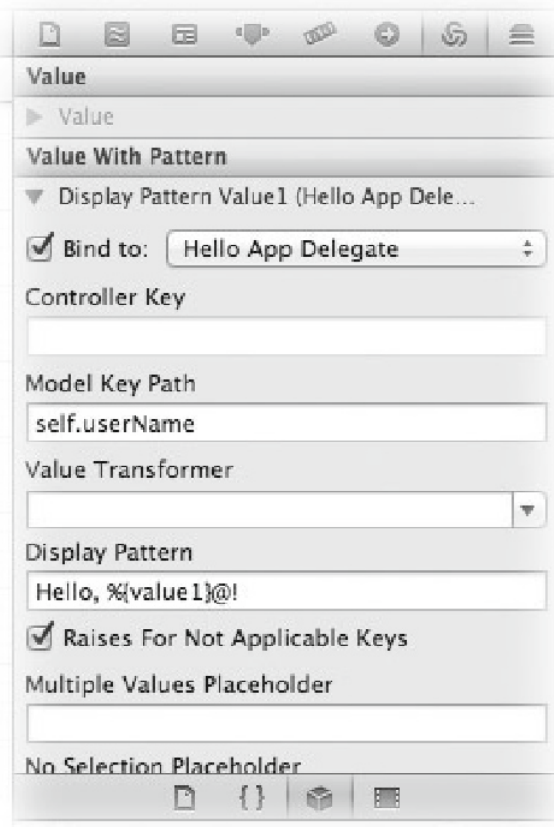


Figure 1-10. Formatting the output field

5. Lastly, switch to the Attributes inspector (the fourth tab on the sidebar) and change the field's alignment to centered, as in Figure 1-11.

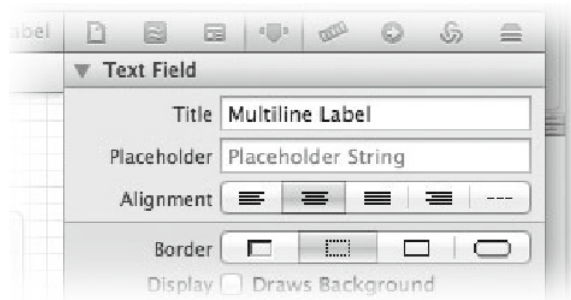


Figure 1-11. Centering the output field

Running the App

It might come as a pleasant surprise to note that the application is now all but finished. You can compile and launch it right now by clicking the Run button in Xcode's toolbar. As you type in the text field, the output field below it updates dynamically.

However, it looks a bit strange at first launch. The text field doesn't contain anything, so the output field reads "Hello, !" and that doesn't really seem very impressive. It might be better to provide a default value when the application launches. In fact, it might be useful to preset the content with the current user's full name. Let's do that.

Open `HelloAppDelegate.m` once more. You're going to fill in the empty method here, which is part of `NSApplication`'s delegation protocol. It currently looks like the code in Listing 1-5.

Listing 1-5. Delegating the App Launch

```
@synthesize userName;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    // Insert code here to initialize your application
}

@end
```

This method is called by `NSApplication` on its delegate once the application has finished launching and is ready to begin showing windows and processing user input. This is where you'll likely set up the initial state for any applications you write. In this instance, you'll fetch the user's name using the handy C function `NSFullUserName()` and assign it to the `userName` property. Assigning and referencing properties uses a structure-like syntax to differentiate it from regular method calls; the compiler swaps in the real Objective-C method calls when compiling the project. Enter the highlighted code from Listing 1-6.