Chapter 1. Views

A *view* (an object whose class is UIView or a subclass of UIView) knows how to draw itself into a rectangular area of the interface. Your app has a visible interface thanks to views; everything the user sees is ultimately because of a view. Creating and configuring a view can be extremely simple: "Set it and forget it." For example, you can configure a UIButton in the nib editor; when the app runs, the button appears, and works properly. But you can also manipulate views in powerful ways, in real time. Your code can do some or all of the view's drawing of itself (Chapter 2); it can make the view appear and disappear, move, resize itself, and display many other physical changes, possibly with animation (Chapter 4).

A view is also a responder (UIView is a subclass of UIResponder). This means that a view is subject to user interactions, such as taps and swipes. Views are the basis not only of the interface that the user sees, but also of the interface that the user touches (Chapter 5). Organizing your views so that the correct view reacts to a given touch allows you to allocate your code neatly and efficiently.

The *view hierarchy* is the chief mode of view organization. A view can have subviews; a subview has exactly one immediate superview. Thus there is a tree of views. This hierarchy allows views to come and go together. If a view is removed from the interface, its subviews are removed; if a view is hidden (made invisible), its subviews are hidden; if a view is moved, its subviews move with it; and other changes in a view are likewise shared with its subviews. The view hierarchy is also the basis of, though it is not identical to, the responder chain.

A view may come from a nib, or you can create it in code. On balance, neither approach is to be preferred over the other; it depends on your needs and inclinations and on the overall architecture of your app.

Window and Root View

The top of the view hierarchy is a window. It is an instance of UIWindow (or your own subclass thereof), which is a UIView subclass.

At launch time, a window is created and displayed; otherwise, the screen would be black. New in iOS 13, your app might support multiple windows on an iPad (Chapter 9); if it doesn't, or if we're running on an iPhone, your app will have exactly one window (the *main* window). A visible window forms the background to, and is the ultimate superview of, all your other visible views. Conversely, all visible views are visible by virtue of being subviews, at some depth, of a visible window.

In Cocoa programming, you do not manually or directly populate a window with subviews. Rather, the link between your window and the interface that it contains is the window's *root view controller*. A view controller is instantiated, and that instance is assigned to the window's `rootViewController` property. That view controller's *main view* — its `view` — henceforth occupies the entirety of the window. It is the window's sole subview; all other visible views are subviews (at some depth) of the root view controller's view. (The root view controller itself will be the top of the *view controller hierarchy,* of which I'll have much more to say in Chapter 6.)

How an App Launches

How does your app, at launch time, come to have its window in the first place, and how does that window come to be populated and displayed? If your app uses a main storyboard, it all happens automatically. But "automatically" does *not* mean "by magic!" The procedure at launch is straightforward and deterministic, and your code can take a hand in it. It is useful to know how an app launches, not least because, if you misconfigure something and app launch goes wrong, you'll be able to figure out why.

Your app consists, ultimately, of a single call to the `UIApplicationMain` function. (Unlike an Objective-C project, a typical Swift project doesn't make this call explicitly, in code; it is called for you, behind the scenes.) This call creates some of your app's most important initial instances; if your app uses a main storyboard, those instances include the window and its root view controller.

Exactly how `UIApplicationMain` proceeds depends on what it discovers as it gets going. New in iOS 13, your app can use scenes and scene-related classes and protocols (UISceneSession, UIScene, UIWindowScene,

UIWindowSceneDelegate). The runtime knows whether your app does this because of the presence of the "Application Scene Manifest" dictionary in the *Info.plist.* By default, new app projects that you create from Xcode's built-in app templates have this dictionary and use scenes (even if they do not support multiple windows on iPad). But you might also have older projects, or you might want your app to be backward compatible to iOS 12 and before. So I'll show two different launch trajectories — what happens in iOS 12 and before, and what happen in iOS 13 in an app that supports window scenes.

IOS 12 AND BEFORE

Here's how `UIApplicationMain` bootstraps your app as it launches on iOS 12 and before:

1. `UIApplicationMain` instantiates UIApplication and retains this instance, to serve as the shared application instance, which your code can later refer to as `UIApplication.shared`. It then instantiates the app delegate class; it knows which class that is because it is marked `@UIApplicationMain`. It retains the app delegate instance, ensuring that it will persist for the lifetime of the app, and assigns it as the application instance's `delegate`.
2. `UIApplicationMain` looks to see whether your app uses a main storyboard; it knows whether you are using a main storyboard, and what its name is, by looking at the *Info.plist* key "Main storyboard file base name" (`UIMainStoryboardFile`). You can easily edit this key, if necessary, by editing the target and, in the General pane, changing the Main Interface value in the Deployment Info section. By default, a new iOS project has a main storyboard called *Main.storyboard*, and the Main Interface value is `Main`.
3. If your app uses a main storyboard, `UIApplicationMain` instantiates that storyboard's initial view controller. (I'll talk more about that in Chapter 6.)
4. If your app uses a main storyboard, `UIApplicationMain` instantiates UIWindow and assigns the window instance to the app delegate's `window` property, which retains it, ensuring that the window will persist for the lifetime of the app. Alternatively, your app delegate can substitute an instance of a custom UIWindow subclass.

5. If your app uses a main storyboard, `UIApplicationMain` assigns the initial view controller instance to the window's `rootViewController` property, which retains it. The view controller's view becomes the window's sole subview.
6. `UIApplicationMain` calls the app delegate's `application(_:didFinishLaunchingWithOptions:)`.
7. Your app's interface is not visible until the window, which contains it, is made the app's key window. Therefore, if your app uses a main storyboard, `UIApplicationMain` calls the window's instance method `makeKeyAndVisible`.

iOS 13 with Window Scene Support

Here's how `UIApplicationMain` bootstraps your app with window scene support as it launches on iOS 13:

1. `UIApplicationMain` instantiates UIApplication and retains this instance, to serve as the shared application instance, which your code can later refer to as `UIApplication.shared`. It then instantiates the app delegate class; it knows which class that is because it is marked `@UIApplicationMain`. It retains the app delegate instance, ensuring that it will persist for the lifetime of the app, and assigns it as the application instance's `delegate`.
2. `UIApplicationMain` calls the app delegate's `application(_:didFinishLaunchingWithOptions:)`.
3. `UIApplicationMain` creates a UISceneSession, a UIWindowScene, and an instance that will serve as the window scene's delegate. The *Info.plist* specifies, as a string, what the class of the window scene delegate instance should be ("Delegate Class Name" inside the "Application Scene Manifest" dictionary's "Scene Configuration"). In the built-in app templates, it is the SceneDelegate class. This is written in the *Info.plist* as `$(PRODUCT_MODULE_NAME).SceneDelegate` to take account of Swift "name mangling."
4. `UIApplicationMain` looks to see whether your initial scene uses a storyboard. The *Info.plist* specifies, as a string, the name of its storyboard ("Storyboard Name" inside the "Application Scene Manifest" dictionary's "Scene Configuration"). If so, it instantiates that storyboard's initial view controller.

5. If the scene uses a storyboard, `UIApplicationMain` instantiates UIWindow and assigns the window instance to the scene delegate's `window` property, which retains it.
6. If the scene uses a storyboard, `UIApplicationMain` assigns the initial view controller instance to the window instance's `rootViewController` property, which retains it. The view controller's view becomes the window's sole subview.
7. `UIApplicationMain` causes your app's interface to appear, by calling the UIWindow instance method `makeKeyAndVisible`.
8. The scene delegate's `scene(_:willConnectTo:options:)` is called.

The most important differences between this procedure and the iOS 12 procedure are:

- The call to `application(_:didFinishLaunchingWithOptions:)` is much earlier in the sequence. This won't matter if you confine your use of this method to initializations that affect the app as a whole. If you need to know that the launch process is over and your window is visible, implement `scene(_:willConnectTo:options:)`.
- The `window` property belongs to the scene delegate, not the app delegate.

TIP

In iOS 13, `UIApplicationMain` *always* creates a UISceneSession and a UIWindowScene even if your app does not declare support for window scenes (in fact, even if your app is linked against iOS 12 or earlier). They are present as part of the architecture even if your app never needs to refer to them.

App Without a Storyboard

It is possible to write an app that lacks a main storyboard:

- In iOS 12 and before, this means that the *Info.plist* contains no "Main storyboard file base name" entry.
- In iOS 13 with scene support, it means that there is no "Storyboard Name" entry under "Application Scene Configuration" in the "Application Scene Manifest" dictionary.

Such an app simply does in code everything that `UIApplicationMain` does automatically if the app has a main storyboard. In iOS 12 and before, you would do that in the app delegate's `application(_:didFinishLaunchingWithOptions:)`. In iOS 13, with a window scene, you do it in the scene delegate's `scene(_:willConnectTo:options:)`:

```
func scene(_ scene: UIScene,
    willConnectTo session: UISceneSession,
    options connectionOptions: UIScene.ConnectionOptions) {
        if let windowScene = scene as? UIWindowScene {
            self.window = UIWindow(windowScene: windowScene)
            let vc = // ...
            self.window!.rootViewController = vc
            self.window!.makeKeyAndVisible()
        }
}
```

Instantiate UIWindow and assign it as the scene delegate's `window` property. It is crucial to make the connection between the window scene and the window by calling `init(windowScene:)`.

Instantiate a view controller and configure it as needed.

Assign the view controller as the window's `rootViewController` property.

Call `makeKeyAndVisible` on the window, to show it.

(That's how a SwiftUI app works. SwiftUI doesn't use storyboards; step 2 creates a UIHostingController that hosts the app's initial View, and after that the SwiftUI code takes over.)

A variant that is sometimes useful is an app that has a storyboard but doesn't let `UIApplicationMain` see it at launch. That way, we can dictate at launch time which view controller from within that storyboard should be the window's root view controller. A typical scenario is that our app has something like a login or registration screen that appears at launch if the user has not logged in, but doesn't appear on subsequent launches once the user *has* logged in:

```
func scene(_ scene: UIScene,
    willConnectTo session: UISceneSession,
    options connectionOptions: UIScene.ConnectionOptions) {
        if let windowScene = scene as? UIWindowScene {
            self.window = UIWindow(windowScene: windowScene)
            let userHasLoggedIn : Bool = // ...
            let vc = UIStoryboard(name: "Main", bundle: nil)
                .instantiateViewController(identifier:
userHasLoggedIn ?
                    "UserHasLoggedIn" : "LoginScreen")
            self.window!.rootViewController = vc
            self.window!.makeKeyAndVisible()
        }
}
```

Referring to the Window

Once the app is running, there are various ways for your code to refer to the window:

From a view

If a UIView is in the interface, it automatically has a reference to the window that contains it, through its own `window` property. Your code will probably be running in a view controller with a main view, so `self.view.window` is usually the best way to refer to the window.

You can also use a UIView's `window` property as a way of asking whether it is ultimately embedded in the window; if it isn't, its `window` property is `nil`. A UIView whose `window` property is `nil` cannot be visible to the user.

From the scene delegate

The scene delegate instance maintains a reference to the window through its `window` property.

From the application

The shared application maintains a reference to the window through its `windows` property:

```
let w = UIApplication.shared.windows.first!
```

WARNING

Do not expect that the window you know about is the app's only window. The runtime can create additional mysterious windows, such as the UITextEffectsWindow and the UIRemoteKeyboardWindow.

Experimenting with Views

In the course of this and subsequent chapters, you may want to experiment with views in a project of your own. If you start your project with the Single View App template, it gives you the simplest possible app — a main storyboard containing one scene consisting of one view controller instance along with its main view. As I described in the preceding section, when the app runs, that view controller will become the window's `rootViewController`, and its main view will become the

window's root view. If you can get *your* views to become subviews of that view controller's main view, they will be present in the app's interface when it launches.

In the nib editor, you can drag a view from the Library into the main view as a subview, and it will be instantiated in the interface when the app runs. However, my initial examples will all create views and add them to the interface *in code*. So where should that code go? The simplest place is the view controller's `viewDidLoad` method, which is provided as a stub by the project template code; it runs once, before the view appears in the interface for the first time.

The `viewDidLoad` method can refer to the view controller's main view by saying `self.view`. In my code examples, whenever I say `self.view`, you can assume we're in a view controller and that `self.view` is this view controller's main view:

```
override func viewDidLoad() {

    super.viewDidLoad() // this is template code

    let v = UIView(frame:CGRect(x:100, y:100, width:50,
height:50))

    v.backgroundColor = .red // small red square

    self.view.addSubview(v) // add it to main view

}
```

Try it! Make a new project from the Single View App template, and make the ViewController class's `viewDidLoad` look like that. Run the app. You will actually *see* the small red square in the running app's interface.

Subview and Superview

Once upon a time, and not so very long ago, a view owned precisely its own rectangular area. No part of any view that was not a subview of this view could appear inside it, because when this view redrew its rectangle,

it would erase the overlapping portion of the other view. No part of any subview of this view could appear outside it, because the view took responsibility for its own rectangle and no more.

Those rules, however, were gradually relaxed, and starting in OS X 10.5, Apple introduced an entirely new architecture for view drawing that lifted those restrictions completely. iOS view drawing is based on this revised architecture. In iOS, some or all of a subview can appear outside its superview, and a view can overlap another view and can be drawn partially or totally in front of it without being its subview.

Figure 1-1 shows three overlapping views. All three views have a background color, so each is completely represented by a colored rectangle. You have no way of knowing, from this visual representation, exactly how the views are related within the view hierarchy. In actual fact, View 1 is a sibling view of View 2 (they are both direct subviews of the root view), and View 3 is a subview of View 2.

Figure 1-1. Overlapping views

When views are created in the nib, you can examine the view hierarchy in the nib editor's document outline to learn their actual relationship (Figure 1-2). When views are created in code, you know their hierarchical relationship because you created that hierarchy. But the visible interface doesn't tell you, because view overlapping is so flexible.

Figure 1-2. A view hierarchy as displayed in the nib editor

Nevertheless, a view's position within the view hierarchy is extremely significant. For one thing, the view hierarchy dictates the *order* in which views are drawn. Sibling subviews of the same superview have a definite order; an earlier sibling is drawn before a later sibling, so if they overlap, the earlier one will appear to be behind the later one. Similarly, a superview is drawn before its subviews, so if the subviews overlap their superview, the superview will appear to be behind them.

You can see this illustrated in Figure 1-1. View 3 is a subview of View 2 and is drawn on top of it. View 1 is a sibling of View 2, but it is a later sibling, so it is drawn on top of View 2 and on top of View 3. View 1 *cannot* appear behind View 3 but in front of View 2, because Views 2 and 3 are subview and superview and are drawn together — both are drawn either before or after View 1, depending on the ordering of the siblings.

This layering order can be governed in the nib editor by arranging the views in the document outline. (If you click in the canvas, you may be able to use the menu items of the Editor → Arrange menu instead — Send to Front, Send to Back, Send Forward, Send Backward.) In code, there are methods for arranging the sibling order of views, which we'll come to in a moment.

Here are some other effects of the view hierarchy:

- If a view is removed from or moved within its superview, its subviews go with it.
- A view's degree of transparency is inherited by its subviews.
- A view can optionally limit the drawing of its subviews so that any parts of them outside the view are not shown. This is called *clipping* and is set with the view's `clipsToBounds` property.
- A superview *owns* its subviews, in the memory-management sense, much as an array owns its elements; it retains its subviews, and is responsible for releasing a subview when that subview is removed from the collection of this view's subviews, or when the superview itself goes out of existence.
- If a view's size is changed, its subviews can be resized automatically (and I'll have much more to say about that later in this chapter).

A UIView has a `superview` property (a UIView) and a `subviews` property (an array of UIView objects, in back-to-front order), allowing you to trace the view hierarchy in code. There is also a method `isDescendant(of:)` letting you check whether one view is a subview of another at any depth.

If you need a reference to a particular view, you will probably arrange it beforehand as a property, perhaps through an outlet. Alternatively, a

view can have a numeric tag (its `tag` property), and can then be referred to by sending any view higher up the view hierarchy the `viewWithTag(_:)` message. Seeing that all tags of interest are unique within their region of the hierarchy is up to you.

Manipulating the view hierarchy in code is easy. This is part of what gives iOS apps their dynamic quality. It is perfectly reasonable for your code to rip an entire hierarchy of views out of the superview and substitute another, right before the user's very eyes! You can do this directly; you can combine it with animation (Chapter 4); you can govern it through view controllers (Chapter 6).

The method `addSubview(_:)` makes one view a subview of another; `removeFromSuperview` takes a subview out of its superview's view hierarchy. In both cases, if the superview is part of the visible interface, the subview will appear or disappear respectively at that moment; and of course the subview may have subviews of its own that accompany it. Removing a subview from its superview releases it; if you intend to reuse that subview later on, you will need to retain it first by assigning it to a variable.

Events inform a view of these dynamic changes. To respond to these events requires subclassing. Then you'll be able to override any of these methods:

- `willRemoveSubview(_:)`, `didAddSubview(_:)`
- `willMove(toSuperview:)`, `didMoveToSuperview`
- `willMove(toWindow:)`, `didMoveToWindow`

When `addSubview(_:)` is called, the view is placed last among its superview's subviews, so it is drawn last, meaning that it appears frontmost. That might not be what you want. A view's subviews are indexed, starting at `0` which is rearmost, and there are methods for inserting a subview at a given index or below (behind) or above (in front of) a specific view; for swapping two sibling views by index; and for moving a subview all the way to the front or back among its siblings:

- `insertSubview(_:at:)`
- `insertSubview(_:belowSubview:)`, `insertSubview(_:aboveSubview:)`
- `exchangeSubview(at:withSubviewAt:)`
- `bringSubviewToFront(_:)`, `sendSubviewToBack(_:)`

Oddly, there is no command for removing all of a view's subviews at once. However, a view's `subviews` array is an immutable copy of the internal list of subviews, so it is legal to cycle through it and remove each subview one at a time:

```
myView.subviews.forEach {$0.removeFromSuperview()}
```

Color

A view can be assigned a background color through its `backgroundColor` property. A view distinguished by nothing but its background color is a colored rectangle, and is an excellent medium for experimentation, as in Figure 1-1.

A view whose background color is `nil` (the default) has a transparent background. If such a view does no additional drawing of its own, it will be invisible! Such a view is perfectly reasonable, however; a view with a transparent background might act as a convenient superview to other views, making them behave together.

A color is a UIColor, which will typically be specified using `.red`, `.blue`, `.green`, and `.alpha` components, which are CGFloat values between 0 and 1:

```
v.backgroundColor = UIColor(red: 0, green: 0.1, blue: 0.1,
alpha: 1)
```

There are also numerous named colors, vended as static properties of the UIColor class:

```
v.backgroundColor = .red
```

New in iOS 13, however, you may need to be rather more circumspect about the colors you assign to things. The problem is that the user can switch the device between light and dark modes. This can cause a cascade of color changes that can make hard-coded colors look bad. Suppose (in a new project created in Xcode 11) we give the view controller's main view a subview with a dark color:

```
override func viewDidLoad() {

    super.viewDidLoad()

    let v = UIView(frame:CGRect(x:100, y:100, width:50,
height:50))

    v.backgroundColor = UIColor(red: 0, green: 0.1, blue:
0.1, alpha: 1)

    self.view.addSubview(v)

}
```

If we run the project in the simulator, we see a small very dark square against a white background. But now suppose we switch to dark mode. Now the background becomes black, and we don't see our dark square any longer. The reason is that the view controller's main view has a *dynamic* color, which is white in light mode but black in dark mode, and now our dark square is black on black.

One solution is to make our UIColor dynamic. We can do this with the initializer `init(dynamicProvider:)`, giving it as parameter a function that takes a trait collection and returns a color. I'll explain more about what a trait collection is later in this chapter; right now, all you need to know is that its `userInterfaceStyle` may or may not be `.dark`:

```
v.backgroundColor = UIColor { tc in

    switch tc.userInterfaceStyle {

    case .dark:

        return UIColor(red: 0.3, green: 0.4, blue: 0.4,
alpha: 1)

    default:

        return UIColor(red: 0, green: 0.1, blue: 0.1, alpha:
1)
```

```
        }

    }
```

We have created our own custom dynamic color, which is different depending what mode we're in. In dark mode, our view's color is now a dark gray that is visible against a black background.

TIP

To switch to dark mode in the simulator, click the Environment Overrides button in the debug bar. In the popover that appears, click the first switch, at the upper right.

A more compact way to get a dynamic color is to use one of the many dynamic colors vended as static properties by UIColor in iOS 13. Most of these have names that start with `.system`, such as `.systemYellow`; others have *semantic* names describing their role, such as `.label`. For details, see Apple's Human Interface Guidelines (*https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/color/*).

You can also design a custom named color in the asset catalog. When you do, you can choose from the Appearances pop-up menu in the Attributes inspector and switch to Any, Dark. Now there are two colors, one for dark mode and the other for everything else, just as in our earlier code. Let's say we've done that, and our color in the asset catalog is called `myDarkColor`. Then you could say:

```
v.backgroundColor = UIColor(named: "myDarkColor")
```

Custom named colors from the asset catalog also appear in the Library and in the color pop-up menus in the Attributes inspector when you select a view.

Visibility and Opacity

Three properties relate to the visibility and opacity of a view:

`isHidden`

A view can be made invisible by setting its `isHidden` property to `true`, and visible again by setting it to `false`. Hiding a view takes it (and its subviews, of course) out of the visible interface without actually removing it from the view hierarchy. A hidden view does not (normally) receive touch events, so to the user it really is as if the view weren't there. But it is there, so it can still be manipulated in code.

`alpha`

A view can be made partially or completely transparent through its `alpha` property: `1.0` means opaque, `0.0` means transparent, and a value may be anywhere between them, inclusive. This property affects both the apparent transparency of the view's background color and the apparent transparency of its contents. If a view displays an image and has a background color and its `alpha` is less than 1, the background color will seep through the image (and whatever is behind the view will seep through both). Moreover, it affects the apparent transparency of the view's subviews! If a superview has an `alpha` of `0.5`, none of its subviews can have an *apparent* opacity of more than `0.5`, because whatever `alpha` value they have will be drawn relative to `0.5`. A view that is completely transparent (or very close to it) is like a view whose `isHidden` is `true`: it is invisible, along with its subviews, and cannot (normally) be touched.

(Just to make matters more complicated, colors have an alpha value as well. A view can have an `alpha` of `1.0` but still have a transparent background because its `backgroundColor` has an alpha less than `1.0`.)

`isOpaque`

This property is a horse of a different color; changing it has no effect on the view's appearance. Rather, it is a hint to the drawing system. If a view is completely filled with opaque material and its `alpha` is `1.0`, so that the view has no effective transparency, then it can be drawn more efficiently (with less drag on performance) if you inform the drawing system of this fact by setting its `isOpaque` to `true`. Otherwise, you should set its `isOpaque` to `false`. The `isOpaque` value is *not* changed for you when you set a

view's `backgroundColor` or `alpha`! Setting it correctly is entirely up to you; the default, perhaps surprisingly, is `true`.

Frame

A view's `frame` property, a CGRect, is the position of its rectangle within its superview, *in the superview's coordinate system*. By default, the superview's coordinate system will have the origin at its top left, with the x-coordinate growing positively rightward and the y-coordinate growing positively downward.

Setting a view's frame to a different CGRect value repositions the view, or resizes it, or both. If the view is visible, this change will be visibly reflected in the interface. On the other hand, you can also set a view's frame when the view is not visible, such as when you create the view in code. In that case, the frame describes where the view *will* be positioned within its superview when it is given a superview.

UIView's designated initializer is `init(frame:)`, and you'll often assign a frame this way, especially because the default frame might otherwise be `CGRect.zero`, which is rarely what you want. A view with a zero-size frame is effectively invisible (though you might still see its subviews). Forgetting to assign a view a frame when creating it in code, and then wondering why it isn't appearing when added to a superview, is a common beginner mistake. If a view has a standard size that you want it to adopt, especially in relation to its contents (like a UIButton in relation to its title), an alternative is to call its `sizeToFit` method.

We are now in a position to generate programmatically the interface displayed in Figure 1-1; we determine the layering order of `v1` and `v3` (the middle and left views, which are siblings) by the order in which we insert them into the view hierarchy:

```
let v1 = UIView(frame:CGRect(113, 111, 132, 194))

v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1,
alpha: 1)

let v2 = UIView(frame:CGRect(41, 56, 132, 194))
```

```
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0,
alpha: 1)

let v3 = UIView(frame:CGRect(43, 197, 160, 230))

v3.backgroundColor = UIColor(red: 1, green: 0, blue: 0,
alpha: 1)

self.view.addSubview(v1)

v1.addSubview(v2)

self.view.addSubview(v3)
```