What you will learn from this book

- Understand the basic requirements for working with iOS user interfaces
- Get to know UI tools, frameworks, and built-in components
- Plot dynamic layout structures using Auto Layout
- Shape and implement adaptive user interfaces for different screen sizes
- Draw and animate your user interfaces using the CALayer and UIKit animations
- Intercept and handle user touches to create user interface interactivity
- Create and depict totally customized controls
- Design with iOS through core graphics

Who this book is written for

This easy-to-follow guide is perfect for beginner-level iOS developers who want to become proficient in user interface development. It would also be useful for experienced iOS developers who need a complete overview of this broad topic all in one place.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'UI Fundamentals'
- A synopsis of the book’s content
- More information on Learning iOS UI Development
About the Author

**Yari D'areglia** is a developer with 15 years of experience in software architecture and development. During the last 4 years, he built successful iOS and OS X applications for developers as well as the mainstream public.

Yari, together with Nicola Armellini, founded Jumpzero, a company based in Italy that released countless mobile, desktop, and web applications.

Currently, he is employed as a senior iOS engineer at Neato Robotics, a company based in Silicon Valley, where he works on the communication between robots and iOS devices.

Yari writes at [www.ThinkAndBuild.it](http://www.ThinkAndBuild.it), a blog focused on iOS development with a strong focus on user interface development.

You can reach him on Twitter at [@bitwaker](https://twitter.com/bitwaker).
Preface

Through this comprehensive one-stop guide, you’ll get to grips with the entire UIKit framework and creating modern user interfaces for your iOS devices using Swift. Starting with an overview of the iOS drawing system and available tools, you will learn how to use these technologies to create adaptable layouts and custom elements for your applications. You’ll then be introduced to other topics such as animation and code drawing with core graphics, which will give you all the knowledge you need to create astonishing user interfaces. By the time you reach the end of this book, you will have a solid foundation in iOS user interface development and have gained a valuable insight into the process of building firm and complex UIs.

What this book covers

Chapter 1, UI Fundamentals, starts by describing how interfaces are structured and drawn and then presents some really important elements, such as windows and views.

Chapter 2, UI Components Overview – UIKit, is an overview of the UIKit framework. It’s a guided tour through the main UIKit elements, from their usage inside an app to their customization.

Chapter 3, Interface Builder, XIB, and Storyboard, gives an overview of the Xcode tools used to set up and build UIs.

Chapter 4, Auto Layout, is the key to understanding how Auto Layout works. It describes in detail how to create dynamic layouts.

Chapter 5, Adaptive User Interfaces, discusses a very important topic: how to improve user experience and provide interfaces that can adapt to different orientations, screen sizes, and user preferences using the latest advancements introduced with iOS 8 and 9.
Preface

Chapter 6, *Layers and Core Animation*, focuses on CALayer in the context of core animation. It illustrates how to achieve animations in iOS using two different approaches.

Chapter 7, *UI Interactions – Touches and Gestures*, analyzes the main way users interact with UIs—through touch. It answers questions such as "how is this information passed from the screen to the views?" and "how can we build an engaging UI using gestures?"

Chapter 8, *How to Build Custom Controls*, explains how to build custom controls after learning how these controls work in the previous chapters.

Chapter 9, *Introduction to Core Graphics*, is a final quick overview of the core graphics (Quartz 2D) framework to show you how to perform custom drawings with iOS.
Creating a successful application is not only a matter of writing efficient and reliable code. User interfaces and experience are relevant topics that we have to seriously take into account during the development process. If you want to be able to create appealing and well-structured layouts, you need to become familiar with the main building blocks of user interfaces.

In this chapter, we will explore the basics of iOS UI development, which you will use and improve upon for the rest of the book and, hopefully, for the rest of your career.

We will start from the fundamental actors of a user interface, such as windows and views; then, we will connect the dots showing how all these components work together by creating a UI hierarchy and how the system interacts with all these elements.

Let's start our journey from the element at the top of the stack: the window.

**Exploring windows**

A window is an instance of the `UIWindow` class, and it is the topmost element of any application UI's hierarchy. It doesn't draw any visual object and can be considered as a blank container for the UI elements called views. An application must have at least one window that normally fills the entire screen.

One of the main roles of the window is to deliver touches to the underlying views. You'll read more about this topic in Chapter 7, *UI Interactions – Touches and Gestures*. For now, it suffices to say that a window is the first entry point for a touch event. The touch is then pushed down through the view hierarchy until it reaches the right view.
The contents of windows

The contents of your applications are mainly directed by *view controllers* and presented through views, which, in turn, are displayed inside a window. As you will learn in the next section, this sequence is automatically handled by iOS, and all the classes involved in the process are organized to interact seamlessly.

The easiest and most reliable way to send content to a window is by configuring its `rootViewController` property with a `UIViewController` instance. The view controller's view will automatically be set as the contents of the window and presented to the user.

This solution simplifies the window hierarchy, ensuring that contents are all children of the same root. Thanks to this solution, changing the contents of a window is just a matter of updating its root view controller.

While you'll learn more about view controllers and views in the next paragraphs, this image should clarify how all these objects cooperate to present their contents to the user:

```
WINDOW
    rootViewController

VIEW CONTROLLER
    view
```

The view controller is initialized and set as the root view controller of the window. Finally, the window presents the current root view controller's view.

**Configuring windows**

You rarely need to set up a window manually. In most cases, Xcode defines all the needed information for you. Let's take a look at the entire process to better understand what goes on under the hood.
When you create a new Xcode project using the wizard, a Storyboard is created for you. If you check the info.plist file, you’ll see that the main storyboard filebase name key reports the name of the Storyboard by default as Main:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application requires iPhone...</td>
<td>Boolean YES</td>
</tr>
<tr>
<td>Launch screen interface file base...</td>
<td>String LaunchScreen</td>
</tr>
<tr>
<td>Main storyboard file base name</td>
<td>String Main</td>
</tr>
<tr>
<td>Required device capabilities</td>
<td>Array (1 item)</td>
</tr>
<tr>
<td>Status bar style</td>
<td>String UIStatusBarStyleLightContent</td>
</tr>
</tbody>
</table>

This key is really important as it tells Xcode that you want to start the application from the Storyboard or, more precisely, from the Storyboard initial view controller (the one indicated by the grey arrow inside the Storyboard working area).

The @UIApplicationMain attribute in the AppDelegate.swift file is responsible for the launch of the entire application process. It marks an entry point for the application launch, reading the Storyboard's information from the info.plist file and instantiating the initial view controller.

At this point, a UIWindow instance is created and associated with the window property of the AppDelegate class. This property will be a handy reference to the main window for the entire life cycle of the application.

The initial view controller, previously initialized, is now assigned to the rootViewController property of the window; therefore, the initial view controller’s view becomes the current window’s content.

Since windows are invisible by default, there is one last step required to show the content to the user. After the application:didFinishLaunchingWithOptions function finishes its execution, makeKeyAndVisible is called for the window, which now loads its interface from rootViewController and finally displays it contents.

The following image summarizes all these steps:
The same result can be obtained programmatically. If you remove the `Main storyboard filebase name` key from the `info.plist` file, Xcode doesn’t have any information on how to set up a valid window. The `application:didFinishLaunchingWithOptions` function is the right place to manually instantiate it. You can execute the following:

```swift
func application(application: UIApplication,
     didFinishLaunchingWithOptions
      launchOptions: [NSObject: AnyObject]?) -> Bool {

    // Instantiate a window with the same size of the screen
    window = UIWindow(frame: UIScreen.mainScreen().bounds)
    // Instantiate a view controller with the Main storyboard
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    let viewController = storyboard.instantiateViewControllerWithIdentifier("viewController2") as! ViewController
    // Setup and present the window
    window?.rootViewController = viewController
    window?.makeKeyAndVisible()

    return true
}
```

As you can note, this code retraces the same steps that we saw previously. The only noteworthy thing is the way the window frame is defined: `UIScreen` is a class that represents a screen device. In the previous block of code, the `mainScreen` function is used to get the current device bounds and build a window that is the same size as the screen.

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Working with views

Along your path of UI enlightenment, you'll encounter different types of views. You can simplify the categorization by thinking about a view as the atomic element of the user interface, where a view has its own specific properties and can be grouped with other views to create complex hierarchies.

UIView is the base class used to instantiate generic views and it can be subclassed to create custom views. Almost all the UI elements that you will use in your applications are inherited from this class, and in the next chapter, you'll learn a lot about some of the elements provided by Apple within the UIKit framework.

Keep in mind that the UIWindow class too is a subclass of UIView. This is generally a cause of confusion considering that in a UI hierarchy, views are the children of a window and not the other way around.

A UIView instance has properties that you can use to change an aspect of the view. For example, the backgroundColor property accepts UIColor to define a tint for the view's background. The background of a view is transparent by default, and as you'll see later in this chapter, some views can seep through other views.

The alpha property is useful if your view and all the elements it contains need to be transparent. It accepts a value from 0.0 to 1.0, and depending on your settings, it gives the view a "see-through" effect. With the help of this property, you can hide or unhide a view using some interesting animations, as you will learn in Chapter 6, Layers and Core Animation.

Another way to change the visibility of a view is using the hidden property, which can be set to true or false.

It's important to note that when you hide a view using the hidden or alpha properties, you are not removing the view from the hierarchy and memory; the view can, in fact, still be reached, but the interaction with the user is automatically disabled until the view is visible again. This is a neat trick to temporarily remove a view from the screen, thus preventing any unintended interaction.

Sometimes, you might need to disable a view even if the drawing goes on (for example, you may want to be sure that the UI interactions are disabled while loading some external data). In this case, you may find the userInteractionEnabled property useful; setting this property to false tells the view to ignore user events.
Defining the view's geometry

The properties of some view's have only one role: defining how the view is drawn on screen. Before diving into a description of these properties, it's worth introducing the structures that are involved in the definition of the view geometry:

- **CGPoint**: This is defined by two properties, \( x \) and \( y \), and it represents a single point
- **CGSize**: This is defined by two properties, \( \text{width} \) and \( \text{height} \), and it represents a generic size
- **CGRect**: This is defined by two properties, \( \text{origin} \) (CGPoint) and \( \text{size} \) (CGSize), and it represents a quadrilateral area

When working with iOS, you will encounter different coordinate systems. With UIViews, the origin of the coordinate system \((x:0, y:0)\) is on the upper-left side. The \( x \) value increases from left to right, and the \( y \) value increases from top to bottom.

The code needed to initialize these structures is straightforward, and there are a lot of functionalities provided by Apple that simplify our work with these structures.

The following code shows how to create the Rect, Size, and Point instances:

```swift
// Define a point
let point = CGPoint(x: 20, y: 10)

// Define a size
let size = CGSize(width: 20, height: 10)

// Define a rect using size and point
let rect_A = CGRect(origin: point, size: size)

// Define a rect using x, y, width and height data
let rect_B = CGRect(x: 15, y: 10, width: 100, height: 30)
```

The properties that define the view geometry are frame, bounds, and center and they are configured using the geometry structures you just saw.
The bounds property

Let's start with the bounds property. It is a CGRect property that defines information (size and origin) locally to the view. This means that when we speak about bounds, we do not determine how the view is drawn in the UI hierarchy but how it's drawn in a decontextualized, local space. You'll read more about UI hierarchy later in this chapter; for now, just think about it as a structure that organizes trees of views, where a view can be either the child or parent of another view. That said, in most cases, the origin bound is (x:0, y:0), while the size property is the size of the view.

The frame property

On the other hand, the frame property defines how a view is placed inside the hierarchy. It's a CGRect property similar to bounds, but its origin value determines how the view is placed inside its parent view.

At this point, we might conclude that frame and bounds are similar; however, this is not always true. Take a look at the following images to better understand how they work together to define the view geometry:

![Frame and Bounds Example](image)

In this case, frame and bounds have the same size, but the position values are different. Take a look at this image:

![Frame and Bounds Example](image)

This indicates that after a rotation, the same view maintains the bounds, while the frame changes significantly, adapting its size in order to contain the rotated view.
The center property
The center property works in relation to the UI hierarchy (as does the frame), and it's a handy way to define the view's position inside its parent. It differs from frame.origin in the way its position is calculated. With frame, the origin position is calculated using the upper-left corner of the view, while for the center, the anchor is the view's center, as you can see from the following image:

UI hierarchy and views inheritance
In this chapter, we already talked about UI hierarchies. Let's dive into the topic by introducing the concepts of subview and superview and learning how to build and manage complex hierarchies.

Every instance of UIView (or its subclass) can be connected with other views in a parent-child relationship. The parent view is called superview, while the children are called subviews. A view can only have one superview, but it can contain more than one subview:
Thanks to the dedicated UIView properties and functions, you can easily inspect
a view hierarchy and navigate from a root view down through to the last view of
the hierarchy.

A view can access its parent from the `superview` property, as follows:

```swift
let parentView = view.superview
```

The property returns a single UIView reference or `nil` if the view is not added to the
hierarchy yet.

In a similar way, the `subviews` property returns an array of UIView instances that are
the children of the view. Take a look at this code:

```swift
let children = view.subviews
```

**Managing with the hierarchy**

The UIView class offers helpful functions to add, move, and delete elements from the
hierarchy; you can call these functions directly from the view instances.

The `addSubview:` function pushes a view as the child of the caller, as follows:

```swift
containerView.addSubview(childView)
```

Views that are added as subviews of the same view are sibling. Sibling subviews are
assigned to an index based on the order of insertion, which in turn corresponds to the
drawing order—the highest index is drawn at the front, while the lowest is drawn
behind the other sibling views. The `addSubview:` function assigns the first free index
to the view, determining its position, which is in front of all the other views with
lower indexes.
This order can be manipulated using functions that specify the desired index in an absolute or relative way:

```swift
containerView.insertSubview(childView, atIndex: 2)
```

With the `insertSubview:atIndex` function, you can specify an index in which to place the view. If the index is not free, the view at this index gets shifted up by one index, consequently shifting all the other sibling views with indexes greater than the requested one.

With the same logic, a view can be placed into the hierarchy using another view as a reference and specifying that the new view should be inserted above (insertSubview:aboveSubview) or below (insertSubview:belowSubview) the referenced view, as follows:

```swift
containerView.insertSubview(childView, aboveSubview: anotherView)
containerView.insertSubview(childView, belowSubview: anotherView)
```

Removing a view from the hierarchy is extremely simple. If you have a reference of the view that you want to remove, just call the `removeFromSuperview` function for this view. Alternatively, if you want to remove all the subviews from the parent view, just loop through the subviews and remove the views one by one. You can use the following code for this:

```swift
for subview in containerView.subviews {
    subview.removeFromSuperview()
}
```

When you remove a view from the hierarchy, you are removing its subviews as well. If you haven't defined any reference to the views, they are no longer retained in memory; therefore, all chances to get access to the views are lost.

When you need to access a subview on the fly without saving a reference to it, you can use the `tag` property. A `UIView` instance can be easily marked using an integer, and you can obtain the view that corresponds to the same tag using the `viewWithTag` function. This function returns the view itself or the first subview whose tag coincides with the requested tag.
View and subview visibility

A parent view defines its subviews' visibility outside its boundaries through the Boolean property `clipToBounds`. If this property is `true`, the view behaves as a mask for its subviews, preventing them from being drawn outside the boundaries. The following image presents the result of different `clipToBounds` values on the same hierarchy:

![Image showing the effect of `clipToBounds`]

Another important note about subview visibility is related to the `alpha` property, which, as we mentioned previously, defines the opacity of the view. Subviews indirectly inherit the alpha value from their superviews, so the resulting opacity of the subview depends both on their parent and their own alpha.

If your view has a nontransparent background, it is good practice to leave the `opaque` property set to `true` as a hint for the drawing system. Opaque views are optimized for this. The opposite is suggested if the view is fully or partially transparent; set this property to `false`.

Hierarchy events

Changes on the hierarchy can be intercepted and managed through callbacks called on the parent and child views.
When a view is attached to a superview, the `didMoveToSuperview` function is called. If we want to perform a task after this event is triggered, we have to override this function with a `UIView` subclass by executing the following code:

```swift
class CustomView: UIView {

    override func didMoveToSuperview() {
        println("I have a superview!"
    }
}
```

In the same way, the event can be intercepted by the superview overriding the `didAddSubview:subview` function, as follows:

```swift
override func didAddSubview(subview: UIView) {
    println("The subview \(subview) has been added")
}
```

The view will be added to a hierarchy that has a window as root. At this point, the `didMoveToWindow` event is called. After the event is triggered, the `window` property of the view instance becomes a reference to the root window; this turns out to be a handy way to check whether a view has reached the screen.

Remember that the only way to show a view on screen is through a window; so, if the `window` property is `nil`, you can be sure that the view is not added to a window hierarchy yet and that, for this reason, it won’t be visible. Take a look at this code:

```swift
override func didMoveToWindow() {
    println("I’ve been attached to this window hierarchy: \(window")
}
```

Another important method that responds to hierarchy changes is `layoutSubviews`. This function is called as soon as a subview is added to or removed from the hierarchy and every time the bounds of the view change. Within this function, `Auto Layout constraints` (more on this will be discussed in `Chapter 5, Adaptive User Interfaces`) are read, and they act to organize the subview’s layout. You can override this function to perform your customizations to the layout, adding or tweaking the current constraints.
Notes about debug

When views are complex, the hierarchy is hardly straightforward. Xcode helps you, though, with an interesting debug instrument. When your application is running, select Debug -> View Debugging -> Capture View Hierarchy from the Xcode menu, and you'll see an interactive 3D representation of the current view hierarchy showing the depth and the position of all the views contained:

Hands-on code

In this section, you will practice some of the concepts that we just discussed by writing a real application.

Open the Chapter 1 folder and launch the ...\Start\Chapter1 project. You'll find a simple application structure that is the starting point for this exercise; the final result is the ...\Completed\Chapter1 project.
The base structure is really simple: an uppermost area with some controls (two buttons and one segmented control) and a view (with a light gray background) that we call a container.

Your goal is to implement the `createView` function that receives a location (`CGPoint`) and adds a subview at this location inside the container. Depending on the value of the segmented control, you should use the received location as the center (red views) or upper-left corner of the new view (blue views). You need to also implement the `clear` function to remove all the added subviews. The project implements a tap gesture on the container, which invokes `createView` linked to the touch location.

Let's implement the `createView` function for the `viewController.swift` file first by executing the following:

```swift
func createView(location: CGPoint) {
    let viewSize = CGSize(width: 50, height: 50)
    let viewFrame = CGRect(origin: location, size: viewSize)
    // Add subview at viewFrame
}
```
let childView = UIView(frame: viewFrame)
childView.backgroundColor = UIColor.redColor()
viewContainer.addSubview(childView)

if isCenterAligned {
    childView.center = location
    childView.backgroundColor = UIColor.blueColor()
}
}

The `childView` function is instantiated using a fixed size and the received location as its origin. Then, it is simply attached to `viewContainer` using `addSubview`. If the `isCenterAligned` property (which is handled by the segmented control) is true, the `center` property of the view is moved to the received location.

The implementation of the `clear` function is straightforward, as you can note in the following code:

```swift
@IBAction func clear(){
    for subview in viewContainer.subviews{
        subview.removeFromSuperview()
    }
}
```

It just performs a loop through the `viewContainer` subviews to call the `removeFromSuperview` function on these subviews.

A second functionality can be added to this exercise. Push the "Deep" button on the upper-left side of the screen, and debug the current hierarchy using the "capture view hierarchy" function to see this great feature in action!

**View drawing and life cycle**

iOS uses remarkable optimizations in the process of drawing contents on screen. Views are not continuously drawn; the system draws any view just once and creates snapshots for each displayed element. The current snapshot is shown until a view doesn't require an update. The view is then redrawn, and a new snapshot for the updated view is taken. This is a clever way to avoid a wastage of resources. In devices such as smartphones, optimization is mandatory.

The `UIView` content can be invalidated by calling the `setNeedsDisplay:` or `setNeedsDisplayInRect:` function.
UI Fundamentals

This call basically tells the drawing system that the view content needs to be updated with a new version. Later, during the next run loop, the system asks the view to redraw. The main difference between the `setNeedsDisplay:` and `setNeedsDisplayInRect:` functions is that the latter performs an optimization using only a portion of the new view content.

In most cases, the redrawing process is managed for you by UIKit. If you need to create your really custom `UIView` subclass, though, you probably want to draw the contents of the view yourself. In this case, the `drawRect:` function is the place where you will add the drawing functions. You'll learn more about custom drawing in Chapters 6, Layers and Core Animation, and Chapter 9, Introduction to Core Graphics. For now, it suffices to say that you should never call this function directly! When the content of the view needs to be updated, just call `setNeedsDisplay:` and the `drawRect:` function will be executed by the system following the right procedure.

Here is a quick example of custom drawing with which we can create a `UIView` subclass that draws a simple red ellipse at the center of the view:

```swift
import UIKit

class ellipseView: UIView {

    override func drawRect(rect: CGRect) {

        let path = UIBezierPath(ovalInRect: self.bounds)
        UIColor.redColor().setStroke()
        UIColor.orangeColor().setFill()
        path.fill()
        path.stroke()
    }
}
```

This function uses `UIBezierPath` to define an oval shape that has a stroke and fill of red and orange. This shape is finally drawn in the current `Graphic Context`, which can be seen as an empty dashboard where you can design using code. You'll learn more about Graphic Contexts in Chapter 9, Introduction to Core Graphics.

**View controllers and views**

The `UIViewController` property `view` is the root view of the hierarchy that defines the view controller's contents. As we already saw, a view controller is associated with the window's `rootViewController` property, and a connection is established between the window property of the view controller `view` and the window.
During its life cycle, a view controller deals with important events related to its view. Depending on your needs, you might find these events useful for the UI definition.

Here is a quick but complete description of the functions called in relation to these events:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadView</td>
<td>This is called when the view is created and assigned to the view property. Depending on the setup of the view controller, a view can be created using a Storyboard or a .xib file. You will not override this method unless you decide not to implement the view using Interface Builder.</td>
</tr>
<tr>
<td>viewDidLoad</td>
<td>This is called after the view is loaded. You will override this to perform additional adjustments to the user interface. You can set the text value of labels with the data received from a previous controller in the navigation hierarchy, for instance.</td>
</tr>
<tr>
<td>viewWillAppear</td>
<td>This is called just before the view is added to a view hierarchy. It is obviously called after viewDidLoad, and it may be called more than once during the view controller's life cycle.</td>
</tr>
<tr>
<td>viewDidAppear</td>
<td>This is called after the view is added to a view hierarchy. When this method is called, the view is visible, and its window property is no longer nil.</td>
</tr>
<tr>
<td>viewWillAppear</td>
<td>This method is called when the view bounds change or are initialized and the view is about to lay out the subviews.</td>
</tr>
<tr>
<td>viewDidLayoutSubviews</td>
<td>When this method is called, the bounds are set and the view has just called the layoutSubviews method.</td>
</tr>
</tbody>
</table>

The appear-functions are called when the controller is presented for the first time when a back button is pressed or a modal is closed, showing the underlying view controller again. If you override these methods, you are required to call them on super before executing your code.
Summary

With this chapter, you learned some of the very important basics that allow you to understand how UI elements are created, presented, and managed by the system. All these notions are essential to build simple and complex user interfaces, and they define a helpful starting point to easily read and appreciate the rest of this book.

The next chapter introduces the UIKit framework, an important set of classes that you will use in your everyday work. UIKit defines all the prebuilt UI elements of iOS and other classes that are fundamental to creating a complete and interactive layout.
Where to buy this book

You can buy Learning iOS UI Development 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.

Get more information Learning iOS UI Development