CS275
Fall 2020

iOS: Views, View Hierarchy

BNRG CHAPTER 3

Topics

- iOS versions
- iOS UI Basics
- UIView and UIView Hierarchy
- Constraints

iOS Version

- iOS moved in 2019 to version 13
- When we build an iOS app, we must specify the iOS version of the target device
- This leads to a question: which version of iOS should we support?
  - for fun and play, it's OK to use the current version
  - for real apps (which could include apps produced for a stakeholder in CS275), the current version might not be the right answer
  - one rule of thumb: support current version and current version minus 1
  - here's a good blog post: https://www.avanderlee.com/workflow/minimum-ios-version/

iOS Version

- Which version of iOS to use?
- One thing that makes this difficult is that Apple made some changes to app plumbing between iOS 12 and iOS 13
  - the safest thing is for your code to check which version of iOS it's running on:

```
if #available(iOS 13, *) {
    // do the cool new thing
} else {
    // do things the old-fashioned way
}
```

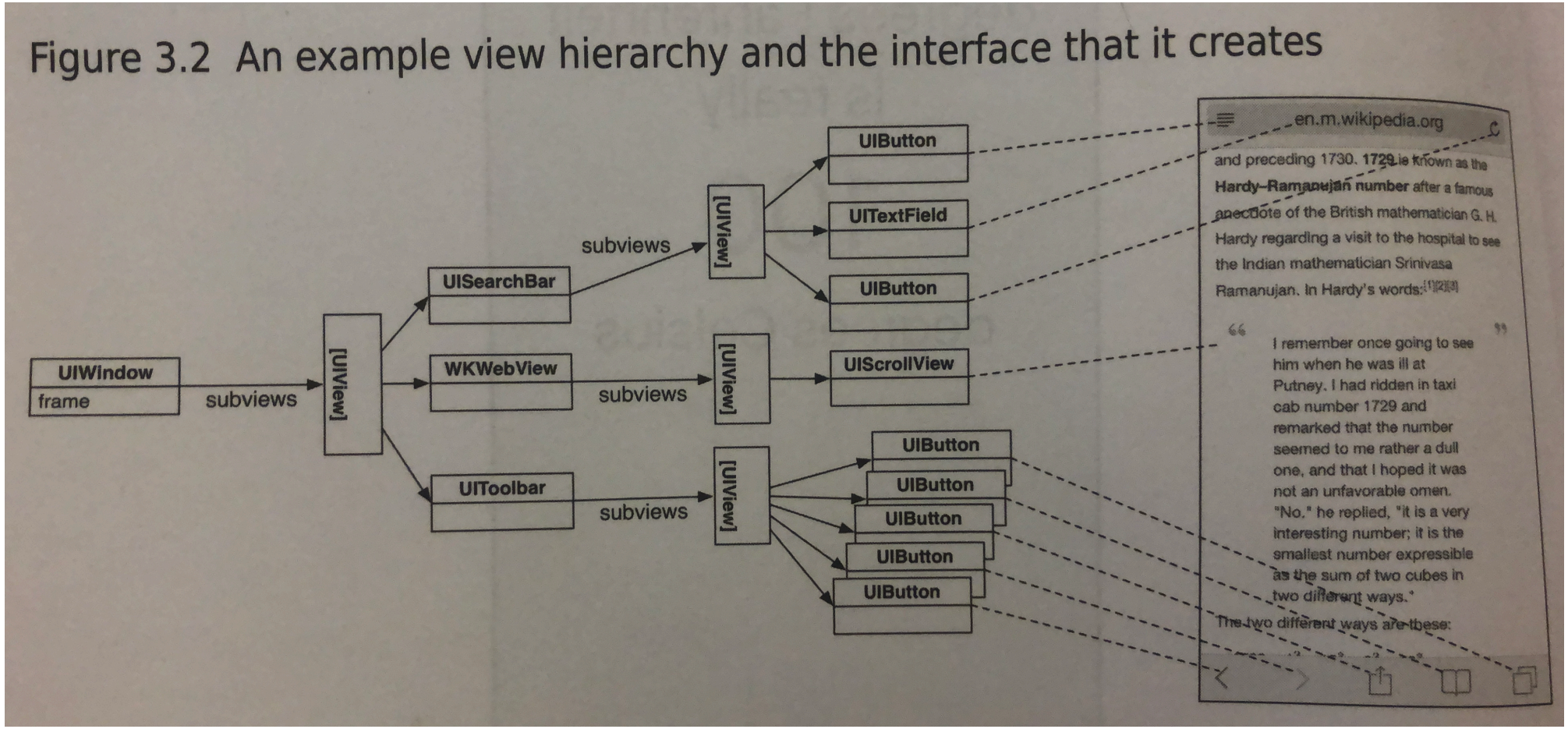
View

- A View is an object that is visible to the user, such as a button, label, slider, etc.
- A View is an instance of `UIView` or of one of its subclasses
- A View can respond to user-generated events, such as touches
- A View exists within a hierarchy of views whose root is the application's window
- A View knows how to draw itself

View Hierarchy

- Every application has a single instance of `UIWindow`
- `UIWindow` is a subclass of `UIView`
- The window is created when the application launches
- This `UIWindow` instance serves as a container for all of the views in the app
- Once the window has been created, other views can be added to it
- Views that have been added to the window are subviews
- Subviews can have subviews

View Hierarchy

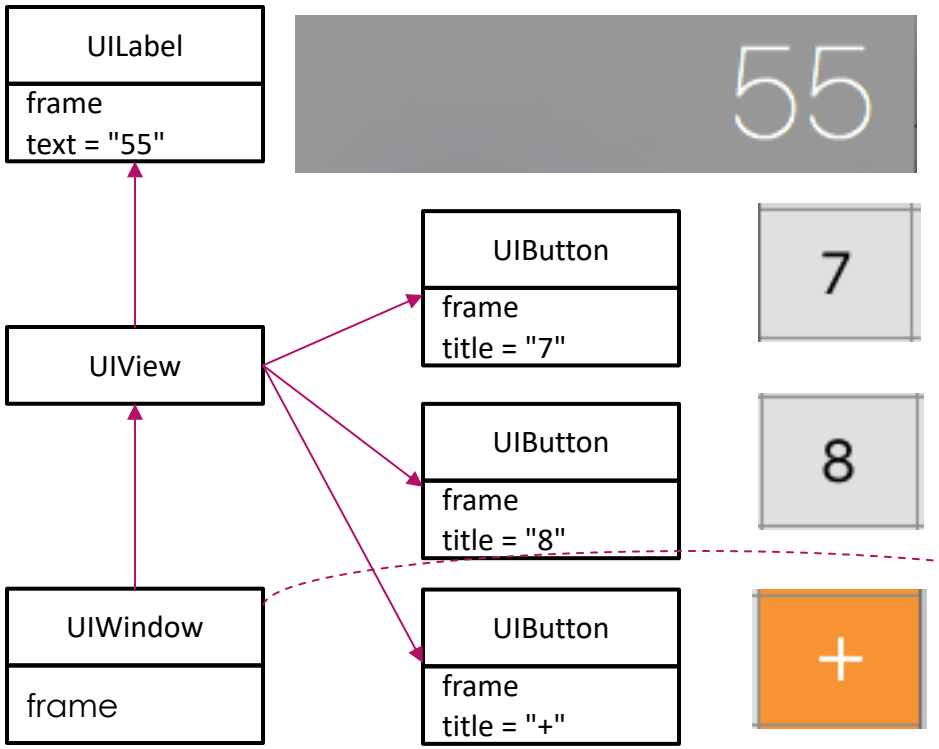


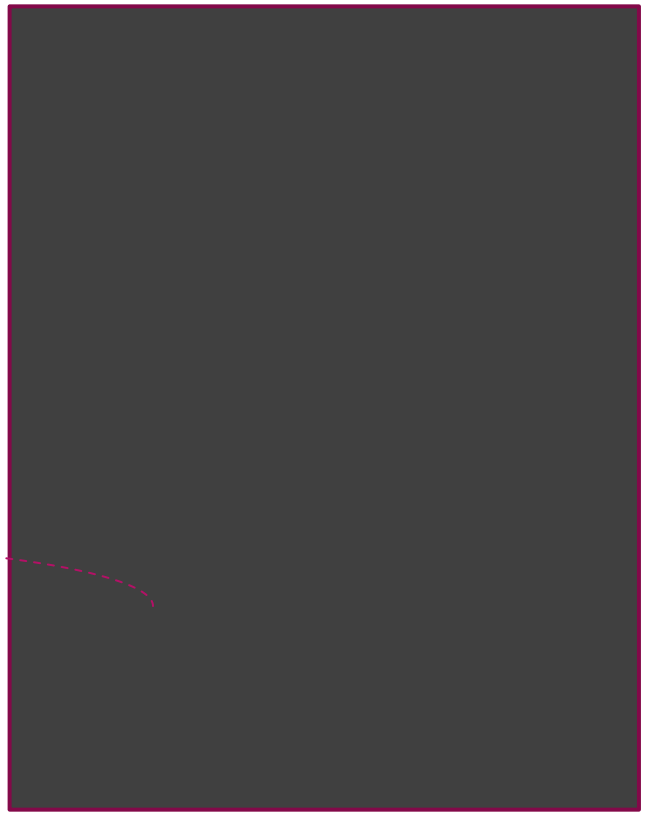
View Hierarchy

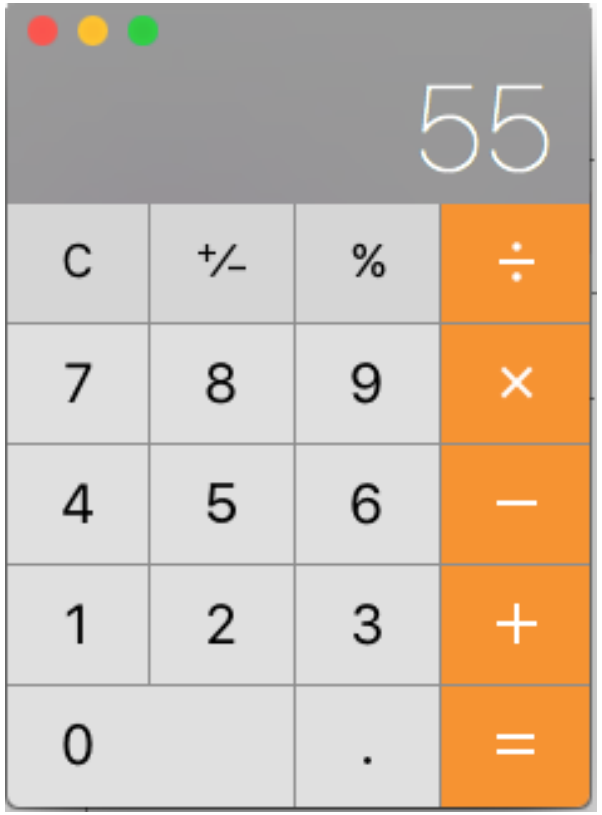
Once the view hierarchy has been created, it will be drawn to the screen:

1. each view in the hierarchy draws itself; it renders itself to its layer, which is like a bitmap image that represents the view
2. the layers of all of the views are composited (i.e. combined) together on the screen

View Hierarchy Example







all of these are children of the UIWindow

Frames

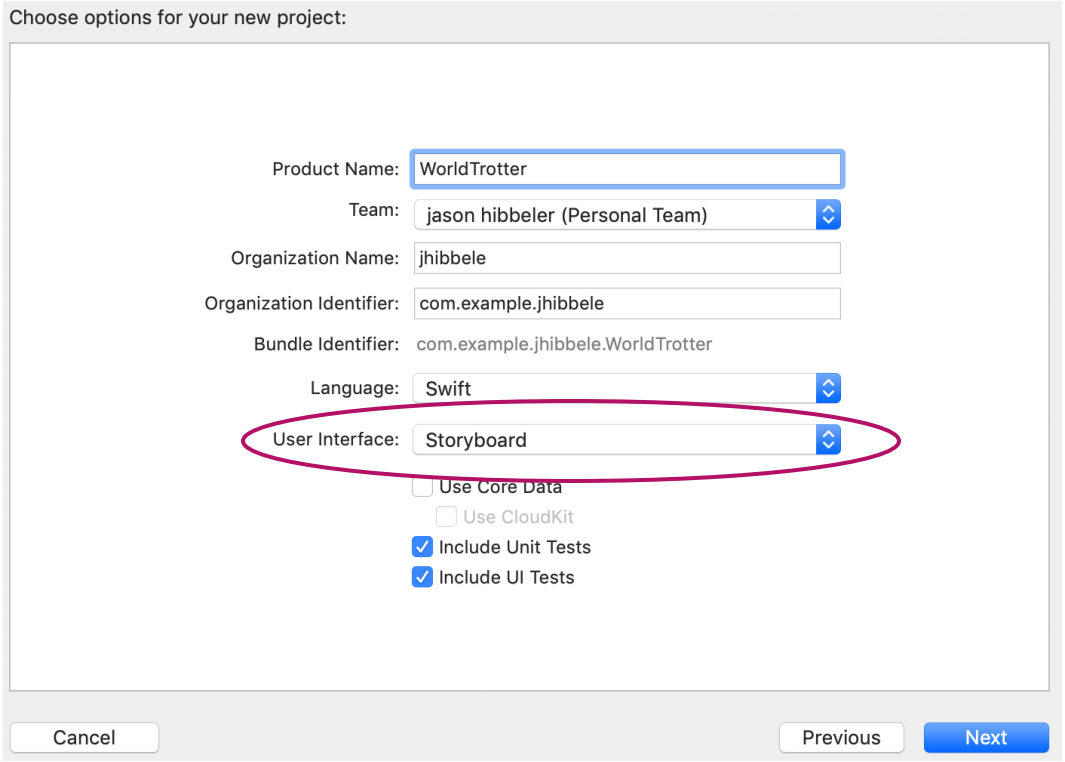
- A view's frame specifies the size of the view (height, width) and its location relative to its superview (x, y)
- To create a view in code, you must specify its frame, using `init(frame:)`
- The view's frame is a property of type `CGRect`
- `CGrect` has two properties
  - `CGPoint`, which contains two `CGFloat` properties: `x` and `y`
  - `CGSize`, which contains two `CGFloat` properties: width and height

Frame and View

- So when we create a view from code, we explicitly provide the frame for the view
- We then add the view to an existing view, which becomes its parent
- When we create a view through Interface Builder, the view still has a frame, and it is initialized when the view is created

WorldTrotter: New Project

- Next step in the book is to create a new project: WorldTrotter
- Specify iOS -> Single View App
- Specify Language as Swift and User Interface as StoryBoard
- For practice: leave the "Create Git repository on my Mac" box checked



WorldTrotter

This app will be used to illustrate several iOS concepts

- text input and text editing
- the software keyboard
- constraints
- navigation, using a tab-bar controller
- programmatic views (creating views from code)

App Initialization

Here's the sequence of steps that occur when an app starts:

1. The user launches an app from the home screen
2. iOS calls a certain hook function in the app's code
3. Each app designates a starting point—the "initial view controller"—and the UI system (Cocoa Touch) puts the `UIView` corresponding to that view controller on the screen
4. The UI framework then calls a function in the view controller that we can overload: `viewDidLoad()`

viewDidLoad()

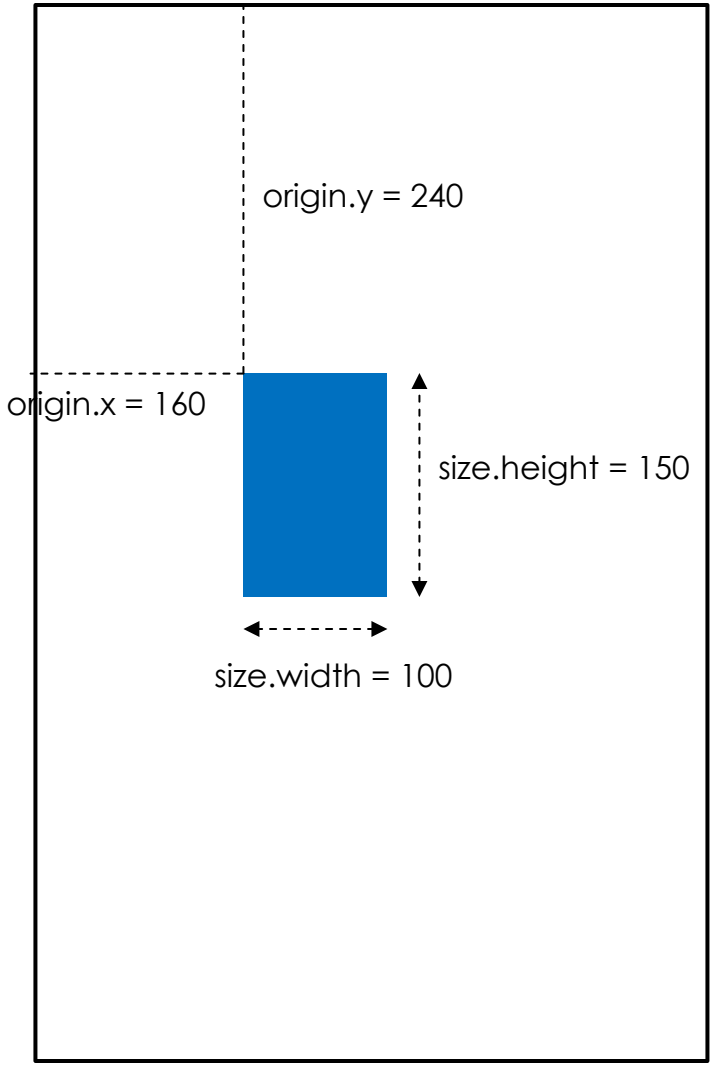
- The function `viewDidLoad()` in the root view controller is an entry point for an iOS app after it's been loaded and is visible
- This gives us a hook to modify the appearance of the app
- For example:

```
override func viewDidLoad() {
    super.viewDidLoad()
    let firstFrame = CGRect(x: 160, y: 240, width: 100, height: 150)
    let firstView = UIView(frame: firstFrame)
    firstView.backgroundColor = UIColor.blue
    view.addSubview(firstView)
}
```

this creates a blue rectangle of the specified size, at the specified position, when the starting view for the app appears

Result

- The origin of the new view is relative to its superview (the top-level view for the app)
- The dimensions are in points--not pixels
- Points are independent of the pixel size
- Pixel size is dependent on the display resolution



A Second View

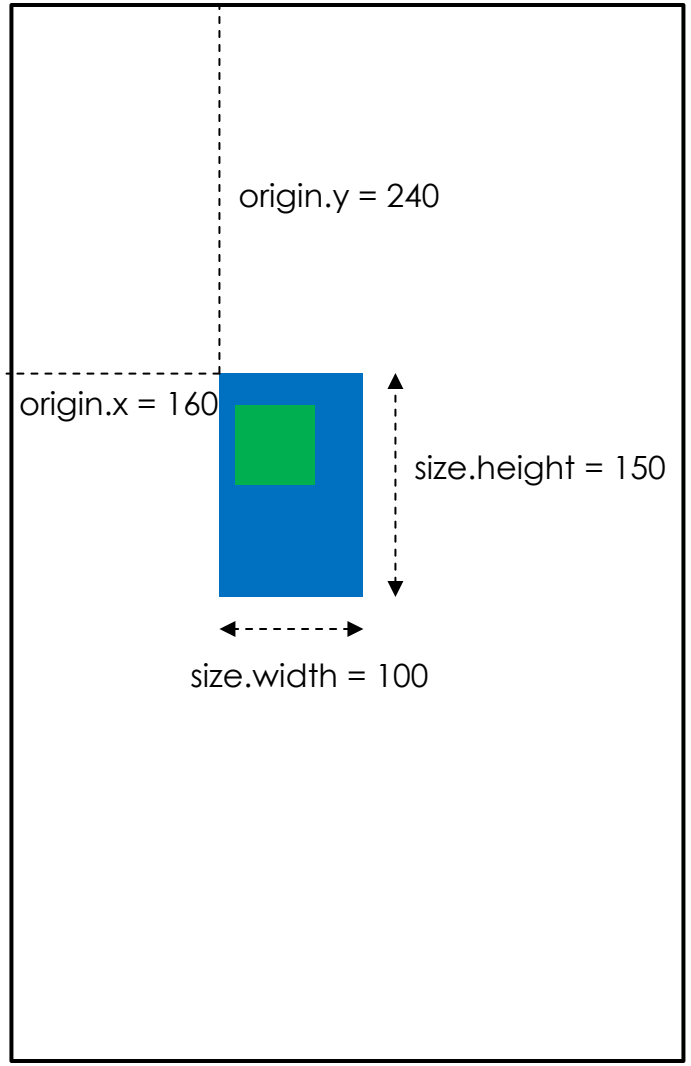
Views are hierarchical

- by adding a new view to the first view, the placement coordinates become relative to the extents of the first view

```
let secondFrame = CGRect(x: 20, y:30, width: 50, height: 50)
let secondView = UIView(frame: secondFrame)
secondView.backgroundColor = UIColor.green
firstView.addSubview(secondView)
```

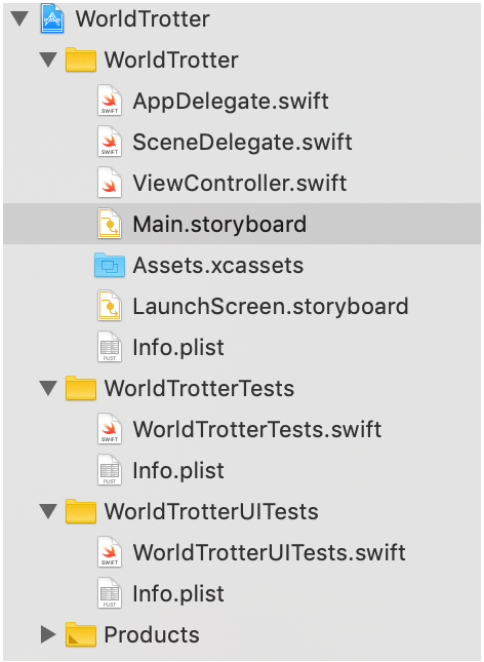
Result

- The second view is a child of the first view
- So its origin is relative to the top-left corner of the first view



Interface Builder (IB)

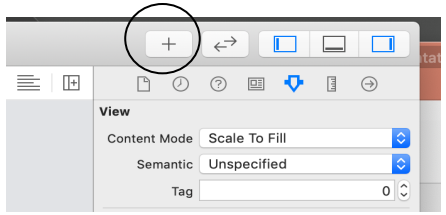
- It's possible to create a user interface programmatically (i.e., in code)
- But it's more convenient to do this outside of the code, using Interface Builder
- IB is essentially a drag-and-drop way to build an interface
- The "canvas" for the interface is by default `Main.storyboard`

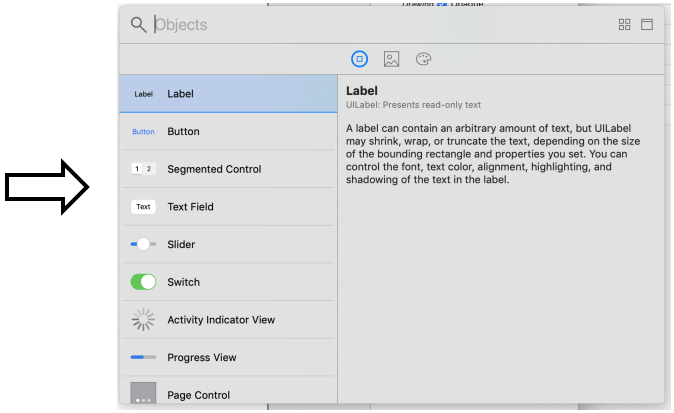


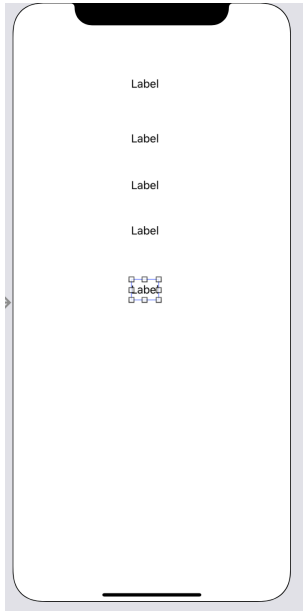
Adding UI Elements

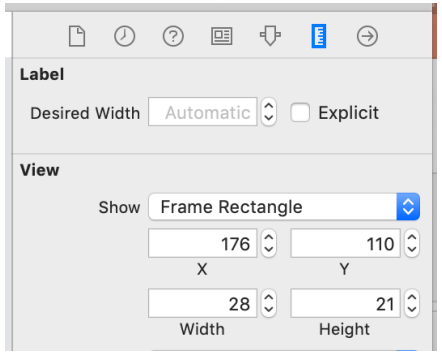
Example:

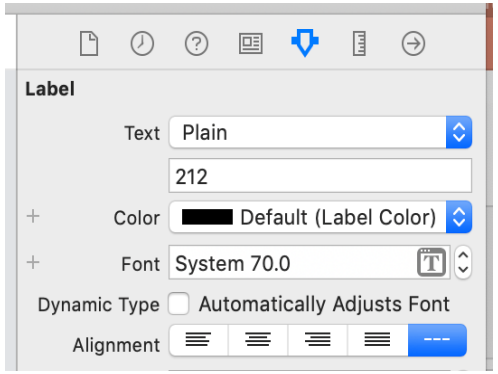
- from the Object Library, drag and drop labels
- set width and height
- change the font size











After Changing Label Attributes

After these first steps:

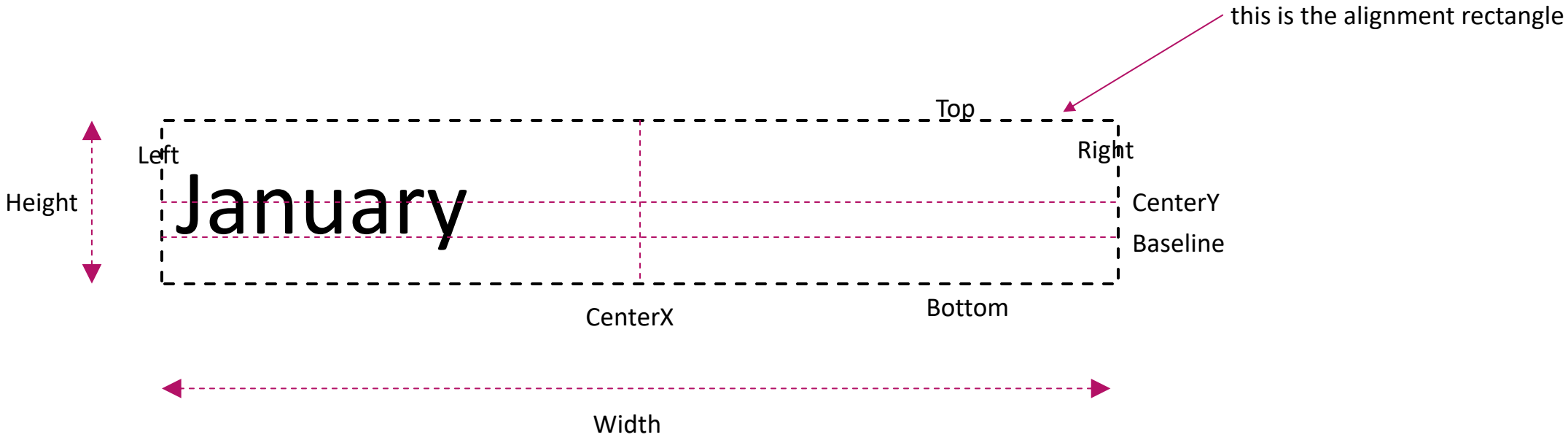
- this shows a significant weakness in the simple drag-and-drop-and-modify technique for designing a UI
- the key thing is that instead of specifying position in absolute terms, we should specify relative positions of the UI elements
  - and this includes the size of the labels: "make the label's width and height large enough to display the label's text"
- solution: build constraint-based layouts
  - simplest way: using Interface Builder (IB)

Alignment Rectangle

- Every view ("view" = UI element) has an alignment rectangle and a frame
- The frame encompasses the entire view
- The alignment rectangle is like an invisible rectangle that is anchored on the view and is used for setting the view's location relative to other UI elements
- It might or might not be the same size as the frame rectangle
- We can't change the alignment rectangle directly

Alignment Rectangle

- Auto Layout knows about the geometry of views
- Here are layout attributes that it uses when it aligns views



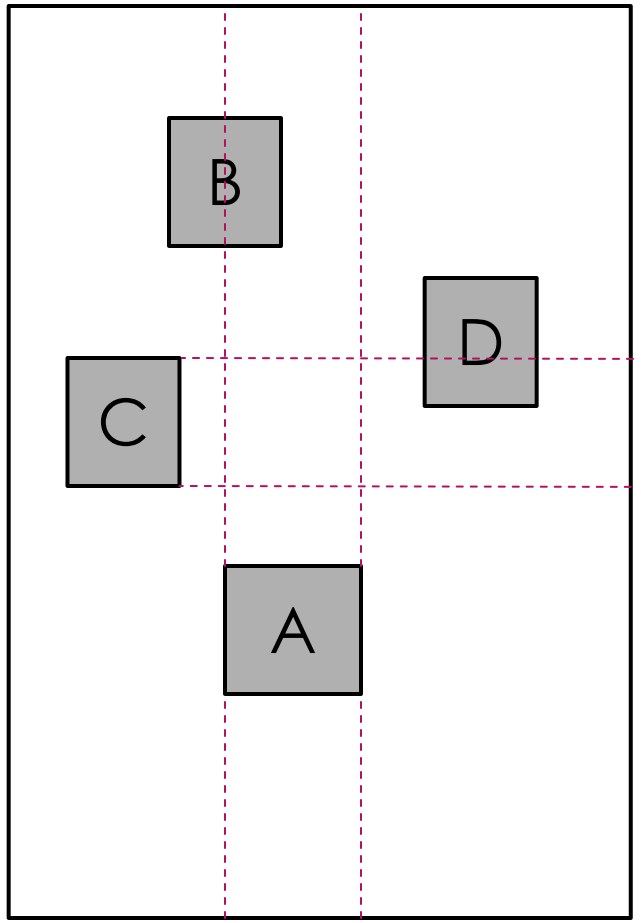
- The alignment rectangle is not the same as the frame
- We can't change the alignment rectangle directly

Constraints

- A constraint defines a specific relationship in a view hierarchy that can be used to determine a layout attribute for one or more views
- For example: "the vertical space between these two views should always be 8 points"
- Or "these two views should always have the same width"
- Or "the CenterX value of this view should be the same as the CenterX value of its parent view"

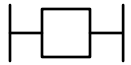
Nearest Neighbor

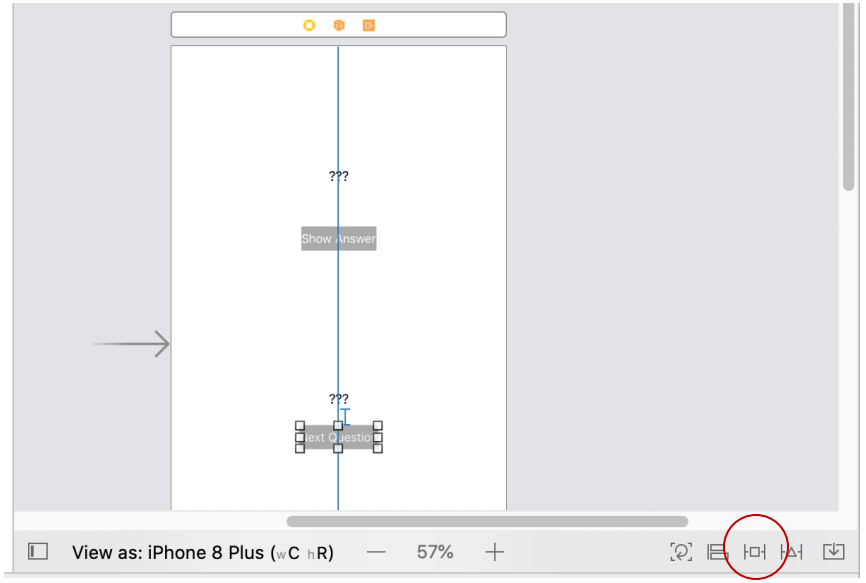
- Auto Layout will position views based on the constraints we provide
- It positions views relative to each other based on the concept of nearest neighbor
- The nearest neighbor is the closest sibling view in the specified direction
  - if the view does not have any siblings in the specified directory, then the nearest neighbor is its superview (also known as its container)
- Nearest top neighbor of A is B
- Nearest right neighbor of C is D
- Nearest bottom neighbor of A is the alignment rectangle of the superview



Interface Builder

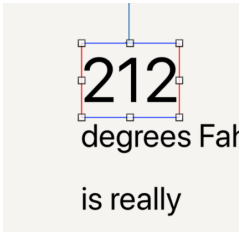
Adding Constraints in IB

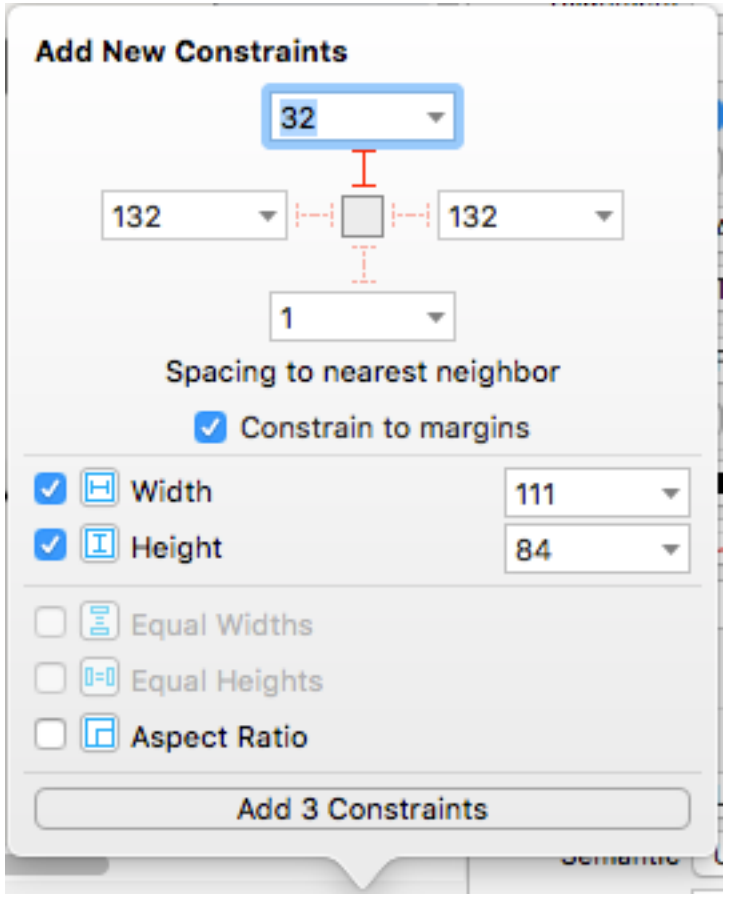
- To add a constraint in IB, select a view and then click the 
- This lets you set size and position for a view
- But note that starting in Xcode 8, there is no "Update Frames"—Auto Layout now does this automatically (the book says to update frames after making a constraint modification in IB)



Adding Constraints in IB

- This shows the width and height of a label set and also the vertical constraint—but the horizontal placement is underconstrained
- Typically, we need two vertical and two horizontal constraints for each view
- There will be a red box around the view in IB if the position is not completely determined



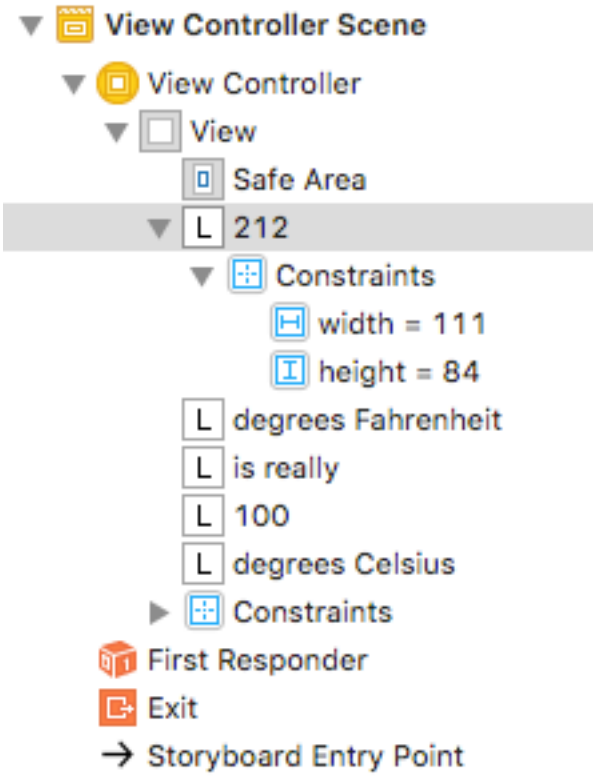


Centering a Label Horizontally in its Parent

- Select the top label and click the 
- Select "Horizontally in Container" and click "Add 1 Constraint" (this is slightly different from what the book says—again, there is no "Update Frames" step necessary)
- The red box around the label will disappear, showing that the alignment rectangle for the label is fully specified (it's no longer underconstrained)

Explicit vs. Intrinsic Height/Width

- The top label ("212") has explicit height and width constraints
- This is a problem—if the content (the text) changes, then the constraint values might no longer be appropriate
- Instead, we want to use the label's *intrinsic content size*—this is like the size that the label wants to be in order to fit the current text it is displaying
- If we don't add explicit constraints to set the width and height, then the intrinsic values are used



More about Constraints

- The book describes a "misplaced view"—the situation when a view in IB does not match what will actually be rendered on the screen because of the constraints in place

- The book shows how to clear all constraints
  - select the top label
  - select Editor -> Resolve Auto Layout Issues
  - select Clear Constraints
  - can also pick the 

Resolve Auto Layout issues

- The book shows how to add constraints—you should end up with 10 constraints

Final Constraints

- Here's what my constraints look like
- Now build and run the app
- Observation #1: it takes a lot of constraints to make even a simple UI look good
- Observation #2: it's difficult to make a good UI