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. Creating and configuring a view can be extremely simple: "Set it and forget it." For example, you can drag an interface object, such as a UIButton, into a view 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. Thus, 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.

The Window

The top of the view hierarchy is the app's window. It is an instance of UIWindow (or your own subclass thereof), which is a UIView subclass. Your app should have *exactly one main window*. It is created at launch time and is never destroyed or replaced. It forms the background to, and is the ultimate superview of, all your other visible views. Other views are visible by virtue of being subviews, at some depth, of your app's window.

If your app can display views on an external screen, you'll create an additional UIWindow to contain those views; but in this chapter I'll behave as if there were just one screen, the device's own screen, and just one window.

The app's window must initially fill the device's screen. This is ensured by setting the window's frame to the screen's bounds as the window is instantiated. (I'll explain later in this chapter what "frame" and "bounds" are.) If you're using a main storyboard, that's taken care of for you automatically behind the scenes by the `UIApplicationMain` function as the app launches; but an app without a main storyboard is possible, and in that case you'd need to create the window and set its frame yourself, very early in the app's lifetime, like this:

```
let w = UIWindow(frame: UIScreen.mainScreen().bounds)
```

New in iOS 9, it's sufficient to instantiate UIWindow with *no* frame; the screen's bounds will be assigned to the window's frame for you:

```
let w = UIWindow()
```

The window must also persist for the lifetime of the app. To make this happen, the app delegate class has a `window` property with a strong retain policy. As the app launches, the `UIApplicationMain` function instantiates the app delegate class and retains the resulting instance. This is the app delegate instance; it is never released, so it persists for the lifetime of the app. The window instance is then assigned to the app delegate instance's `window` property; therefore it, too, persists for the lifetime of the app.

You will typically not put any view content manually and directly inside your main window. Instead, you'll obtain a view controller and assign it to the main window's `rootViewController` property. Once again, if you're using a main storyboard, this is done automatically behind the scenes; the view controller in question will be your storyboard's initial view controller.

When a view controller becomes the main window's `rootViewController`, its main view (its `view`) is made the one and only immediate subview of your main window — the main window's *root view*. All other views in your main window will be subviews of the root view. Thus, the root view is the highest object in the view hierarchy that the user will usually see. There might be just a chance, under certain circumstances, that

the user will catch a glimpse of the window, behind the root view; for this reason, you may want to assign the main window a reasonable `backgroundColor`. But this seems unlikely, and in general you'll have no reason to change anything about the window itself.

Your app's interface is not visible until the window, which contains it, is made the app's key window. This is done by calling the UIWindow instance method `makeKeyAndVisible`.

Let's summarize how the initial creation, configuration, and display of the main window happens. There are two cases to consider:

App with a main storyboard
: If your app has a main storyboard, as specified by its *Info.plist* key "Main storyboard file base name" (`UIMainStoryboardFile`) — the default for all Xcode 7 app templates — then `UIApplicationMain` instantiates UIWindow, sets its frame correctly, and assigns that instance to the app delegate's `window` property. In addition, it instantiates the storyboard's initial view controller, and assigns that instance to the window's `rootViewController` property. All of that happens *before* the app delegate's `application:didFinishLaunchingWithOptions:` is called (Appendix A).

  Finally, `UIApplicationMain` calls `makeKeyAndVisible` on the window, to display your app's interface. This in turn automatically causes the root view controller to obtain its main view (typically by loading it from a nib), which the window adds as its own root view. That happens *after* `application:didFinishLaunchingWithOptions:` is called.

App without a main storyboard
: If your app has no main storyboard, then creation and configuration of the window must be done in some other way. Typically, it is done in code. No Xcode 7 app template lacks a main storyboard, but if you start with, say, the Single View Application template, you can experiment as follows:

  1. Edit the target. In the General pane, select "Main" in the Main Interface field and delete it (and press Tab to set this change).
  2. Delete *Main.storyboard* and *ViewController.swift* from the project.
  3. Delete the entire content of *AppDelegate.swift*.

  You now have a project with an app target but no storyboard and no code. To make a minimal working app, you need to edit *AppDelegate.swift* in such a way as to recreate the AppDelegate class with just enough code to create and show the window at launch time, as demonstrated in Example 1-1.

Example 1-1. An App Delegate class with no storyboard

```
import UIKit
@UIApplicationMain
class AppDelegate : UIResponder, UIApplicationDelegate {
    var window : UIWindow?
    func application(application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
        -> Bool {
            self.window = UIWindow()
            self.window!.rootViewController = UIViewController()
            self.window!.backgroundColor = UIColor.whiteColor()
            self.window!.makeKeyAndVisible()
            return true
    }
}
```

The result of implementing Example 1-1 is a minimal working app with an empty white window; you can prove to yourself that your code is creating the window by changing its `backgroundColor` to something else (such as `UIColor.redColor()`) and running the app again. The app is extremely simple and rather inflexible because its root view controller is a generic UIViewController, but it's a legal working app and is sufficient for some basic experimentation with views; in real life, you'd set the window's `rootViewController` to a new instance of your own custom UIViewController subclass (and I'll talk more about that in Chapter 6).

It is extremely improbable that you would ever need to subclass UIWindow. If, however, you wanted to create a UIWindow subclass and make an instance of that subclass your app's main window, then how you proceed depends on how the window is instantiated in the first place:

App with a main storyboard
: As the app launches, after `UIApplicationMain` has instantiated the app delegate, it asks the app delegate instance for the value of its `window` property. If that value is `nil`, `UIApplicationMain` instantiates UIWindow and assigns that instance to the app delegate's `window` property. If that value is *not* `nil`, `UIApplicationMain` leaves it alone and uses it as the app's main window. Therefore, to make your app's main window be an instance of your UIWindow subclass, you'll make that instance the default value for the app delegate's `window` property, like this:

```
lazy var window : UIWindow? = {
    return MyWindow()
}()
```

App without a main storyboard
: You're already instantiating UIWindow and assigning that instance to the app delegate's `self.window` property, in code (Example 1-1). So instantiate your UIWindow subclass instead:

```
// ...
self.window = MyWindow()
// ...
```

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

- If a UIView is in the interface, it automatically has a reference to the window through its own `window` property.

  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.

- The app delegate instance maintains a reference to the window through its `window` property. You can get a reference to the app delegate from elsewhere through the shared application's `delegate` property, and through it you can refer to the window:

  ```
  let w = UIApplication.sharedApplication().delegate!.window!!
  ```

  If you prefer something less generic (and requiring less extreme unwrapping of Optionals), cast the `delegate` explicitly to your app delegate class:

  ```
  let w = (UIApplication.sharedApplication().delegate as! AppDelegate).window!
  ```

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

  ```
  let w = UIApplication.sharedApplication().keyWindow!
  ```

  That reference, however, is slightly volatile, because the system can create temporary windows and interpose them as the application's key window.

Experimenting With Views

In the course of this and subsequent chapters, you may want to experiment with views in a project of your own. Since view controllers aren't formally explained until Chapter 6, I'll just outline two simple approaches.

Single View Application template
: If you start your project with the Single View Application template, it gives you a main storyboard containing one scene containing one view controller instance containing its own main view; when the app runs, that view controller will become the app's main window's `rootViewController`, and its main view will become the window's root view.

  You will now want some subviews of the main view, to experiment with. In the nib editor, you can drag a view from the Object library into the main view as a subview, and it will be instantiated in the interface when the app runs. Alternatively, you can write code to create views and add them to the interface; the simplest place to do

this, for now, is the view controller's `viewDidLoad` method, which has a reference to the view controller's main view as `self.view`. For example:

```
override func viewDidLoad() {
    super.viewDidLoad()
    let mainview = self.view
    let v = UIView(frame:CGRectMake(100,100,50,50))
    v.backgroundColor = UIColor.redColor() // small red square
    mainview.addSubview(v) // add it to main view
}
```

App without a main storyboard

If you start with the empty application without a storyboard that I described in Example 1-1, it has no *.xib* or *.storyboard* file, so your views will have to be created entirely in code. Our root view controller is a purely generic UIViewController, so we have no `viewDidLoad` override, but we can access its main view through its `view` property. For example:

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
    -> Bool {
        self.window = UIWindow()
        self.window!.rootViewController = UIViewController()
        // here we can add subviews
        let mainview = self.window!.rootViewController!.view
        let v = UIView(frame:CGRectMake(100,100,50,50))
        v.backgroundColor = UIColor.redColor() // small red square
        mainview.addSubview(v) // add it to main view
        // and the rest is as before...
        self.window!.backgroundColor = UIColor.whiteColor()
        self.window!.makeKeyAndVisible()
        return true
}
```

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.

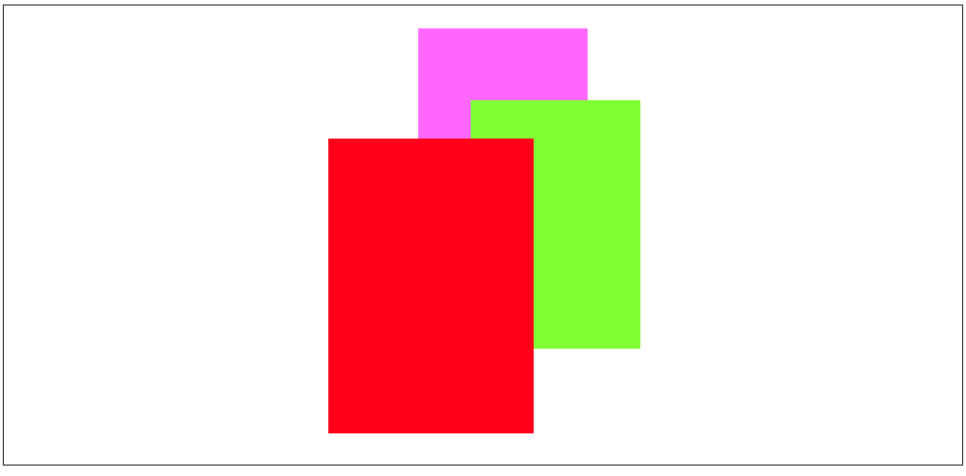

Figure 1-1. Overlapping views

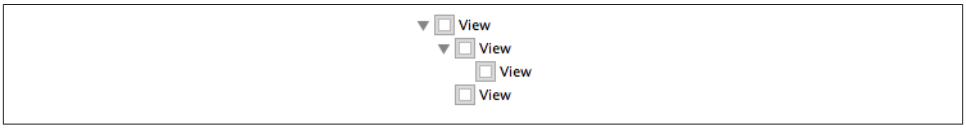

Figure 1-2. A view hierarchy as displayed in the nib editor

For example, 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, the view in the middle (horizontally) is a sibling view of the view on the left (they are both direct subviews of the root view), and the view on the right is a subview of the middle view.

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.

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: one is drawn before the other, so if they overlap, it will appear to be behind its sibling. Similarly, a superview is drawn before its subviews, so if they overlap it, it will appear to be behind them.

You can see this illustrated in Figure 1-1. The view on the right is a subview of the view in the middle and is drawn on top of it. The view on the left is a sibling of the view in

the middle, but it is a later sibling, so it is drawn on top of the view in the middle and on top of the view on the right. The view on the left *cannot* appear behind the view on the right but in front of the view in the middle, because those two views are subview and superview and are drawn together — both are drawn either before or after the view on the left, 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 them and is responsible for releasing a subview when that subview ceases to be its subview (it is removed from the collection of this view's subviews) or when it 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 `isDescendantOfView:` 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 this 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, and it compensates for the fact that there is basically just a single window. It is perfectly reasonable for your code to rip an entire hierarchy of views out of the superview and substitute another! 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; and of course this view may itself have subviews that accompany it. Just remember that removing a subview from its superview releases it; if you intend to reuse that subview later on, you will wish to retain it first. This is often taken care of by assignment to a property.

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:

- `didAddSubview:`, `willRemoveSubview:`
- `didMoveToSuperview`, `willMoveToSuperview:`
- `didMoveToWindow`, `willMoveToWindow:`

When `addSubview:` is called, the view is placed last among its superview's subviews; thus it is drawn last, meaning that it appears frontmost. A view's subviews are indexed, starting at `0`, which is rearmost. There are additional 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:atIndex:`
- `insertSubview:belowSubview:`, `insertSubview:aboveSubview:`
- `exchangeSubviewAtIndex:withSubviewAtIndex:`
- `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()}
```

Visibility and Opacity

A view can be made invisible by setting its `hidden` 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 the overhead of 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.

A view can be assigned a background color through its `backgroundColor` property. A color is a UIColor; this is not a difficult class to use, and I'm not going to go into details. A view whose background color is `nil` (the default) has a transparent background. It is perfectly reasonable for a view to have a transparent background and to do no additional

drawing of its own, just so that it can act as a convenient superview to other views, making them behave together.

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 affects 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`. (Just to make matters more complicated, colors have an alpha value as well. So, for example, 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`.) A view that is completely transparent (or very close to it) is like a view whose `hidden` is `true`: it is invisible, along with its subviews, and cannot (normally) be touched.

A view's `alpha` property value affects both the apparent transparency of its background color and the apparent transparency of its contents. For example, 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).

A view's `opaque` property, on the other hand, is a horse of a different color; changing it has no effect on the view's appearance. Rather, this property is a hint to the drawing system. If a view completely fills its bounds with ultimately 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 `opaque` to `true`. Otherwise, you should set its `opaque` to `false`. The `opaque` 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 — for example, 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 `CGRectZero`, which is rarely what you want.

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. A view with a zero-size frame is effectively invisible. 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:

```
let v1 = UIView(frame:CGRectMake(113, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:CGRectMake(41, 56, 132, 194))
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
let v3 = UIView(frame:CGRectMake(43, 197, 160, 230))
v3.backgroundColor = UIColor(red: 1, green: 0, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
mainview.addSubview(v3)
```

In that code, we determined the layering order of `v1` and `v3` (the middle and left views, which are siblings) by the order in which we inserted them into the view hierarchy with `addSubview:`.

Bounds and Center

Suppose we have a superview and a subview, and the subview is to appear inset by 10 points, as in Figure 1-3. The Foundation utility function `CGRectInset` and the Swift CGRect method `insetBy` make it easy to derive one rectangle as an inset from another, so we'll use one of them to determine the subview's frame. But *what* rectangle should this be inset from? Not the superview's frame; the frame represents a view's position within *its* superview, and in that superview's coordinates. What we're after is a CGRect describing our superview's rectangle in its *own* coordinates, because those are the coordinates in which the subview's frame is to be expressed. The CGRect that describes a view's rectangle in its own coordinates is the view's `bounds` property.

So, the code to generate Figure 1-3 looks like this:

```
let v1 = UIView(frame:CGRectMake(113, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:v1.bounds.insetBy(dx: 10, dy: 10))
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
```

You'll very often use a view's `bounds` in this way. When you need coordinates for positioning content inside a view, whether drawing manually or placing a subview, you'll often refer to the view's `bounds`.

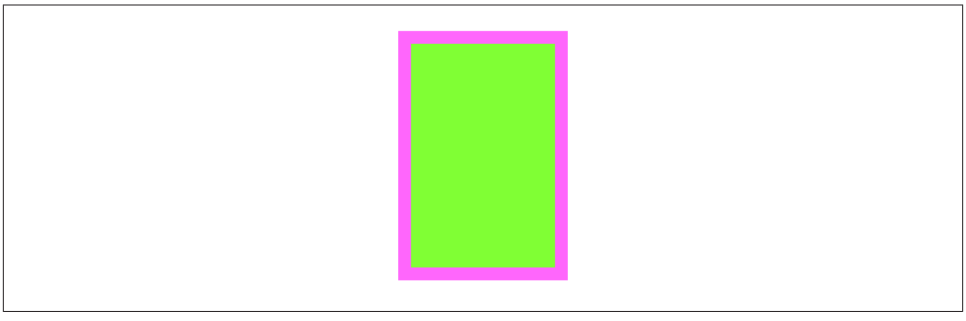

Figure 1-3. A subview inset from its superview

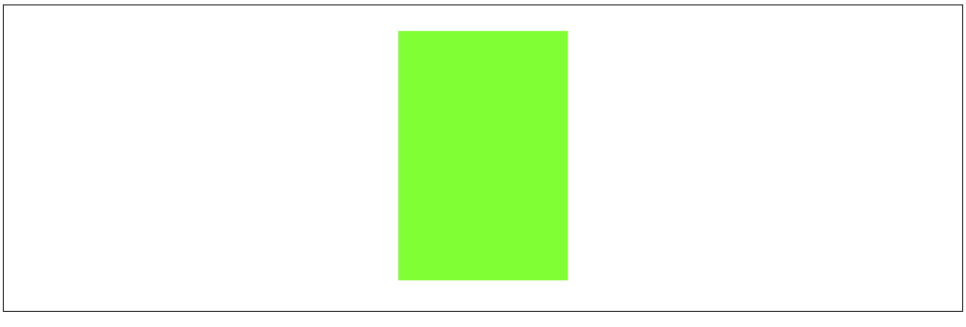

Figure 1-4. A subview exactly covering its superview

Interesting things happen when you set a view's bounds. If you change a view's bounds *size*, you change its *frame*. The change in the view's frame takes place around its *center*, which remains unchanged. So, for example:

```
let v1 = UIView(frame:CGRectMake(113, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:v1.bounds.insetBy(dx: 10, dy: 10))
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
v2.bounds.size.height += 20
v2.bounds.size.width += 20
```

What appears is a single rectangle; the subview completely and exactly covers its superview, its frame being the same as the superview's bounds. The call to `insetBy` started with the superview's bounds and shaved 10 points off the left, right, top, and bottom to set the subview's frame (Figure 1-3). But then we added 20 points to the subview's bounds height and width, and thus added 20 points to the subview's frame height and width as well (Figure 1-4). The center didn't move, so we effectively put the 10 points back onto the left, right, top, and bottom of the subview's frame.

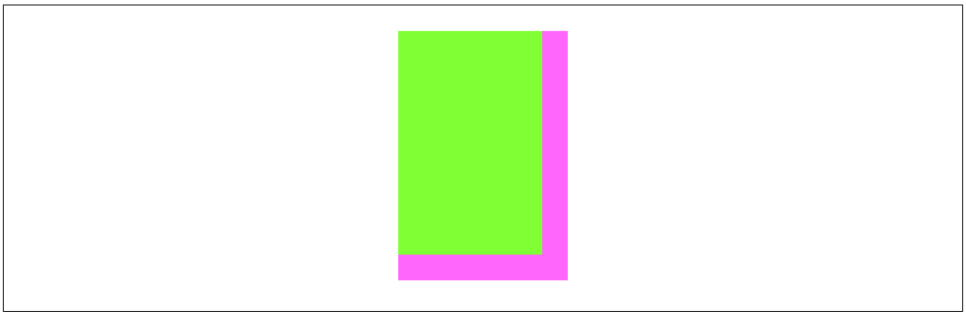

Figure 1-5. The superview's bounds origin has been shifted

When you create a UIView, its bounds coordinate system's zero point (`0.0,0.0`) is at its top left. If you change a view's bounds *origin*, you move the *origin of its internal coordinate system*. Because a subview is positioned in its superview with respect to its superview's coordinate system, a change in the bounds origin of the superview will change the apparent position of a subview. To illustrate, we start once again with our subview inset evenly within its superview, and then change the bounds origin of the superview:

```
let v1 = UIView(frame:CGRectMake(113, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:v1.bounds.insetBy(dx: 10, dy: 10))
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
v1.bounds.origin.x += 10
v1.bounds.origin.y += 10
```

Nothing happens to the superview's size or position. But the subview has moved up and to the left so that it is flush with its superview's top-left corner (Figure 1-5). Basically, what we've done is to say to the superview, "Instead of calling the point at your upper left (`0.0,0.0`), call that point (`10.0,10.0`)." Because the subview's frame origin is itself at (`10.0,10.0`), the subview now touches the superview's top-left corner. The effect of changing a view's bounds origin may seem directionally backward — we increased the superview's origin in the positive direction, but the subview moved in the negative direction — but think of it this way: a view's bounds origin point coincides with its frame's top left.

We have seen that changing a view's bounds size affects its frame size. The converse is also true: changing a view's frame size affects its bounds size. What is *not* affected by changing a view's bounds size is the view's `center`. This property, like the `frame` property, represents a subview's position within its superview, in the superview's coordinates; in particular, it is the position within the superview of the subview's bounds center, the point derived from the bounds like this:

```
let c = CGPointMake(theView.bounds.midX, theView.bounds.midY)
```

A view's center is thus a single point establishing the positional relationship between the view's bounds and its superview's bounds.

Changing a view's bounds does not change its center; changing a view's center does not change its bounds. Thus, a view's bounds and center are orthogonal (independent), and describe (among other things) both the view's size and its position within its superview. The view's frame is therefore superfluous! In fact, the `frame` property is merely a convenient expression of the `center` and `bounds` values. In most cases, this won't matter to you; you'll use the `frame` property anyway. When you first create a view from scratch, the designated initializer is `init(frame:)`. You can change the frame, and the bounds size and center will change to match. You can change the bounds size or the center, and the frame will change to match. Nevertheless, the proper and most reliable way to position and size a view within its superview is to use its bounds and center, not its frame; there are some situations in which the frame is meaningless (or will at least behave very oddly), but the bounds and center will always work.

We have seen that every view has its own coordinate system, expressed by its `bounds`, and that a view's coordinate system has a clear relationship to its superview's coordinate system, expressed by its `center`. This is true of every view in a window, so it is possible to convert between the coordinates of any two views in the same window. Convenience methods are supplied to perform this conversion both for a CGPoint and for a CGRect:

- `convertPoint:fromView:`, `convertPoint:toView:`
- `convertRect:fromView:`, `convertRect:toView:`

If the second parameter is `nil`, it is taken to be the window.

For example, if `v2` is a subview of `v1`, then to center `v2` within `v1` you could say:

```
v2.center = v1.convertPoint(v1.center, fromView:v1.superview)
```

When setting a view's position by setting its center, if the height or width of the view is not an integer (or, on a single-resolution screen, not an even integer), the view can end up *misaligned*: its point values in one or both dimensions are located between the screen pixels. This can cause the view to be displayed incorrectly; for example, if the view contains text, the text may be blurry. You can detect this situation in the Simulator by checking Debug → Color Misaligned Images. A simple solution is to set the view's frame, after positioning it, to the `CGRectIntegral` of its frame, or (in Swift) to call `makeIntegralInPlace` on the view's frame.

Window Coordinates and Screen Coordinates

The device screen has no frame, but it has bounds. The main window has no superview, but its frame is set with respect to the screen's bounds, as I showed earlier:

```
let w = UIWindow(frame: UIScreen.mainScreen().bounds)
```

In iOS 9, you can omit the `frame` parameter, as a shortcut, but the effect is exactly the same:

```
let w = UIWindow()
```

The window thus starts out life filling the screen, and generally continues to fill the screen, and so, for the most part, *window coordinates are screen coordinates*. (However, new in iOS 9, there are circumstances under which that won't be the case; I'll discuss them in Chapter 9.)

In iOS 7 and before, the screen's coordinates were invariant, regardless of the orientation of the device and of the rotation of the app to compensate. iOS 8 introduced a major change in this coordinate system: when the app rotates to compensate for the rotation of the device, the screen (and therefore the window) is what rotates. This change is expressed in part as a transposition of the size components of the bounds of the screen and window (and the frame of the window): in portrait orientation, the size is taller than wide, but in landscape orientation, the size is wider than tall.

Nevertheless, you might still want to obtain device coordinates, independent of the rotation of the app. Therefore, the screen reports its coordinates through two different properties; their values are typed as UICoordinateSpace, a protocol (also adopted by UIView) that provides a `bounds` property:

UIScreen's `coordinateSpace` *property*
: This coordinate space rotates, so that its `bounds` height and width are transposed when the app rotates to compensate for a change in the orientation of the device; its `(0.0,0.0)` point is at the app's top left.

UIScreen's `fixedCoordinateSpace` *property*
: This coordinate space is invariant, meaning that its `bounds` top left represents the physical top left of the device *qua* physical device; its `(0.0,0.0)` point thus might be in any corner (from the user's perspective).

To help you convert between coordinate spaces, UICoordinateSpace also provides four methods parallel to the coordinate-conversion methods I listed in the previous section:

- `convertPoint:fromCoordinateSpace:`, `convertPoint:toCoordinateSpace:`
- `convertRect:fromCoordinateSpace:`, `convertRect:toCoordinateSpace:`

So, for example, suppose we have a UIView `v` in our interface, and we wish to learn its position in fixed device coordinates. We could do it like this:

```
let r = v.superview!.convertRect(
    v.frame, toCoordinateSpace: UIScreen.mainScreen().fixedCoordinateSpace)
```

Occasions where you need such information, however, will be rare. Indeed, my experience is that it is rare even to worry about window coordinates. All of your app's visible action takes place within your root view controller's main view, and the bounds of that view, which are adjusted for you automatically when the app rotates to compensate for a change in device orientation, are probably the highest coordinate system that will interest you.

Transform

A view's `transform` property alters how the view is drawn — it may, for example, change the view's perceived size and orientation — without affecting its bounds and center. A transformed view continues to behave correctly: a rotated button, for example, is still a button, and can be tapped in its apparent location and orientation.

A transform value is a CGAffineTransform, which is a struct representing six of the nine values of a 3×3 transformation matrix (the other three values are constants, so there's no need to represent them in the struct). You may have forgotten your high-school linear algebra, so you may not recall what a transformation matrix is. For the details, which are quite simple really, see the "Transforms" chapter of Apple's *Quartz 2D Programming Guide*, especially the section called "The Math Behind the Matrices." But you don't really need to know those details, because convenience functions, whose names start with `CGAffineTransformMake...`, are provided for creating three of the basic types of transform: rotation, scaling, and translation (i.e., changing the view's apparent position). A fourth basic transform type, skewing or shearing, has no convenience function.

By default, a view's transformation matrix is `CGAffineTransformIdentity`, the identity transform. It has no visible effect, so you're unaware of it. Any transform that you do apply takes place around the view's center, which is held constant.

Here's some code to illustrate use of a transform:

```
let v1 = UIView(frame:CGRectMake(113, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:v1.bounds.insetBy(dx: 10, dy: 10))
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
v1.transform = CGAffineTransformMakeRotation(45 * CGFloat(M_PI)/180.0)
```

The `transform` property of the view `v1` is set to a rotation transform. The result (Figure 1-6) is that the view appears to be rocked 45 degrees clockwise. (I think in

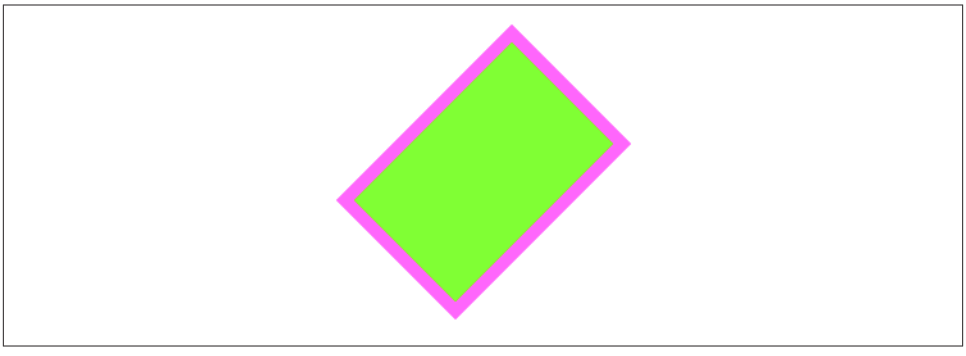

Figure 1-6. A rotation transform

degrees, but Core Graphics thinks in radians, so my code has to convert.) Observe that the view's `center` property is unaffected, so that the rotation seems to have occurred around the view's center. Moreover, the view's `bounds` property is unaffected; the internal coordinate system is unchanged, so the subview is drawn in the same place relative to its superview. The view's `frame`, however, is now useless, as no mere rectangle can describe the region of the superview apparently occupied by the view; the frame's actual value, roughly `(63.7,92.7,230.5,230.5)`, describes the minimal bounding rectangle surrounding the view's apparent position. The rule is that if a view's `transform` is not the identity transform, you should not set its `frame`; also, automatic resizing of a subview, discussed later in this chapter, requires that the superview's transform be the identity transform.

Suppose, instead of `CGAffineTransformMakeRotation`, we call `CGAffineTransformMakeScale`, like this:

```
v1.transform = CGAffineTransformMakeScale(1.8, 1)
```

The `bounds` property of the view `v1` is still unaffected, so the subview is still drawn in the same place relative to its superview; this means that the two views seem to have stretched horizontally together (Figure 1-7). No bounds or centers were harmed by the application of this transform!

Transformation matrices can be chained. There are convenience functions for applying one transform to another. Their names do *not* contain `Make`. These functions are not commutative; that is, order matters. (That high school math is starting to come back to you now, isn't it?) If you start with a transform that translates a view to the right and then apply a rotation of 45 degrees, the rotated view appears to the right of its original position; on the other hand, if you start with a transform that rotates a view 45 degrees and then apply a translation to the right, the meaning of "right" has changed, so the rotated view appears 45 degrees down from its original position. To demonstrate the difference, I'll start with a subview that exactly overlaps its superview:

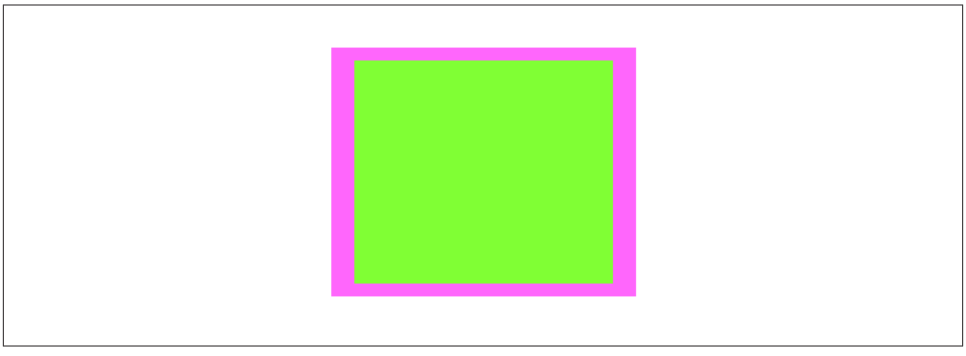

Figure 1-7. A scale transform

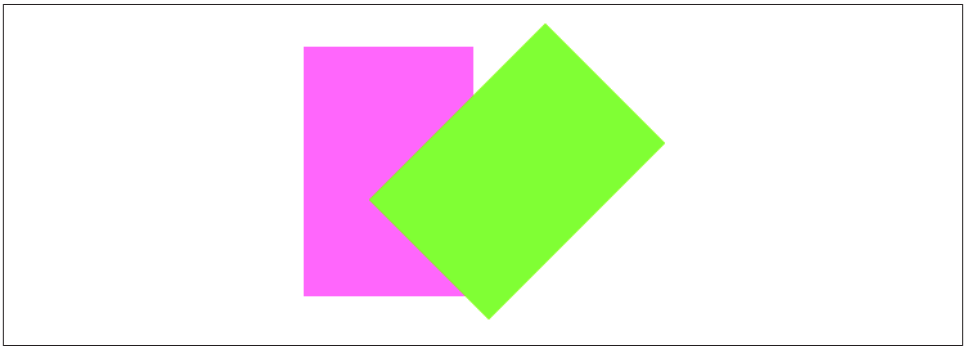

Figure 1-8. Translation, then rotation

```
let v1 = UIView(frame:CGRectMake(20, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:v1.bounds)
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
```

Then I'll apply two successive transforms to the subview, leaving the superview to show where the subview was originally. In this example, I translate and then rotate (Figure 1-8):

```
v2.transform = CGAffineTransformMakeTranslation(100, 0)
v2.transform = CGAffineTransformRotate(v2.transform, 45 * CGFloat(M_PI)/180.0)
```

In this example, I rotate and then translate (Figure 1-9):

```
v2.transform = CGAffineTransformMakeRotation(45 * CGFloat(M_PI)/180.0)
v2.transform = CGAffineTransformTranslate(v2.transform, 100, 0)
```

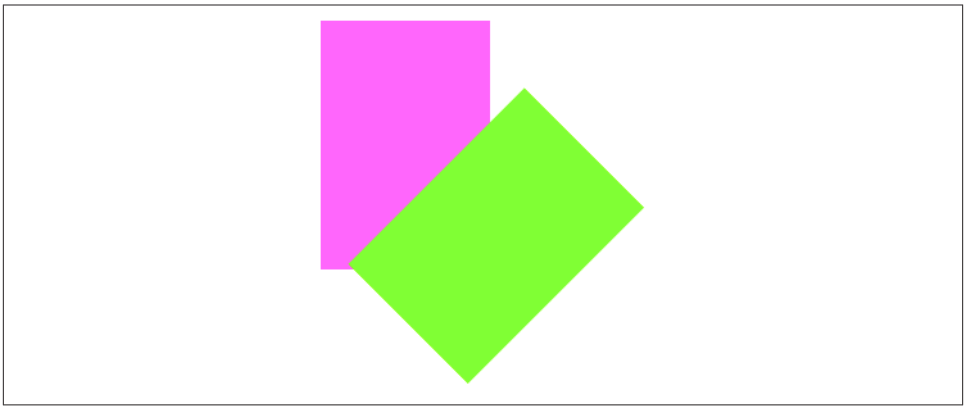

Figure 1-9. Rotation, then translation

The function `CGAffineTransformConcat` concatenates two transform matrices using matrix multiplication. Again, this operation is not commutative. The order is the *opposite* of the order when using convenience functions for applying one transform to another. For example, this gives the same result as Figure 1-9:

```
let r = CGAffineTransformMakeRotation(45 * CGFloat(M_PI)/180.0)
let t = CGAffineTransformMakeTranslation(100, 0)
v2.transform = CGAffineTransformConcat(t,r) // not r,t
```

To remove a transform from a combination of transforms, apply its inverse. A convenience function lets you obtain the inverse of a given affine transform. Again, order matters. In this example, I rotate the subview and shift it to its "right," and then remove the rotation (Figure 1-10):

```
let r = CGAffineTransformMakeRotation(45 * CGFloat(M_PI)/180.0)
let t = CGAffineTransformMakeTranslation(100, 0)
v2.transform = CGAffineTransformConcat(t,r)
v2.transform = CGAffineTransformConcat(
    CGAffineTransformInvert(r), v2.transform)
```

Finally, as there are no convenience methods for creating a skew (shear) transform, I'll illustrate by creating one manually, without further explanation (Figure 1-11):

```
v1.transform = CGAffineTransformMake(1, 0, -0.2, 1, 0, 0)
```

Transforms are useful particularly as temporary visual indicators. For example, you might call attention to a view by applying a transform that scales it up slightly, and then applying the identity transform to restore it to its original size, and animating those changes (Chapter 4).

In iOS 7 and before, the `transform` property lay at the heart of an iOS app's ability to rotate its interface: the window's frame and bounds were invariant, locked to the screen,

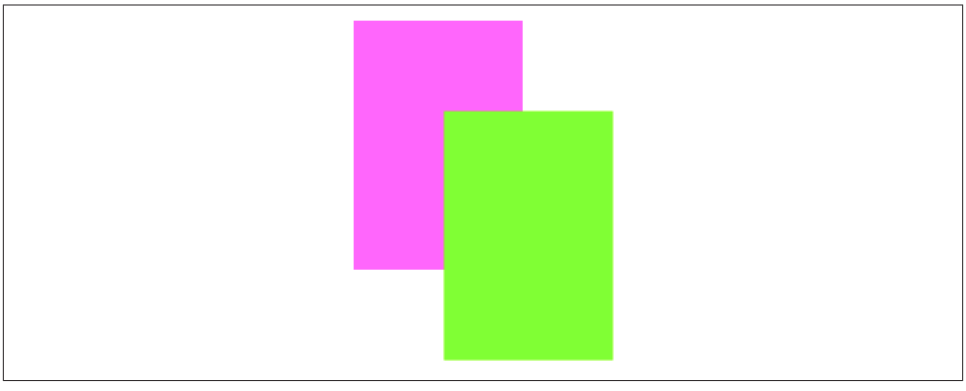

Figure 1-10. Rotation, then translation, then inversion of the rotation

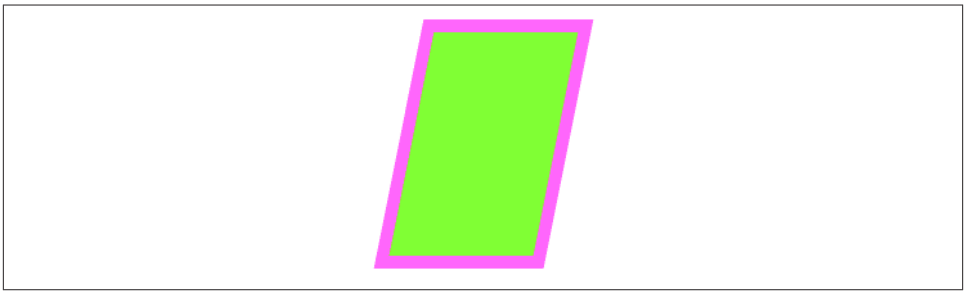

Figure 1-11. Skew (shear)

and an app's interface rotated to compensate for a change in device orientation by applying a rotation transform to the root view, so that its `(0.0,0.0)` point moved to what the user now saw as the top left of the view.

In iOS 8 and later, as I've already mentioned, this is no longer the case. The screen's coordinate space is effectively rotated, but a coordinate space doesn't have a `transform` property, so the rotation transform applied to that coordinate space is fictitious: you can work out what has happened, if you really want to, by comparing the screen's `coordinateSpace` with its `fixedCoordinateSpace`, but none of the views in the story — neither the window, nor the root view, nor any of its subviews — receives a rotation transform when the app's interface rotates.

Instead, you are expected to concentrate on the *dimensions* of the window, the root view, and so forth. This might mean their absolute dimensions, but it will often mean their dimensions as embodied in a set of *size classes* which are vended by a view's `traitCollection` property as a UITraitCollection object. I'll discuss trait collections and size classes further in the next section.

You can thus treat app rotation as effectively nothing more than a change in the interface's *proportions*: when the app rotates, the long dimension (of the root view, the window, and the screen's coordinate space bounds) becomes its short dimension and *vice versa*. This, after all, is what your interface needs to take into account in order to keep working when the app rotates.

Consider, for example, a subview of the root view, located at the bottom right of the screen when the device is in portrait orientation. If the root view's bounds width and bounds height are effectively transposed, then that poor old subview will now be outside the bounds height, and therefore off the screen — unless your app responds in some way to this change to reposition it. Such a response is called *layout*, a subject that will occupy most of the rest of this chapter. The point, however, is that what you're responding *to* is just a change in the window's proportions; the fact that this change stems from rotation of the app's interface is all but irrelevant.

Trait Collections and Size Classes

Every view in the interface, from the window on down, as well as any view controller whose view is part of the interface, inherits from the environment the value of its `traitCollection` property, which it has by virtue of implementing the UITraitEnvironment protocol. The `traitCollection` is a UITraitCollection, a value class consisting of four properties:

`displayScale`
: The scale inherited from the current screen, typically 1 or 2 for a single- or double-resolution screen respectively — or 3 for the iPhone 6 Plus. (This will be the same, by default, as the UIScreen `scale` property.)

`userInterfaceIdiom`
: A UserInterfaceIdiom value, either `.Phone` or `.Pad`, stating generically what kind of device we're running on. (This will be the same, by default, as the UIDevice `userInterfaceIdiom` property.)

`horizontalSizeClass`, `verticalSizeClass`
: A UIUserInterfaceSizeClass value, either `.Regular` or `.Compact`. These are called *size classes*. The size classes, in combination, have the following meanings:

  Both the vertical and horizontal size classes are `.Regular`
  : We're running on an iPad.

  The vertical size class is `.Regular`*, but the horizontal size class is* `.Compact`
  : We're running on an iPhone with the app in portrait orientation. (Alternatively, we might be running on an iPad in a splitscreen iPad multitasking configuration; see Chapter 9.)

Both the vertical and horizontal size classes are `.Compact`
: We're running on an iPhone (except an iPhone 6 Plus) with the app in landscape orientation.

The vertical size class is `.Compact`*, but the horizontal size class is* `.Regular`
: We're running on an iPhone 6 Plus with the app in landscape orientation.

The trait collection properties can differ from one run of an app to another. For example, if you write a universal app, one that runs natively on different device types (iPhone and iPad), you will probably want your interface to differ depending on which device type we're running on; trait collections are the way to detect that.

Moreover, some trait collection properties can change while the app is running. For example, the size classes on an iPhone reflect the orientation of the app — which can change as the app rotates in response to a change in the orientation of the device.

Therefore, both at app launch time and, thereafter, if the trait collection changes while the app is running, the `traitCollectionDidChange:` message is propagated down the hierarchy of UITraitEnvironments (meaning primarily, for our purposes, view controllers and views); the old trait collection (if any) is provided as the parameter, and the new trait collection can be retrieved as `self.traitCollection`.

It is also possible to create a trait collection yourself. (It may not be immediately obvious why this would be a useful thing to do; I'll give an example in the next chapter.) Oddly, however, you can't set any trait collection properties directly; instead, you form a trait collection through an initializer that determines just *one* property, and if you want to add further property settings, you have to combine trait collections by calling `init(traitsFromCollections:)`. For example:

```
let tcdisp = UITraitCollection(displayScale: 2.0)
let tcphone = UITraitCollection(userInterfaceIdiom: .Phone)
let tcreg = UITraitCollection(verticalSizeClass: .Regular)
let tc = UITraitCollection(traitsFromCollections: [tcdisp, tcphone, tcreg])
```

When combining trait collections with `init(traitsFromCollections:)`, an *ordered intersection* is performed. If two trait collections are combined, and one sets a property and the other doesn't (the property isn't set or its value isn't yet known), the one that sets the property wins; if they both set the property, the winner is the trait collection that appears later in the array.

Similarly, if you create a trait collection and you don't specify a property, this means that the value for that property is to be inherited if the trait collection finds itself in the inheritance hierarchy.

To compare trait collections, call `containsTraitsInCollection:`. This returns `true` if the value of every *specified* property of the second trait collection (the argument) matches that of the first trait collection (the target of the message).

You cannot insert a trait collection directly into the inheritance hierarchy simply by setting a view's trait collection; `traitCollection` isn't a settable property. Instead, you'll call a special `setOverrideTraitCollection:...` method; I'll give an example in Chapter 6.

Layout

We have seen that a subview moves when its superview's bounds *origin* is changed. But what happens to a subview when its superview's bounds (or frame) *size* is changed?

Of its own accord, nothing happens. The subview's bounds and center haven't changed, and the superview's bounds origin hasn't moved, so the subview stays in the same position relative to the top left of its superview. In real life, however, that often won't be what you want. You'll want subviews to be resized and repositioned when their superview's bounds size is changed. This is called *layout*.

Here are some ways in which a superview might be resized dynamically:

- Your app might compensate for the user rotating the device 90 degrees by rotating itself so that its top moves to the new top of the screen, matching its new orientation — and, as a consequence, transposing its bounds width and height values.
- An iPhone app might launch on screens with different aspect ratios: for example, the screen of the iPhone 4s is relatively shorter than the screen of later iPhone models, and the app's interface may need to adapt to this difference.
- A universal app might launch on an iPad or on an iPhone. The app's interface may need to adapt to the size of the screen on which it finds itself running.
- A view instantiated from a nib, such as a view controller's main view or a table view cell, might be resized to fit the interface into which it is placed.
- A view might respond to a change in its surrounding views. For example, when a navigation bar is shown or hidden dynamically, the remaining interface might shrink or grow to compensate, filling the available space.
- New in iOS 9, the user might alter the width of your app's window on an iPad, as part of the iPad multitasking interface.

In any of those situations, and others, layout will probably be needed.

Layout is performed in three primary ways:

Manual layout
: The superview is sent the `layoutSubviews` message whenever it is resized; so, to lay out subviews manually, provide your own subclass and override `layoutSubviews`. Clearly this could turn out to be a lot of work, but it means you can do anything you like.

Autoresizing
: Autoresizing is the pre-iOS 6 way of performing layout automatically. When its superview is resized, a subview will respond in accordance with the rules prescribed by its own `autoresizingMask` property value.

Autolayout
: Autolayout, introduced in iOS 6, depends on the *constraints* of views. A constraint (an instance of NSLayoutConstraint) is a full-fledged object with numeric values describing some aspect of the size or position of a view, often in terms of some other view; it is much more sophisticated, descriptive, and powerful than the `autoresizingMask`. Multiple constraints can apply to an individual view, and they can describe a relationship between *any* two views (not just a subview and its superview). Autolayout is implemented behind the scenes in `layoutSubviews`; in effect, constraints allow you to write sophisticated `layoutSubviews` functionality without code.

Your layout strategy can involve any combination of these. The need for manual layout is rare, but it's there if you need it. Autoresizing is used automatically unless you deliberately turn it off by setting a superview's `autoresizesSubviews` property to `false`, or unless a view uses autolayout instead. Autolayout is an opt-in technology, and can be used for whatever areas of your interface you find appropriate; a view that uses autolayout can live side by side with a view that uses autoresizing.

One of the chief places where you opt in to autolayout is the nib file, and in Xcode 7 all new *.storyboard* and *.xib* files do opt in — they have autolayout turned on, by default. To see this, select the file in the Project navigator, show the File inspector, and examine the "Use Auto Layout" checkbox. On the other hand, a view that your code creates and adds to the interface, by default, uses autoresizing, not autolayout.

Autoresizing

Autoresizing is a matter of conceptually assigning a subview "springs and struts." A spring can stretch; a strut can't. Springs and struts can be assigned internally or externally, horizontally or vertically. Thus you can specify (using internal springs and struts) whether and how the view can be resized, and (using external springs and struts) whether and how the view can be repositioned. For example:

- Imagine a subview that is centered in its superview and is to stay centered, but is to resize itself as the superview is resized. It would have struts externally and springs internally.
- Imagine a subview that is centered in its superview and is to stay centered, and is *not* to resize itself as the superview is resized. It would have springs externally and struts internally.

- Imagine an OK button that is to stay in the lower right of its superview. It would have struts internally, struts externally to its right and bottom, and springs externally to its top and left.
- Imagine a text field that is to stay at the top of its superview. It is to widen as the superview widens. It would have struts externally, but a spring to its bottom; internally it would have a vertical strut and a horizontal spring.

In code, a combination of springs and struts is set through a view's `autoresizingMask` property, which is a bitmask so that you can combine options. The options, members of the `UIViewAutoresizing` struct, represent springs; whatever isn't specified is a strut. The default is `.None`, apparently meaning all struts — but of course it can't really be *all* struts, because if the superview is resized, *something* needs to change; in reality, `.None` is the same as `.FlexibleRightMargin` together with `.FlexibleBottomMargin`.

In debugging, when you log a UIView to the console, its `autoresizingMask` is reported using the word "autoresize" and a list of the springs. The margins are `LM`, `RM`, `TM`, and `BM`; the internal dimensions are `W` and `H`. For example, `autoresize = LM+TM` means that what's flexible is the left and top margins; `autoresize = W+BM` means that what's flexible is the width and the bottom margin.

To demonstrate autoresizing, I'll start with a view and two subviews, one stretched across the top, the other confined to the lower right (Figure 1-12):

```
let v1 = UIView(frame:CGRectMake(100, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:CGRectMake(0, 0, 132, 10))
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
let v3 = UIView(frame:CGRectMake(
    v1.bounds.width-20, v1.bounds.height-20, 20, 20))
v3.backgroundColor = UIColor(red: 1, green: 0, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
v1.addSubview(v3)
```

To that example, I'll add code applying springs and struts to the two subviews to make them behave like the text field and the OK button I was hypothesizing earlier:

```
v2.autoresizingMask = .FlexibleWidth
v3.autoresizingMask = [.FlexibleTopMargin, .FlexibleLeftMargin]
```

Now I'll resize the superview, thus bringing autoresizing into play; as you can see (Figure 1-13), the subviews remain pinned in their correct relative positions:

```
v1.bounds.size.width += 40
v1.bounds.size.height -= 50
```

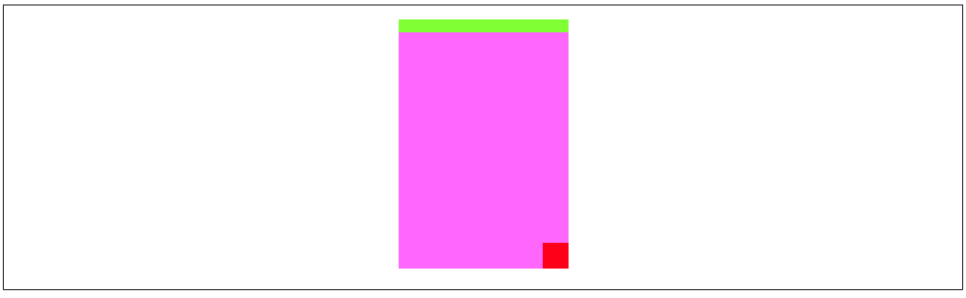

Figure 1-12. Before autoresizing

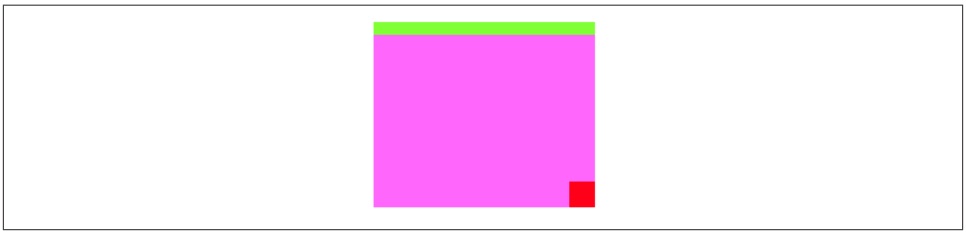

Figure 1-13. After autoresizing

That example shows exactly what autoresizing is about, but it's a little artificial; in real life, the superview is more likely to be resized, not because you resize it in code, but because of automatic behavior, such as compensatory resizing of the interface when the device is rotated. To see this, you might modify the previous example to pin the size of `v1` to the size of the root view, and then run the app and rotate the device. Thus you might initially configure `v1` like this:

```
v1.frame = mainview.bounds
v1.autoresizingMask = [.FlexibleHeight, .FlexibleWidth]
```

Now run the app and rotate the device (in the Simulator, repeatedly choose Hardware → Rotate Left). The view `v1` now fills the screen as the interface rotates, and its subviews stay pinned in their correct relative positions.

Autoresizing is effective but simple — sometimes too simple. The only relationship it describes is between a subview and its superview; it can't help you do such things as space a row of views evenly across the screen relative to one another. Before autolayout, the way to achieve more sophisticated goals of that sort was to combine autoresizing with manual layout in `layoutSubviews`. Autoresizing happens before `layoutSubviews` is called, so your `layoutSubviews` code is free to come marching in and tidy up whatever autoresizing didn't get quite right. Nowadays, though, autolayout is the norm.

Autolayout

Autolayout is an opt-in technology, at the level of each individual view. A view may opt in to autolayout in any of three ways:

- Your code adds an autolayout constraint to a view. The views involved in this constraint use autolayout.
- Your app loads a nib for which "Use Auto Layout" is checked. *Every* view instantiated from that nib uses autolayout.
- A view in the interface, which would be an instance of a custom UIView subclass of yours, returns `true` from the class method `requiresConstraintBasedLayout`. That view uses autolayout.

  The reason for this third approach to opting in to autolayout is that you might need autolayout to be switched on in order to add autolayout constraints in code. A common place to create constraints in code is in a view's `updateConstraints` implementation (discussed later in this chapter). However, if autolayout isn't switched on, `updateConstraints` won't be called. So `requiresConstraintBasedLayout` provides a way of switching it on.

One sibling view can use autolayout while another sibling view does not, and a superview can use autolayout while some or all of its subviews do not. However, autolayout is implemented through the superview chain, so if a view uses autolayout, then automatically so do all its superviews; and if (as will almost certainly be the case) one of those views is the main view of a view controller, that view controller receives autolayout-related events that it would not have received otherwise.

You can't turn off autolayout for just part of a nib. Either all views instantiated from a nib use autolayout or they all use autoresizing. To generate different parts of your interface from nibs, one part with autoresizing, another part with autolayout, separate those parts into different nibs (different *.storyboard* or *.xib* files) and then load and combine them at runtime.

Constraints

An autolayout constraint, or simply *constraint*, is an NSLayoutConstraint instance, and describes either the absolute width or height of a view or a relationship between an attribute of one view and an attribute of another view. In the latter case, the attributes don't have to be the same attribute, and the two views don't have to be siblings (subviews of the same superview) or parent and child (superview and subview) — the only requirement is that they share a common ancestor (a superview somewhere up the view hierarchy).

Here are the chief properties of an NSLayoutConstraint:

`firstItem`, `firstAttribute`, `secondItem`, `secondAttribute`
: The two views and their respective attributes (NSLayoutAttribute) involved in this constraint. If the constraint is describing a view's absolute height or width, the second view will be `nil` and the second attribute will be `.NotAnAttribute`. Additional NSLayoutAttribute values are:

  - `.Top`, `.Bottom`
  - `.Left`, `.Right`, `.Leading`, `.Trailing`
  - `.Width`, `.Height`
  - `.CenterX`, `.CenterY`
  - `.FirstBaseline`, `.LastBaseline`

  `.FirstBaseline` applies primarily to multiline labels, and is some distance down from the top of the label (Chapter 10); `.LastBaseline` is some distance up from the bottom of the label.

  The meanings of the other attributes are intuitively obvious, except that you might wonder what "leading" and "trailing" mean: they are the international equivalent of "left" and "right," automatically reversing their meaning on systems for which your app is localized and whose language is written right-to-left. New in iOS 9, the *entire* interface is automatically reversed on such systems — but that will work properly only if you've used "leading" and "trailing" constraints throughout.

`multiplier`, `constant`
: These numbers will be applied to the second attribute's value to determine the first attribute's value. The `multiplier` is multiplied by the second attribute's value; the `constant` is added to that product. The first attribute is set to the result. (The name *constant* is a very poor choice, as this value isn't constant; have the Apple folks never heard the term *addend*?) Basically, you're writing an equation $a_1 = ma_2 + c$, where a_1 and a_2 are the two attributes, and m and c are the multiplier and the constant. Thus, in the degenerate case where the first attribute's value is to equal the second attribute's value, the multiplier will be 1 and the constant will be 0. If you're describing a view's width or height absolutely, the multiplier will be 1 and the constant will be the width or height value.

`relation`
: An NSLayoutRelation stating how the two attribute values are to be related to one another, as modified by the `multiplier` and the `constant`. This is the operator that goes in the spot where I put the equal sign in the equation in the preceding paragraph. It might be an equal sign (`.Equal`), but inequalities are also permitted (`.LessThanOrEqual`, `.GreaterThanOrEqual`).

`priority`
: Priority values range from 1000 (required) down to 1, and certain standard behaviors have standard priorities. Constraints can have different priorities, determining the order in which they are applied.

A constraint belongs to a view. A view can have many constraints: a UIView has a `constraints` property, along with these instance methods:

- `addConstraint:`, `addConstraints:`
- `removeConstraint:`, `removeConstraints:`

The question then is *which* view a given constraint will belong to. The answer is: the view that is closest up the view hierarchy from both views involved in the constraint. If possible, it should *be* one of those views. Thus, for example, if the constraint dictates a view's absolute width, it belongs to that view; if it sets the top of a view in relation to the top of its superview, it belongs to that superview; if it aligns the tops of two sibling views, it belongs to their common superview.

Starting in iOS 8, however, instead of adding a constraint to a particular view explicitly, you can *activate* the constraint using the NSLayoutConstraint class method `activateConstraints:`, which takes an array of constraints. The activated constraints are *added to the correct view automatically*, relieving the programmer from having to determine what view that would be. There is also a method `deactivateConstraints:` which removes constraints from their view. And a constraint has an `active` property; you can set it to activate or deactivate a single constraint, plus it tells you whether a given constraint is part of the interface at this moment.

NSLayoutConstraint properties are read-only, except for `priority` and `constant`. If you want to change anything else about an existing constraint, you must remove the constraint and add a new one.

Once you are using explicit constraints to position and size a view, *do not set its frame* (or bounds and center) subsequently; use constraints alone. Otherwise, when `layoutSubviews` is called, the view will jump back to where its constraints position it. (The exception is that you *may* set a view's frame if you are *in* `layoutSubviews`, as I'll explain later.)

Autoresizing constraints

The mechanism whereby individual views can opt in to autolayout can suddenly involve other views in autolayout, even though those other views were not using autolayout previously. Therefore, there needs to be a way, when such a view becomes involved in autolayout, to determine that view's position and layout through constraints in the same

way they were previously being determined through its frame and its `autoresizingMask`. The runtime takes care of this for you: it translates the view's frame and `autoresizingMask` settings into constraints. The result is a set of implicit constraints, of class NSAutoresizingMaskLayoutConstraint, affecting this view (though they may be attached to a different view). Thanks to these implicit constraints, the layout dictated by the view's `autoresizingMask` continues to work.

For example, suppose I have a UILabel whose frame is `(20.0,20.0,42.0,22.0)`, and whose `autoresizingMask` is `.None`. If this label were suddenly to come under autolayout, then its superview would acquire four implicit constraints setting its width and height at 42 and 22 and its center `x` and `y` at 41 and 31.

This conversion is performed only if the view in question has its `translatesAutoresizingMaskIntoConstraints` property set to `true`. That is, in fact, the default if the view came into existence either in code or by instantiation from a nib where "Use Auto Layout" is not checked. The assumption is that if a view came into existence in either of those ways, you want its frame and `autoresizingMask` to act as its constraints if it becomes involved in autolayout.

That's a sensible rule, but it means that if you intend to apply any explicit constraints of your own to such a view, you'll probably want to remember to turn off this automatic behavior by setting the view's `translatesAutoresizingMaskIntoConstraints` property to `false`. If you don't, you're going to end up with both implicit constraints and explicit constraints affecting this view, and it's unlikely that you would want that. Typically, that sort of situation will result in a conflict between constraints, as I'll explain a little later; indeed, what usually happens to me is that I *don't* remember to set the view's `translatesAutoresizingMaskIntoConstraints` property to `false`, and am reminded to do so only when I *do* get a conflict between constraints.

Creating constraints in code

We are now ready to write some code involving constraints! I'll start by using the NSLayoutConstraint initializer `init(item:attribute:relatedBy:toItem:attribute: multiplier:constant:)`, which sets every property of the constraint as I described them a moment ago (except the `priority`, which defaults to 1000 and can be set later if necessary).

I'll generate the same views and subviews and layout behavior as in Figures 1-12 and 1-13, but using constraints. Observe that I don't bother to assign the subviews `v2` and `v3` explicit frames as I create them, because constraints will take care of positioning them, and that I remember (for once) to set their `translatesAutoresizingMaskIntoConstraints` properties to `false`:

```
let v1 = UIView(frame:CGRectMake(100, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView()
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
let v3 = UIView()
v3.backgroundColor = UIColor(red: 1, green: 0, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
v1.addSubview(v3)
v2.translatesAutoresizingMaskIntoConstraints = false
v3.translatesAutoresizingMaskIntoConstraints = false
v1.addConstraint(
    NSLayoutConstraint(item: v2,
        attribute: .Leading,
        relatedBy: .Equal,
        toItem: v1,
        attribute: .Leading,
        multiplier: 1, constant: 0)
)
v1.addConstraint(
    NSLayoutConstraint(item: v2,
        attribute: .Trailing,
        relatedBy: .Equal,
        toItem: v1,
        attribute: .Trailing,
        multiplier: 1, constant: 0)
)
v1.addConstraint(
    NSLayoutConstraint(item: v2,
        attribute: .Top,
        relatedBy: .Equal,
        toItem: v1,
        attribute: .Top,
        multiplier: 1, constant: 0)
)
v2.addConstraint(
    NSLayoutConstraint(item: v2,
        attribute: .Height,
        relatedBy: .Equal,
        toItem: nil,
        attribute: .NotAnAttribute,
        multiplier: 1, constant: 10)
)
v3.addConstraint(
    NSLayoutConstraint(item: v3,
        attribute: .Width,
        relatedBy: .Equal,
        toItem: nil,
        attribute: .NotAnAttribute,
        multiplier: 1, constant: 20)
)
v3.addConstraint(
```

```
        NSLayoutConstraint(item: v3,
            attribute: .Height,
            relatedBy: .Equal,
            toItem: nil,
            attribute: .NotAnAttribute,
            multiplier: 1, constant: 20)
    )
    v1.addConstraint(
        NSLayoutConstraint(item: v3,
            attribute: .Trailing,
            relatedBy: .Equal,
            toItem: v1,
            attribute: .Trailing,
            multiplier: 1, constant: 0)
    )
    v1.addConstraint(
        NSLayoutConstraint(item: v3,
            attribute: .Bottom,
            relatedBy: .Equal,
            toItem: v1,
            attribute: .Bottom,
            multiplier: 1, constant: 0)
    )
```

Now, I know what you're thinking. You're thinking: "What are you, nuts? That is a boatload of code!" (Except that you probably used another four-letter word instead of "boat.") But that's something of an illusion. I'd argue that what we're doing here is actually *simpler* than the code with which we created Figure 1-12 using explicit frames and autoresizing.

After all, we merely create eight constraints in eight simple commands. (I've broken each command into multiple lines, but that's just a matter of formatting.) They're verbose, but they are the same command repeated with different parameters, so creating them is just a matter of copy-and-paste. Moreover, our eight constraints determine the *position, size, and layout behavior* of our two subviews, so we're getting a lot of bang for our buck.

Even more telling, these constraints are a far clearer expression of what's supposed to happen than setting a frame and `autoresizingMask`. The position of our subviews is described once and for all, both as they will initially appear and as they will appear if their superview is resized. And it is described meaningfully; we don't have to use arbitrary math. Recall what we had to say before:

```
let v3 = UIView(frame:CGRectMake(
    v1.bounds.width-20, v1.bounds.height-20, 20, 20))
```

That business of subtracting the view's height and width from its superview's bounds height and width in order to position the view is confusing and error-prone. With constraints, we can speak the truth directly; our constraints say, plainly and simply, "`v3` is 20 points wide and 20 points high and flush with the bottom-right corner of `v1`."

In addition, of course, constraints can express things that autoresizing can't. For example, instead of applying an absolute height to `v2`, we could require that its height be exactly one-tenth of `v1`'s height, regardless of how `v1` is resized. To do that without constraints, you'd have to implement `layoutSubviews` and enforce it manually, in code.

Anchor notation

New in iOS 9, it's possible to do everything I just did, making exactly the same eight constraints and adding them to the same views, using a much more compact notation. The old notation has the virtue of singularity: one NSLayoutConstraint initializer method can create any constraint. The new notation takes the opposite approach: it concentrates on brevity but sacrifices singularity. To do so, instead of focusing on the constraint, it focuses on the attributes to which the constraint relates. These attributes are expressed as *anchor* properties of a UIView:

- `topAnchor`, `bottomAnchor`
- `leftAnchor`, `rightAnchor`, `leadingAnchor`, `trailingAnchor`
- `centerXAnchor`, `centerYAnchor`
- `firstBaselineAnchor`, `lastBaselineAnchor`

The anchor values are all NSLayoutAnchor instances (some are instances of NSLayoutAnchor subclasses). The constraint-forming methods are then NSLayoutAnchor instance methods, and there are a lot of them, with your choice depending on whether your constraint needs to specify just another anchor (with the `constant` and the `multiplier` defaulting to `0` and `1`), a `constant` or a `multiplier` or both, and upon your choice of relation:

- `constraintEqualToConstant:`
- `constraintGreaterThanOrEqualToConstant:`
- `constraintLessThanOrEqualToConstant:`
- `constraintEqualToAnchor:`
- `constraintGreaterThanOrEqualToAnchor:`
- `constraintLessThanOrEqualToAnchor:`
- `constraintEqualToAnchor:constant:`
- `constraintGreaterThanOrEqualToAnchor:constant:`
- `constraintLessThanOrEqualToAnchor:constant:`
- `constraintEqualToAnchor:multiplier:`
- `constraintGreaterThanOrEqualToAnchor:multiplier:`

- `constraintLessThanOrEqualToAnchor:multiplier:`
- `constraintEqualToAnchor:multiplier:constant:`
- `constraintGreaterThanOrEqualToAnchor:multiplier:constant:`
- `constraintLessThanOrEqualToAnchor:multiplier:constant:`

All of that may sound very elaborate when I describe it, but when you see it in action, you will appreciate immediately the benefit of this compact notation: it's easy to write (especially thanks to Xcode's code completion), easy to read, and easy to maintain. The new notation is particularly convenient in connection with `activateConstraints:`, as we don't have to worry about specifying what view each constraint should be added to:

```
NSLayoutConstraint.activateConstraints([
    v2.leadingAnchor.constraintEqualToAnchor(v1.leadingAnchor),
    v2.trailingAnchor.constraintEqualToAnchor(v1.trailingAnchor),
    v2.topAnchor.constraintEqualToAnchor(v1.topAnchor),
    v2.heightAnchor.constraintEqualToConstant(10),
    v3.widthAnchor.constraintEqualToConstant(20),
    v3.heightAnchor.constraintEqualToConstant(20),
    v3.trailingAnchor.constraintEqualToAnchor(v1.trailingAnchor),
    v3.bottomAnchor.constraintEqualToAnchor(v1.bottomAnchor)
])
```

That's eight constraints in eight lines of code — plus the surrounding `activateConstraints` call to put those constraints into our interface. It isn't strictly necessary to activate all one's constraints at once, but it's best to try to do so.

Visual format notation

Another way to abbreviate your creation of constraints is to use a sort of text-based shorthand, called a *visual format*. This has the advantage of allowing you to describe multiple constraints simultaneously, and is appropriate particularly when you're arranging a series of views horizontally or vertically. I'll start with a simple example:

```
"V:|[v2(10)]"
```

In that expression, `V:` means that the vertical dimension is under discussion; the alternative is `H:`, which is also the default (so it is permitted to specify no dimension). A view's name appears in square brackets, and a pipe (`|`) signifies the superview, so here we're portraying `v2`'s top edge as butting up against its superview's top edge. Numeric dimensions appear in parentheses, and a numeric dimension accompanying a view's name sets that dimension of that view, so here we're also setting `v2`'s height to 10.

To use a visual format, you have to provide a dictionary mapping the string name of each view mentioned to the actual view. For example, the dictionary accompanying the preceding expression might be `["v2":v2]`. So here's another way of expressing of the preceding code example, using the visual format shorthand throughout:

```
let d = ["v2":v2,"v3":v3]
NSLayoutConstraint.activateConstraints([
    NSLayoutConstraint.constraintsWithVisualFormat(
        "H:|[v2]|", options: [], metrics: nil, views: d),
    NSLayoutConstraint.constraintsWithVisualFormat(
        "V:|[v2(10)]", options: [], metrics: nil, views: d),
    NSLayoutConstraint.constraintsWithVisualFormat(
        "H:[v3(20)]|", options: [], metrics: nil, views: d),
    NSLayoutConstraint.constraintsWithVisualFormat(
        "V:[v3(20)]|", options: [], metrics: nil, views: d)
].flatten().map{$0})
```

That example creates the same constraints as the previous example, but in four commands instead of eight. (The `constraintsWithVisualFormat:` method yields an array of constraints, so my literal array is an array of arrays of constraints. But `activateConstraints` expects an array of constraints, so I flatten my literal array.)

The visual format syntax shows itself to best advantage when multiple views are laid out in relation to one another along the same dimension; in that situation, you get a lot of bang for your buck (many constraints generated by one visual format string). The syntax, however, is somewhat limited in what constraints it can readily express; it conceals the number and exact nature of the constraints that it produces; and personally I find it easier to make a mistake with the visual format syntax than with the complete expression of each constraint. Still, you'll want to become familiar with the visual format syntax, not least because console messages describing a constraint sometimes use it.

Here are some further things to know when generating constraints with the visual format syntax:

- The `metrics:` parameter is a dictionary with numeric values. This lets you use a name in the visual format string where a numeric value needs to go.
- The `options:` parameter is a bitmask (NSLayoutFormatOptions) chiefly letting you do things like add alignments. The alignments you specify are applied to all the views mentioned in the visual format string.
- To specify the distance between two successive views, use hyphens surrounding the numeric value, like this: `"[v1]-20-[v2]"`. The numeric value may optionally be surrounded by parentheses.
- A numeric value in parentheses may be preceded by an equality or inequality operator, and may be followed by an at sign with a priority. Multiple numeric values, separated by comma, may appear in parentheses together. For example: `"[v1(>=20@400,<=30)]"`.

For formal details of the visual format syntax, see the "Visual Format Syntax" chapter of Apple's *Auto Layout Guide*.

In Objective-C, you can form a dictionary for mapping view names to view references more or less automatically, thanks to the `NSDictionaryOfVariableBindings` macro; for example, `NSDictionaryOfVariableBindings(v2,v3)` yields the Objective-C equivalent of the dictionary `["v2":v2,"v3":v3]` that we formed manually in the preceding code. But Swift lacks macros; there's no preprocessor, so the textual transformation needed to generate a literal dictionary from a literal list of view variable names is impossible. For an alternative, see the `dictionaryOfNames` utility function in Appendix B.

Constraints as objects

Although the examples so far have involved creating constraints and adding them directly to the interface — and then forgetting about them — it is frequently useful to form constraints and keep them on hand for future use (typically in a property). A common use case is where you intend, at some future time, to change the interface in some radical way, such as by inserting or removing a view; you'll probably find it convenient to keep multiple sets of constraints on hand, each set being appropriate to a particular configuration of the interface. It is then trivial to swap constraints out of and into the interface along with views that they affect.

In this example, we create within our main view (`mainview`) three views, `v1`, `v2`, and `v3`, which are red, yellow, and blue rectangles respectively. We keep strong references (as properties) to all three views. For some reason, we will later want to remove the yellow view (`v2`) dynamically as the app runs, moving the blue view to where the yellow view was; and then, still later, we will want to insert the yellow view once again (Figure 1-14). So we have two alternating view configurations. Therefore we create *two* sets of constraints, one describing the positions of `v1`, `v2`, and `v3` when all three are present, the other describing the positions of `v1` and `v3` when `v2` is absent. For purposes of maintaining these sets of constraints, we have already prepared two properties, `constraintsWith` and `constraintsWithout`, initialized as empty arrays of NSLayoutConstraint:

```
var constraintsWith = [NSLayoutConstraint]()
var constraintsWithout = [NSLayoutConstraint]()
```

Here's the code for creating the views and the two sets of constraints. We start with `v2` present, so it is the first set of constraints that we initially make active:

```
let v1 = UIView()
v1.backgroundColor = UIColor.redColor()
v1.translatesAutoresizingMaskIntoConstraints = false
let v2 = UIView()
v2.backgroundColor = UIColor.yellowColor()
v2.translatesAutoresizingMaskIntoConstraints = false
let v3 = UIView()
v3.backgroundColor = UIColor.blueColor()
```

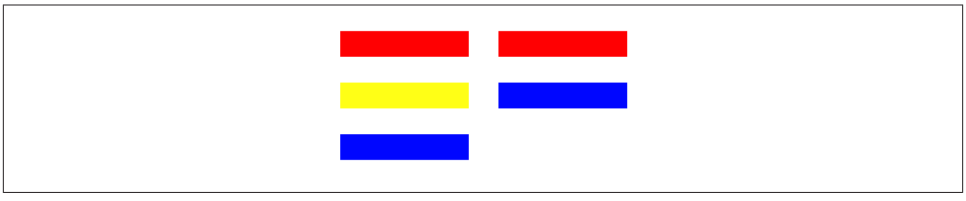

Figure 1-14. Alternate sets of views and constraints

```
v3.translatesAutoresizingMaskIntoConstraints = false
mainview.addSubview(v1)
mainview.addSubview(v2)
mainview.addSubview(v3)
self.v1 = v1
self.v2 = v2
self.v3 = v3
// construct constraints
let c1 = NSLayoutConstraint.constraintsWithVisualFormat(
    "H:|-(20)-[v(100)]", options: [], metrics: nil, views: ["v":v1])
let c2 = NSLayoutConstraint.constraintsWithVisualFormat(
    "H:|-(20)-[v(100)]", options: [], metrics: nil, views: ["v":v2])
let c3 = NSLayoutConstraint.constraintsWithVisualFormat(
    "H:|-(20)-[v(100)]", options: [], metrics: nil, views: ["v":v3])
let c4 = NSLayoutConstraint.constraintsWithVisualFormat(
    "V:|-(100)-[v(20)]", options: [], metrics: nil, views: ["v":v1])
let c5with = NSLayoutConstraint.constraintsWithVisualFormat(
    "V:[v1]-(20)-[v2(20)]-(20)-[v3(20)]", options: [], metrics: nil,
    views: ["v1":v1, "v2":v2, "v3":v3])
let c5without = NSLayoutConstraint.constraintsWithVisualFormat(
    "V:[v1]-(20)-[v3(20)]", options: [], metrics: nil,
    views: ["v1":v1, "v3":v3])
// first set of constraints
self.constraintsWith.appendContentsOf(c1)
self.constraintsWith.appendContentsOf(c2)
self.constraintsWith.appendContentsOf(c3)
self.constraintsWith.appendContentsOf(c4)
self.constraintsWith.appendContentsOf(c5with)
// second set of constraints
self.constraintsWithout.appendContentsOf(c1)
self.constraintsWithout.appendContentsOf(c3)
self.constraintsWithout.appendContentsOf(c4)
self.constraintsWithout.appendContentsOf(c5without)
// apply first set
NSLayoutConstraint.activateConstraints(self.constraintsWith)
```

All that preparation may seem extraordinarily elaborate, but the result is that when the time comes to swap `v2` out of or into the interface, swapping the appropriate constraints is trivial:

```
if self.v2.superview != nil {
    self.v2.removeFromSuperview()
    NSLayoutConstraint.deactivateConstraints(self.constraintsWith)
    NSLayoutConstraint.activateConstraints(self.constraintsWithout)
} else {
    mainview.addSubview(v2)
    NSLayoutConstraint.deactivateConstraints(self.constraintsWithout)
    NSLayoutConstraint.activateConstraints(self.constraintsWith)
}
```

Guides and margins

So far, I've been assuming that the attributes (anchors) between which you want to create constraints are, for the most part, the exact edges and centers of views, and that these will stay put. But that's not always the case. Sometimes, you want a view to vend some *other* anchor, a kind of *secondary* anchor, to which another view can be constrained. It is even possible that you'll want this anchor to be able to move, and thus move the constrained view along with it. This notion of a secondary anchor has evolved by accretion over the past several iterations of iOS, so that now it is expressed in several different ways.

Consider, to begin with, the business of pinning your subviews to the bottom and (especially) the top of a view controller's main view. The top and bottom of the interface are often occupied by a bar (status bar, navigation bar, toolbar, tab bar — see Chapter 12). Your layout of subviews will typically occupy the region *between* these bars. Since iOS 7, however, the main view can extend vertically to the edges of the window *behind* those bars. Moreover, such bars can come and go dynamically, and can change their heights; for example, since iOS 8 the default behavior has been for the status bar to vanish when an iPhone app is in landscape orientation, and a navigation bar is taller when an iPhone app is in portrait orientation than when the same app is in landscape orientation.

Therefore, you need something else, other than the *literal* top and bottom of a view controller's main view, to which to anchor the vertical constraints that position its subviews — something that will move vertically to reflect the current location of the bars. Otherwise, an interface that looks right under some circumstances will look wrong in others.

For example, consider a view whose top is literally constrained to the top of the view controller's main view, which is its superview:

```
let arr = NSLayoutConstraint.constraintsWithVisualFormat(
    "V:|-0-[v]", options: [], metrics: nil, views: ["v":v])
```

When the app is in landscape orientation, with the status bar removed by default, this view will be right up against the top of the screen, which is fine. But in portrait orientation, this view will *still* be right up against the top of the screen — which looks bad because the status bar reappears and overlaps it.

To solve this problem, UIViewController supplies and maintains two invisible views, the *top layout guide* and the *bottom layout guide*, which it injects as subviews into the view hierarchy of its main view. Your topmost and bottommost vertical constraints will usually not be between a subview and the top or bottom of the main view, but between a subview and the bottom of the top layout guide, or a subview and the top of the bottom layout guide. The bottom of the top layout guide matches the bottom of the lowest top bar, or the top of the main view if there is no top bar; the top of the bottom layout guide matches the top of the bottom bar, or the bottom of the main view if there is no bottom bar. Most important, these layout guides change their size as the situation changes — the top or bottom bar changes its height, or vanishes entirely — and so your views constrained to them move to track the edges of the main view's visible area.

You can access these layout guides programmatically through the UIViewController properties `topLayoutGuide` and `bottomLayoutGuide`. For example (this code is in a view controller, so the top layout guide is `self.topLayoutGuide`):

```
let arr = NSLayoutConstraint.constraintsWithVisualFormat(
    "V:[tlg]-0-[v]", options: [], metrics: nil,
    views: ["tlg":self.topLayoutGuide, "v":v])
```

In iOS 9, the `topLayoutGuide` has a `bottomAnchor` and the `bottomLayoutGuide` has a `topAnchor`:

```
let tlg = self.topLayoutGuide
let c = v.topAnchor.constraintEqualToAnchor(tlg.bottomAnchor)
```

In iOS 8, views acquired *margins*. These are things you can constrain to that are *inset* from the edge of a UIView. The main idea, clearly, is that you might want subviews to keep a minimum standard distance from the edge of the superview. Thus, a UIView has a `layoutMargins` property which is a UIEdgeInsets, a struct consisting of four floats representing inset values starting at the top and proceeding counterclockwise — top, left, bottom, right. The default for a view controller's main view is a top and bottom margin of 0 and a right and left margin of 16; for any other view, it's 8 for all four margins.

You can constrain things to these margins. A visual format string that pins a subview's edge to its superview's edge, expressed as a pipe character (`|`), using *a single hyphen with no explicit distance value*, is interpreted as a constraint to the superview's margin. Thus, for example, here's a view that's butting up against its superview's left margin:

```
let arr = NSLayoutConstraint.constraintsWithVisualFormat(
    "H:|-[v]", options: [], metrics: nil, views: ["v":v])
```

As an NSLayoutAttribute value, a view's margin is expressed as one of the following:

- `.TopMargin`, `.BottomMargin`
- `.LeftMargin`, `.RightMargin`, `.LeadingMargin`, `.TrailingMargin`

- `.CenterXWithinMargins`, `.CenterYWithinMargins`

So here's another way to form the same constraint, with a view placed against its superview's left margin:

```
let c = NSLayoutConstraint(item: v,
    attribute: .Leading,
    relatedBy: .Equal,
    toItem: mainview,
    attribute: .LeadingMargin,
    multiplier: 1, constant: 0)
```

In the compact anchor notation, use the view's `layoutMarginsGuide` property (new in iOS 9). It is a UILayoutGuide object (another iOS 9 innovation), which itself has anchors just like a UIView. So here's the same constraint once more:

```
let c = v.leadingAnchor.constraintEqualToAnchor(
    mainview.layoutMarginsGuide.leadingAnchor)
```

An additional UIView property, `preservesSuperviewLayoutMargins`, if `true`, causes a view to adopt as its `layoutMargins` the intersection of its own and its superview's `layoutMargins`. To put it another way, the superview's `layoutMargins` are allowed to supervene if the subview overlaps them. For example, suppose the view `v` has default layout margins `{8,8,8,8}`. And suppose its superview, `mainview`, is the view controller's main view, and has the default layout margins `{0,16,0,16}`. And suppose `v` is constrained exactly to the literal edges of its superview `mainview` — it covers it completely. Then if `v`'s `preservesSuperviewLayoutMargins` is `true`, its effective layout margins are `{8,16,8,16}`.

iOS 9 introduces a second set of margins, a UIView's `readableContentGuide` (a UILayoutGuide), which you cannot change. The idea is that a subview consisting of text should not be allowed to grow as wide as an iPad in landscape, because that's too wide to read easily. By constraining such a subview horizontally to its superview's `readableContentGuide`, you ensure that that won't happen.

Finally, we come to a major iOS 9 innovation: it permits you to add your own custom UILayoutGuide objects to a view. They constitute a view's `layoutGuides` array, and are managed by calling `addLayoutGuide:` or `removeLayoutGuide:`. Such custom layout guide objects must be configured entirely using constraints. A UILayoutGuide has anchors, but that's effectively all it has. Thus, it can participate in layout as if it were a subview, but it is *not* a subview, and therefore it avoids all the overhead and complexity that a UIView would have.

Why is that useful? Well, consider the question of how to distribute views equally within their superview. This is easy to arrange initially, but it is not obvious how to design evenly spaced views that will remain evenly spaced when their superview is resized. The problem is that constraints describe relationships between *views*, not between *con-*

straints; there is no way to constrain the spacing constraints between views to remain equal to one another automatically as the superview is resized.

You can, on the other hand, constrain the heights or widths of *views* to remain equal to one another. The traditional solution, therefore, has been to resort to spacer views with their `hidden` set to `true`. But spacer views are views; hidden or not, they add overhead with respect to drawing, memory, touch detection, and more. Custom UILayoutGuides solve the problem; they can serve the same purpose as spacer views, but they are *not* views.

Suppose, for example, that I have four views that are to remain equally distributed vertically. I constrain their left and right edges, their heights, and the top of the first view and the bottom of the last view. This leaves open the question of how we will determine the vertical position of the two middle views; they must move in such a way that they are always equidistant from their vertical neighbors.

To solve the problem, I introduce three UILayoutGuide objects between my real views. A custom UILayoutGuide object is added to a UIView, so I'll add mine to the superview of my four real views:

```
let guides = [UILayoutGuide(), UILayoutGuide(), UILayoutGuide()]
for guide in guides {
    mainview.addLayoutGuide(guide)
}
```

I then involve my three layout guides in the layout. Remember, they must be configured entirely using constraints (the three layout guides are referenced through my `guides` array, and the four views are referenced through another array, `views`):

```
NSLayoutConstraint.activateConstraints([
    // guide left is arbitrary, let's say superview margin ❶
    guides[0].leadingAnchor.constraintEqualToAnchor(mainview.leadingAnchor),
    guides[1].leadingAnchor.constraintEqualToAnchor(mainview.leadingAnchor),
    guides[2].leadingAnchor.constraintEqualToAnchor(mainview.leadingAnchor),
    // guide widths are arbitrary, let's say 10
    guides[0].widthAnchor.constraintEqualToConstant(10),
    guides[1].widthAnchor.constraintEqualToConstant(10),
    guides[2].widthAnchor.constraintEqualToConstant(10),
    // bottom of each view is top of following guide ❷
    views[0].bottomAnchor.constraintEqualToAnchor(guides[0].topAnchor),
    views[1].bottomAnchor.constraintEqualToAnchor(guides[1].topAnchor),
    views[2].bottomAnchor.constraintEqualToAnchor(guides[2].topAnchor),
    // top of each view is bottom of preceding guide
    views[1].topAnchor.constraintEqualToAnchor(guides[0].bottomAnchor),
    views[2].topAnchor.constraintEqualToAnchor(guides[1].bottomAnchor),
    views[3].topAnchor.constraintEqualToAnchor(guides[2].bottomAnchor),
    // guide heights are equal! ❸
    guides[1].heightAnchor.constraintEqualToAnchor(guides[0].heightAnchor),
    guides[2].heightAnchor.constraintEqualToAnchor(guides[0].heightAnchor),
])
```

❶ I constrain the leading edges of the layout guides (arbitrarily, to the leading edge of their superview) and their widths (arbitrarily).

❷ I constrain each layout guide to the bottom of the view above it and the top of the view below it.

❸ Finally, our whole purpose is to distribute our views *equally*, so the heights of our layout guides must be *equal to one another*.

In that code, I clearly could have (and should have) generated each group of constraints as a loop, thus making this approach suitable for any number of distributed views; I have deliberately unrolled those loops for the sake of the example.

In real life, you are unlikely to use this technique directly, because you will use a UIStackView instead, and let the UIStackView generate all of that code — as I will explain a little later.

Intrinsic content size and alignment rects

Some built-in interface objects, when using autolayout, have an inherent size in one or both dimensions. For example:

- A UIButton, by default, has a standard height, and its width is determined by its title.
- A UIImageView, by default, adopts the size of the image it is displaying.
- A UILabel, by default, if it consists of multiple lines and if its width is constrained, adopts a height sufficient to display all of its text.

This inherent size is the object's *intrinsic content size*. The intrinsic content size is used to generate constraints implicitly (of class NSContentSizeLayoutConstraint).

A change in the characteristics or content of a built-in interface object — a button's title, an image view's image, a label's text or font, and so forth — may thus cause its intrinsic content size to change. This, in turn, may alter your layout. You will want to configure your autolayout constraints so that your interface responds gracefully to such changes.

You do not have to supply explicit constraints configuring a dimension of a view whose intrinsic content size configures that dimension. But you might! And when you do, the tendency of an interface object to size itself to its intrinsic content size must not be allowed to conflict with its tendency to obey your explicit constraints. Therefore the constraints generated from a view's intrinsic content size have a lowered priority, and come into force only if no constraint of a higher priority prevents them. The following methods allow you to access these priorities (the parameter is a UILayoutConstraintAxis, either `.Horizontal` or `.Vertical`):

`contentHuggingPriorityForAxis:`
: A view's resistance to growing larger than its intrinsic size in this dimension. In effect, there is an inequality constraint saying that the view's size in this dimension should be less than or equal to its intrinsic size. The default prority is usually 250 (the same as `UILayoutPriorityDefaultLow`), though some interface classes will default to 251 if initialized in a nib.

`contentCompressionResistancePriorityForAxis:`
: A view's resistance to shrinking smaller than its intrinsic size in this dimension. In effect, there is an inequality constraint saying that the view's size in this dimension should be greater than or equal to its intrinsic size. The default priority is usually 750 (the same as `UILayoutPriorityDefaultHigh`).

Those methods are getters; there are corresponding setters. Situations where you would need to change the priorities of these tendencies are few, but they do exist. For example, here are the visual formats configuring two adjacent labels (`lab1` and `lab2`) pinned to the superview and to one another:

```
"V:|-20-[lab1]"
"V:|-20-[lab2]"
"H:|-20-[lab1]"
"H:[lab2]-20-|"
"H:[lab1(>=100)]-(>=20)-[lab2(>=100)]"
```

The inequalities ensure that as the superview becomes narrower or the text of the labels becomes longer, a reasonable amount of text will remain visible in both labels. At the same time, one label will be squeezed down to 100 points width, while the other label will be allowed to grow to fill the remaining horizontal space. The question is: which label is which? You need to answer that question. To do so, it suffices to raise the compression resistance priority of one of the labels by a single point above that of the other:

```
let p = lab2.contentCompressionResistancePriorityForAxis(.Horizontal)
lab1.setContentCompressionResistancePriority(p+1, forAxis: .Horizontal)
```

You can supply an intrinsic size in your own custom UIView subclass by implementing `intrinsicContentSize`. Obviously you should do this only if your view's size depends on its contents. If you need the runtime to call `intrinsicContentSize` again, because that size has changed and the view needs to be laid out afresh, it's up to you to call your view's `invalidateIntrinsicContentSize` method.

Another question with which your custom UIView subclass might be concerned is what it should mean for another view to be aligned with it. It might mean aligned with your view's frame edges, but then again it might not. A possible example is a view that draws, internally, a rectangle with a shadow; you probably want to align things with that drawn rectangle, not with the outside of the shadow. To determine this, you can override your view's `alignmentRectInsets` method (or, more elaborately, its `alignmentRectForFrame:` and `frameForAlignmentRect:` methods).

By the same token, you may want to be able to align your custom UIView with another view by their baselines. The assumption here is that your view has a subview containing text and, therefore, possessing a baseline. Your custom view will return that subview in its implementation of `viewForFirstBaselineLayout` or `viewForLastBaselineLayout`.

Stack views

A stack view (UIStackView), new in iOS 9, is a view whose primary task is to generate constraints for some or all of its subviews. These are its *arranged subviews*. In particular, a stack view solves the problem of providing constraints when subviews are to be configured linearly in a horizontal row or a vertical column. In practice, it turns out that the vast majority of complex layouts can be expressed as an arrangement, possibly nested, of simple rows and columns of subviews. Thus, you are likely to resort to stack views to make your layout easier to construct and maintain.

Configuration of a stack view is extremely simple and yet remarkably powerful. First, you supply it with arranged subviews, usually by calling its initializer `init(arrangedSubviews:)`. The arranged subviews become the stack view's `arrangedSubviews` read-only property. You can also manage the arranged subviews with these methods:

- `addArrangedSubview:`
- `insertArrangedSubview:atIndex:`
- `removeArrangedSubview:`

The `arrangedSubviews` is different from, but is a subset of, the stack view's `subviews`. To put it another way, it's perfectly fine for the stack view to have subviews that are *not* arranged (and which you'll have to provide with constraints yourself), but any subview that *is* arranged must in fact be a subview plain and simple as well, and if you set a view as an arranged subview and it is *not* already a subview, the stack view will adopt it as a subview at that moment. The *order* of the `arrangedSubviews` is independent of the order of the `subviews`; the `subviews` order, you remember, determines the order in which the subviews are drawn, but the `arrangedSubviews` order determines how the stack view will *position* those subviews.

You will also want to set the following properties of the stack view:

`axis`
: Which way should the arranged subviews be arranged? Your choices are:

  - `.Horizontal`
  - `.Vertical`

`distribution`
: How should the arranged subviews be positioned along the `axis`? Your choices are:

- `.Fill`
- `.FillEqually`
- `.FillProportionally`
- `.EqualSpacing`
- `.EqualCentering`

Except for `.FillEqually`, the exact interpretation of your instructions may depend upon the intrinsic content sizes of the views. For example, `.FillProportionally` fills the full `axis` dimension, sizing the views in proportion to their intrinsic content sizes in this dimension. The stack view's `spacing` property determines the spacing (or minimum spacing) between the views.

`alignment`
: This describes how the arranged subviews should be laid out with respect to the *other* dimension. Your choices are:

  - `Fill`
  - `Leading` (or `Top`)
  - `Center`
  - `Trailing` (or `Bottom`)
  - `FirstBaseline` or `LastBaseline` (if the `axis` is `.Horizontal`)

  Again, except in the case of `.Fill`, the interpretation here will depend upon the views having intrinsic content size (or baselines). If the `axis` is `.Vertical`, you can still involve the subviews' baselines in their spacing by setting the stack view's `baselineRelativeArrangement` to `true`.

`layoutMarginsRelativeArrangement`
: If `true`, the stack view's internal `layoutMargins` are involved in the constraints of its arranged subviews. If `false` (the default), the stack view's literal edges are used.

Finally, note that although you will not have to constrain the arranged views — it is exactly the job of the stack view to do that — you *will* have to constrain the stack view *itself* (though of course this, too, might be done automatically because the stack view is an arranged view of a containing stack view).

To illustrate, I'll rewrite the code from earlier in this chapter. I have four views, with height constraints. I want to distribute them vertically in my main view. This time, I'll have a stack view do all the work for me:

```
// give the stack view arranged subviews
let sv = UIStackView(arrangedSubviews: views)
// configure the stack view
sv.axis = .Vertical
sv.alignment = .Fill
sv.distribution = .EqualSpacing
// constrain the stack view
sv.translatesAutoresizingMaskIntoConstraints = false
mainview.addSubview(sv)
NSLayoutConstraint.activateConstraints([
    sv.topAnchor.constraintEqualToAnchor(self.topLayoutGuide.bottomAnchor),
    sv.leadingAnchor.constraintEqualToAnchor(mainview.leadingAnchor),
    sv.trailingAnchor.constraintEqualToAnchor(mainview.trailingAnchor),
    sv.bottomAnchor.constraintEqualToAnchor(mainview.bottomAnchor),
])
```

Inspecting the resulting constraints, you can see that the stack view is doing for us effectively just what we did earlier (generating UILayoutGuide objects and using them as spacers). But letting the stack view do it is a lot easier!

Another nice feature of UIStackView is that it responds intelligently to changes. For example, if we were to make one of our arranged subviews invisible (set its `hidden` to `true`), the stack view would respond by distributing the remaining subviews evenly, as if the hidden subview didn't exist. Similarly, we can change properties of the stack view itself in real time. (Moreover, all such changes are animatable, as I'll explain in Chapter 4.)

Internationalization

New in iOS 9, the entire interface and its behavior are reversed when the app runs on a system for which the app is localized and whose language is right-to-left. Wherever you use leading and trailing constraints instead of left and right constraints, or if your constraints are constructed using the visual format language, your app's layout will participate in this reversal more or less automatically.

There may, however, be exceptions. Apple gives the example of a horizontal row of transport controls that mimic the buttons on a CD player: you wouldn't want the Rewind button and the Fast Forward button to be reversed just because the user's language reads right-to-left. Therefore, a UIView is endowed with a `semanticContentAttribute` property stating whether it should be flipped; the default is `.Unspecified`, but a value of `.Playback` or `.Spatial` will prevent flipping.

If you are constructing a view's subviews in code in real time, you can feed a subview's `semanticContentAttribute` value to the UIView class method `userInterfaceLayoutDirectionForSemanticContentAttribute:` to find out whether directionality is `.LeftToRight` or `.RightToLeft`.

Mistakes with constraints

Creating constraints manually, as I've been doing so far in this chapter, is an invitation to make a mistake. Your totality of constraints constitute instructions for view layout, and it is all too easy, as soon as more than one or two views are involved, to generate faulty instructions. You can (and will) make two major kinds of mistake with constraints:

Conflict
: You have applied constraints that can't be satisfied simultaneously. This will be reported in the console (at great length).

Underdetermination (ambiguity)
: A view uses autolayout, but you haven't supplied sufficient information to determine its size and position. This is a far more insidious problem, because nothing bad may seem to happen, so you might not discover it until much later. If you're lucky, the view will at least fail to appear, or will appear in an undesirable place, alerting you to the problem.

Only required constraints (priority 1000) can contribute to a conflict, as the runtime is free to ignore lower-priority constraints that it can't satisfy. Constraints with different priorities do not conflict with one another. Nonrequired constraints with the same priority can contribute to ambiguity.

Let's start by generating a conflict. In this example, we return to our small red square in the lower right corner of a big purple square (Figure 1-12) and append a contradictory constraint:

```
NSLayoutConstraint.activateConstraints([
    // ...
    NSLayoutConstraint.constraintsWithVisualFormat(
        "H:[v3(20)]|", options: [], metrics: nil, views: d),
    NSLayoutConstraint.constraintsWithVisualFormat(
        "V:[v3(20)]|", options: [], metrics: nil, views: d),
    NSLayoutConstraint.constraintsWithVisualFormat(
        "V:[v3(10)]|", options: [], metrics: nil, views: d), // *
].flatten().map{$0})
```

The height of `v3` can't be both 10 and 20. The runtime reports the conflict, and tells you which constraints are causing it:

```
Unable to simultaneously satisfy constraints. Probably at least one of the
constraints in the following list is one you don't want...

    "<NSLayoutConstraint:0x7f7fabc10750 V:[UIView:0x7f7fabc059d0(20)]>",
    "<NSLayoutConstraint:0x7f7fabc10d10 V:[UIView:0x7f7fabc059d0(10)]>"
```

You can assign a constraint (or a UILayoutGuide) an `identifier` string; this can make it easier to determine which constraint in a conflict report is which.

Now we'll generate an ambiguity. Here, we neglect to give our small red square a height:

```
NSLayoutConstraint.activateConstraints([
    // ...
    NSLayoutConstraint.constraintsWithVisualFormat(
        "H:[v3(20)]|", options: [], metrics: nil, views: d),
].flatten().map{$0})
```

No console message alerts us to our mistake. Fortunately, however, `v3` fails to appear in the interface, so we know something's wrong. *If your views fail to appear, suspect ambiguity*. In a less fortunate case, the view might appear, but (if we're lucky) in the wrong place. In a truly unfortunate case, the view might appear in the right place, but not consistently.

Suspecting ambiguity is one thing; tracking it down and proving it is another. A useful trick is to pause in the debugger and give the following mystical command in the console:

```
(lldb) expr -l objc++ -O -- [[UIWindow keyWindow] _autolayoutTrace]
```

The result is a graphical tree describing the view hierarchy and marking any ambiguously laid out views:

```
UIWindow:0x7fe8d0d9dbd0
|   •UIView:0x7fe8d0c2bf00
|   |   +UIView:0x7fe8d0c2c290
|   |   |   *UIView:0x7fe8d0c2c7e0
|   |   |   *UIView:0x7fe8d0c2c9e0- AMBIGUOUS LAYOUT
```

UIView also has a `hasAmbiguousLayout` method; I find it useful to set up a utility method that lets me check a view and all its subviews at any depth for ambiguity:

```
extension NSLayoutConstraint {
    class func reportAmbiguity (var v:UIView?) {
        if v == nil {
            v = UIApplication.sharedApplication().keyWindow
        }
        for vv in v!.subviews {
            print("\(vv) \(vv.hasAmbiguousLayout())")
            if vv.subviews.count > 0 {
                self.reportAmbiguity(vv)
            }
        }
    }
}
```

You can call that method in your code, or while paused in the debugger:

```
(lldb) expr NSLayoutConstraint.reportAmbiguity(nil)
```

To get a full list of the constraints responsible for positioning a particular view within its superview, log the results of calling the UIView instance method `constraintsAffectingLayoutForAxis:`. These constraints do not necessarily belong to this view (and the output doesn't tell you what view they do belong to). Your choices of axis

(UILayoutConstraintAxis) are `.Horizontal` and `.Vertical`. If a view doesn't participate in autolayout, the result will be an empty array. Again, a utility method can come in handy:

```
extension NSLayoutConstraint {
    class func listConstraints (var v:UIView?) {
        if v == nil {
            v = UIApplication.sharedApplication().keyWindow
        }
        for vv in v!.subviews {
            let arr1 = vv.constraintsAffectingLayoutForAxis(.Horizontal)
            let arr2 = vv.constraintsAffectingLayoutForAxis(.Vertical)
            NSLog("\n\n%@\nH: %@\nV:%@", vv, arr1, arr2)
            if vv.subviews.count > 0 {
                self.listConstraints(vv)
            }
        }
    }
}
```

And here's how to call it from the debugger:

```
(lldb) expr NSLayoutConstraint.listConstraints(nil)
```

Xcode's view debugging feature can also be a great help (Figure 1-15). With the app running, choose Debug → View Debugging → Capture View Hierarchy, or click the Debug View Hierarchy button in the debug toolbar. At the left, the Debug navigator lists your window and all its views hierarchically, along with their constraints. What's more, when a view is selected in this list or in the canvas, the Size inspector at the right lists its bounds and all the constraints that determine those bounds. This, along with the layered graphical display of your views and constraints in the canvas, is very likely to help you penetrate to the cause of any constraint-related difficulties.

Given the notions of conflict and ambiguity, we can understand what priorities are for. Imagine that all constraints have been placed in boxes, where each box is a priority value, in descending order. The first box (1000) contains all the required constraints, so all required constraints are obeyed first. (If they conflict, that's bad, and a report appears in the log; meanwhile, the system implicitly lowers the priority of one of the conflicting constraints, so that it doesn't have to obey it and can continue with layout by satisfying the remaining required constraints.) If there still isn't enough information to perform unambiguous layout given the required priorities alone, we pull the constraints out of the next box and try to obey them. If we can, consistently with what we've already done, fine; if we can't, or if ambiguity remains, we look in the *next* box — and so on. For a box after the first, we don't care about obeying exactly the constraints it contains; if an ambiguity remains, we can use a lower-priority constraint value to give us something to aim at, resolving the ambiguity, without fully obeying the lower-priority constraint's desires. For example, an inequality is an ambiguity, because an infinite number of values

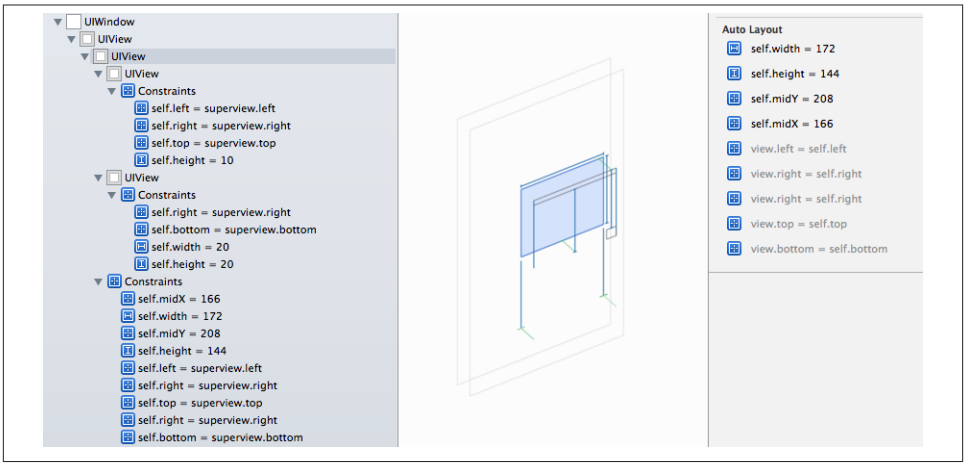

Figure 1-15. View debugging

will satisfy it; a lower-priority equality can tell us what value to prefer, resolving the ambiguity, but there's no conflict even if we can't fully achieve that preferred value.

Configuring Layout in the Nib

The focus of the discussion so far has been on configuring layout in code. This, however, will often be unnecessary; instead, you'll set up your layout (autoresizing or autolayout) in the nib, using the nib editor (Interface Builder). It would not be strictly true to say that you can do absolutely anything in the nib that you could do in code, but the nib editor is certainly a remarkably powerful way of configuring layout (and where it falls short, you can always supplement it with some code in addition).

Autoresizing in the nib

To configure autoresizing in the nib editor, you'll need to ensure that autolayout is turned off for the *.storyboard* or *.xib* file you're editing. To do so, select that file in the Project navigator and show the File inspector: uncheck "Use Auto Layout."

When editing a nib file with autolayout turned off, you can assign a view springs and struts in the Size inspector. A solid line externally represents a strut; a solid line internally represents a spring. A helpful animation shows you the effect on your view's position as its superview is resized.

Constraints in the nib

In a *.xib* or *.storyboard* file where "Use Auto Layout" is checked, a vast array of tools springs to life in the nib editor to help you create constraints that will be instantiated

from the nib along with the views. What's more, the nib editor will help prevent you from ending up with conflicting or ambiguous constraints.

Even in a nib with "Use Auto Layout" checked, the nib editor *doesn't generate any constraints* unless you ask it to. However, it doesn't want the app to run with ambiguous layout, because then you might not see any views at all; you wouldn't be able to test your app until you'd fully worked out all the constraints throughout the interface. Therefore, if your views lack needed constraints, the nib supplies them implicitly behind the scenes so that they are present at runtime:

No constraints
: If a view is affected by no constraints at all, it is given constraints tagged in the debugger as "IB auto generated at build time for view with fixed frame."

Ambiguous constraints
: If a view is affected by some constraints but not enough to disambiguate fully, it is given additional constraints tagged in the debugger as "IB auto generated at build time for view with ambiguity."

The nib editor also *doesn't change any constraints* unless you ask it to. If you create constraints and then move or resize a view affected by those constraints, the constraints are *not* automatically changed. This means that the constraints no longer match the way the view is portrayed; if the constraints were to position the view, they wouldn't put it where you've put it. The nib editor will alert you to this situation (a Misplaced Views issue), and can readily resolve it for you, but it won't do so unless you explicitly ask it to.

Creating a constraint

The nib editor provides two primary ways to create a constraint:

- Control-drag from one view to another. A HUD appears, listing constraints that you can create (Figure 1-16). Either view can be in the canvas or in the document outline. To create an internal width or height constraint, control-drag from a view to itself! When you control-drag within the canvas, the direction of the drag is used to winnow the options presented in the HUD; for example, if you control-drag horizontally within a view in the canvas, the HUD lists Width but not Height.
- Choose from the Editor → Align or Editor → Pin hierarchical menus, or click the Align or Pin button at the right end of the layout bar below the canvas.

The buttons in the layout bar are very powerful! They present little popover dialogs where you can choose multiple constraints to create (possibly for multiple views, if that's what you've selected beforehand) and provide them with numeric values (Figure 1-17). Constraints are not actually added until you click Add Constraints at the bottom. Before clicking Add Constraints, think about the Update Frames pop-up menu; if you don't

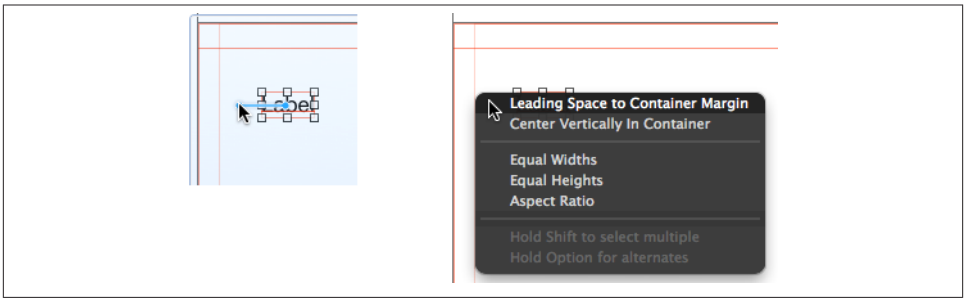

Figure 1-16. Creating a constraint by control-dragging

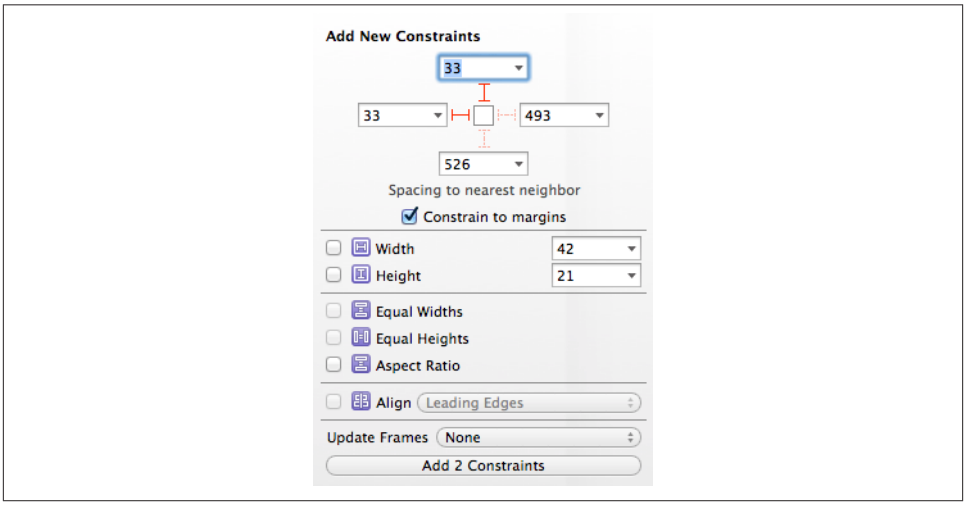

Figure 1-17. Creating constraints from the layout bar

update frames, the views may end up being drawn in the canvas differently from how the constraints describe them (a Misplaced Views issue).

The nib editor displays layout margins as faint lines in the canvas. (The faint vertical line at the left of Figure 1-16 is a margin.) When you Control-drag to form a constraint from a view to its superview, you might want to toggle the Option key to see some alternatives; for example, this might make the difference between an edge-based constraint and a margin-based constraint. Similarly, the popover dialog from the Pin button in the layout bar has a "Constrain to margins" checkbox (Figure 1-17). Finally, when editing a constraint (as I describe in the next section), the First Item and Second Item pop-up menus have a "Relative to margin" option.

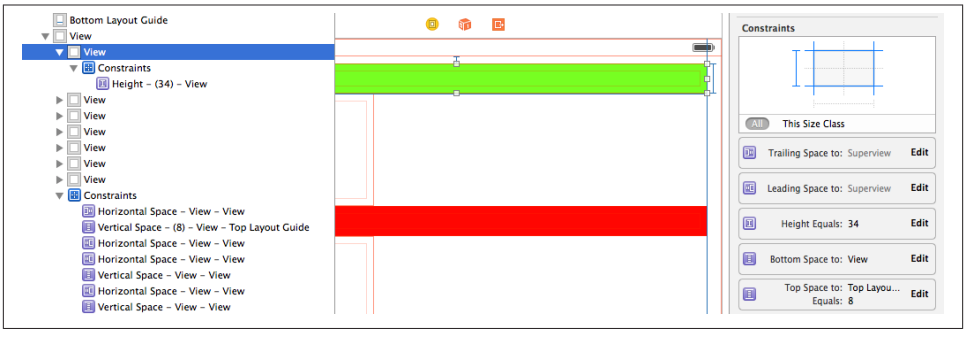

Figure 1-18. A view's constraints displayed in the nib

To set a view's layout margins explicitly, switch to the Size inspector and change the Layout Margins pop-up menu to Explicit. To make a view's layout margins behave as `readableContentGuide` margins, check Follow Readable Width.

In Xcode 7, it is possible to create a constraint in the nib editor between a subview and the *literal* top or bottom of a view controller's main view — something that was absolutely impossible in earlier versions of Xcode. When Control-dragging from a subview to the main view, the default is a constraint to the top layout guide or the bottom layout guide, but holding Option lets you make a constraint to the margin. You can then edit the resulting constraint to switch off "Relative to margin."

Viewing and editing constraints

Constraints in the nib are full-fledged objects. They can be selected, edited, and deleted. Moreover, you can create an outlet to a constraint (and there are reasons why you might want to do so).

Constraints in the nib are visible in three places (Figure 1-18):

In the document outline
: Constraints are listed in a special category, "Constraints," under the view to which they belong. (You'll have a much easier time distinguishing these constraints if you give your views meaningful labels!)

In the canvas
: Constraints appear graphically as dimension lines when you select a view that they affect.

In the Size inspector
: When a view affected by constraints is selected, the Size inspector lists those constraints, and a Constraints grid displays the view's constraints graphically.

When you select a constraint in the document outline or the canvas, you can view and edit its values in the Attributes or Size inspector. The inspector gives you access to almost all of a constraint's features: both anchors involved in the constraint, the relation, the constant and multiplier, and the priority. You can also set the identifier here (useful when debugging, as I mentioned earlier).

Alternatively, for simple editing of a constraint's constant, relation, and priority, double-click the constraint in the canvas to summon a little popover dialog. Similarly, when a constraint is listed in a view's Size inspector, double-click it to edit it in its own inspector, or click its Edit button to summon the little popover dialog.

A view's Size inspector also provides access to its content hugging and content compression resistance priority settings. Beneath these, there's an Intrinsic Size pop-up menu. The idea here is that your custom view might have an intrinsic size, but the nib editor doesn't know this, so it will report an ambiguity when you fail to provide (say) a width constraint that you know isn't actually needed; choose Placeholder to supply an intrinsic size and relieve the nib editor's worries (and to prevent the missing constraints from being generated automatically at runtime).

In a constraint's Attributes or Size inspector, there is a Placeholder checkbox ("Remove at build time"). If you check this checkbox, the constraint you're editing *won't* be instantiated when the nib is loaded, and it will *not* be replaced by an automatically generated constraint: in effect, you are deliberately generating ambiguous layout when the views and constraints are instantiated from the nib. Why might you want to do such a thing? One reason might be that you intend to substitute, in code, a constraint that you weren't quite able to configure in the nib. You need *some* constraint in the nib, because otherwise the nib editor will complain of ambiguity and will generate a constraint anyway at load time.

Problems with constraints

I've already said that generating constraints manually, in code, is error-prone. The nib editor, however, knows whether it contains problematic constraints. If a view is affected by any constraints, the Xcode nib editor will permit them to be ambiguous or conflicting, but it will also complain helpfully. You should pay attention to such complaints! The nib editor will bring the situation to your attention in various places:

Canvas
: Constraints drawn in the canvas when you select a view that they affect use color coding to express their status:

  Satisfactory constraints
  : Drawn in blue.

  Problematic constraints
  : Drawn in red.

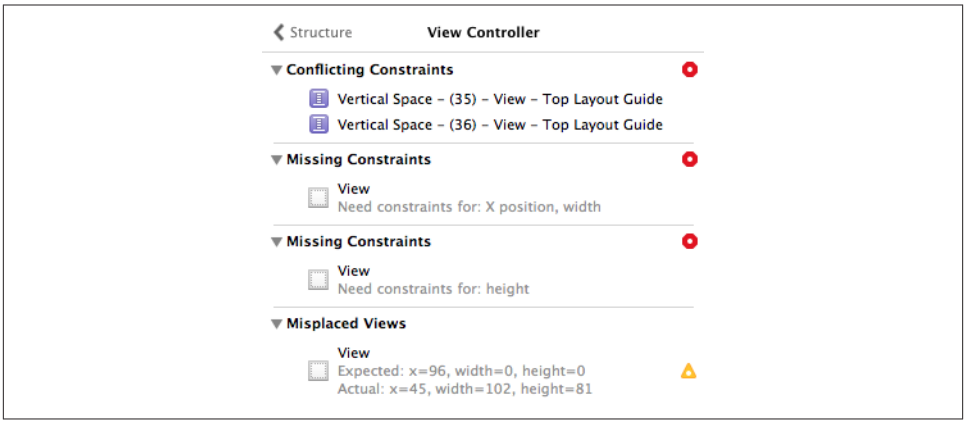

Figure 1-19. Layout issues in the document outline

Misplacement constraints
: Drawn in orange; these constraints are valid, but they are inconsistent with the frame you have imposed upon the view. I'll discuss misplaced views in the next paragraph.

Document outline
: If there are layout issues, the document outline displays a right arrow in a red or orange circle. Click it to see a detailed list of the issues (Figure 1-19). Hover the mouse over a title to see an Info button which you can click to learn more about the nature of this issue. The icons at the right are buttons: click one for a list of things the nib editor is offering to do to fix the issue for you. The chief issues are:

Conflicting Constraints
: A conflict between constraints.

Missing Constraints
: Ambiguous layout.

Misplaced Views
: If you manually change the frame of a view that is affected by constraints (including its intrinsic size), then the nib editor canvas may be displaying that view differently from how it would really appear if the current constraints were obeyed. A Misplaced Views situation is also reflected in the canvas:

  - The constraints in the canvas, drawn in orange, display the numeric *difference* between their values and the view's frame.
  - A dotted outline in the canvas may show where the view would be drawn if the existing constraints were obeyed.

A hierarchical menu, Editor → Resolve Auto Layout Issues, also available from the Resolve Auto Layout Issues button in the layout bar, proposes five large-scale moves involving *all* the constraints affecting selected views or all views:

Update Frames
: Changes the way the view is drawn in the canvas, to show how it would really appear in the running app under the constraints as they stand. Be careful: if constraints are ambiguous, *this can cause a view to disappear*.

  Alternatively, if you have resized a view with intrinsic size constraints, such as a button or a label, and you want it to resume the size it would have according to those intrinsic size constraints, select the view and choose Editor → Size to Fit Content.

Update Constraints
: Choose this menu item to change numerically all the existing constraints affecting a view to match the way the canvas is currently drawing the view's frame.

Add Missing Constraints
: Create new constraints so that the view has sufficient constraints to describe its frame unambiguously. The added constraints correspond to the way the canvas is currently drawing the view's frame.

  Not everything that this command does may be what you ultimately want; you should regard it as a starting point. For example, the nib editor doesn't know whether you think a certain view's width should be determined by an internal width constraint or by pinning it to the left and right of its superview; and it may generate alignment constraints with other views that you never intended.

Reset to Suggested Constraints
: This is as if you chose Clear Constraints followed by Add Missing Constraints: it removes all constraints affecting the view, and replaces them with a complete set of automatically generated constraints describing the way the canvas is currently drawing the view's frame.

Clear Constraints
: Removes all constraints affecting the view.

Conditional constraints

Constraints and views can be made conditional in the nib editor. The conditions on which they depend are the size classes that I discussed earlier. This means that you can design your interface's constraints, and even the presence or absence of views, to depend on the size of the screen. In effect, the interface detects `traitCollectionDidChange:` and responds to it. Thus:

- You can design directly into your interface a complex rearrangement of that interface when an iPhone app rotates to compensate for a change in device orientation.
- A single *.storyboard* or *.xib* file can be used to design the interface of a universal app, even though the iPad interface and the iPhone interface may be quite different from one another.

Use of conditional constraints is an opt-in feature of the nib editor. You have opted in if "Use Size Classes" is checked in the File inspector for this *.storyboard* or *.xib*. In that case, the following nib editor interface features are present:

- The main view in the canvas, by default, is portrayed as a square, to suggest the dimension-agnostic nature of the design process.
- The center of the layout bar, below the canvas, acquires a pop-up menu where you can choose any combination of size classes. Your choices are Compact, Regular, and (between them) Any, so the choice is represented as a 3×3 grid.

The idea is that you design your interface initially for the general case (Any width, Any height, which is the default). You then use the pop-up grid to switch to a specific case — *the layout bar turns blue* to indicate that you're in specific-case mode — and modify the design for that case. Do *not* impulsively start moving interface items around! Instead, use the inspectors:

- Use a view's Attributes inspector to determine whether that view is present for a particular size class combination. Note the Installed checkbox! Initially, it applies to the general case. Click the Plus button, at its left, to add *another* Installed checkbox applicable to a particular set of size classes. Now you can check or uncheck Installed checkboxes so that this view is present for some size class combinations but removed for others.
- Use a constraint's Attributes or Size inspector to determine:
  - Whether that constraint is present for a particular size class combination. There is an Installed checkbox, which works just like the Installed checkbox for a view.
  - The value of that constraint's constant for a particular size class combination. The Constant field has a Plus button, at its left, just like the Installed checkbox; click it to add *another* Constant field applicable to a particular set of size classes.

A constraint or view that is not installed for the set of size classes you're looking at is listed in gray in the document outline.

In Xcode 7, other features of a view can be conditional on the size class. For example, the font of a button, label, or text field, and *every* feature of a stack view, can be different for different size classes. Keep your eyes peeled for that telltale Plus button in the Attributes or Size inspector.

View Debugging, Previewing, and Designing

Xcode has several features for helping you visualize and understand your view hierarchy and the effect of your constraints. This section calls attention to some of these.

View debugger

To enter the view debugger, choose Debug → View Debugging → Capture View Hierarchy, or click the Debug View Hierarchy button in the debug bar. The result is that your app's current view hierarchy is analyzed and displayed (Figure 1-15):

- On the left, in the Debug navigator, the views and their constraints are listed hierarchically.
- At the top, the jump bar shows the same hierarchy in a different way, and helps you navigate it.
- In the center, in the canvas, the views and their constraints are displayed graphically. The window starts out facing front, much as if you were looking at the screen with the app running; but if you swipe sideways a little in the canvas, the window rotates and its subviews are displayed in front of it, in layers. The slider at the lower left changes the distance between these layers. The double-slider at the lower right lets you eliminate the display of views from the front or back of the layering order (or both). You can switch to wireframe mode. You can display constraints for the currently selected view.
- On the right, the Object inspector and the Size inspector tell you details about the currently selected object (view or constraint).

Previewing your interface

When you're displaying the nib editor in Xcode, the assistant pane's Tracking menu (the first component in its jump bar, Control-4) includes the Preview option. Choose it to see a preview of the currently selected view controller's view (or, in a *.xib* file, the top-level view). The Plus button at the lower left lets you add previews for different devices and device sizes. At the bottom of each preview, a Rotate button lets you switch this preview to the other orientation.

The previews take account of constraints, including conditional constraints. At the lower right, a language pop-up menu lets you switch your app's text (buttons and labels)

to another language for which you have localized your app, or to an artificial "double-length" language.

Designable views and inspectable properties

Your view can appear more or less correctly in the nib editor canvas and preview *even if it is configured in code*. To take advantage of this feature, you need a UIView subclass declared `@IBDesignable`:

```
@IBDesignable class MyView: UIView {
    // ... your code goes here ...
}
```

If an instance of this UIView subclass appears in the nib, then its self-configuration methods, such as `init(coder:)` and `willMoveToSuperview:`, will be compiled and run as the nib editor prepares to portray your view. For example, if your view's `init(coder:)` method adds a UILabel as a subview of this view, then the nib editor will show that label.

In addition, your view can implement `prepareForInterfaceBuilder` to perform visual configurations aimed specifically at how it will be portrayed in the nib editor. For example, if your view contains a UILabel that is created and configured empty but will eventually contain text, you could implement `prepareForInterfaceBuilder` to give the label some sample text to be displayed in the nib editor.

In Figure 1-20, the nib editor displays a MyView instance; the green and red subviews come from MyView's initializer, and the purple background is added in `prepareForInterfaceBuilder`.

You can also configure custom view properties directly in the nib editor. If your UIView subclass has a property whose value type is understood by the nib editor, and if this property is declared `@IBInspectable`, then if an instance of this UIView subclass appears in the nib, that property will get a field of its own at the top of the view's Attributes inspector. Thus, when a custom UIView subclass is to be instantiated from the nib, its custom properties can be set in the nib editor rather than having to be set in code. (This feature is actually a convenient equivalent of setting a nib object's User Defined Runtime Attributes in the Identity inspector.)

Inspectable property types are: Bool, number, String, CGRect, CGPoint, CGSize, UIColor, or UIImage — or an Optional wrapping any of these. You can assign a default value in code; Interface Builder won't portray this value as the default, but you can tell Interface Builder to use the default by leaving the field empty (or, if you've entered a value, by deleting that value).

In Figure 1-20, the nib editor displays MyView's custom `name` property.

`@IBDesignable` and `@IBInspectable` are unrelated, but the former is aware of the latter. Thus, you can use an inspectable property to change the nib editor's display of your

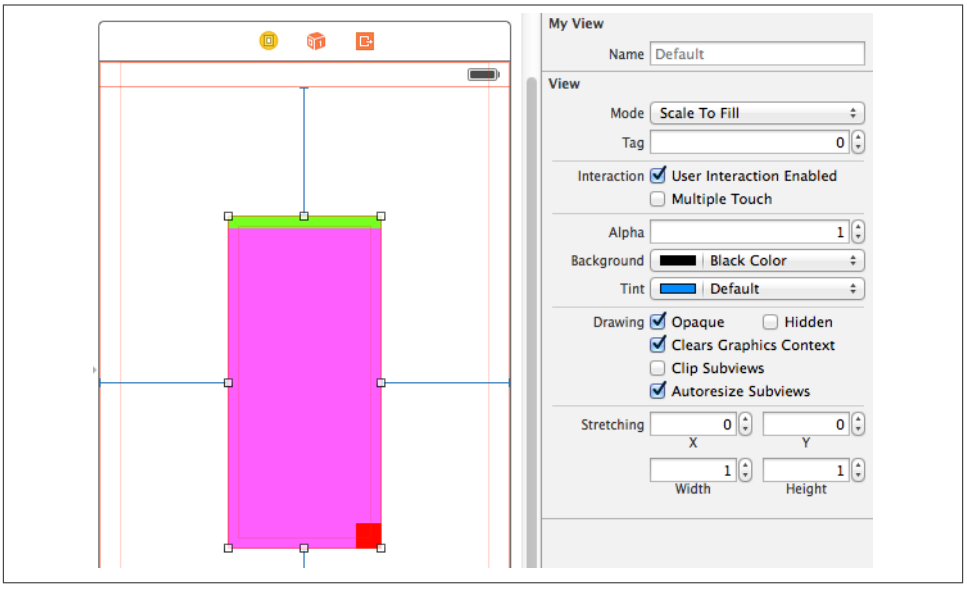

Figure 1-20. A designable view with an inspectable property

interface in real time. For example, if my view is `@IBDesignable`, and if its `prepareForInterfaceBuilder` creates and adds to the interface a label whose text is its `name` property, and if `name` is `@IBInspectable`, then if I change the `name` property in the nib editor (Figure 1-20), the label's text changes in the nib editor canvas or preview.

View Events Related to Layout

This section summarizes the chief UIView events related to layout. These are events that you can receive and respond to by overriding them in your UIView subclass. You might want to do this in situations where layout is complex — for example, when you need to supplement autoresizing or autolayout with manual layout in code, or when your autoresizing or autolayout configuration itself needs to change in response to changing conditions.

These UIView events are not the same as the layout-related events you can receive and respond to in a UIViewController. I'll discuss those in Chapter 6.

At launch time, and if the environment's trait collection changes thereafter, the `traitCollectionDidChange:` message is propagated *down* the hierarchy of UITraitEnvironments.

Thus, if your interface needs to respond to a change in the trait collection — by changing constraints, adding or removing subviews, or what have you — an override of `traitCollectionDidChange:` is the place to do it. For example, earlier in this chapter I showed some code for swapping a view into or out of the interface together with the entire set of constraints laying out that interface. But I left open the matter of the conditions under which we wanted such swapping to occur; `traitCollectionDidChange:` might be an appropriate moment.

If your interface involves autolayout and constraints, then `updateConstraints` is propagated *up* the hierarchy, starting at the deepest subview. This event may be omitted for a view if its constraints have not changed, but it will certainly be called for the view at the top of the hierarchy.

You might override `updateConstraints` in a UIView subclass if your subclass is capable of altering its own constraints and you need a signal that now is the time to do so. You must call `super` or the app will crash (with a helpful error message).

You should never call `updateConstraints` directly. To trigger an immediate call to `updateConstraints`, send a view the `updateConstraintsIfNeeded` message. But `updateConstraints` may still not be sent unless constraints have changed or the view is at the top of the hierarchy. To force `updateConstraints` to be sent to a view, send it the `setNeedsUpdateConstraints` message.

Finally, we come to layout itself. Layout can be triggered even if the trait collection didn't change; for example, perhaps a constraint was changed, or the text of a label was changed, or a superview's size changed. The `layoutSubviews` message is propagated *down* the hierarchy, starting at the top (typically the root view) and working down to the deepest subview.

You can override `layoutSubviews` in a UIView subclass in order to take a hand in the layout process. If you're not using autolayout, `layoutSubviews` does nothing by default; `layoutSubviews` is your opportunity to perform manual layout after autoresizing has taken place. If you are using autolayout, you must call `super` or the app will crash (with a helpful error message).

You should never call `layoutSubviews` directly; to trigger an immediate call to `layoutSubviews`, send a view the `layoutIfNeeded` message (which may cause layout of the entire view tree, not only below but also above this view), or send `setNeedsLayout` to trigger a call to `layoutSubviews` later on, after your code finishes running, when layout would normally take place.

When you're using autolayout, what happens in `layoutSubviews`? The runtime examines all the constraints affecting this view's subviews, works out values for their center and bounds, and assigns those views those center and bounds values. In other words, `layoutSubviews` performs manual layout! The constraints are merely instructions

attached to the views; `layoutSubviews` reads them and responds accordingly, sizing and positioning views in the good old-fashioned way, by setting their frames, bounds, and centers. (Thus, `layoutSubviews` is a place where it is legal — and indeed necessary — to set the size and position of a view governed by explicit constraints.)

Knowing this, you might override `layoutSubviews` when you're using autolayout, in order to tweak the outcome. A typical structure is: first you call `super`, causing all the subviews to adopt their new frames; then you examine those frames; if you don't like the outcome, you can change things; and finally you call `super` *again*, to get a new layout outcome. Keep in mind, however, that you are cooperating with an elaborate existing layout operation that is already in train. Do not call `setNeedsUpdateConstraints` (that moment has passed), and do not stray beyond the subviews *of this view*.

It is also possible to *simulate* layout of a view in accordance with its constraints and those of its subviews. This is useful for discovering ahead of time what a view's size would be if layout were performed at this moment. Send the view the `systemLayoutSizeFittingSize:` message. The system will attempt to reach or at least approach the size you specify, at a very low priority; mostly likely you'll specify either `UILayoutFittingCompressedSize` or `UILayoutFittingExpandedSize`, depending on whether what you're after is the smallest or largest size the view can legally attain. You can dictate the individual axis priorities explicitly (`systemLayoutSizeFittingSize:withHorizontalFittingPriority:verticalFittingPriority:`). I'll show an example in Chapter 7.

Unless you explicitly demand immediate layout, layout isn't performed until your code finishes running (and then only if needed). Moreover, ambiguous layout isn't ambiguous until layout actually takes place. Thus, for example, it's perfectly reasonable to cause an ambiguous layout temporarily, provided you resolve the ambiguity before `layoutSubviews` is called. On the other hand, a conflicting constraint conflicts the instant it is added.