CS275
Fall 2020

iOS: Stack Views

BNRG CHAPTER 13

Topics

- Stack views and stack-view distribution
- Implicit constraints
- Content-hugging priority
- Content-compression-resistance priority
- Segues

Stack View

A stack view simplifies the construction of interfaces

- specifically, it eliminates the need for creating the constraints required to place UI elements correctly relative to each other

A stack view is a natural UI organization for parts of an interface that are linear in nature

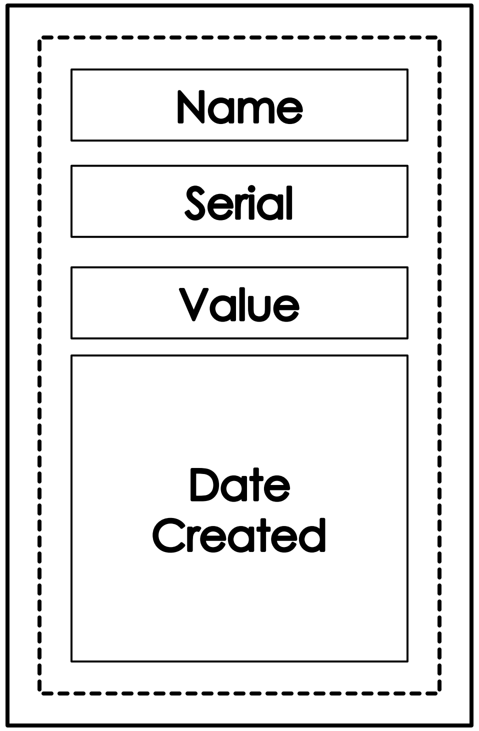
- for example, a set of similar elements
- or a label-text pair

In iOS, a stack view is an instance of `UIStackView`

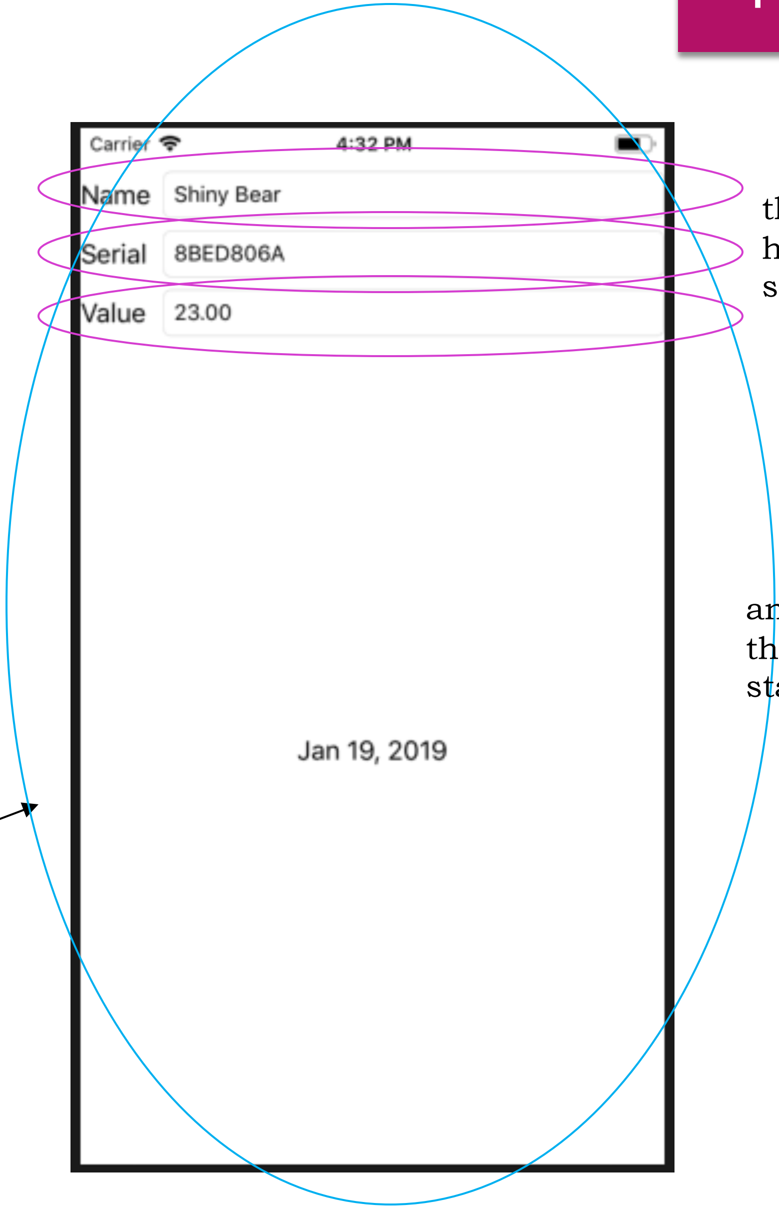
- it has a property saying whether it's a horizontal or vertical stack view

Stack View

A stack view is a linear layout of elements



here is an example of a vertical stack view



these are horizontal stack views

and the whole thing is a vertical stack view

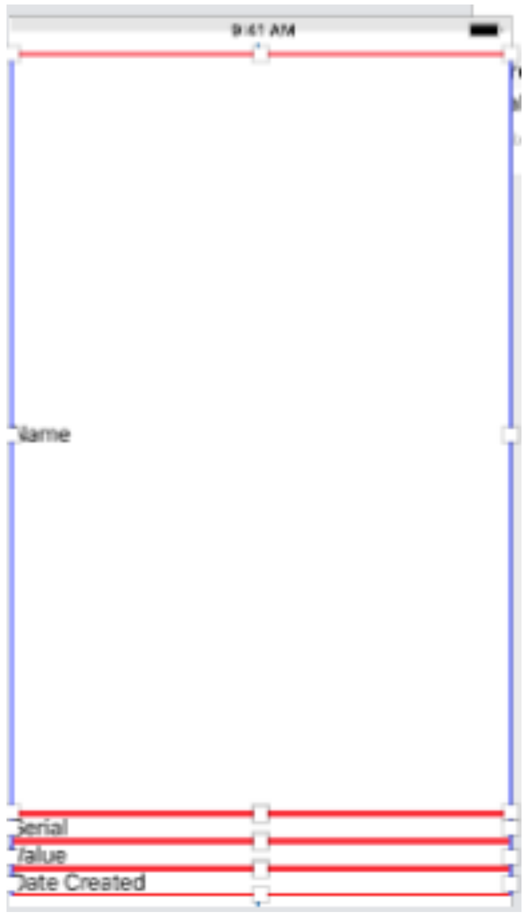
and this is the layout for the detail view of the Homepwner project

Creating a Vertical Stack View

The object library in IB has a Vertical Stack View and a Horizontal Stack View

Filling in a Stack View

- Drag four UI Labels onto the Stack View
- Change their text values
  - Name
  - Serial
  - Value
  - Date Created
- Red lines show a problem from Auto Layout
- The red boxes indicate that the placement of the labels is ambiguous



Implicit Constraints

For the UI runtime to be able to place views (UI elements), it has to know how big each view is

Intrinsic content size: the size that a view wants to be in order to display everything that it contains

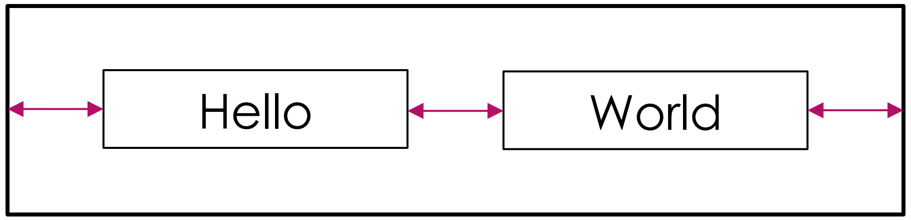
Every view has an intrinsic content size

A view calculates its intrinsic content using two sets of properties

- content hugging priorities (vertical and horizontal)
- compression resistance priorities (vertical and horizontal)

Content Hugging Priority

Consider two labels side by side in a superview



Content Hugging Priority

Suppose these two labels are fully constrained in the horizontal direction

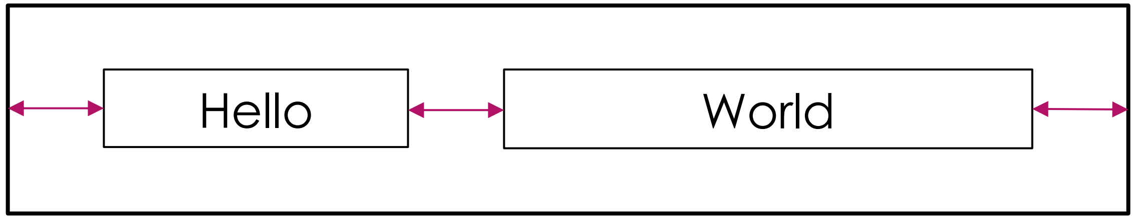
Now suppose the superview becomes wider

- for example, the user rotates the device

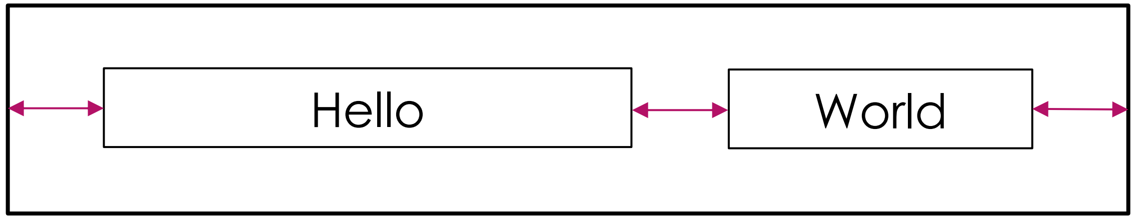
Content Hugging Priority

If the interface becomes wider, there are two ways that this can be accommodated

- the first label or the second label must become wider
- but which one?
- without further specification, the interface is ambiguous

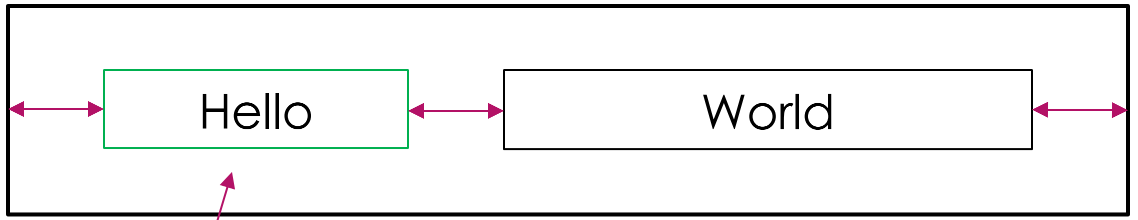


this?

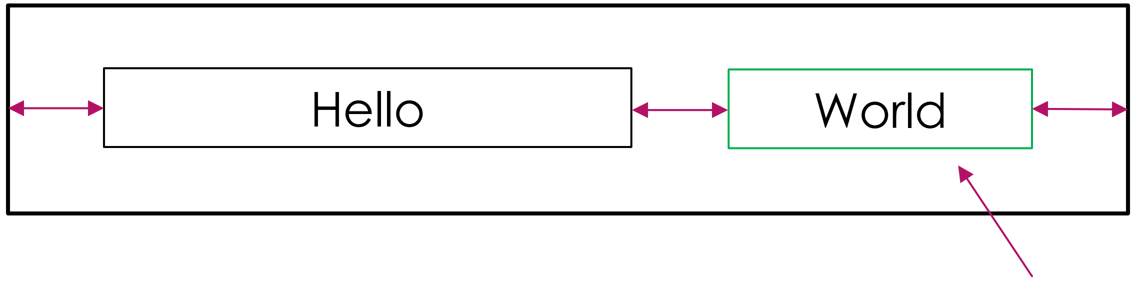


or this?

Content Hugging Priority



result when this label has greater CHP



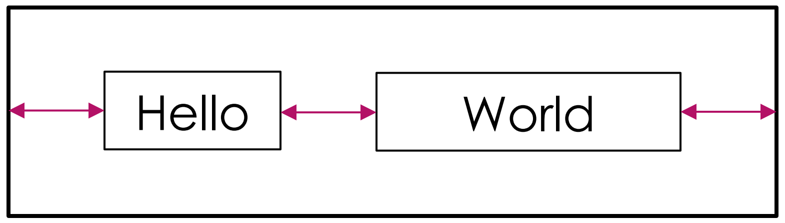
result when this label has greater CHP

The view with the greater content-hugging priority will not stretch

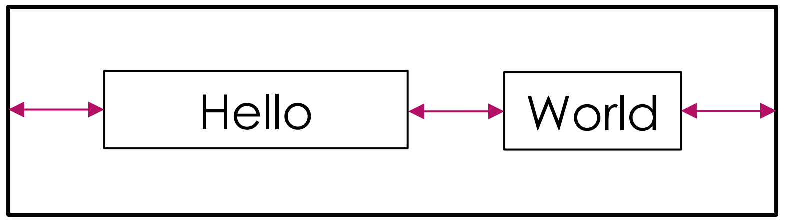
- think of the content-hugging priority as a rubber band around the view (the label, in this case)
- the "strength" of the rubber band (the content-hugging priority) can be a value between 0 and 1000
- a value of 1000 means "don't stretch"—the view cannot become bigger than its intrinsic content size

Content Compression Resistance Priority

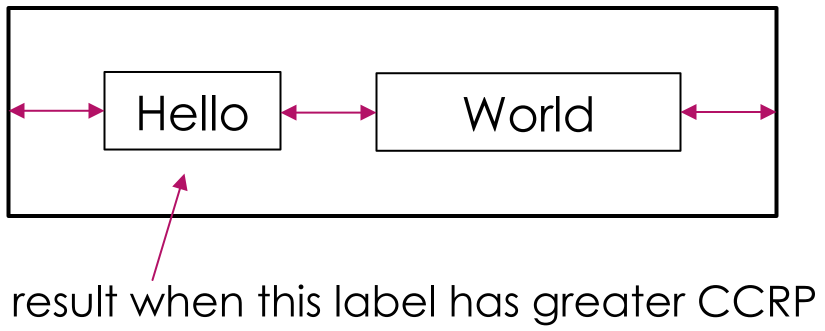
Same situation, except now suppose that the superview becomes smaller

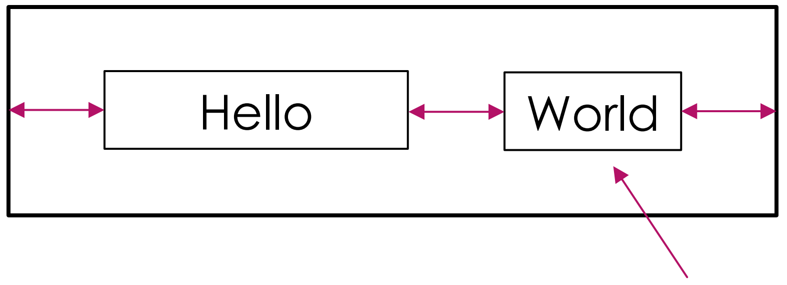


Which one will shrink?



Content Compression Resistance Priority





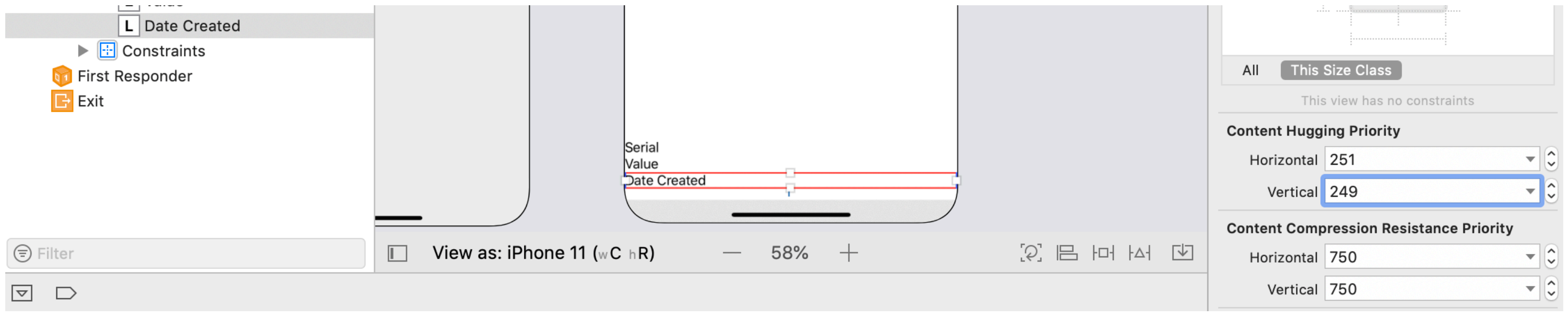
result when this label has greater CCRP

The view with the greater content compression resistance priority is the one that will resist compression

- think of this like a spring that's inside the subview
- and again, the priority can have a value between 0 and 1000

Fixing the Layout

- Change the vertical content-hugging priority for the "Date Created" label
- Set it to 249, which will be smaller than the value for the other three labels (the default value is 250)
- By reducing this value, it tells Auto Layout that it's OK to expand this label vertically in order to establish a complete set of vertical constraints for the four labels



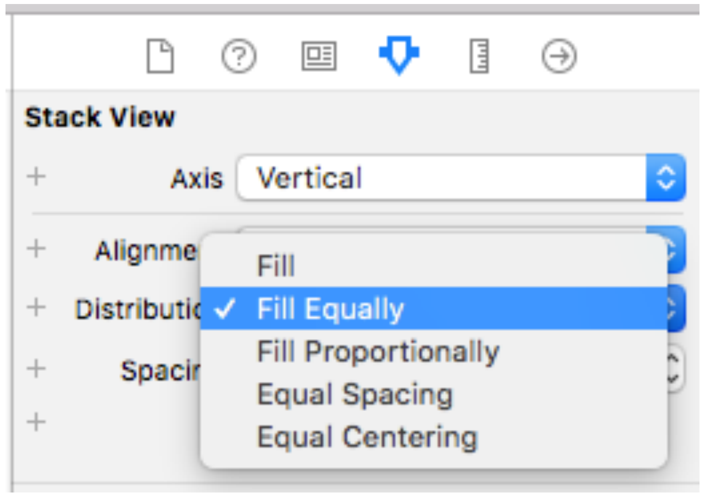
Content Hugging Priority
Horizontal 251
Vertical 249

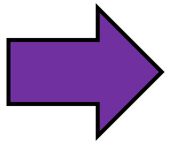
Content Compression Resistance Priority
Horizontal 750
Vertical 750

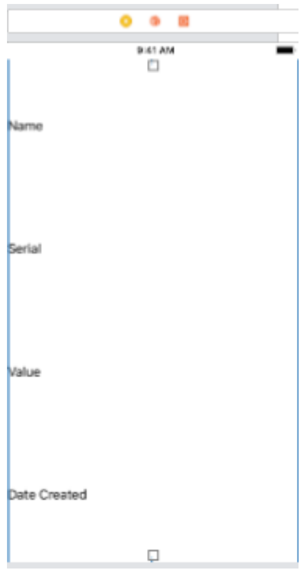
Another Solution: Stack View Distribution

Stack views have several properties that determine how their content is laid out

- Fill: views lay out their content based on their intrinsic content size
- Fill Equally: resizes the views so that they all have the same height

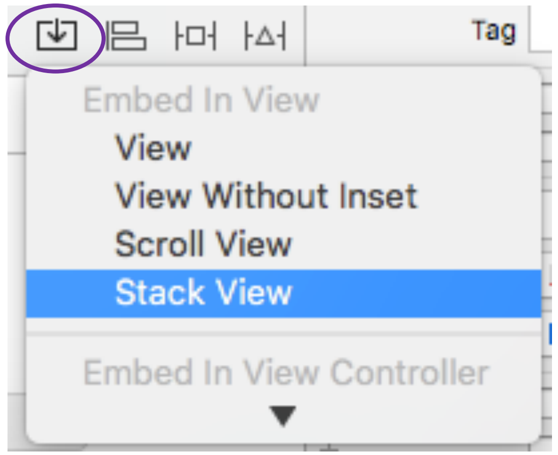




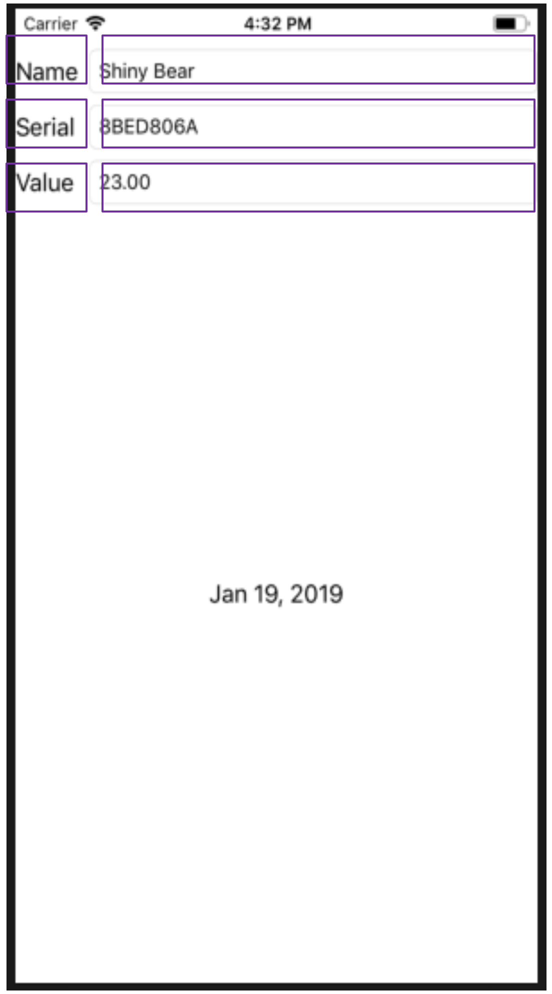


Nested Stack Views

- We can put one stack view inside another one
- And this lets us organize our layouts hierarchically
- The Auto Layout GUI makes it easy to embed a view within a nested stack view



vertical SV [horizontal SV
horizontal SV
horizontal SV
label]



Name Shiny Bear
Serial 8BED806A
Value 23.00

Jan 19, 2019

Nested Stack Views

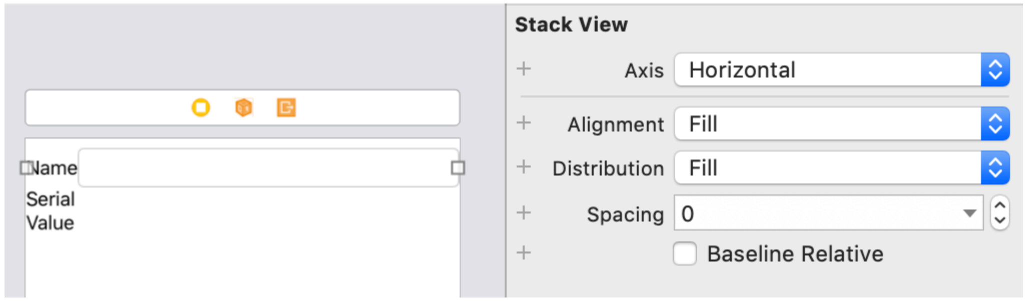
Description of steps to modify the interface:

- Embed each of the top three labels inside a stack view
- Change the stack view's orientation to horizontal
- Add a text field to each horizontal SV
- Change the text field's CCR priority to 749
  - the CCR priority for the label will be the default value, 750
  - so if there's not enough room for both, then the text field will be truncated

Stack-View Spacing

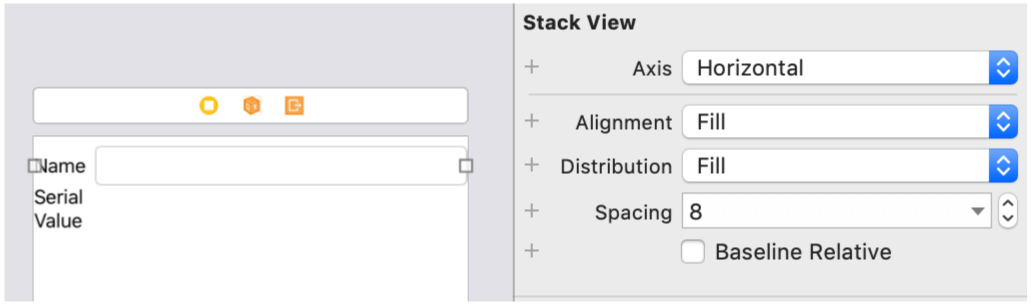
A stack view has a property called *spacing*

- this is the spacing it puts between each of its elements as they're laid out



Stack View

Axis: Horizontal
Alignment: Fill
Distribution: Fill
Spacing: 0
Baseline Relative: ☐

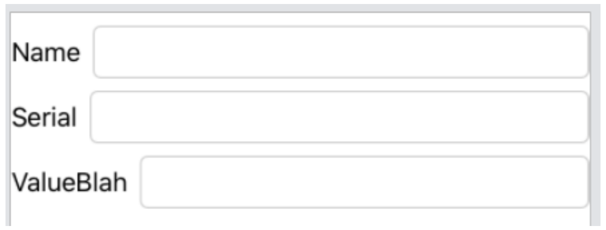


Stack View

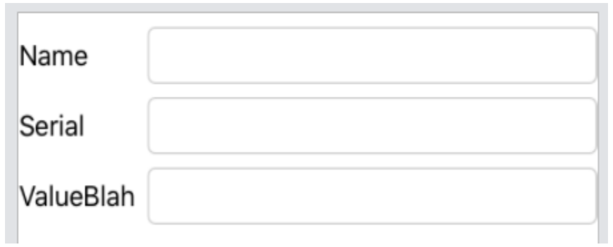
Axis: Horizontal
Alignment: Fill
Distribution: Fill
Spacing: 8
Baseline Relative: ☐

More Polishing

- Change the spacing of the vertical stack view
- Change the alignment of the "Date Created" label
- And align the left edges of the text fields
- I changed one of the labels to show this misalignment:



unaligned left edges



aligned left edges

Stack Views: Summary

Stack views provide a great mechanism for building interfaces with groups of elements

It's much easier and faster and efficient to build constraints between elements using a stack view than by building the constraints manually

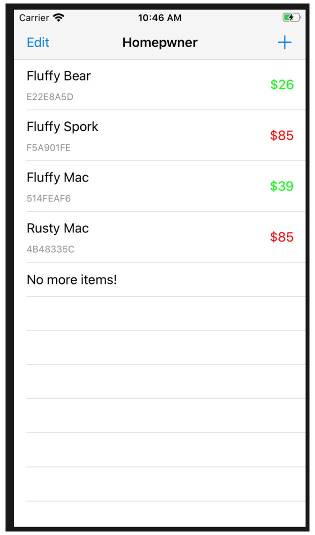
It's even possible to control stack views dynamically, from code

- add, remove, hide elements

~~Master-Detail~~ Overview-Detail

A common UI pattern for apps

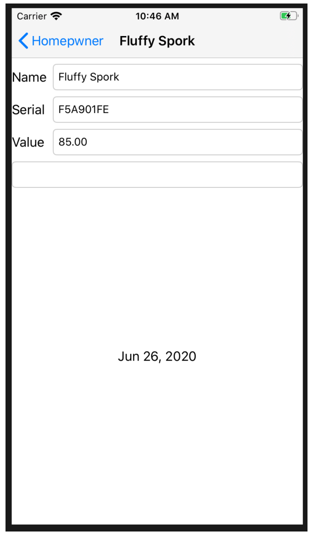
- the overview shows all of the model objects, in an abbreviated form
- the detail view provides complete information about one of the model objects



Fluffy Bear E22E8A5D $26
Fluffy Spork F5A901FE $85
Fluffy Mac 514FEAF6 $39
Rusty Mac 4B48335C $85
No more items!

overview

segue



Name: Fluffy Spork
Serial: F5A901FE
Value: 85.00

Jun 26, 2020

detail

Segues

- Most apps have several different view controllers between which users of the app navigate
- As a user interacts with the app, we will want to move between different view controllers
- Storyboards enable us to specify how this movement occurs, without the need for writing code
- A *segue* is an action that moves a view controller onto the screen

segue: borrowed from Italian *segue* ("it follows"), from *seguire* ("to follow"), from Latin *sequor;* originally a term used in a musical score to indicate that the next movement or passage is to follow without a break.

Segways

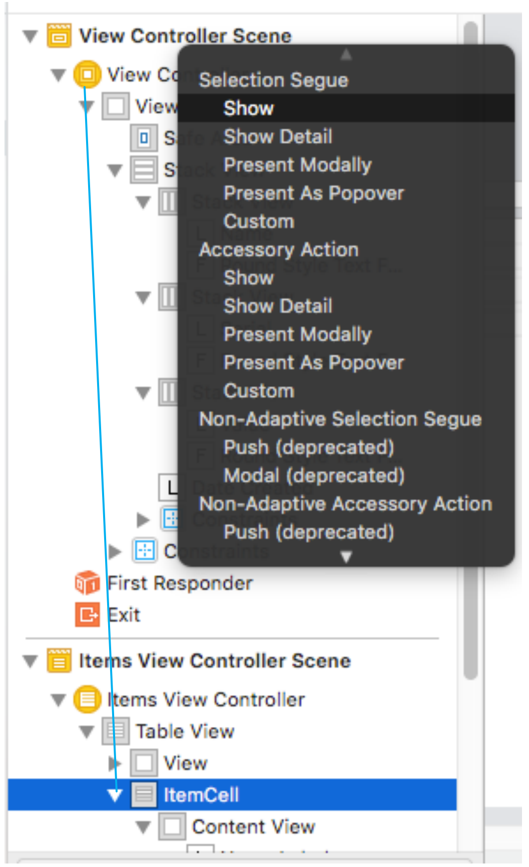
Segues

Segues have characteristics that we specify

- style: this determines how the view controller will be presented
- action item: this is the view object that triggers the segue (e.g., a button)
- identifier: this is a hook that lets us access the segue programmatically; useful when we want to trigger the segue when some other event (not from the action item) happens; for example a shake event

Show Segue

- *Show segue*: straightforward—display a different view controller
- To build a show segue
  - control-drag from `ItemCell` to the new view controller
  - and select "Show"
- Then build and run: you should see that tapping on one of the cells in the table view causes the new view controller to slide up from the bottom (this is the default behavior for the show segue)
- The new view controller won't actually display any info yet
- And, there's no way to go back again



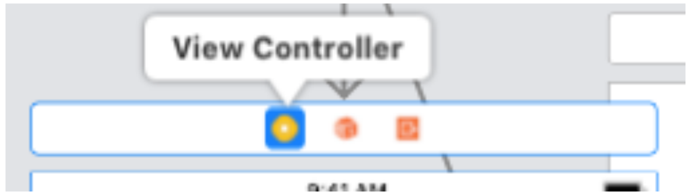
Hooking up the Content

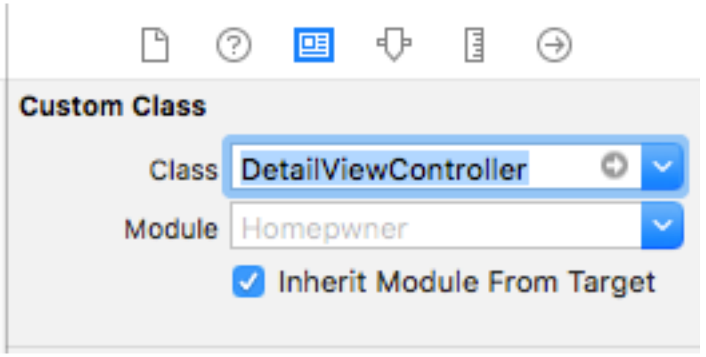
To fill in detail for the second view controller, we create a new file with a new class subclassed from `UIViewController`

```
class DetailViewController: UIViewController {

}
```

Must also set the class for the new view controller to this new class




Custom Class

Class: DetailViewController

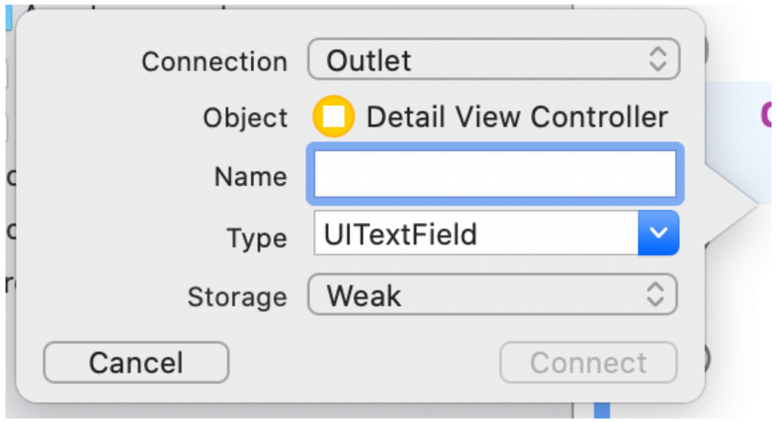
Module: Homepwner

✓ Inherit Module From Target

Hooking up the Content

The book describes how to autogenerate outlets (hooks from code to UI elements) for the three text fields

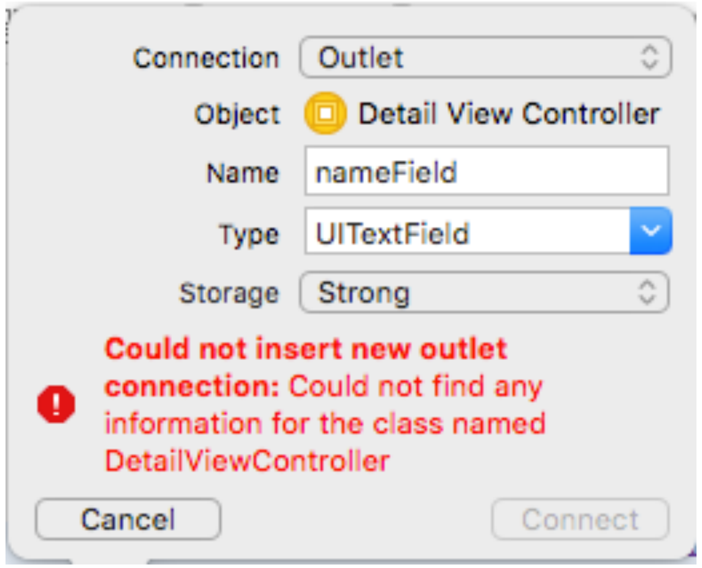
- control-drag from the `UITextField` on the Storyboard to the top of the `DetailViewController` code
- and you should see a pop-up like this:


Connection Outlet
Object Detail View Controller
Name
Type UITextField
Storage Weak

Hooking up the Content

The book describes how to autogenerate outlets (hooks from code to UI elements) for the three text fields

- If you get this error:



Connection: Outlet
Object: Detail View Controller
Name: nameField
Type: UITextField
Storage: Strong

Could not insert new outlet connection: Could not find any information for the class named DetailViewController

then do Product -> Clean Build Folder and exit and restart Xcode

- (at one time) this was a bug in Xcode

https://stackoverflow.com/questions/29923881/could-not-insert-new-outlet-connection-could-not-find-any-information-for-the-c

Hooking up the Content

Hook up the other two fields and also the data label:

```
@IBOutlet var nameField: UITextField!
@IBOutlet var serialNumberField: UITextField!
@IBOutlet var valueField: UITextField!
@IBOutlet var dateLabel: UILabel!
```

- Important general comment about storyboards and app code: the code for a `UIViewController` must stay in sync with the connections to the GUI
- If you change the name of a property but don't update the connection, then the app will crash

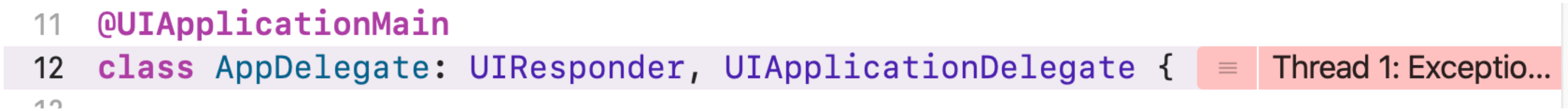
Storyboard - Code: Must Stay in Sync

I changed the name of one of the properties that is connected to a `UITextField`

- Here's the runtime error I get:

```
2020-06-27 09:33:27.825657-0400 HomePwnerCh13[47669:64963817] ***
Terminating app due to uncaught exception 'NSUnknownKeyException', reason:
'[<HomePwnerCh13.DetailViewController 0x7fe9d430e940>
setValue:forUndefinedKey:]: this class is not key value coding-compliant
for the key valueField.'
```

- and it crashed here:



```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
```

Thread 1: Exceptio...

Passing Data between View Controllers

When we touch on one of the items in the `UITableView`, we want to pass information about that item to the `DetailViewController`

- more accurately: we want to pass a reference to the item itself

Whenever a segue is triggered, this method is called on the view controller that initiates the segue:

```
prepare(for:sender:)
```

Segue Info

The `prepare(for:sender:)` method has two arguments

1. `UIStoryboardSegue`, which has information about which segue is happening
2. the sender, which is the object that triggered the segue (here, this will be an instance of `UITableViewCell`)

The `UIStoryBoardSegue` has this information:

- the source view controller (the view controller that initiated the segue)
- the destination view controller
- the identifier of the segue—this lets us create more than one segue between two view controllers

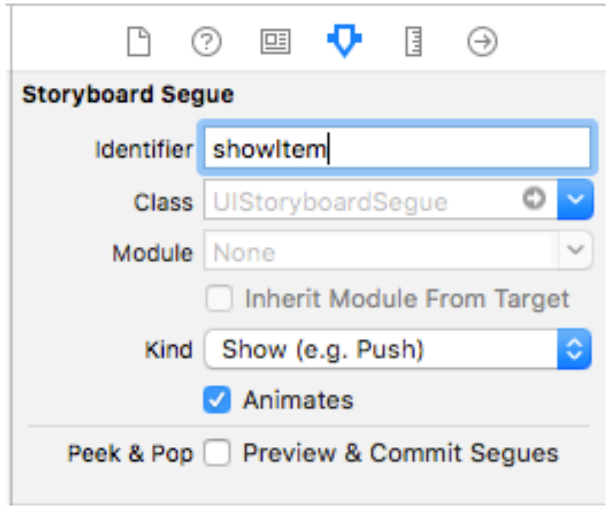
Must Give the Segue an Identifier

In order to refer to a segue event in code, we have to know what its name is

- so, we must give the segue an identifier

Select the arrow between the two view controllers

- Then set the name in the attributes inspector



Storyboard Segue

Identifier: showItem
Class: UIStoryboardSegue
Module: None
☐ Inherit Module From Target
Kind: Show (e.g. Push)
✓ Animates
Peek & Pop: ☐ Preview & Commit Segues

Passing Data from One VC to Another VC

`prepare(for:sender:)` is the glue

- called on the source view controller
- lets us query information from the source VC (such as the row number that was touched)
- and then, using the destination VC that is in the segue, lets us pass info to the destination VC
  - for example, by setting a property in the destination VC

The `prepare(for:sender:)` function tells us where we are, and where we're going

Example

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // if the triggered segue is the "showItem" segue
    switch segue.identifier {
    case "showItem"?:
    // figure out which row was just tapped
        if let row = tableView.indexPathForSelectedRow?.row {
            // get the item associated with this row and pass it along
            let item = itemStore.allItems[row]
            let detailViewController = segue.destination as! DetailViewController
            detailViewController.item = item
        }
    default:
        preconditionFailure("Unexpected segue identifier.")
    }
}
```

Final Things

This way, we touch a row, and the `DetailViewController` with the correct item info filled in will appear

- there's no way to dismiss the `DetailViewController` – that's in the next chapter

This is a forced exit, useful for development and debugging:

```
preconditionFailure(_:file:line:)
```

preconditionFailure(_:file:line:)
Indicates that a precondition was violated.
Use this function to stop the program when control flow can only reach the call if your API was improperly used.