Apple’s Xcode technology is making the development curve smoother than it has ever been with its easy-to-develop features and enhancements. With the latest release of Xcode 7, Apple has also added great support for Swift development.

This book will introduce you to all the new features of Xcode 7 and demonstrate how Swift programming can be much easier, faster, and simply better with Xcode!

Even if you’re starting with just a little knowledge of Swift or Xcode, you will learn the basics of the language as well as the tool. You will then use this knowledge to create simple applications and will learn how to debug and optimize your code. At the end of this book, you will have learned enough to build, run, and submit your very own application to the App Store.

Who this book is written for
If you are a novice programmer who is familiar with the concepts of object-oriented programming, but has little to no knowledge of Swift, Xcode, or Apple’s Cocoa APIs, then this book is for you. Starting with an introduction to the basics of Xcode and the Swift programming language, you will learn all of the skills that are essential to building an app and submitting it to the App Store.

What you will learn from this book
- Get an introduction to Xcode and get to know how to navigate and use the tool
- Build playgrounds to learn about and explore the environment
- Create an adaptive UI with the interface builder that will help your app to scale through a wide range of devices
- Understand Table, Collection, and Stack views and find out how they can be implemented
- Create custom components for your application through frameworks and live previews
- Exploit Xcode’s advanced features to elevate your debugging and testing capabilities
- Run your application on a physical device and submit it to the App Store

In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 4 'Interface Builder'
- A synopsis of the book’s content
- More information on Xcode 7 Essentials (Second Edition)
About the Authors

**Brett Ohland** develops software, likes to tinker and fix things (especially bikes), and keeps himself and others caffeinated. He has well over a decade of experience of converting keystrokes and staring at the ceiling into functional applications for clients.

He started his career by building Flash websites when those were still an amazing thing, transitioned to web development when everyone was loving JavaScript, and then moved on to iOS development so that he could build his dream photography app. Throughout this period, Brett has worked for major advertising agencies, consultancies of all sizes, a stock photo agency, two guys in a basement, and himself for a while. Several teaching opportunities have allowed him to share his knowledge at start-up incubators and even a university.

Currently, he writes about his technological discoveries and interests at [ampersandsoftworks.com](http://ampersandsoftworks.com) when he isn't riding his bike, chasing his toddler around the house, or pulling a shot of espresso.
Jayant Varma is a technophile with a career spanning more than two decades and was introduced to computing in the days of 8-bit computers and Z80 chips. While managing the IT and telecom department at BMW’s dealerships in India and Oman, and Nissan in Qatar, he worked extensively on Windows, AS/400, and Unix. His love for traveling inspired him to work and travel to several countries. He is currently based in Australia.

Jayant’s technological journey began as a Microsoft technologies developer and then diversified. His focus is now on Apple and mobile technologies. He has a master’s degree in business administration and IT from James Cook University, Australia. He also lectured at James Cook University and coordinated the onshore and off-shore teaching of Linux/Unix administration. Jayant has worked closely with the Australian Computer Society (ACS) and Apple University Consortium (AUC) on workshops and projects.

He authored the book *Learn Lua for iOS Game Development*, Apress, and has also been a technical reviewer on several titles. As a founder, consultant, and developer at OZApps (www.oz-apps.com), he helps organizations and individuals integrate technology into their businesses and strategies. He also conducts training sessions and workshops, and writes blogs to share his knowledge with the community.
Preface

Welcome to Xcode 7 Essentials. Developing apps for desktop and mobile computers has never been easier for anyone to do. The technical and commuting requirements are such that practically anyone can make and release an app to millions of users. With well over a billion iOS devices and nearly half a billion Macs in people's homes and businesses, there is a real market for any piece of software, no matter how specialized. Apple has fostered this market by continually updating Xcode, their free development tool, to keep it modern, easy to use, and powerful.

This book will be covering Xcode 7, the current version of the app, as well as the Swift programming language. The book starts with a basic tour of Xcode, takes you through many of the features of making the app, and ends with how to submit your app to the app store of your choice using a combination of theory and step-by-step guides.

What this book covers

Chapter 1, Introduction to Xcode, introduces Xcode and shows you how to find, install, and set it up for use.

Chapter 2, Tour of Xcode, takes a deeper look at the tool itself. Here, you become familiar with the interface and its features.

Chapter 3, Playgrounds, introduces Playgrounds, an interactive environment for learning and experimenting with the Swift programming language. This chapter will then walk you through many of Swift's features and its syntax to help you express yourself in code.

Chapter 4, Interface Builder, covers Interface Builder and Storyboards, Xcode's tools for creating user interfaces by dragging and dropping elements on a screen.
Chapter 5, *Table, Collection, and Stack Views*, uses the knowledge gained in the previous chapters and shows you how three advanced view types will let you build complicated-looking layouts quickly and easily.

Chapter 6, *Custom Controls*, shows you how you can create custom, live-updating controls for your applications, and how to easily share them across projects as a framework.

Chapter 7, *Debugging*, shows you the tools that Xcode provides to help you become a great detective and exterminator of bugs and crashes in your code.

Chapter 8, *Testing Your Code*, introduces the concepts behind test-driven development and the unit test and UI testing tools that Xcode provides to help you write great code.

Chapter 9, *Sideloading, Optimizing, and Submitting Your App*, shows you how to run your app on a physical device, how it can be optimized, and the steps necessary to submit it to the app store of your choice.

Appendix, *Everything Else*, covers more advanced topics. These include distributing beta versions of your app using TestFlight, collecting and reading crash reports submitted to you by those testers, and how to simplify your user interfaces with Storyboard references.
In the last chapter, you learned about Playgrounds and were introduced to the Swift programming language. In this chapter, we’re going to learn about another tool that Xcode offers: **Interface Builder (IB)**. We’ll also see how it will let you lay out your user interface (UI) graphically without writing a line of code.

The main user interface file that’s available to you is a .storyboard file. In this file, you can lay out your entire application and visually connect them through segues. Creating a new application using one of Xcode’s templates will automatically create a .storyboard file for you to immediately start building your UI.

Another type of UI file that is supported by Xcode is the .xib file. These were replaced by storyboards in Xcode 4.3 as the recommended user interface file by Apple. While storyboards can contain multiple views and many connections, XIB files are limited to one view and all transitions to other screens must be done in code.

We will only be working with .storyboard files for the rest of the book.

In this chapter, we will cover the following topics:

- An introduction to Interface Builder
- Creating a user interface
- Adaptive layouts
- Auto layout
- Connecting your UI to code
- UINavigationController
- Subclassing UIViewController
Model View Controller

Developing iOS, watchOS, tvOS, and OS X applications means that you're interacting with a set of Application Programming Interfaces (APIs) provided to you by Apple. On OS X, these are called Cocoa and they're called Cocoa Touch on iOS, watchOS, and tvOS.

Cocoa APIs follow an architecture pattern called Model View Controller (MVC). This breaks your app down into three types of object:

- **Models**: These represent data structures (the code)
- **Views**: These represent anything the user sees and handles user interaction
- **Controllers**: These link Views and Models, handle logic, and maintain state

Using MVC isn't enforced by the Swift compiler or Xcode; it's up to you as the developer to keep your code from acquiring functionality it shouldn't. Views represent what the user sees or touches; Models represent the classes, structures, and enumerations that power your app; and Controllers act as a bridge between your views and your models. A view shouldn't interact with a model and a controller shouldn't model data.

IB is the tool used to create our views in a very user-friendly way. We drag and drop items onto a canvas and then use a mouse and keyboard shortcut to create connections between other objects and even code.

Understanding Interface Builder

When a UI file is opened in Xcode, an IB canvas will replace the standard editor area. The options in the Utility sidebar will be expanded and that will be where you will find the object library that contains all of the UI elements you can use.

To start off, let's create a new iOS project by navigating to **File | New | Project**. Then, select **Single View Application** and save the project on your Mac; click on the Main.storyboard file in the project navigator to start editing your UI.
Until Xcode 4 was released in 2011, IB was a separate program from Xcode. IB functionality is only used when interacting with storyboard or XIB files.

The editor area of Xcode has now become the canvas. This is the area where you will be dragging and dropping all of your UI elements. Because we selected the Single View Application template, our canvas is prepopulated with a single view:

The canvas with an empty view

The arrow on the left hand side of the central white box says that this is the initial view. This is important as you build more complex user interfaces.
There are three icons at the top of each view:

- **View Controller**: This represents the controller that handles the logic of this view. It needs to extend the UIViewController class provided by the Cocoa APIs.

- **First Responder**: This represents the element in the view that will be the first element to handle any user interaction. This is helpful if you have multiple text views and you’d like to direct focus towards any one.

- **Exit**: This represents code that will run as the view exits.

At the bottom of the canvas area, there are three groups of icons. From left to right, they are as follows:

- **Show/Hide Document Outline**: This opens and closes the document outline. The document outline area shows the view hierarchy and allows for quick access to view elements.
• **Size Class selector**: This allows you to select the Size Class that's currently open for editing.

• **Auto Layout tools**: These tools help you lay out your code using Auto Layout.

**Views and View Controllers**

As we learned in the MVC section, if we want to change our view or handle any user interaction, a view needs a view controller. Open up the Utility Inspector sidebar, select the identity inspector, and press the **View Controller** button at the top of our view in the canvas:

![View Controller button](image)

The **View Controller** button is the leftmost button at the top of the view, within the canvas. The Identity Inspector is currently selected in the Utility sidebar on the right.
The class area of the Identity Inspector Utility shows which class is controlling this view. In this case, it is a class called ViewController. Clicking on the circular arrow icon next to the name will open the controller in the editor area. You can press the back arrow in the jump bar to come back to the Main.storyboard file.

ViewController is simply the default name that is generated by Xcode when using the Single View Application template. Any views that you create for your application should be named to better reflect what each view controller will do: LandingPage, Sidebar, ImageViewer, ContactList, and so on.

Understanding the relationship between views and their controllers is an important part of developing a Cocoa app.

**Adding elements to your view**

At the bottom of the Utility area, you can find the objects library by selecting the object library icon:

![Object Library]

The object library
Find and drag a **Button** object onto the view using the blue dotted line guides to snap the button in the exact center of the view.

You can filter the objects listed in the object library by using the filter bar. Typing `button` will filter the results to objects with button in their names.

We now have a button in a view; you'll see that the button has been added to the view hierarchy in the document outline area.

You can run the application at this point by hitting the **Run** button or using the `Cmd + R` keyboard shortcut. Using the scheme editor, select different devices on which this application will be run. Depending on the size of the device, you'll see a different amount of the button. This is because without any Auto Layout information, the view is simply using the current X and Y coordinates you provided by placing the button in the center of this view.

**Adaptive UI**

To understand the adaptive UI system, you need to understand how Size Classes and Auto Layout solve the challenges of developing modern apps.

iOS 9 supports four different sizes of iPhone and three different sizes of iPad. This doesn't include the new split view supported on iPads, any higher density iOS devices that are in development at Apple, or even supporting both portrait and landscape orientations on current devices.

Before Adaptive UI and Size Classes, a developer was forced to include a large amount of code that would manually resize, hide, show, and animate all of your elements. This was cumbersome and error prone, and it would break if Apple announced a new class of device to support. This is what the Adaptive UI system is meant to handle for you.

Developers tend to think in terms of a grid. An application's UI elements are simply placed onto the screen at fixed coordinates with fixed dimensions (an object's frame and bounds). The iPhone version of an app would have a certain set of hardcoded values and the iPad version would simply have different values.

This is a brittle way of building applications; instead Apple wants you to think in terms of Size Classes and lay out your screen using Auto Layout.
Size Classes

At the bottom of the canvas in the editor area, there is an area that says wAny hAny. Selecting this with your mouse will make a Size Class selector pop up. Go ahead and select different combinations of widths and heights. You’ll see a square box that represents your view’s change in proportions.

The Any Size Class is specifically used to create universal layouts that support every Size Class. It’s a good habit to start designing using wAny/hAny as you’ll support the widest variety of devices possible.

There are nine different combinations that can be selected, but only three different classes: Compact, Regular, and Any. Alone, these size classes are a bit confusing, but consider the following:

- An iPhone uses wCompact/hRegular in portrait and wCompact/hCompact in landscape
- An iPad uses regular for both dimensions, regardless of orientation
- An iPhone 6 Plus uses wRegular/hCompact in landscape
Instead of thinking in terms of grids and dimensions, you need to think in terms of how your layout will differ between these Size Classes. By moving your layout information from code to IB, you're able to support an iPad and an iPhone using one single Storyboard and lay out everything visually. By using the universal Any Size Class, your design is future-proof.

For the remainder of this example, we will be using the master Size Class of wAny/hAny.

**Auto Layout**

Instead of hardcoding dimensions and sizes to each object on screen, Auto Layout is a system where you describe the relationship between objects.

![Three views with their constraints (relationships) to each other and their container](image)

At runtime, the app takes these rules (called constraints) and uses them to calculate the final positions for each object. At times, you'll want to pin an element to a specific corner of the screen, and sometimes you may want to always place one element below another element regardless of heights. Auto Layout gives you all of this power, and more.

Auto Layout is an advanced and, at times, complicated feature. We will cover some of the basics in this book. For more information, see Apple's developer resources: https://developer.apple.com/library/ios/recipes/xcode_help-IB_auto_layout/chapters/UnderstandingAutolayout.html
Using Auto Layout, every element needs to have enough constraints set so that the view can correctly place items on screen. This is called an element's alignment rectangle. You need to describe the element's position as well as its size in order to satisfy Auto Layout.

As an example, let's add some constraints to our button element from earlier to properly center it on screen, no matter which device is running the app.

Select our button, hold down the Ctrl key, and drag upwards. Release the mouse button once you're inside the ViewController's view.

This Ctrl + drag action is telling IB to set up a relationship between two objects. In this case, you're setting up a relationship between the button and the ViewController's view. Now, select **Center Horizontally in Container**. The button will now be centered horizontally and you'll see a visual representation of this new constraint as a red line that splits the view. The constraint is red because just adding a horizontal constraint isn't enough to satisfy the alignment rectangle. The next step will fix this.

Repeat the Ctrl + drag action, but this time drag to the right and release it on the container. Now, select **Center Vertically in Container**. The new constraint is represented by a horizontal blue line. We've now satisfied Auto Layout and both of the constraints we created are appearing blue on our canvas.

Another way to do this would be to open the alignment tool from the canvas' bottom bar and selecting the same options:
Running the app now in a device of any size will place the button in the middle of the screen.

The constraints currently affecting an object on screen will only appear when you've selected that item. You can actually click on each constraint in the canvas and view its properties in the Utility Inspector bar to make changes.

Run your app in various simulators to see how the button is always at the center.
Delete the button.

**Debugging Auto Layout**
Xcode will let you know if there aren't enough constraints on an object by adding an arrow in the document outline sidebar. Clicking on the button will show you any structural errors as well as give you options to try and fix the issue.

**Tweaking values**
While an object with constraints has been selected, you can modify various Auto Layout values in the Size Inspector section of the Utility sidebar.
**Pinning**

Another option in creating constraints is to use the **Pin** functionality. Drag a new button from the object library onto the view, select it, and press the **Pin** button:

![The Pin pop-up](image-url)
Chapter 4

The four boxes at the top of this pop up represents the distance from the current object to the container view. You can choose to pin the button any distance to the top, bottom, left or right. Enter 0 in all four boxes (the red constraint marker will change from a dotted line to a solid line) and press the **Add Constraints** button.

![Adding pins to each dimension](image)
If you happen to have added the pin at a 0 value already, the constraint indicator won’t change from dotted to solid. Click on the dotted constraint indicator to force IB to add it.

Right now, Auto Layout has an issue: the yellow dotted line area is the area that Auto Layout wants the button to fill. The four lines with numbers are telling you how far off the current location is compared to the Auto Layout constraint. We need to update the frame: press the Resolve Auto Layout button in the bottom toolbar, and press Update Frames.

Updating the frames simply tells Auto Layout to use its constraints and lay out every element in the view. This is helpful to return your view to how they’ll look at runtime.

Once the frames have been updated, you’ll see a bit of an odd sight. The button that you’ve pinned to each edge of the screen is now filling up the entire frame. Pinning an object pins the object and forces it to expand in width and height to fit the screen. To better visualize this, select the button that is now pinned, go to the property inspector in the Utility sidebar, and change the background color by scrolling down to the View subsection.
The default background for a button is clear as denoted by a box with a red slash across it.

Finally, delete the button.

**Relative positioning**

You can position elements relative to other elements by creating constraints between them. Let's create a layout that includes a label at the top of the screen and a button below it:

Drag a **Label** object from the object library onto the view and place it in the top-left corner of the screen.
Using the pin pop-up, pin the label to the left and right of the view (do not pin it to the bottom or top). Check the box next to **Height**. Click on the **Add Constraints** button.

If any of the values are already 0, you can simply click on the dotted red constraint indicator to add a constraint.

Instead of pinning the top of the label to the top of the view, we're going to pin it to the **Top Layout Guide**:

The **Top Layout Guide** is a special object that represents the top-most visible point on screen at any time. Pinning an object to this instead of the top of the view will guarantee that your object won't be covered by any title or navigation bars.
Before we update the frame to expand the label to its Auto Layout constraints, let’s add a button underneath:

Pin the button to the left and the right side of the view; this will expand the button to the full width of the view. Also, check the height constraint checkbox before selecting Add Constraints. To satisfy the X and Y position requirements, select the button and Ctrl + drag to the label and select **Vertical Spacing**.

Using the Attributes inspector in the Utility sidebar, let’s change the background of the button as well as set the tint to make the button more visible. Then, update frames by using the Fix Auto Layout Issues pop-up.

Now, the vertical position of the button is directly related to the bottom of the label. If the label ever expanded in height, the button would be pushed down by that same amount. This is the power of Auto Layout: you create relatively positioned elements so that no matter the screen size, your layout will be smart enough to intelligently grow or shrink as needed.
Run your application in a variety of simulators to see how the layout expands:

![The app running in an iPhone 4S and an iPad Air simulator](image)

**Adding more views**

Our current view has a button that does nothing when the user presses it. Let's make it transition to a new screen.

You can zoom out by using pinch to zoom mouse gestures.

First, double-click on the button object and change the text within the button. Change it to Open. Then, drag a View Controller object from the object library and place it to the right of our already created view.

A View that has a View Controller powering it can be called a screen. Yellow tinted objects in the object library are considered to be screens (View Controller for example), while black tinted objects are simple views (View for example).
Our newly created View Controller isn't connected to any other views. First, change the background of the newly created view by changing the background color property in the Attributes Inspector. Then, use the Ctrl + click + drag gesture to connect the button to our new view:

Choose Present Modally from the segue selector popup. You'll now see that there's a line with an arrow connecting the two views. This is known as a segue and it's how a storyboard visually represents a transition or relationship between two views.

A modal window converts the current view controller. Show would replace the current view controller with the destination view on screen. Each has their pros and cons and is covered in detail on the Apple Developer site: https://developer.apple.com

Running the application and pressing the button will now open that second screen in a modal window. We've created an app with two screens without writing a single line of code.
Navigation controllers

A Navigation controller is a special kind of controller. It knows which view is currently on screen, it keeps track of the hierarchy of views that make up an application, it handles the transition between various views, and it shows all of this in a standardized Navigation bar with standard buttons.

You'll notice that in our last example, with our Present Modally segue, when the second view is on screen we have no way to go back to the first view. Let's fix that by adding a Navigation controller.

First, delete the Present Modally segue that we created by selecting the connector between the two views and pressing the Delete key on the keyboard:

Select the first view and go to Editor | Embed In | Navigation Controller. You'll immediately see a Navigation Controller is added. Add a relationship segue to our original view and it then sets itself to be the initial view controller.

Our label and button, however, are partially covered. Since title and navigation bars in iOS 9 are transparent and will cover content, we luckily pinned our label (and by extension our button) to the Top Layout Guide and not the top of the view. Update our frames and our label and button will be properly positioned without making any changes to our constraints.

Create a segue between our first and second views by selecting the Open button and Ctrl + drag to the second view. This time, select Show.

Run the application and click on the button. You'll see that a happens transition and a back button is automatically created for you. Again, we have created an application that has multiple views without writing any code.

Connecting views and controllers

We now have an app with two pages. While impressive, we haven't tapped into the real power of IB to connect our view and our controllers.

Subclassing UIViewController

When we created a new project using the Single View Application template, Xcode created a View and View Controller for us in the storyboard. This is the view on the canvas that we've been dragging buttons and labels onto.
Remember our discussion of an MVC application; the view that we see on the IB canvas is the V portion and Xcode created the C portion for you as a file called ViewController.swift.

Our newly created View Controller that was placed on the canvas is only a View; we need to specifically create the Controller and associate these two pieces ourselves.

Our first step to create the controller is to go to File | New | File | Cocoa Touch Class.

The first window of the New File wizard is similar to the New Project wizard. In the Subclass Of area, select UIViewController and type in SecondViewController in the Class area. Leave Create XIB file unchecked and keep the Language selector as Swift. Click on Next.

SecondViewController is simply an example name. You can name the file anything that you'd like. Notice that Xcode will append ViewController to the end of any name you create to help keep the purpose of the file clear.

On the next window, save the file inside your project folder and click on Create. This new controller will be opened by Xcode in the source editor. You'll notice that we've created a file called SecondViewController.swift and that inside that file is a class called SecondViewController.

Understanding subclassing

A class is a type of object that can be subclassed. This simply means that you can take an object, inherit all of its functionality, and then be able to further customize it.

Earlier, we talked about how we interact with Cocoa Touch APIs when we're writing a native iOS application. Apple offers various objects that have specific purposes and the most basic view controller is named UIViewController. This class has the basic functionality you need to get a view on screen, know when it's about to appear or disappear, and do advanced things like handling animations and transitions.
When we created our `SecondViewController` class and file, Xcode included some boilerplate:

```swift
import UIKit

class SecondViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    /*
    // MARK: - Navigation
    // In a storyboard-based application, you will often want to
do a little preparation before navigation
    override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
        // Get the new view controller using
        segue.destinationViewController.
        // Pass the selected object to the new view controller.
    }
    */
}
```

Notice the following line:

```swift
class SecondViewController: UIViewController {

It simply says that our `SecondViewController` is subclassing `UIViewController`, where `UIViewController` is the superclass.
```

We now have a view and a controller. Unfortunately, at this point our second view in IB has no idea that we've created a controller for it to use. Let's connect them now:

Open up the `Main.Storyboard` file.
You can use the quick launch keyboard shortcut (Cmd + Shift + O) to open up the quick launch box. This box has fuzzy logic search built in and typing Main will autofill the Main.Storyboard file. Pressing Enter will open the file for editing.

Press the View Controller button at the top of the second view and open up the Identity Inspector in the Utility sidebar:

In the Class textbox, you'll notice that the default UIViewController is currently powering this view. Start typing SecondViewController in the box. Once the full class name appears, hit Enter on the keyboard.

IB is smart about classes and will do its best to guess which class is supposed to be powering the currently selected object as you type.

We've now associated our new view controller with our view.

**Creating connections, outlets and actions**

Now that we've associated our view with its controller, we have a screen. Let's add some functionality:

- Add a label whose text changes at runtime
- Add a button that will cause an alert dialogue box to appear when touched

Drag a label and a button and position or pin these two objects in a similar way to the first view. Change the label text (by double-clicking on the label) to **THIS IS GOING TO CHANGE** and the button text to show alert.

Close the document overview and Utility sidebars and click on the Assistant editor button in the editor area of the top toolbar.
You will now see the IB canvas on the left and a standard editor on the right, which contains the `SecondViewController.swift` file we just created:

If the Assistant editor isn't showing the contents of the `SecondViewController.swift` file, make sure that the jump bar has selected *Automatic* and that the *view controller* button on the view (in IB) has been selected.
Select the label and Ctrl + click + drag from the canvas on the left into the standard editor on the right. A tooltip will show Insert Outlet or Outlet Collection and show a line indicator. Release the mouse button and the add connection popup will show:

Keep the Connection selector as Outlet, type titleLabel into the Name input box, and click on Connect. The following line is added to your Swift class:

```swift
@IBOutlet var titleLabel: UILabel!
```

We’ve created our first outlet. What this means is that when we interact with the titleLabel object in our code, we will interact with the associated label in our view.

We now have the ability to change the text in the label in our code. Add the following line to the viewDidLoad() function:

```swift
override func viewDidLoad() {
    super.viewDidLoad()
    titleLabel.text = "Changed it" // Add this line
    // Do any additional setup after loading the view.
}
```
On running the app, you'll now see that the text in the label has changed:

```
super.viewDidLoad()
```

is an important line. This tells the parent class to run all of the code contained in `viewDidLoad()`. Nothing will show on screen if this line is omitted.

Outlets are one-way. We can change objects on screen but we can't know when a user interacts with it. Let's add an action to our button to fire an alert.

The `Ctrl +` drag from the button in the IB canvas to a line underneath the `didReceiveMemoryWarning()` function. Release the mouse and make the following changes in the connections popup:
Change the **Connection** type to **Action**, give it the name `userDidTouch`, and press **Connect**:

![Diagram of the connection setting](image)

<table>
<thead>
<tr>
<th>Connection</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Second View Controller</td>
</tr>
<tr>
<td>Name</td>
<td>userDidTouch</td>
</tr>
<tr>
<td>Type</td>
<td>AnyObject</td>
</tr>
<tr>
<td>Event</td>
<td>Touch Up Inside</td>
</tr>
<tr>
<td>Arguments</td>
<td>Sender</td>
</tr>
</tbody>
</table>

There are many different **Events** that are supported by buttons. **Touch Up Inside** is the most common as it represents the user pressing a button on screen and releasing their finger.

The following code is added:

```swift
@IBAction func userDidTouch(sender: AnyObject) {
}
```

This function will now be called whenever a user presses the button. You can test this action by adding a line of debug code:

```swift
@IBAction func userDidTouch(sender: AnyObject) {
    print("Button Pressed!")
}
```

Running the application and pressing the button will now cause **Button Pressed!** to show up in the debug console every time the button is pressed.

Let's add the code that will cause an alert to show on screen:

```swift
@IBAction func userDidTouch(sender: AnyObject) {
    print("Button Pressed!")
    let alertController = UIAlertController(title: "Button pressed", message: "You pressed a button", preferredStyle: .Alert)
    let cancelAction = UIAlertAction(title: "Great", style: .Cancel, handler: nil)
    alertController.addAction(cancelAction)
    present(alertController, animated: true, completion: nil)
}
```
alertController.addAction(cancelAction)
presentViewController(alertController, animated: true,
completion: nil)
}

Pressing the button on the second view will now show a popup:

![Popup example]

**Different elements in different Size Classes**

By using the master Size Class of `wAny/hAny` and Auto Layout, we've now created a two-screen app with some functionality that supports every possible type of screen size with very little work. But let's say that we want change how our buttons look on iPads only, how would we do that?

The Size Class selector at the bottom hides some powerful functionality: When a different size class is selected, you can add Size Class customizations for the selected size class.

Open `Main.storyboard` and select our first view. Then, choose the `wRegular/hRegular` size class using the Size Class selector.
The bottom bar will turn blue, indicating that you are only editing properties for this one set of Size Classes. Drag a new button onto the first view from the object library.

If you once again select the \textit{wAny/hAny} Size Class, you'll see that the button has disappeared.

That newly added button is still there; you can verify that by looking in the Document Overview sidebar. Any outlets or actions created will also persist. This is simply a way of customizing your UI to be optimized for a device of any size. It's useful if you'd like to add extra functionality to larger screens and pare down functionality for smaller screens.

\section*{Layout previews}

You can preview how your interface will look from within IB. It is no longer necessary to change simulators and run your app. Open up the \textbf{Assistant editor} by selecting it in the Editors section of the top toolbar. Inside the newly opened Assistant editor, press the text that says \textbf{Automatic} and select \textbf{Preview} instead:

Pressing the + icon in the bottom-left corner and selecting a device will add a new device preview to the canvas. Each canvas can be rotated using the \textit{rotation} button.
Managing connections
You can quickly see how an object is connected to its controller by using the Connections Inspector in the Utility sidebar. Select the Show Alert button in the second view and open up the inspector (Opt + Cmd + 6).

This shows that there is one action attached to the Touch Up Inside Event

You can add and remove connections directly from this inspector.

Adding gesture recognizers
Instead of only reacting to actions on specific objects, you can also add gesture recognizers to react to a wide array of user gestures.
Drag a rotation gesture recognizer from the **object library** onto our first view controller (make sure to not drop it on the label or button). It will be added to the menu at the top of the view on the canvas:

The Rotation Gesture Recognizer icon now appears in the top toolbar of our view.

Open the Assistant editor to the ViewController class and Ctrl + drag a connection from the gesture recognizer to the code. Add an **Action type** connection and name it `userRotationGesture`.

If the preview Assistant editor is still selected, choose **Automatic** in the jump bar to add the connection.

Inside the newly created Action, add the following code:

```swift
@IBAction func userRotationGesture(sender: AnyObject) {
    guard let gesture = sender as? UIRotationGestureRecognizer else {
        return
    }
    gesture.view?.transform = CGAffineTransformRotate((gesture.view?.transform)!, gesture.rotation)
    gesture.rotation = 0
}
```

Now, using the two finger rotation gesture in the application will cause the entire view to rotate.
Rotation can be tested in the simulator by holding the Opt key and dragging with the mouse. You will notice two dots appear on screen; this indicates rotation input.

Our view in the simulator being rotated to the right
There are several other gesture recognizers available for use:

- **Tap Gesture Recognizer**: This recognizes simple taps within the view
- **Pinch Gesture Recognizer**: This recognized a pinch gesture (think pinch to zoom)
- **Swipe Gesture Recognizer**: This recognizes a swipe action, can be horizontal or vertical
- **Screen Edge Pan Gesture Recognizer**: This recognizes a swipe that starts at the edge of a screen
- **Long Press Gesture Recognizer**: This recognizes a long press and hold on the view

Gesture recognizers are powerful, advanced tools that allow you to create all sorts of custom interactions in your application.

**Summary**

In this chapter, you learned what Interface Builder is and how to use it to create the user interfaces in your applications using Storyboards. You learned about the MVC design pattern and how Interface Builder helps you create the View portion of the MVC pattern. We also discussed how you can easily support many screen sizes by creating an adaptive user interfaces by using Size Classes and Auto Layout, and even add or remove UI elements for specific screen sizes. Finally, you learned how to connect your views to your controllers using IB's Ctrl + drag and drop connections functionality to add actions and outlets.

In our next chapter, you will learn about Table, Collection, and Stack Views. We'll discuss what they are and how we can use them to create complex user interfaces quickly and easily.
Where to buy this book
You can buy Xcode 7 Essentials (Second Edition) from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.