Stanford CS193p

Developing Applications for iOS

Fall 2017-18

Today

- Drag and Drop

  Transferring information around within and between apps.

  EmojiArt Demo — Drag and drop an image to get our EmojiArt masterpieces started.

- `UITableView` and `UICollectionView`

  Ways to display arbitrary amounts of data in a list or collection.

Drag and Drop

- Very interoperable way to move data around
  - Between apps on iPad and within an app on all iOS 11 devices.
  - Your app continues to work normally while drag and drop is going on.
  - Multitouch allows some fingers to be doing drag and drop and other fingers working your app.
  - New mulittasking UI in iOS 11 makes drag and drop really useful.

Drag and Drop

- Interactions

  A view "signs up" to participate in drag and/or drop using an interaction.
  It's kind of like a "gesture recognizer" for drag and drop.

```
let drag/dropInteraction = UIDrag/DropInteraction(delegate: theDelegate)
view.addInteraction(drag/dropInteraction)
```

  Now the `theDelegate` will get involved if a drag or drop occurs in the `view`.

Drag and Drop

- Starting a drag

Now, when the user makes a dragging gesture, the delegate gets …

```
func dragInteraction(_ interaction: UIDragInteraction,
     itemsForBeginning session: UIDragSession
) -> [UIDragItem]
```

… and can return the items it is willing to have dragged from the view.
Returning an empty array means "don't drag anything after all."

A `UIDragItem` is created like this …

```
let dragItem = UIDragItem(itemProvider: NSItemProvider(object: provider))
```

Providers: `NSAttributedString`, `NSString`, `UIImage`, `NSURL`, `UIColor`, `MKMapItem`, `CNContact`.
You can drag your own types of data, but that's beyond the scope of this course.
Note that some of these types start with NS … you might have to use `as?` to cast them.

You can also provide an object that will be passed to drop targets inside your own app …

```
dragItem.localObject = someObject
```

Drag and Drop

- Adding to a drag

  Even in the middle of a drag, users can add more to their drag if you implement …

```
func dragInteraction(_ interaction: UIDragInteraction,
        itemsForAddingTo session: UIDragSession
) -> [UIDragItem]
```

  … and returns more items to drag.

Drag and Drop

- Accepting a drop

When a drag moves over a view with a `UIDropInteraction`, the delegate gets …

```
func dropInteraction(_ interaction: UIDropInteraction,
    canHandle session: UIDragSession
) -> Bool
```

… at which point the delegate can refuse the drop before it even gets started.

To figure that out, the delegate can ask what kind of objects can be provided …

```
let stringAvailable = session.canLoadObjects(ofClass: NSAttributedString.self)
let imageAvailable = session.canLoadObjects(ofClass: UIImage.self)
```

… and refuse the drop if it isn't to your liking.

Drag and Drop

- Accepting a drop

  If you don't refuse it in canHandle:, then as the drag progresses, you'll start getting …

```
func dropInteraction(_ interaction: UIDropInteraction,
    sessionDidUpdate session: UIDragSession
) -> UIDropProposal
```

  … to which you will respond with `UIDropProposal(operation: .copy/.move/.cancel)`.

  `.cancel` means the drop would be refused

  `.copy` means drop would be accepted

  `.move` means drop would be accepted and would move the item (only for drags within an app)

  If it matters, you can find out where the touch is with `session.location(in: view)`.

Drag and Drop

- Accepting a drop

If all that goes well and the user let's go of the drop, you get to go fetch the data …

```
func dropInteraction(_ interaction: UIDropInteraction,
    performDrop session: UIDropSession
)
```

You will implement this method by calling `loadObjects(ofClass:)` on the `session`.
This will go and fetch the data asynchronously from whomever the drag source is.

```
session.loadObjects(ofClass: NSAttributedString.self) { theStrings in
    // do something with the dropped NSAttributedStrings
}
```

The passed closure will be executed some time later on the main thread.
You can call multiple `loadObjects(ofClass:)` for different classes.
You don't usually do anything else in `dropInteraction(performDrop:)`.

Drag and Drop

- Demo

  We're going to start a new app: EmojiArt

  The first thing we'll do is allow drag and drop to create our EmojiArt document background

Table and Collection Views

- UITableView and UICollectionView

  These are UIScrollView subclasses used to display unbounded amounts of information.
  Table View presents the information in a long (possibly sectioned) list.
  Collection View presents the information in a 2D format (usually "flowing" like text flows).
  They are very similar in their API, so we will learn about them at the same time.

Table and Collection Views

- UITableView

  The list can be very simple …

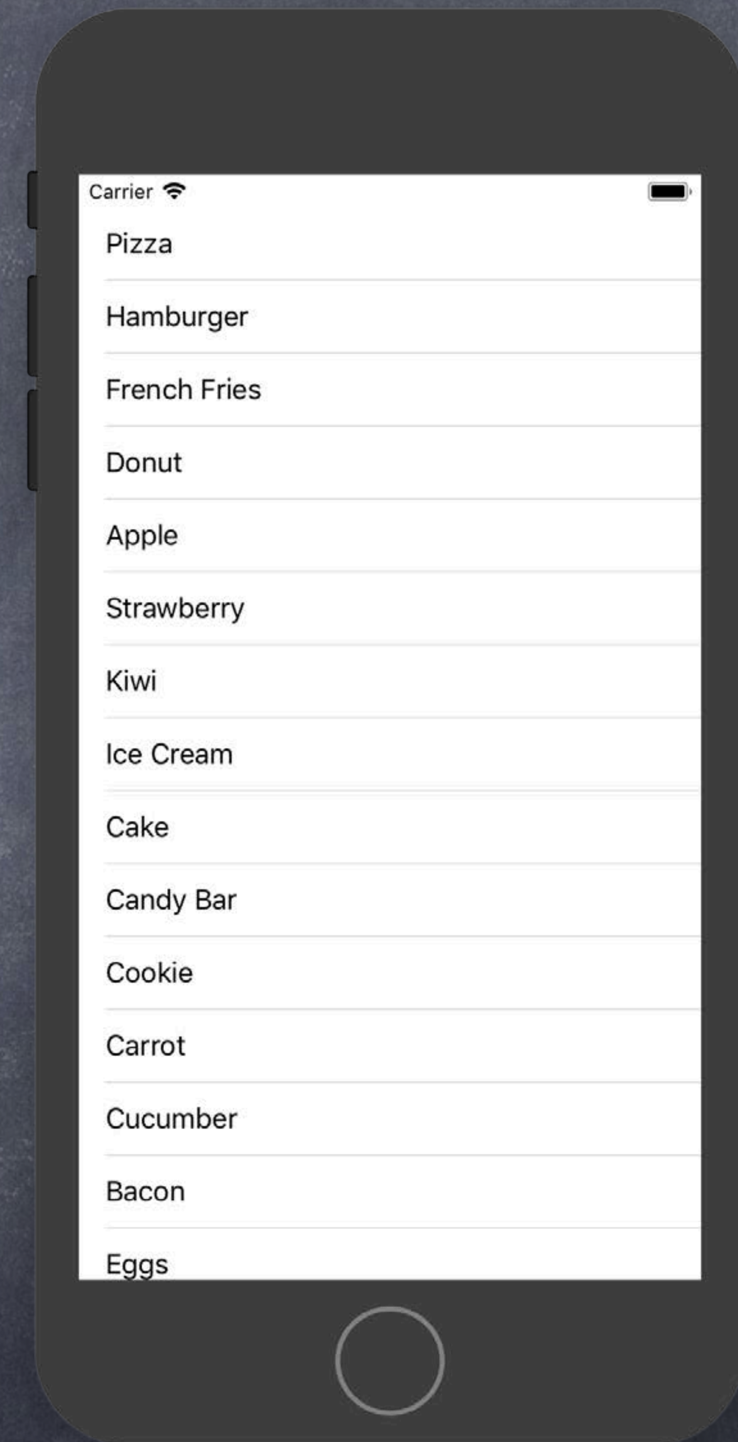


Table and Collection Views

- UITableView

  Or divided into sections ...

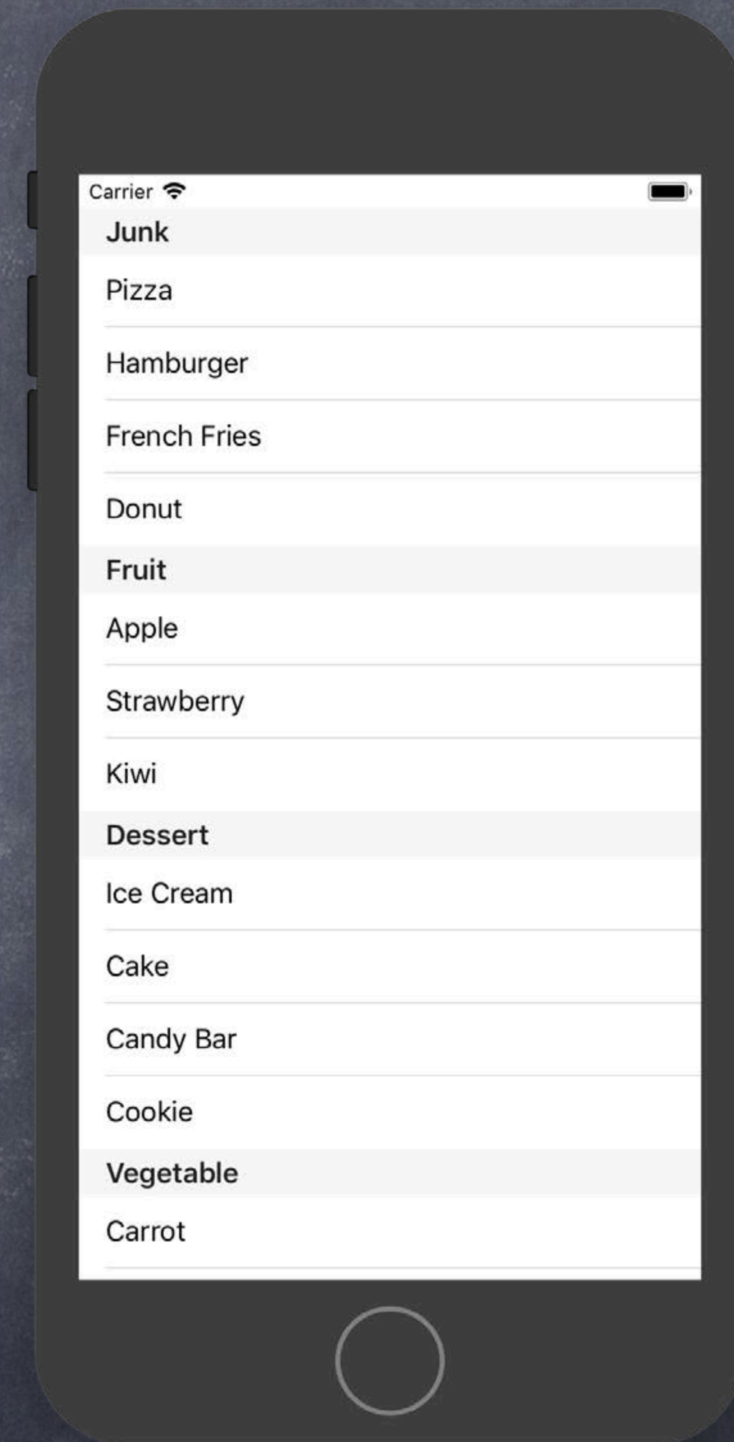


Table and Collection Views

- UITableView

  It can show simple ancillary information …

Subtitle style

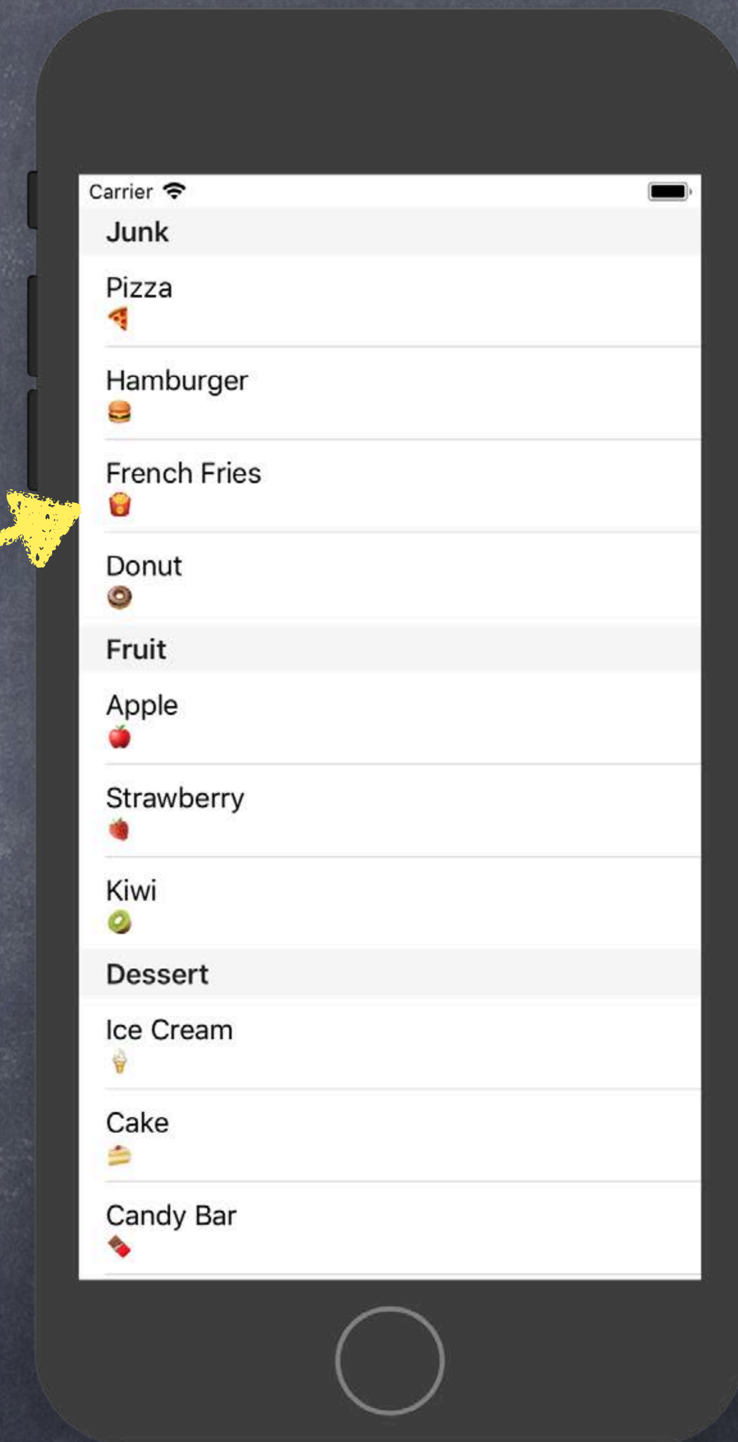


Table and Collection Views

- `UITableView`

It can show simple ancillary information …

Left Detail style

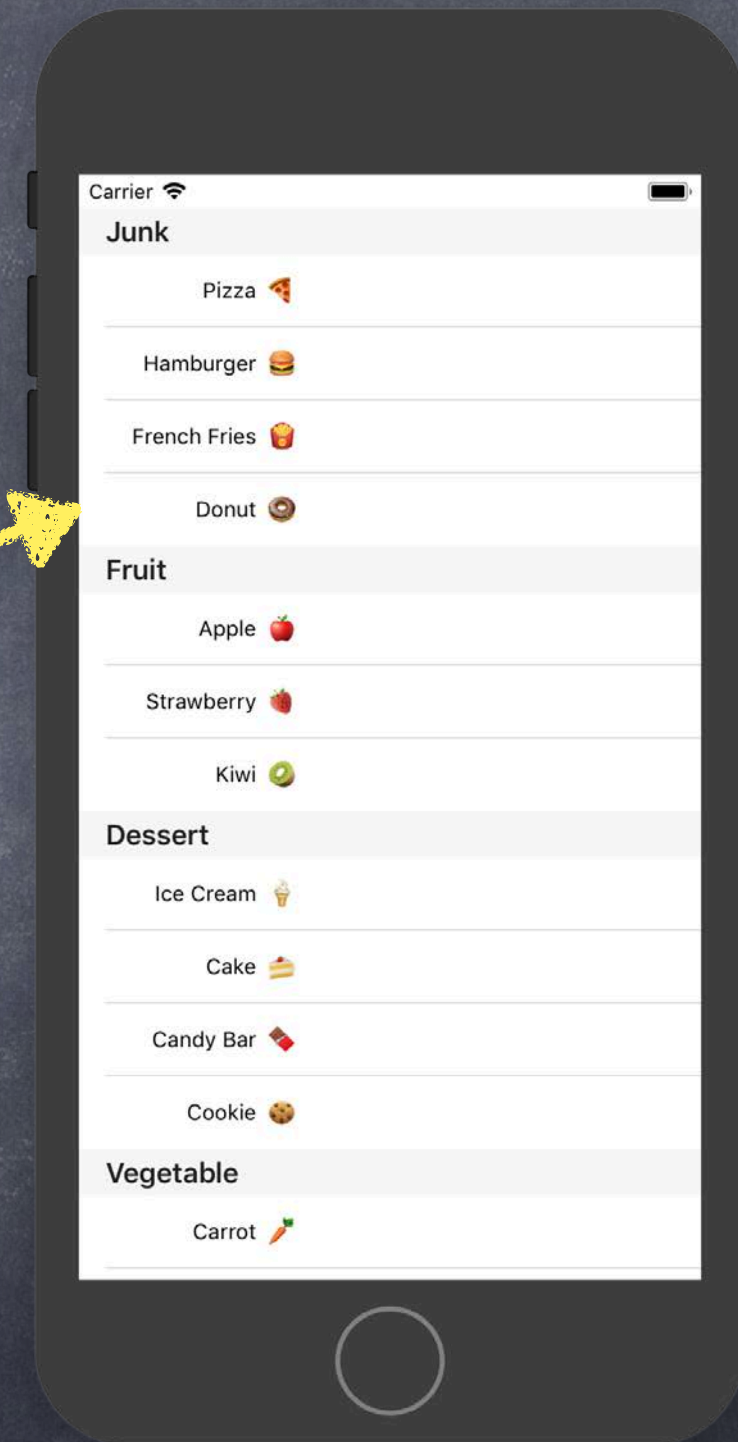


Table and Collection Views

- UITableView

It can show simple ancillary information ...

Right Detail style

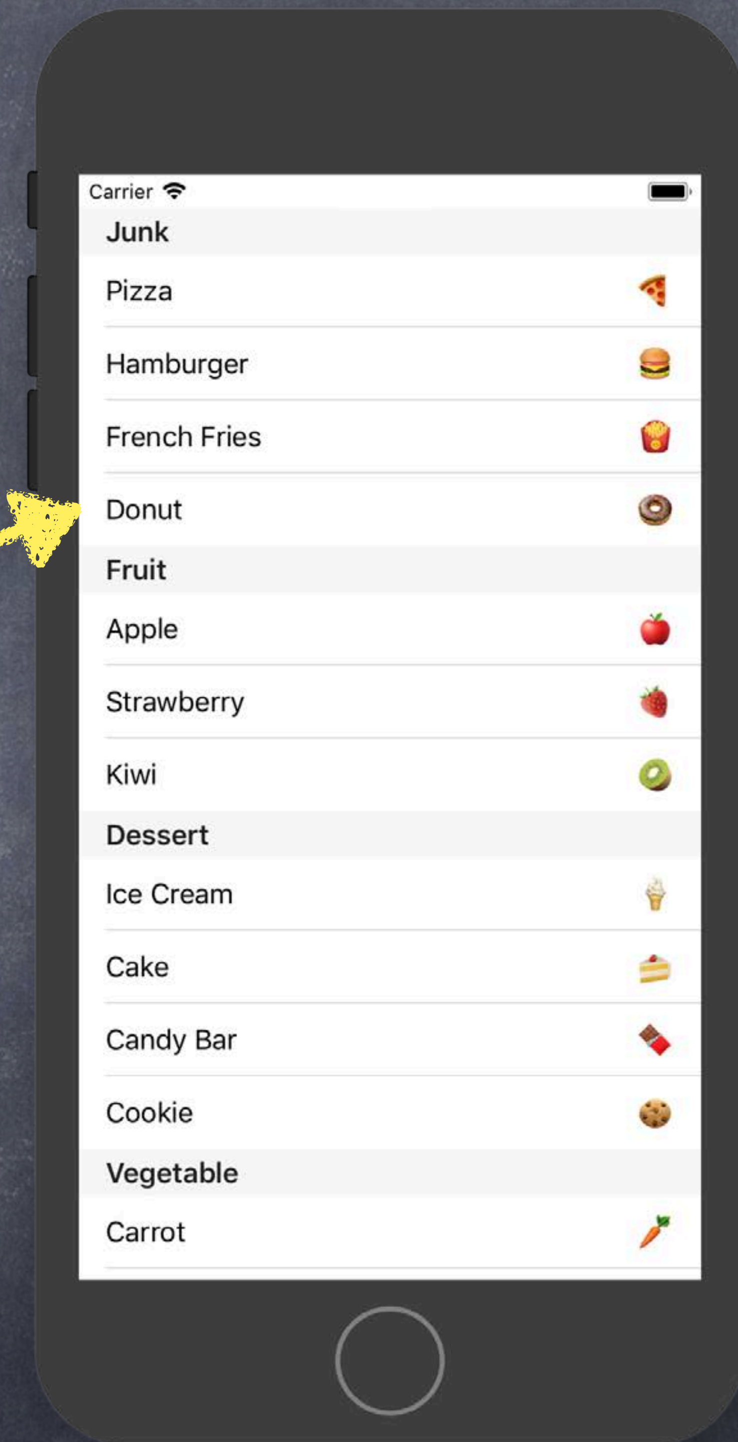


Table and Collection Views

- UITableView

It can show simple ancillary information ...

Basic style

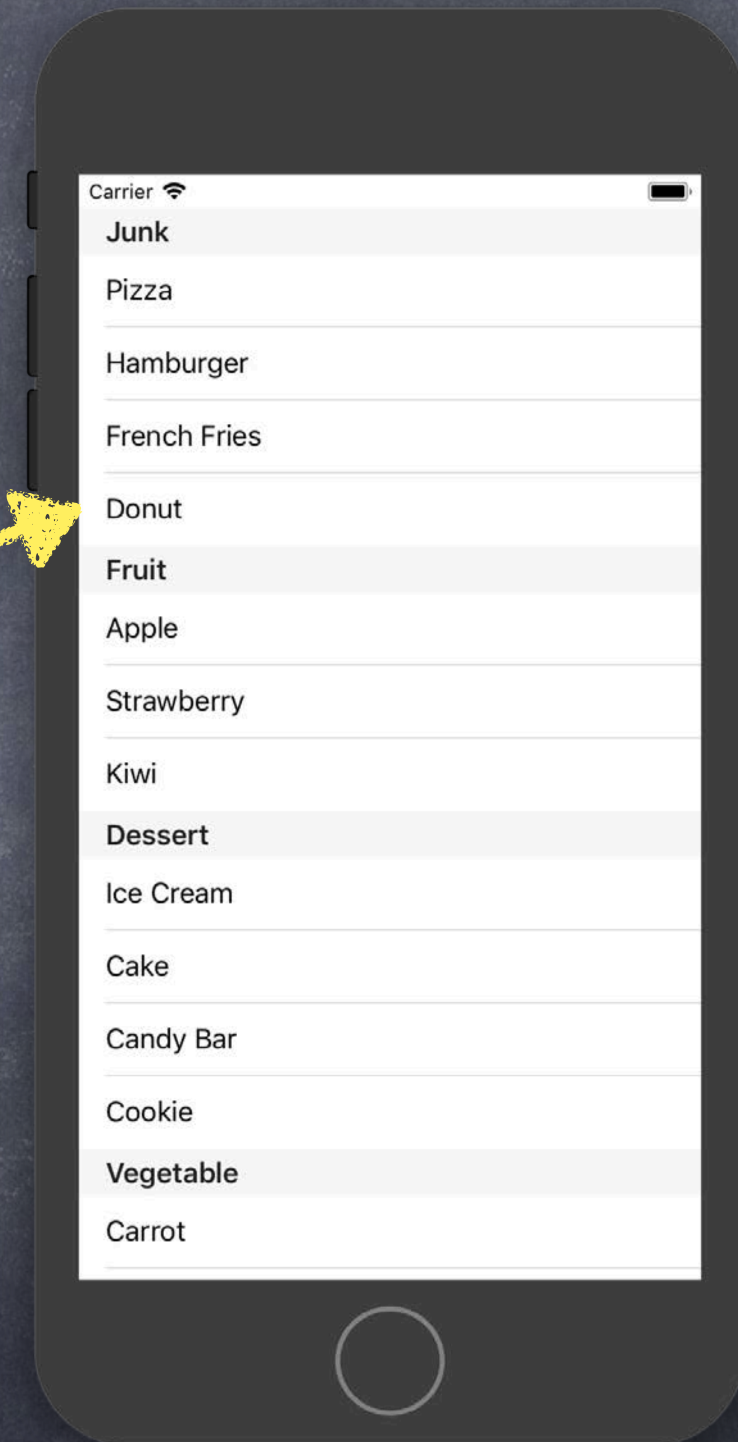


Table and Collection Views

- UITableView

  Or arbitrarily complex information ...

Custom style

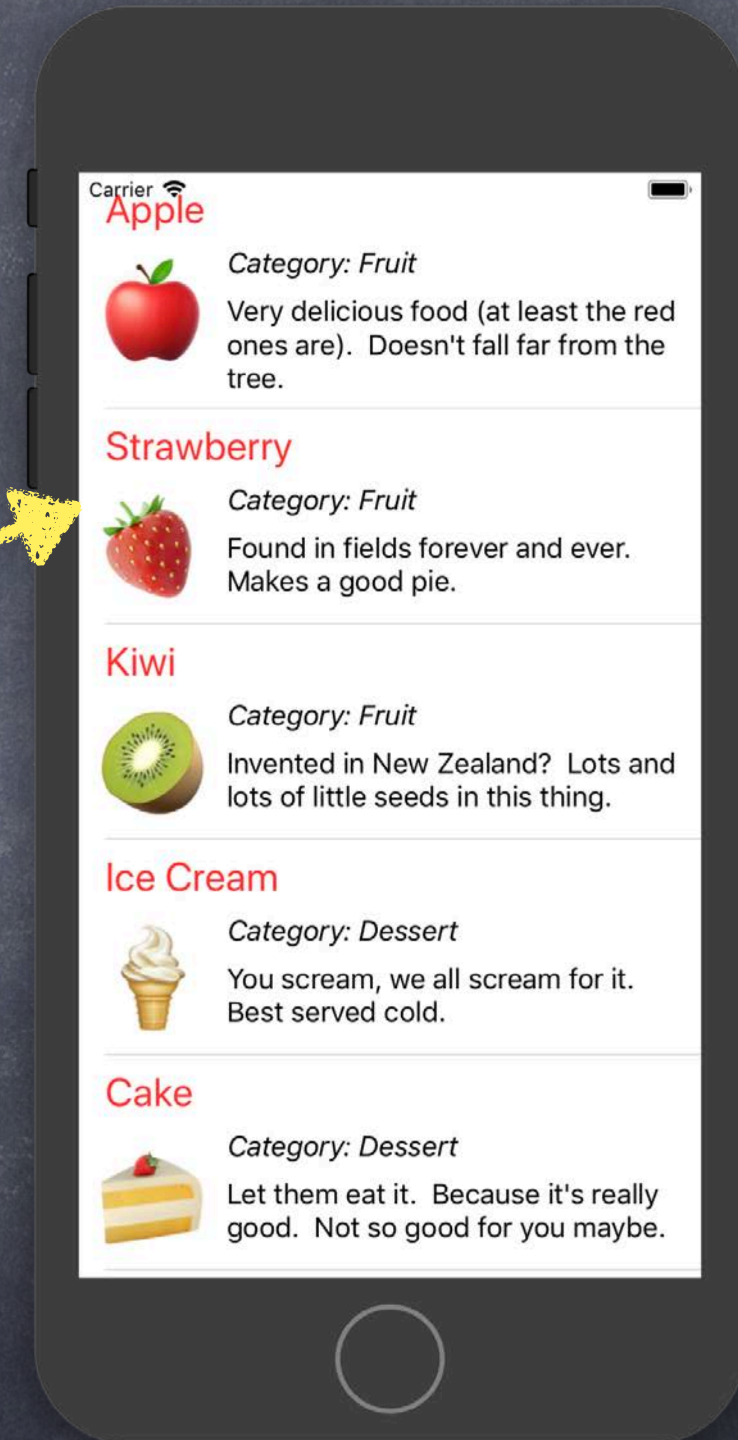


Table and Collection Views

- UITableView

The rows can also be Grouped …
(but usually only when the information in the table is fixed)

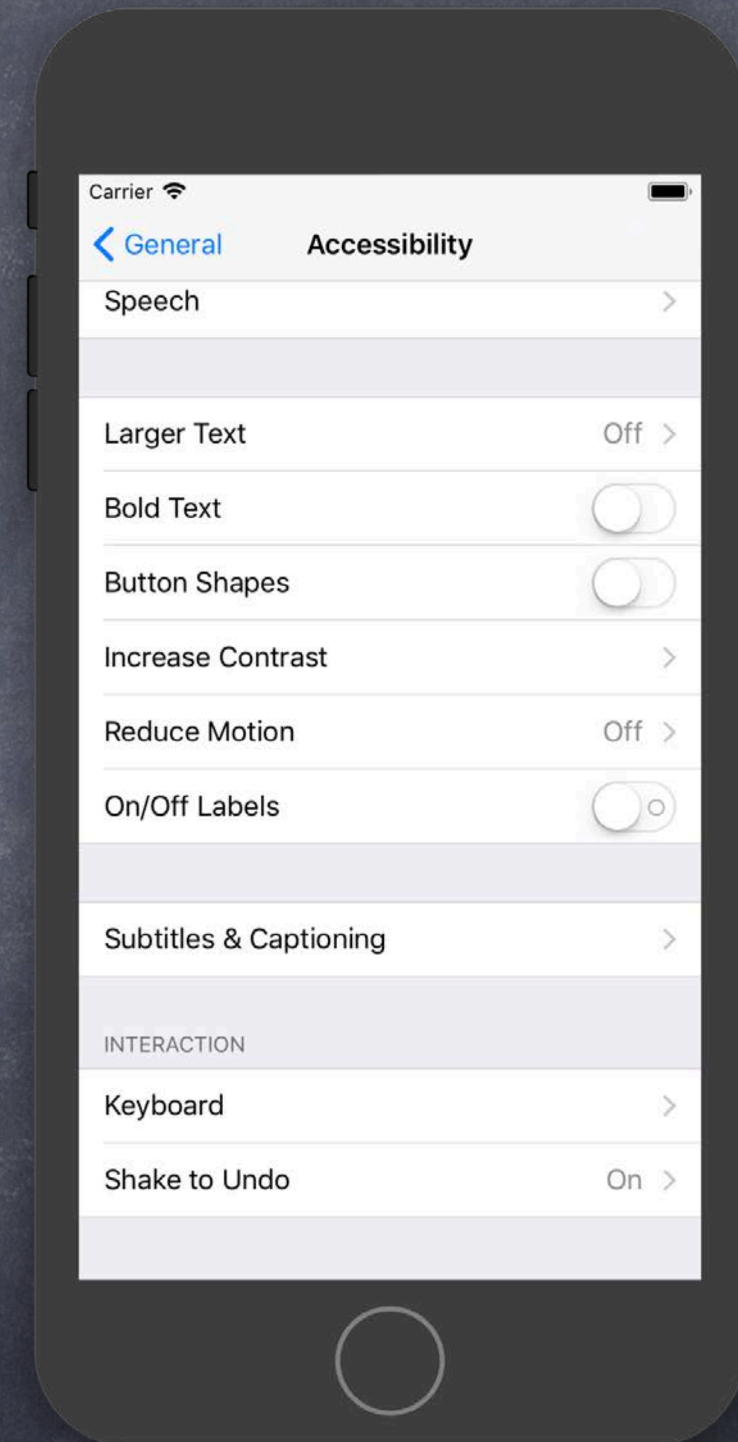


Table and Collection Views

- UICollectionView

  Is configurable to show information in any 2D arrangement.

  But by default it "flows" the items it shows like text flows.

  There is only "custom" layout of information.

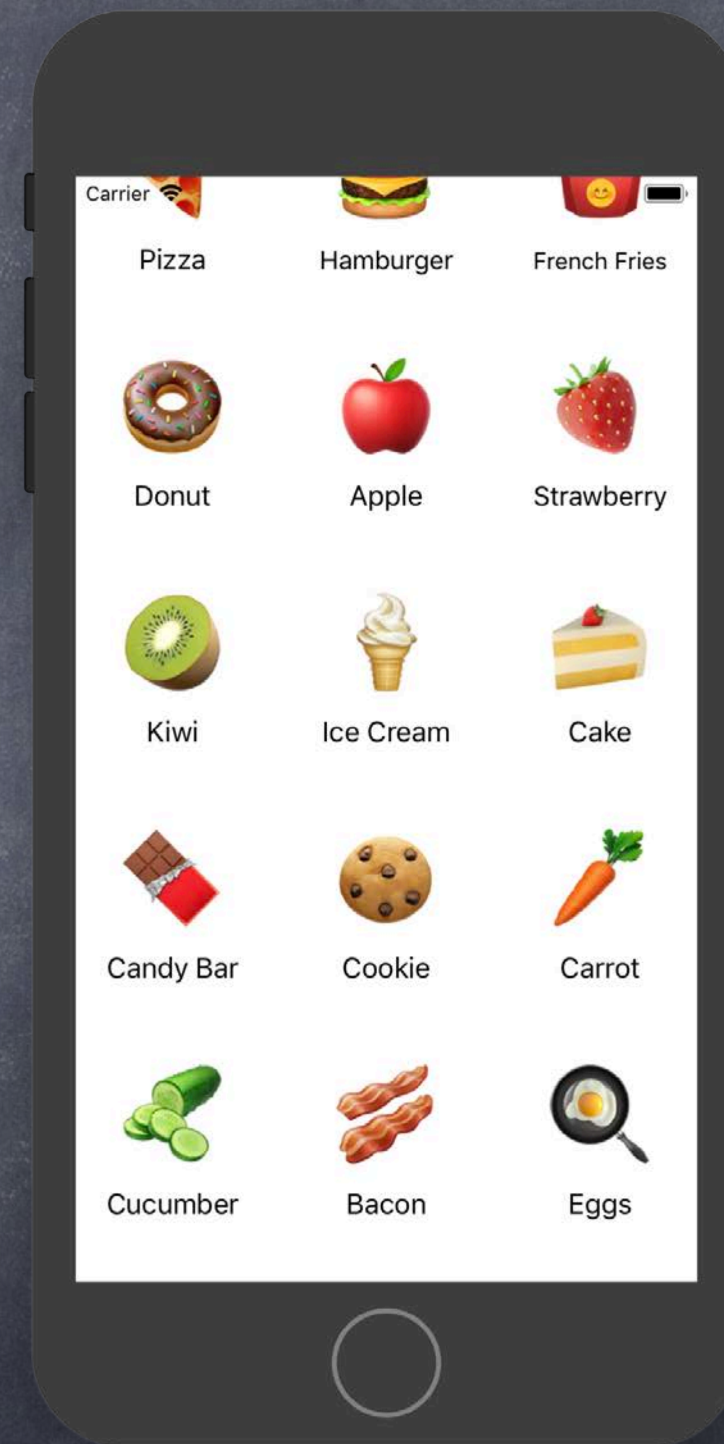


Table and Collection Views

- UICollectionView

  Is configurable to show information in any 2D arrangement.

  But by default it "flows" the items it shows like text flows.

  There is only "custom" layout of information.

  Like Table View, can also be divided into sections …

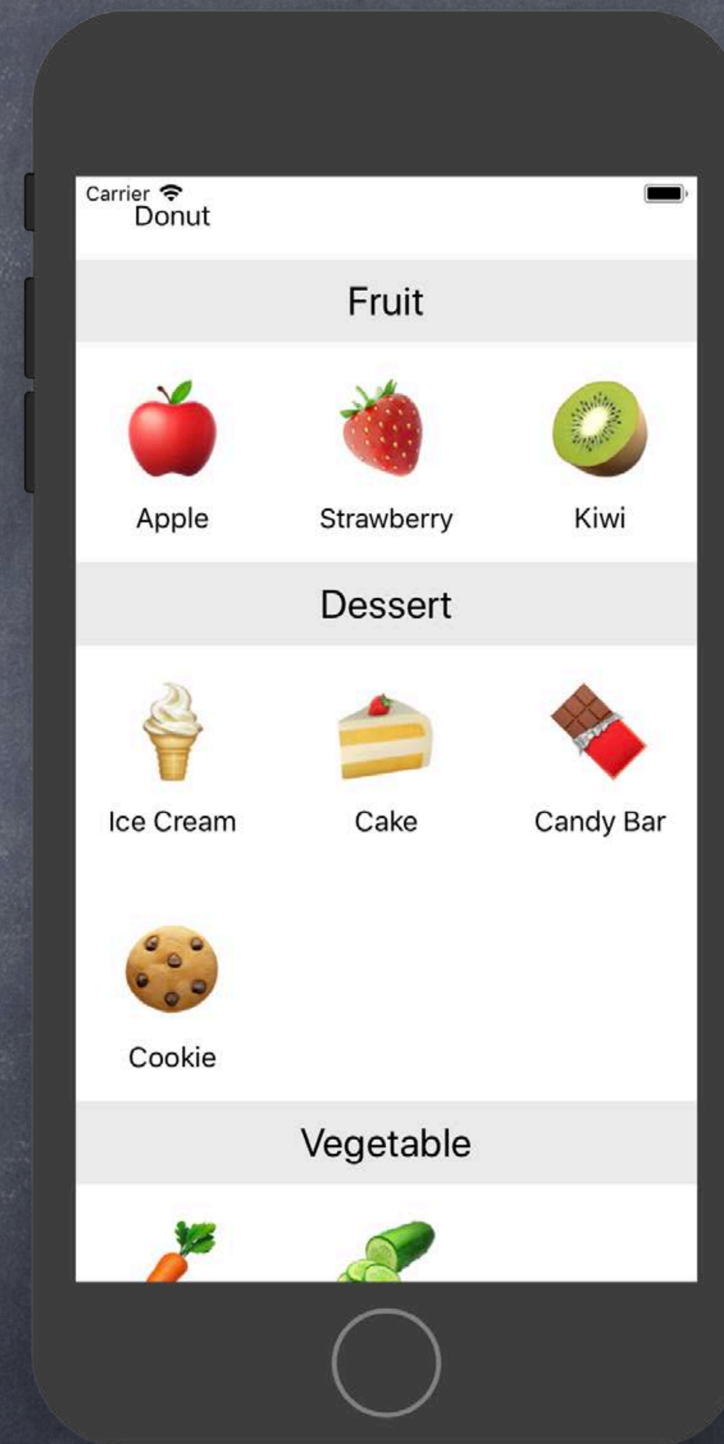
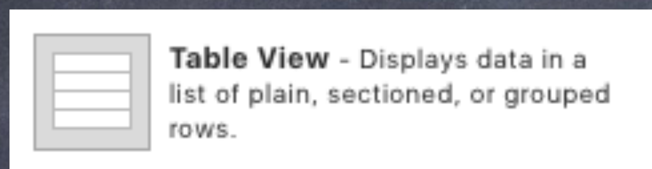
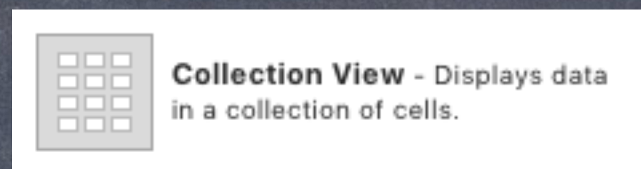


Table and Collection Views

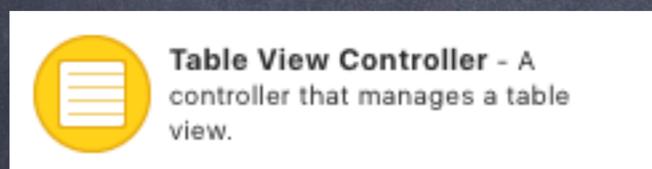
- How do you get one?

  As usual, we drag them into our storyboard …





  There are also "prepackaged" MVCs whose entire view is the table or collection view …



  If you are going to have your entire view be the table or collection view, use the latter.

Table and Collection Views

- Where does the data come from?
  The most important thing to understand about both of them is where they get their data.
  Remember that, per MVC, "views are not allowed to own their data".
  So we can't just somehow set the data in some `var`.
  Instead, we set a `var` called `dataSource`.
  The type of the `dataSource` `var` is a `protocol` with methods that supply the data.
  `dataSource` is exactly like a `delegate` in how it works.
  Table View and Collection View also have a `delegate`.
  Their `delegate` controls how they look, not what data they display (that's the `dataSource`).

Table and Collection Views

- Setting the dataSource and delegate

  In UITableView …

```
var dataSource: UITableViewDataSource
var delegate: UITableViewDelegate
```

  In UICollectionView …

```
var dataSource: UICollectionViewDataSource
var delegate: UICollectionViewDelegate
```

  These are automatically set for you if you use the prepackaged MVCs.
  If you drag out a UITableView or UICollectionView, you must set these vars yourself.
  99.99% of the time, these vars will want to be set to the Controller of the MVC.

Table and Collection Views

- The UITableView/CollectionViewDataSource protocol

  The "data retrieving" protocol has many methods.

  But these 3 are the core (UITableView abbreviated to UITV and UICollectionView to UICV) …

  UITableView

```
func numberOfSections(in tableView: UITV) -> Int
```

  UICollectionView

```
func numberOfSections(in collectionView: UICV) -> Int
```

Table and Collection Views

- The `UITableView/CollectionViewDataSource` protocol

  The "data retrieving" protocol has many methods.

  But these 3 are the core (`UITableView` abbreviated to `UITV` and `UICollectionView` to `UICV`) …

```
UITableView
func numberOfSections(in tableView: UITV) -> Int
func tableView(_ tv: UITV, numberOfRowsInSection section: Int) -> Int

UICollectionView
func numberOfSections(in collectionView: UICV) -> Int
func collectionView(_ cv: UICV, numberOfItemsInSection section: Int) -> Int
```

Table and Collection Views

- The `UITableView/CollectionViewDataSource` protocol

  The "data retrieving" protocol has many methods.

  But these 3 are the core (`UITableView` abbreviated to `UITV` and `UICollectionView` to `UICV`) ...

```
UITableView
func numberOfSections(in tableView: UITV) -> Int
func tableView(_ tv: UITV, numberOfRowsInSection section: Int) -> Int
func tableView(_ tv: UITV, cellForRowAt indexPath: IndexPath) -> UITableViewCell
UICollectionView
func numberOfSections(in collectionView: UICV) -> Int
func collectionView(_ cv: UICV, numberOfItemsInSection section: Int) -> Int
func collectionView(_ cv: UICV, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell
```

  `IndexPath` specifies which row (in TV) or item (in CV) we're talking about.

  In both, you get the section the row or item is in from `indexPath.section`.

  In TV, you get which row from `indexPath.row`; in CV you get which item from `indexPath.item`.

  CV might seem like rows and columns, but it's not, it's just items "flowing" like text.

Loading up Cells

- Putting data into the UI

  Let's focus on how we implement that last method.
  We'll look at it in the context of UITableView, but it's the same for UICollectionView.

```
func tableView(_ tv: UITV, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tv.dequeueReusableCell(withIdentifier: "MyCellId", for: indexPath)



}
```

  This gets the UITableViewCell we are going to load up with our Model data and return.
  The UITableView will then use that UITableViewCell to draw the row at the given indexPath.
  We need to understand a few things to parse this line of code …

Loading up Cells

- Cell Reuse

  A `UITableView` might have 1000s of rows (all your Music Library songs maybe?).

  If it had to create a `UIView` for all of them, it would be very inefficient.

  So it reuses the cells.

  When a `UITableViewCell` scrolls off the screen, it gets put in a pool to be reused.

  The `dequeueReusableCell(withIdentifier:)` method grabs one out of that reuse pool.

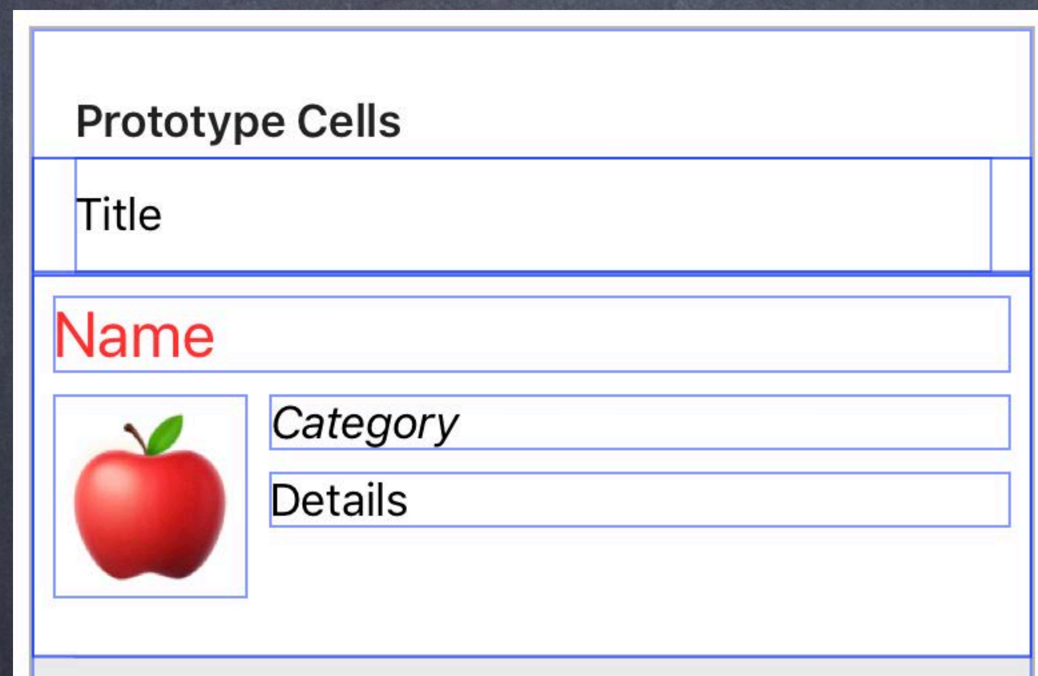
  But what if the reuse pool is empty (like when the table first appears)?

Loading up Cells

- Cell Creation

  How do new (non-reused) cells get created?

  They get created by copying a prototype cell you configure in your storyboard.



Prototype #1 (a Basic cell)

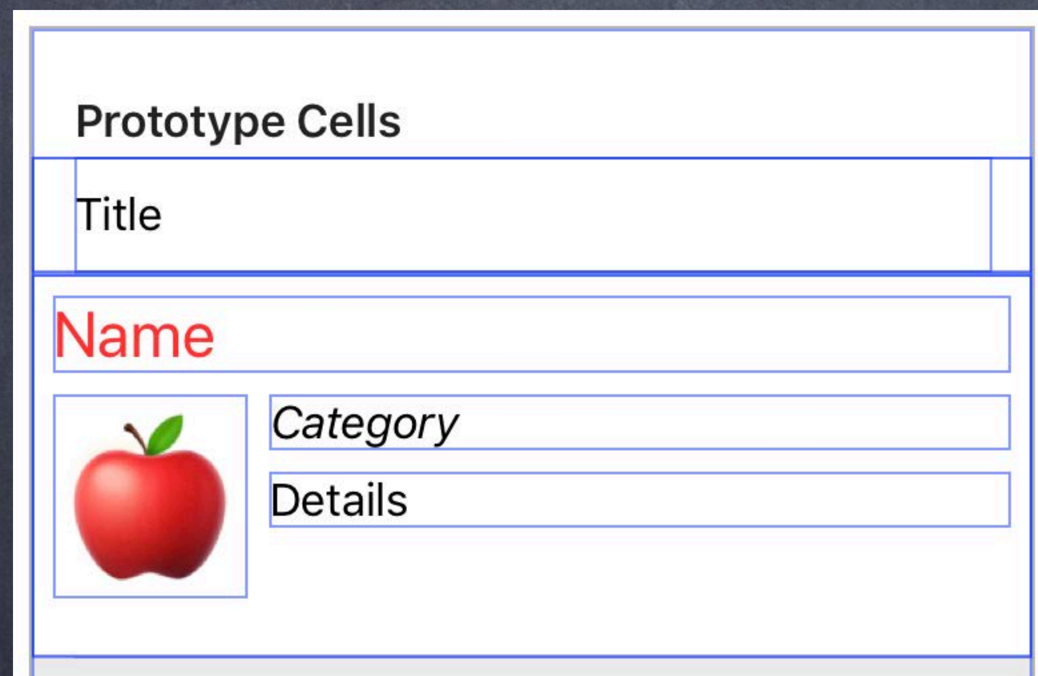
Prototype #2 (a Custom cell)

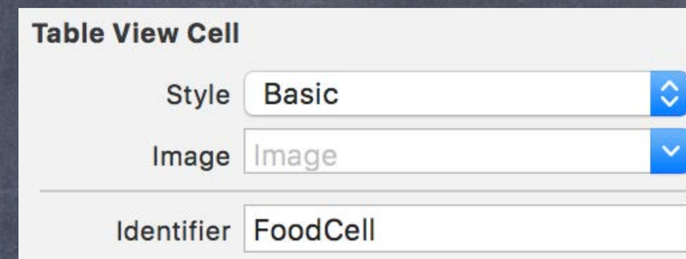
Loading up Cells

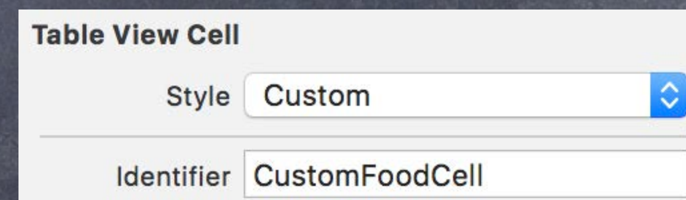
Cell Creation

How do new (non-reused) cells get created?

They get created by copying a prototype cell you configure in your storyboard.







Each prototype has an identifier you set in the Inspector.

Loading up Cells

- Implementing `cellForRowAt`

  Let's focus on how we implement that last method.
  We'll look at it in the context of `UITableView`, but it's the same for `UICollectionView`.

```
func tableView(_ tv: UITV, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tv.dequeueReusableCell(withIdentifier: "MyCellId", for: indexPath)




}
```

  So now we can understand this line of code.
  It is reusing a `UITableViewCell` with the given identifier if possible.
  Otherwise it is making a copy of the prototype in the storyboard.
  The fact that cells are reused has serious implications for multithreading!
  By the time something returns from another thread, a cell might have been reused.

Loading up Cells

- Implementing `cellForRowAt`

  Let's focus on how we implement that last method.
  We'll look at it in the context of `UITableView`, but it's the same for `UICollectionView`.

```
func tableView(_ tv: UITV, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let prototype = decision ? "FoodCell" : "CustomFoodCell"
    let cell = tv.dequeueReusableCell(withIdentifier: prototype , for: indexPath)


}
```

The `decision` can be made based on many factors.
Usually its based on the `indexPath` (i.e. which row we're displaying here).
But it might also be based on the data in our Model at that `indexPath`.
Some data might be an image, whereas other data is text, etc.

Loading up Cells

- Implementing `cellForRowAt`

  Let's focus on how we implement that last method.

  We'll look at it in the context of `UITableView`, but it's the same for `UICollectionView`.

```
func tableView(_ tv: UITV, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let prototype = decision ? "FoodCell" : "CustomFoodCell"
    let cell = tv.dequeueReusableCell(withIdentifier: prototype, for: indexPath)


}
```

So what API can we use to configure this cell that we just reused/created?

Loading up Cells

- Implementing `cellForRowAt`

  Let's focus on how we implement that last method.
  We'll look at it in the context of `UITableView`, but it's the same for `UICollectionView`.

```
func tableView(_ tv: UITV, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let prototype = decision ? "FoodCell" : "CustomFoodCell"
    let cell = tv.dequeueReusableCell(withIdentifier: prototype, for: indexPath)
    cell.textLabel?.text = food(at: indexPath)
    cell.detailTextLabel?.text = emoji(at: indexPath)
}
```

  So what API can we use to configure this cell that we just reused/created?
  Well, for `UITableView` only, the default `UITableViewCell` has a few basic things …
  `textLabel`, `detailTextLabel` and `imageView`

Loading up Cells

- Implementing cellForRowAt

Let's focus on how we implement that last method.
We'll look at it in the context of UITableView, but it's the same for UICollectionView.

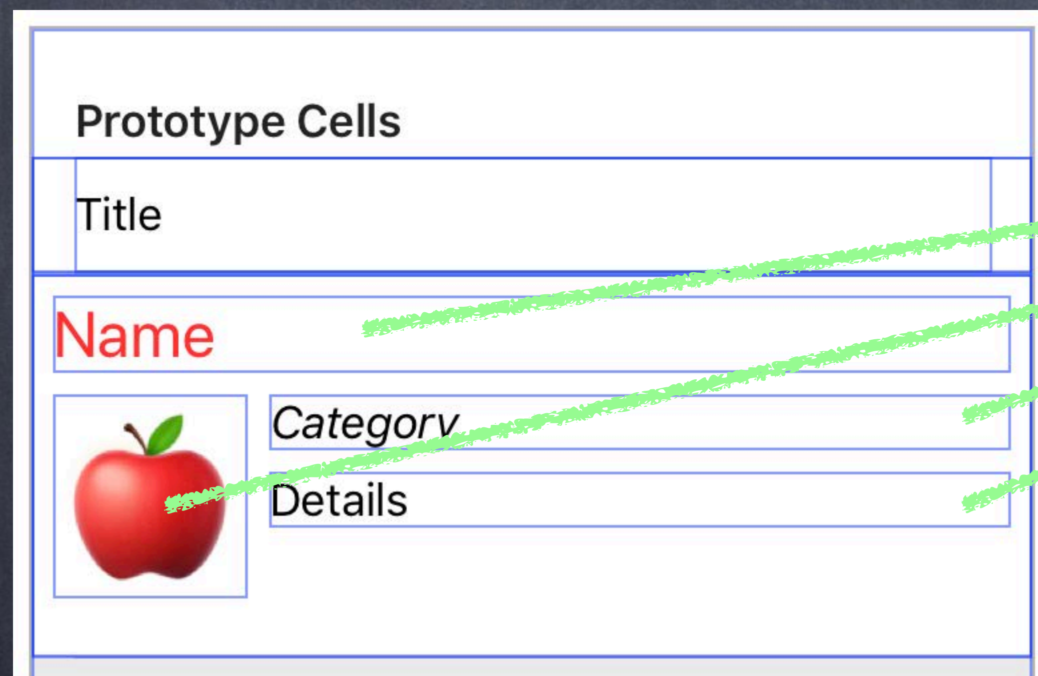
```
func tableView(_ tv: UITV, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let prototype = decision ? "FoodCell" : "CustomFoodCell"
    let cell = tv.dequeueReusableCell(withIdentifier: prototype, for: indexPath)


}
```

So what API can we use to configure this cell that we just reused/created?
Well, for UITableView only, the default UITableViewCell has a few basic things …
textLabel, detailTextLabel and imageView
But for UICollectionView and for custom UITableViewCells, WE have to provide the API.
Let's see how we do that …

Loading up Cells

- Custom `UITableViewCell` subclass

  When we put custom UI into a `UITableViewCell` prototype, we probably need outlets to it.

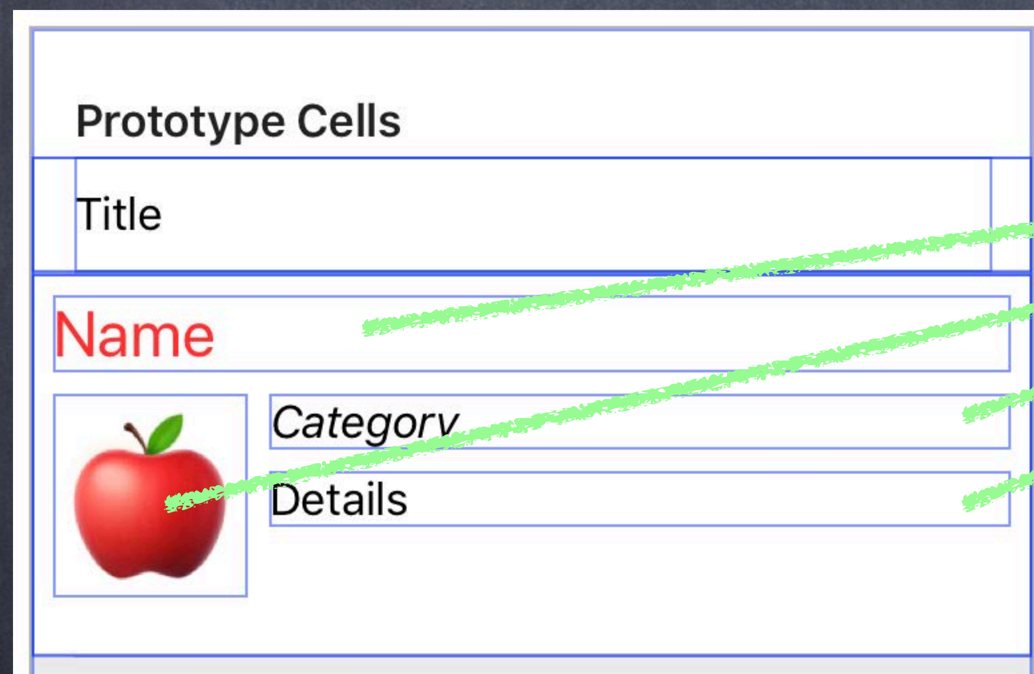



```
{
    @IBOutlet var name: UILabel
    @IBOutlet var emoji: UILabel
    @IBOutlet var category: UILabel
    @IBOutlet var details: UILabel
}
```

Loading up Cells

- Custom `UITableViewCell` subclass

  When we put custom UI into a `UITableViewCell` prototype, we probably need outlets to it.
  Can we hook them up directly to our Controller?




```
class MyTVC: UITableViewController
{
    @IBOutlet var name: UILabel
    @IBOutlet var emoji: UILabel
    @IBOutlet var category: UILabel
    @IBOutlet var details: UILabel
}
```

Loading up Cells

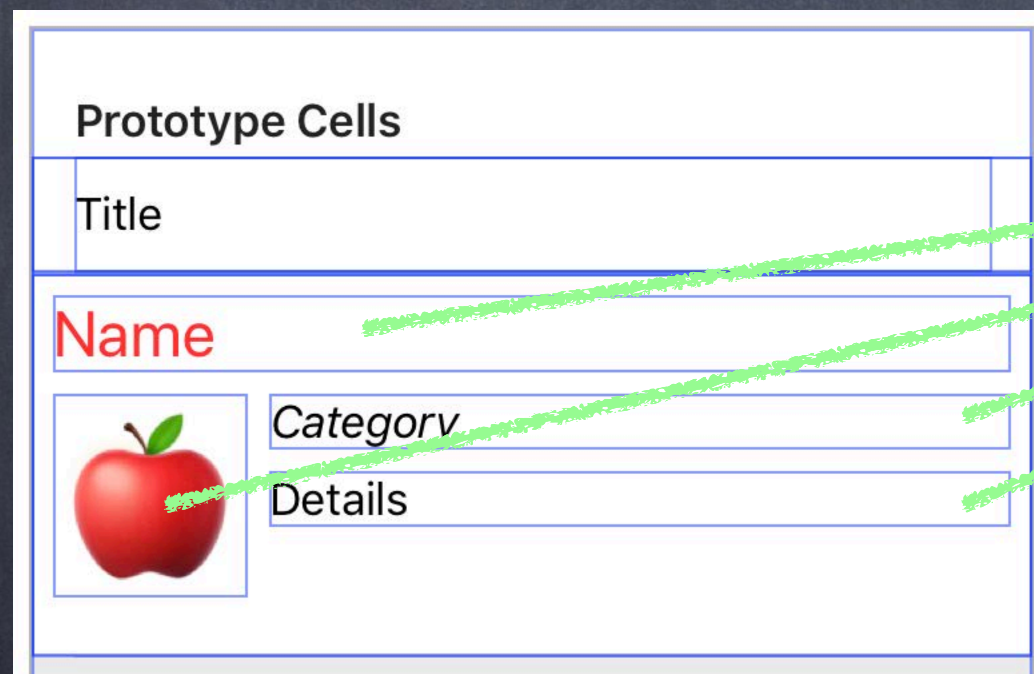
- Custom `UITableViewCell` subclass

  When we put custom UI into a `UITableViewCell` prototype, we probably need outlets to it.

  Can we hook them up directly to our Controller?

  No, we can't, because there might be multiple rows with that type of cell.

  They can't all be hooked up to the same single outlet!

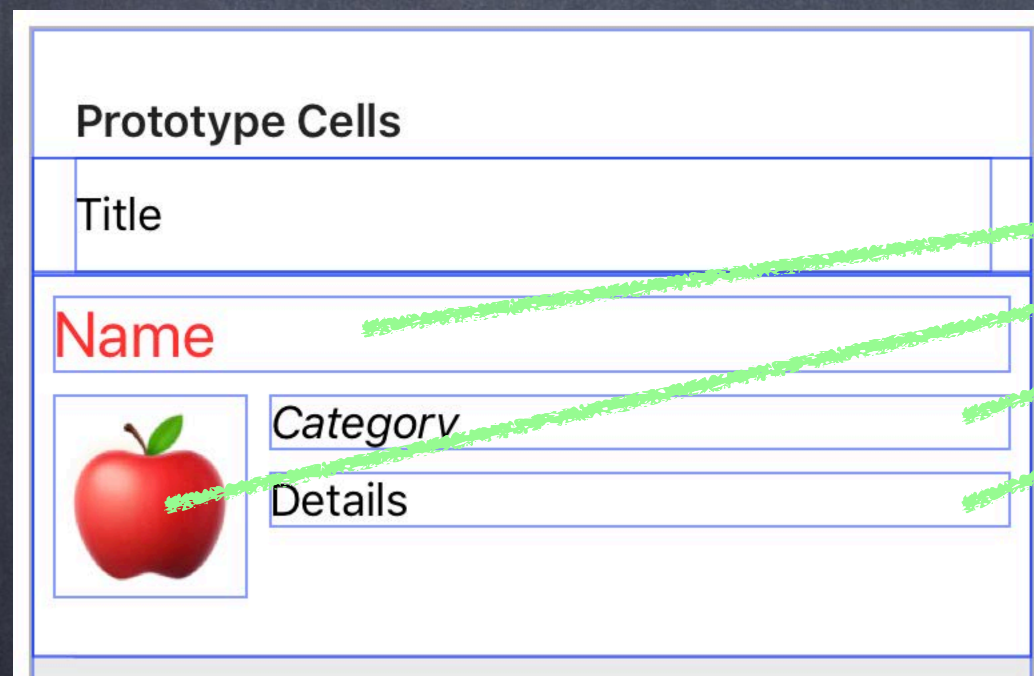



```
class MyTVC: UITableViewController
{
    @IBOutlet var name: UILabel
    @IBOutlet var emoji: UILabel
    @IBOutlet var category: UILabel
    @IBOutlet var details: UILabel
}
```

Loading up Cells

- Custom UITableViewCell subclass

When we put custom UI into a UITableViewCell prototype, we probably need outlets to it.
Can we hook them up directly to our Controller?
No, we can't, because there might be multiple rows with that type of cell.
They can't all be hooked up to the same single outlet!
Instead, we have to subclass UITableViewCell and put the outlets in there.

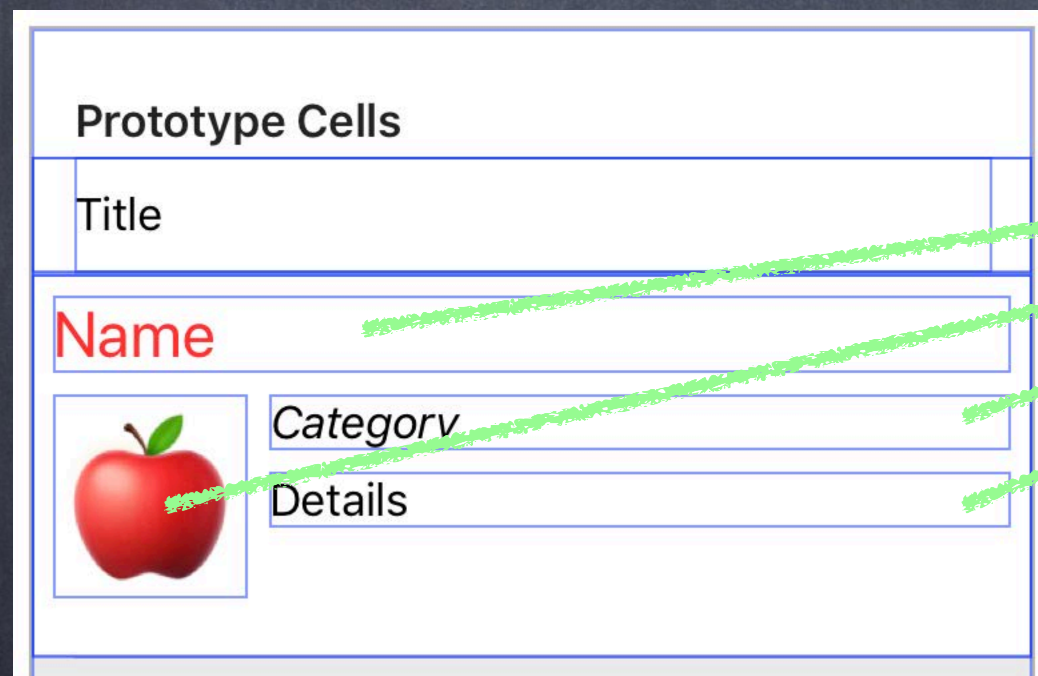



```
class MyTVC: UITableViewCell
{
    @IBOutlet var name: UILabel
    @IBOutlet var emoji: UILabel
    @IBOutlet var category: UILabel
    @IBOutlet var details: UILabel
}
```

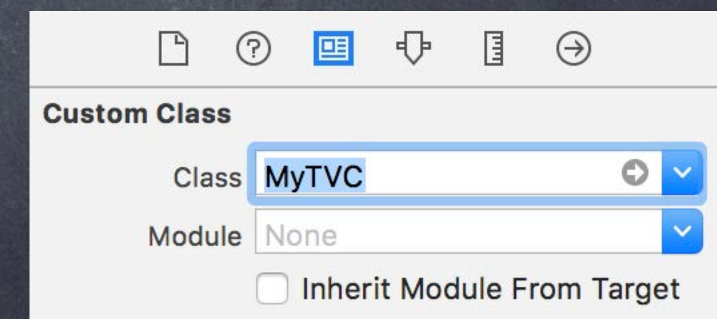
Loading up Cells

- Custom UITableViewCell subclass

When we put custom UI into a UITableViewCell prototype, we probably need outlets to it.
Can we hook them up directly to our Controller?
No, we can't, because there might be multiple rows with that type of cell.
They can't all be hooked up to the same single outlet!
Instead, we have to subclass UITableViewCell and put the outlets in there.




```
class MyTVC: UITableViewCell
{
    @IBOutlet var name: UILabel
    @IBOutlet var emoji: UILabel
    @IBOutlet var category: UILabel
    @IBOutlet var details: UILabel
}
```



Then we inspect the cell in the Identity Inspector
and change its class from UITableViewCell to MyTVC

Loading up Cells

- Implementing `cellForRowAt`

  Let's focus on how we implement that last method.
  We'll look at it in the context of `UITableView`, but it's the same for `UICollectionView`.

```
func tableView(_ tv: UITV, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let prototype = decision ? "FoodCell" : "CustomFoodCell"
    let cell = tv.dequeueReusableCell(withIdentifier: prototype, for: indexPath)
    if let myTVCell = cell as? MyTVC {

    }
}
```

  In order to get at those outlets, we need to cast our `UITableViewCell` to our subclass.

Loading up Cells

- Implementing `cellForRowAt`

  Let's focus on how we implement that last method.
  We'll look at it in the context of `UITableView`, but it's the same for `UICollectionView`.

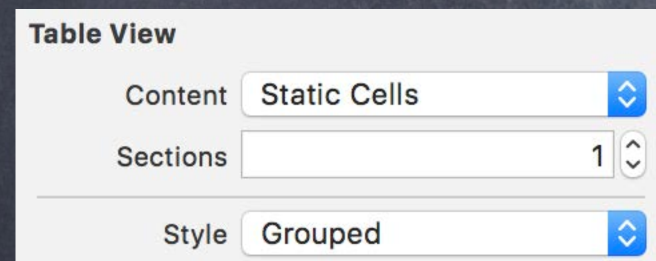
```
func tableView(_ tv: UITV, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let prototype = decision ? "FoodCell" : "CustomFoodCell"
    let cell = tv.dequeueReusableCell(withIdentifier: prototype, for: indexPath)
    if let myTVCell = cell as? MyTVC {
        myTVC.name = food(at: indexPath); myTVC.emoji = emoji(at: indexPath)
    }
}
```

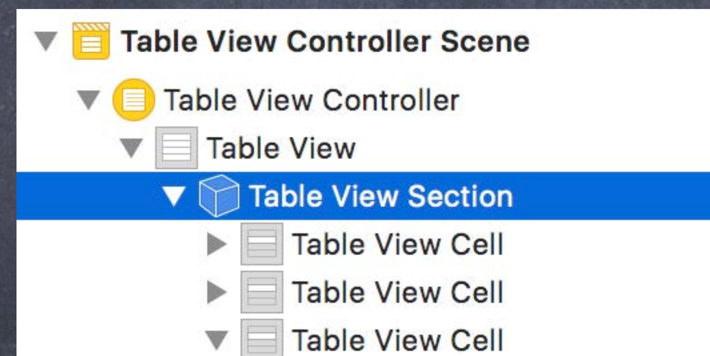
  In order to get at those outlets, we need to cast our `UITableViewCell` to our subclass.
  Then we can access its outlets (or any other API it wants to make public).
  In Collection View, we always have to do this (there are only "Custom" cells).
  In Table View, we do it when the simple Basic, Subtitle, etc. styles aren't enough.

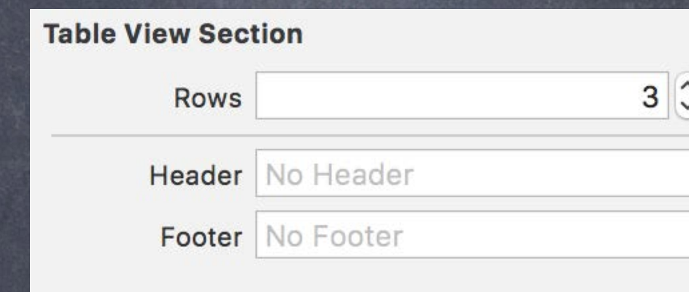
Static Table View

- Using Table View purely for UI layout

  Sometimes we just use a table view to lay out UI elements.
  A fantastic example of this is the iOS Settings app.
  In this case, you do not need to do any of the `UITableViewDataSource` stuff.
  And you can connect outlets directly to your Controller (because there's only one of each cell).
  To do this, just set your `UITableView` to have Static Cells instead of Dynamic Prototypes.
  Usually static table views are Style Grouped.
  Then pick the section in the Document Outline you want to add cells to and add them.

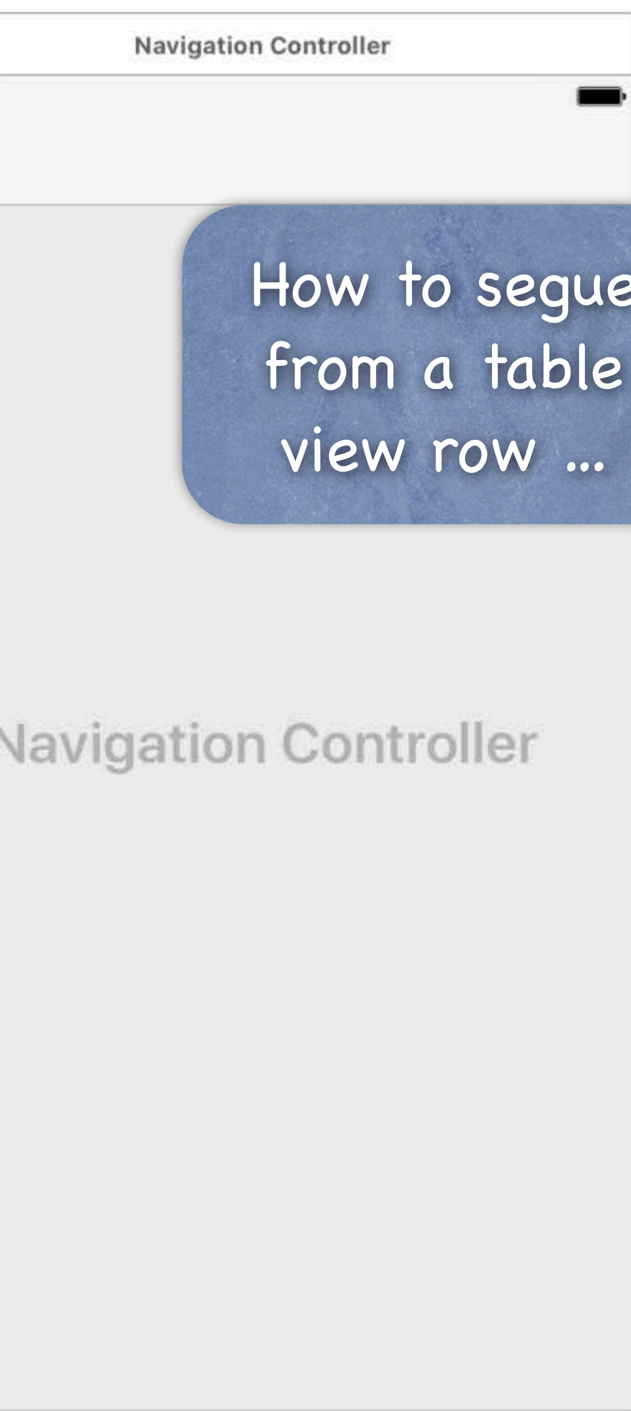




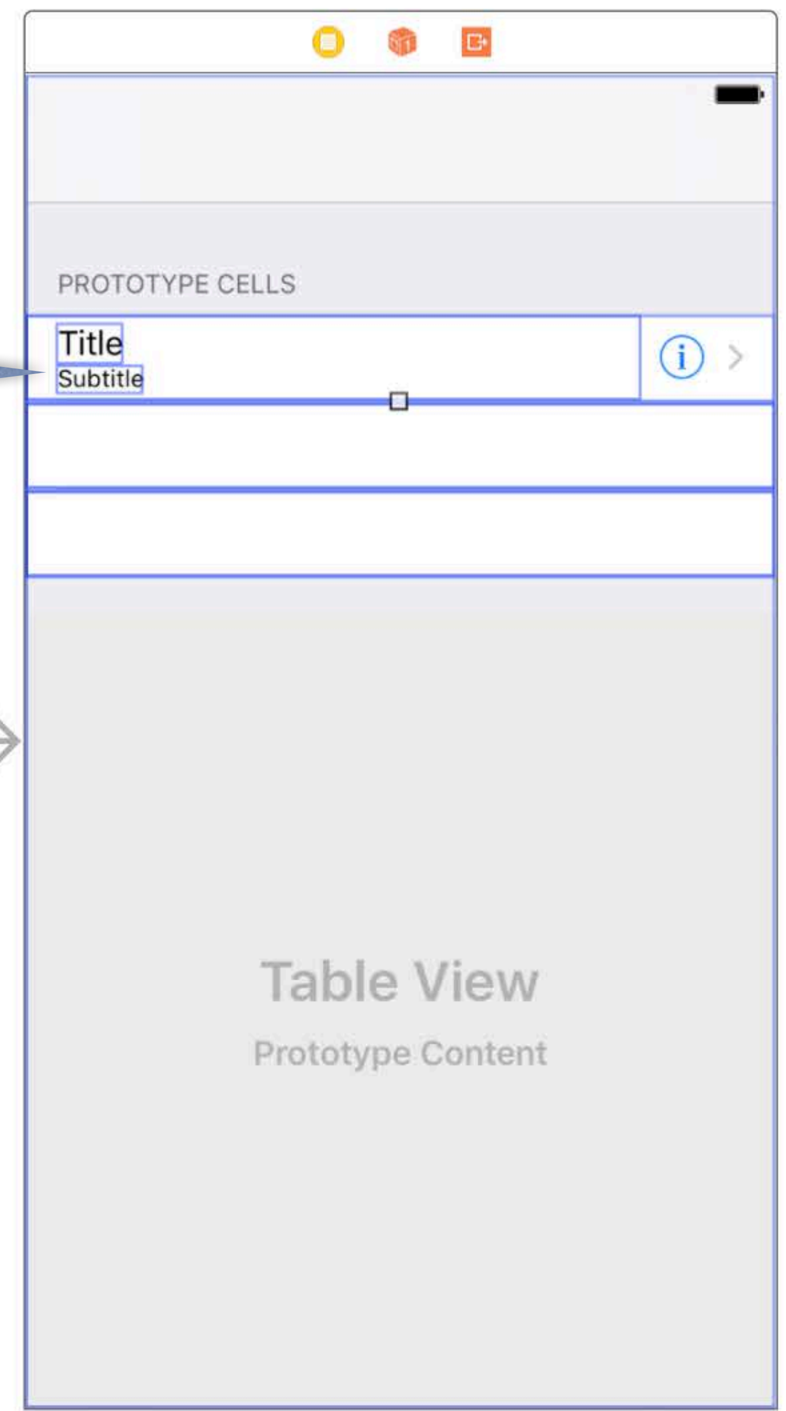


TVCExample 〉 iPhone 7 TVCExample: **Ready**

TVCExample 〉 TVCExample



How to segue from a table view row ...



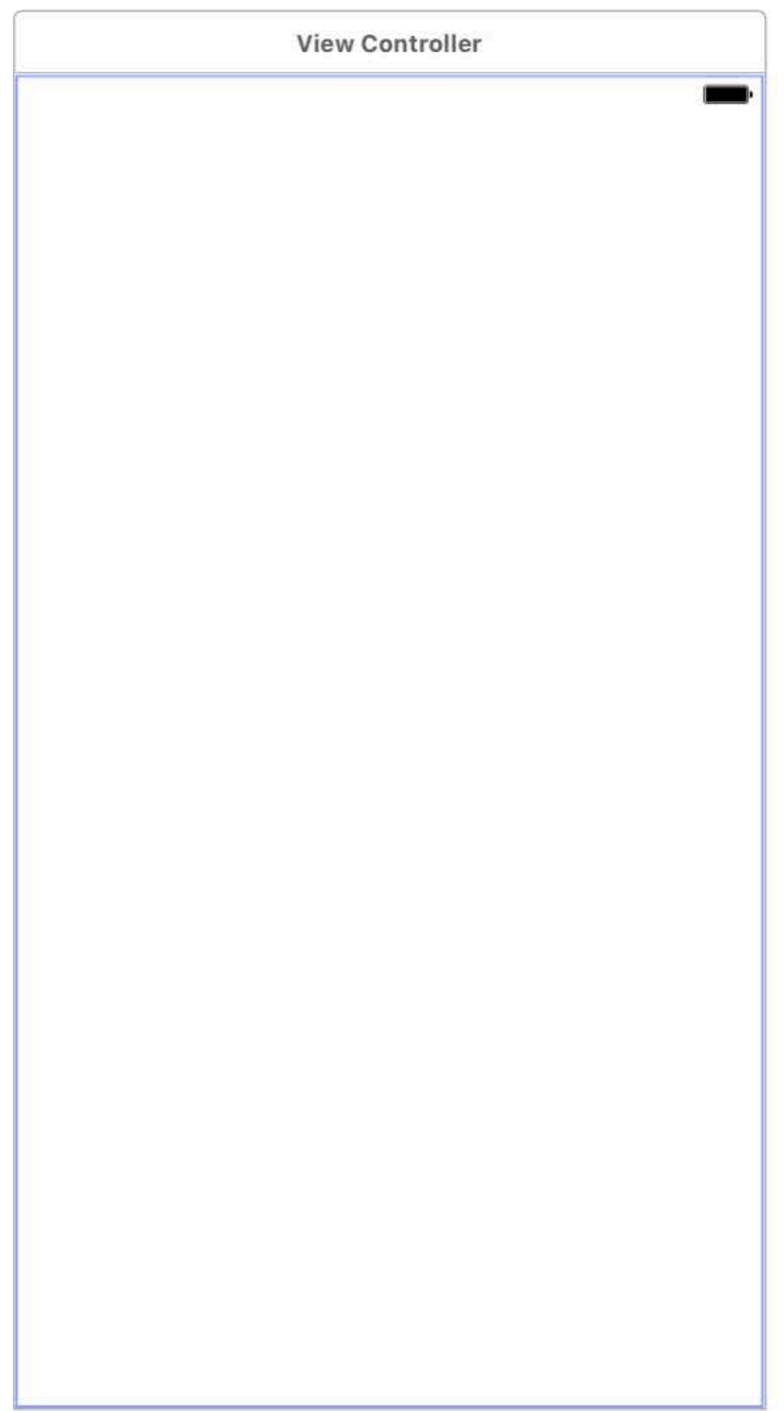


Table View Cell

Style: Subtitle
Image: Image
Identifier: MyCell
Selection: Default
Accessory: Detail Disclosure
Editing Acc.: None
Focus Style: Default
Indentation: 0 (Level), 10 (Width)
☑ Indent While Editing
☐ Shows Re-order Controls
Separator: Default Insets

View

Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: ☑ User Interaction Enabled
☐ Multiple Touch
Alpha: 1
Background
Tint: Default
Drawing: ☑ Opaque
☐ Hidden
☑ Clears Graphics Context
☑ Clip To Bounds
☑ Autoresize Subviews
Stretching: 0 0

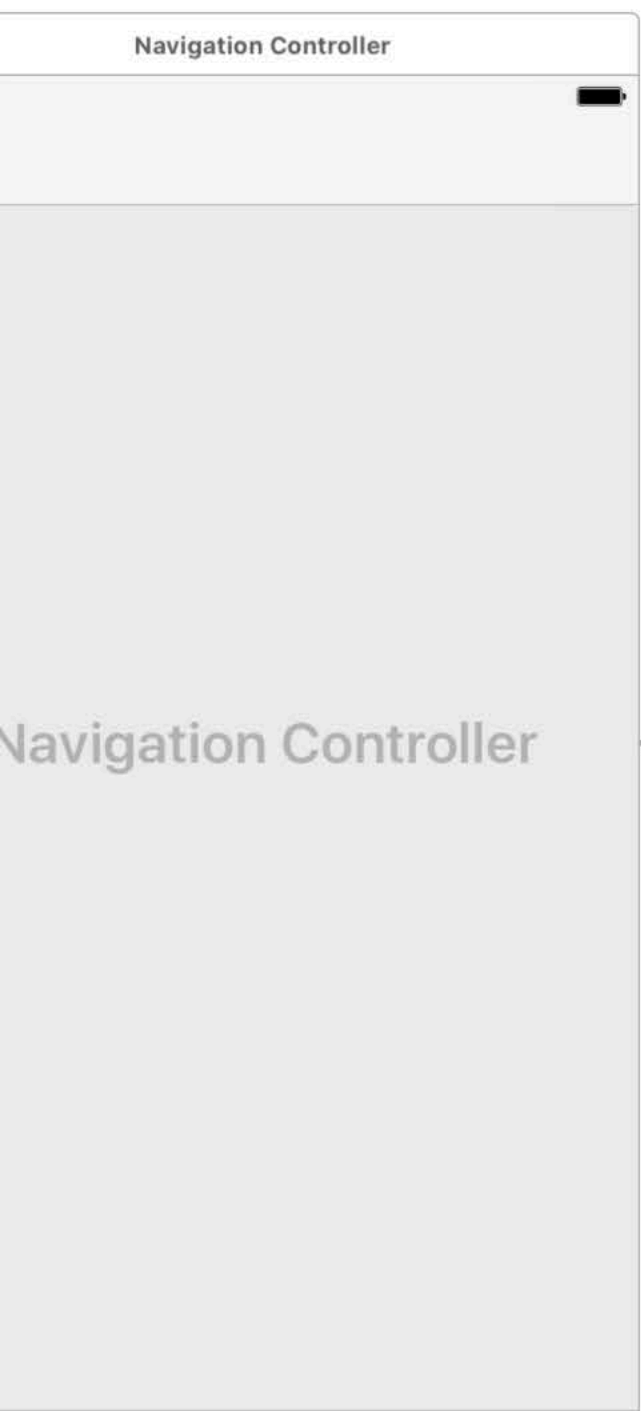
View as: iPhone 7 (wC hR) — 75%

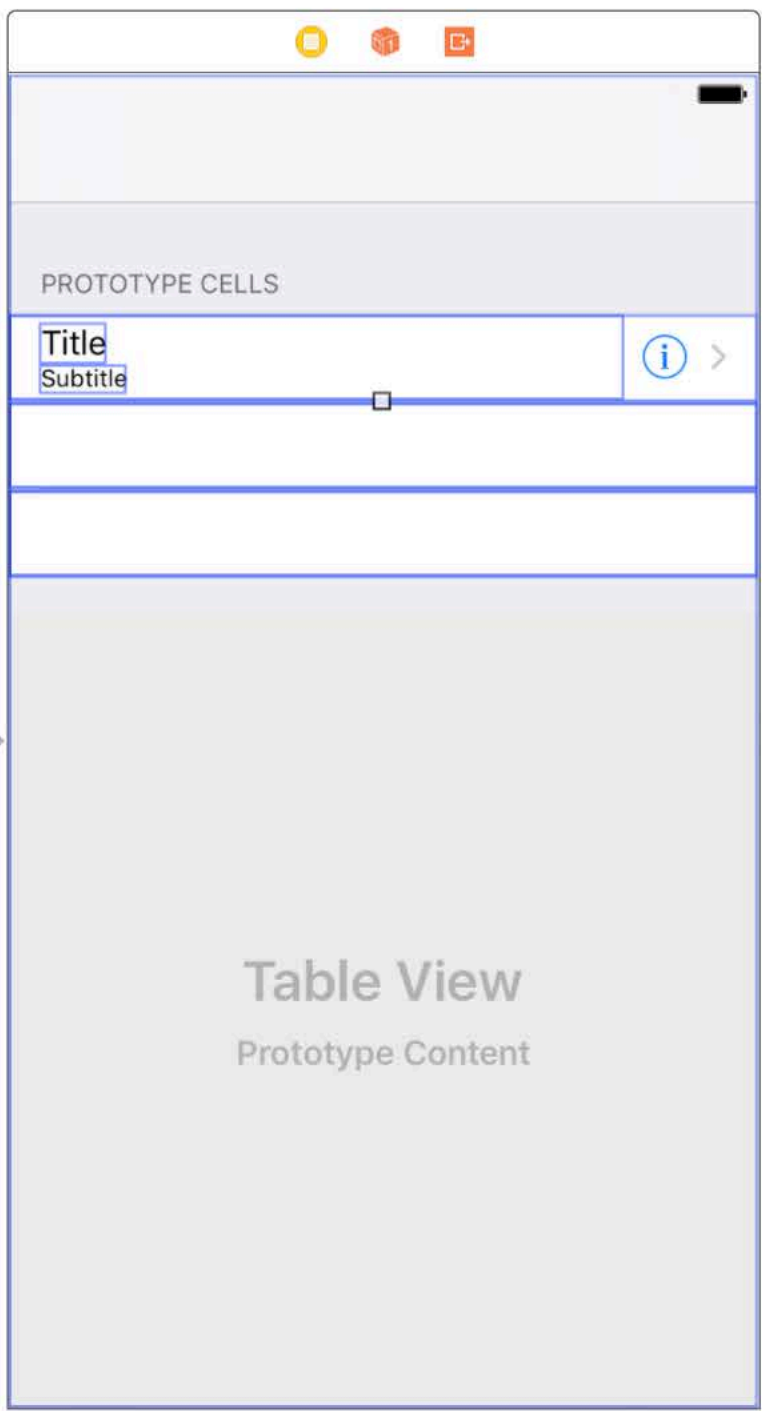
CS193p
Fall 2017-18

TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample





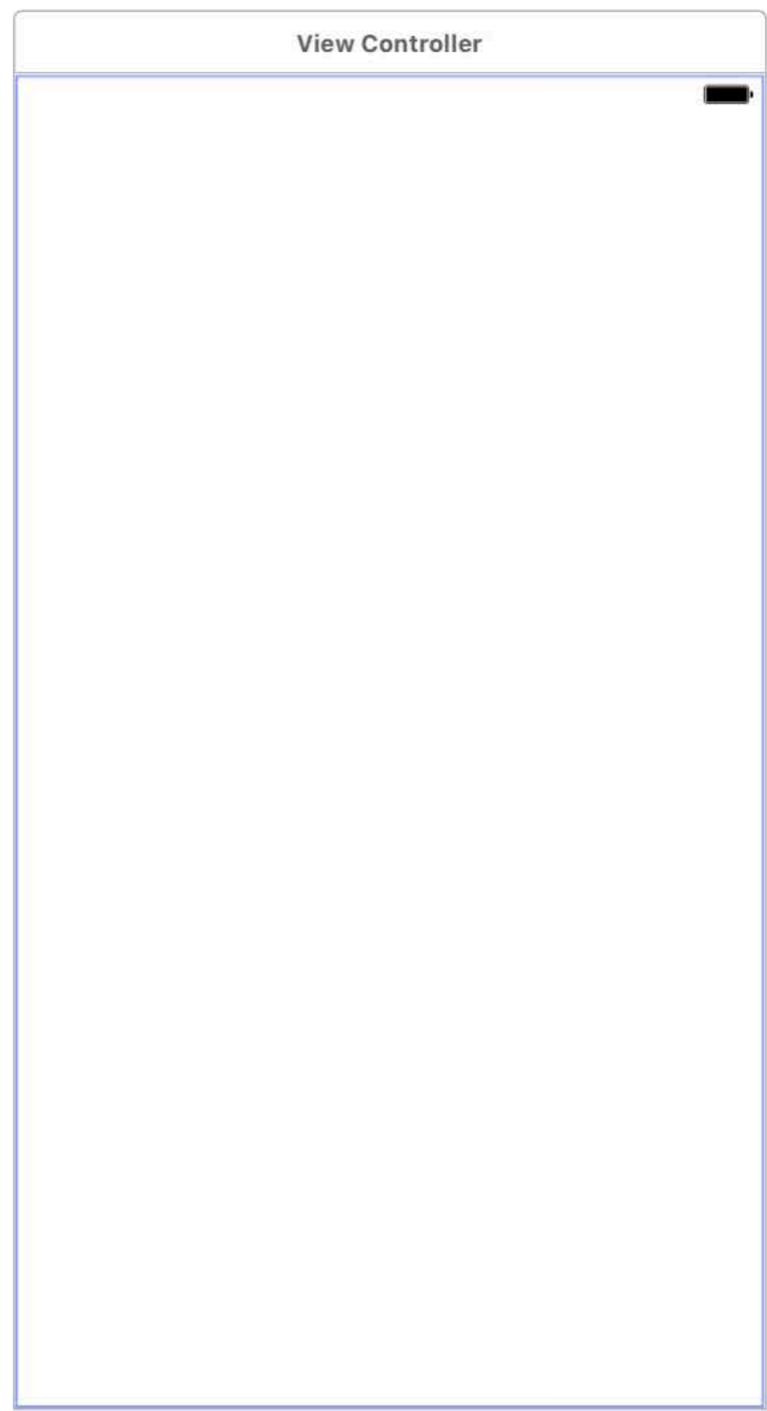


Table View Cell

Style: Subtitle

Image: Image

Identifier: MyCell

Selectio…

Accesso…

- None
- Disclosure Indicator
- ✓ Detail Disclosure
- Checkmark
- Detail

Editing Ac…

Focus Style: Default

Indentation: 0 (Level), 10 (Width)

- ☑ Indent While Editing
- ☐ Shows Re-order Controls

Separator: Default Insets

View

Content Mode: Scale To Fill

Semantic: Unspecified

Tag: 0

Interaction:
- ☑ User Interaction Enabled
- ☐ Multiple Touch

Alpha: 1

Background

Tint: Default

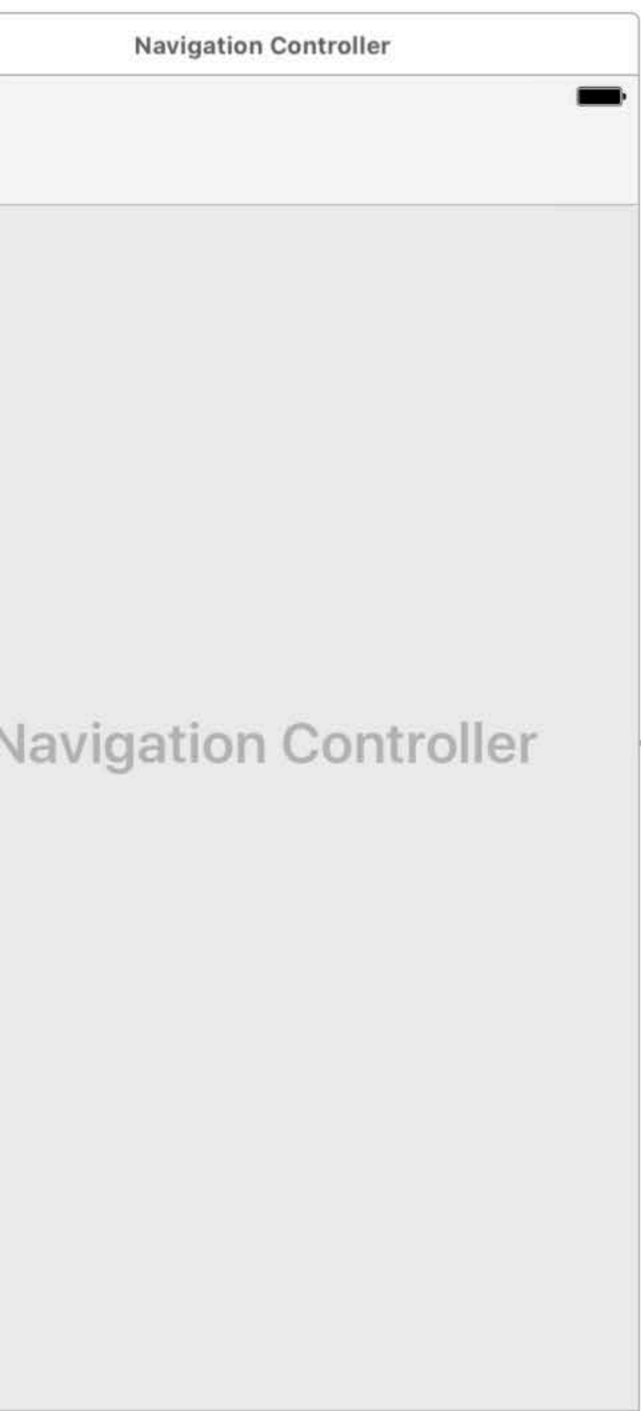
Drawing:
- ☑ Opaque
- ☐ Hidden
- ☑ Clears Graphics Context
- ☑ Clip To Bounds
- ☑ Autoresize Subviews

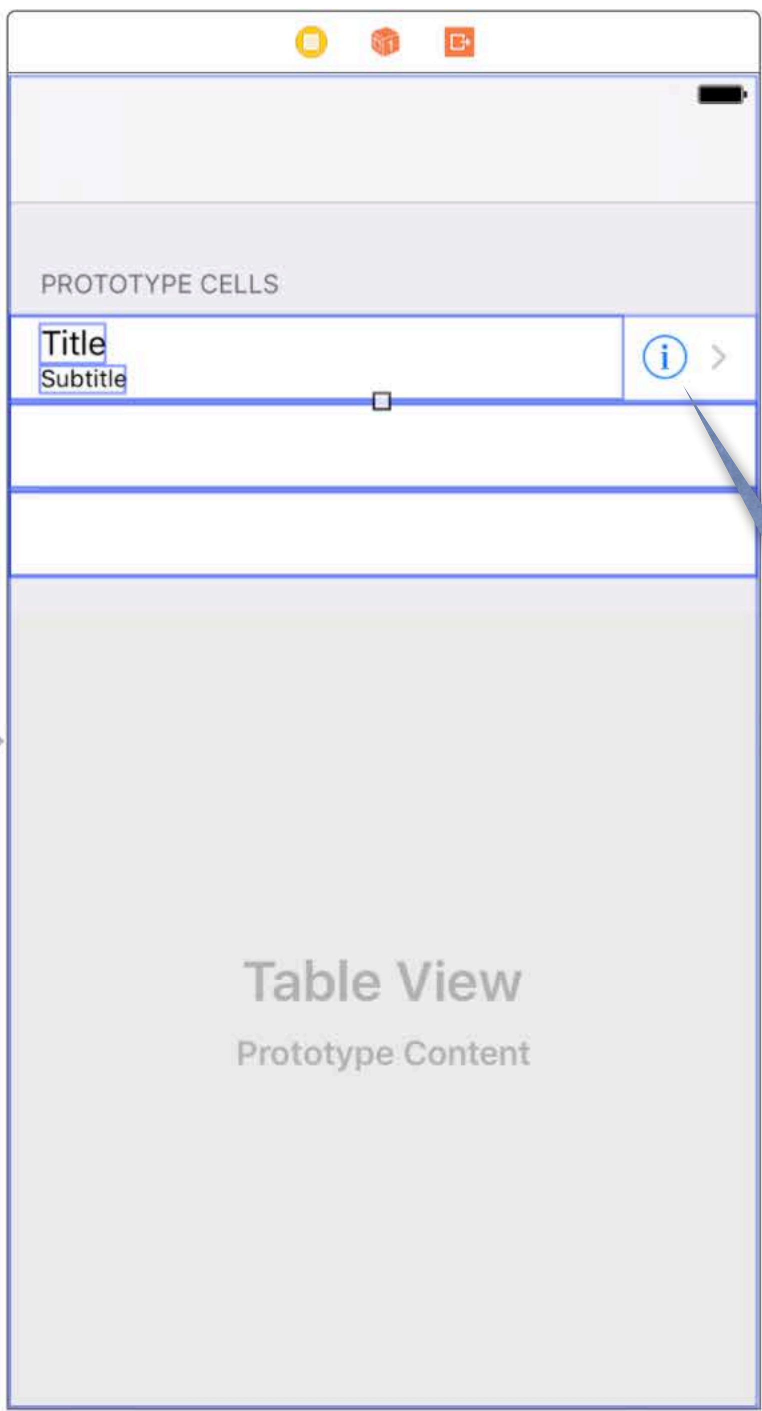
Stretching: 0 0

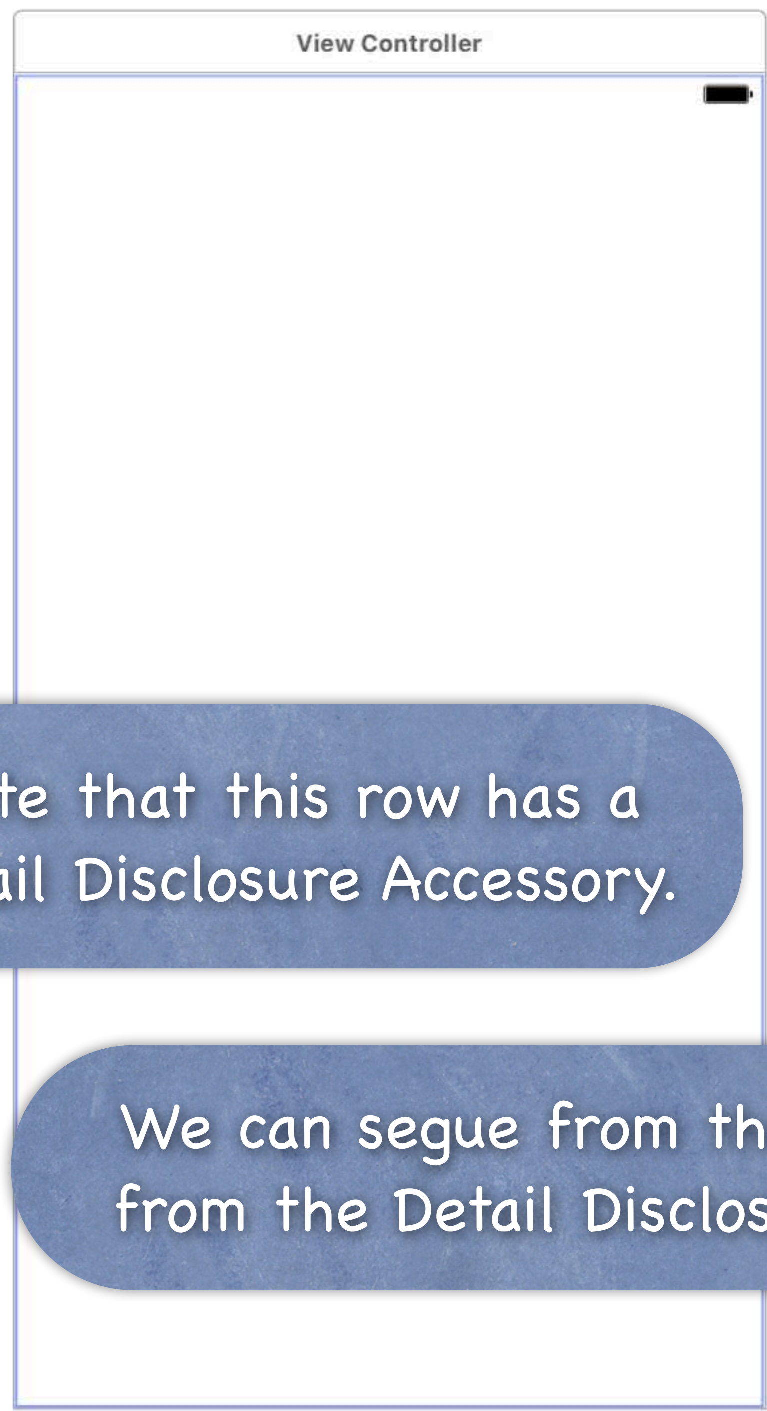
View as: iPhone 7 (wC hR)

75%

CS193p
Fall 2017-18







Note that this row has a Detail Disclosure Accessory.

We can segue from the row and/or from the Detail Disclosure Accssory.

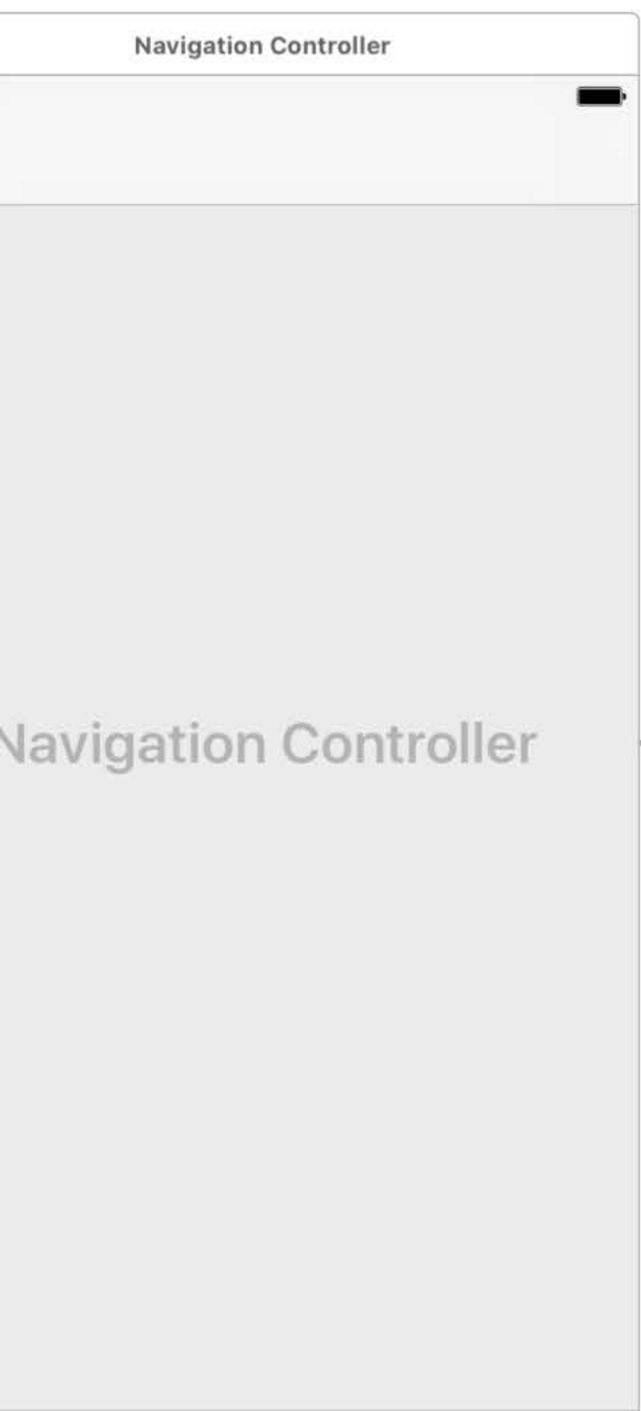
Table View Cell

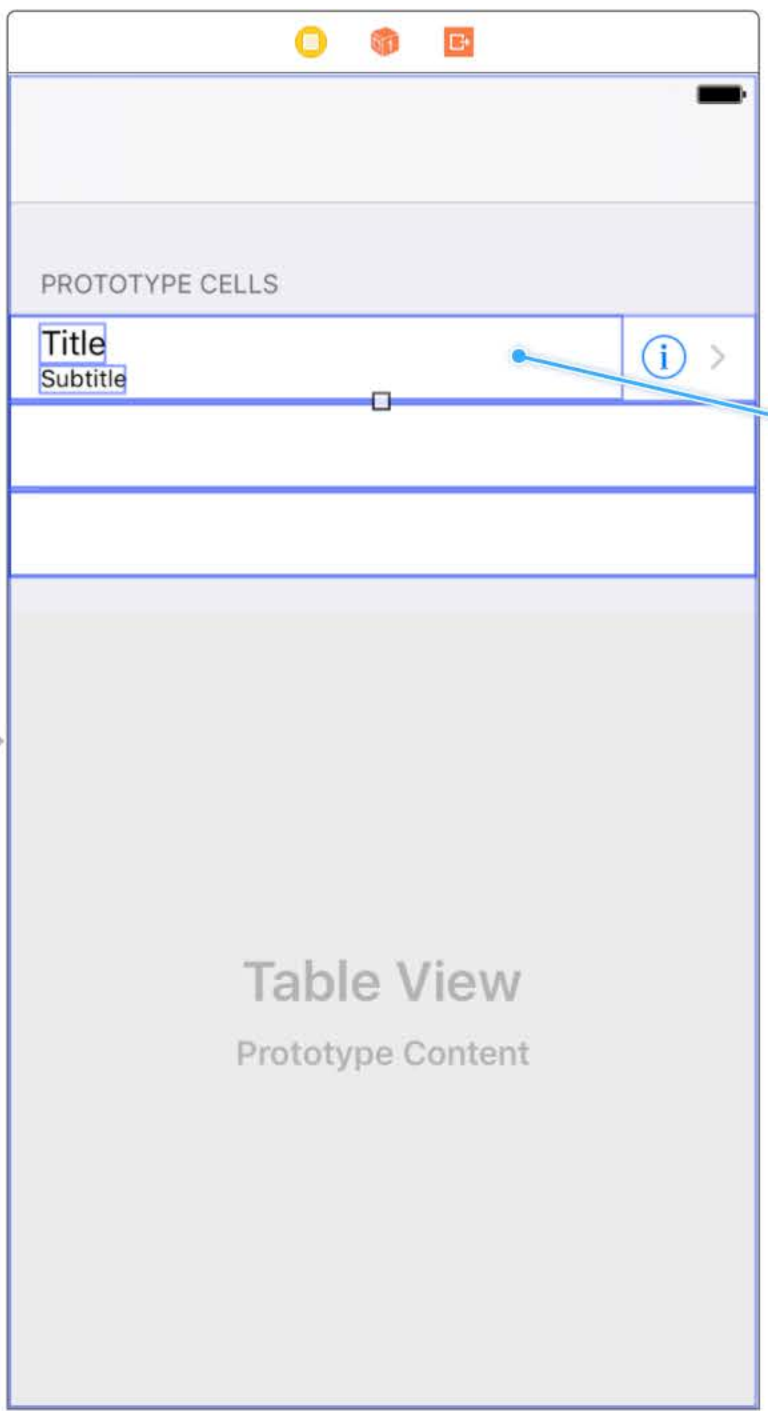
Style: Subtitle
Image: Image
Identifier: MyCell
Accessory: None / Disclosure Indicator / ✓ Detail Disclosure / Checkmark / Detail

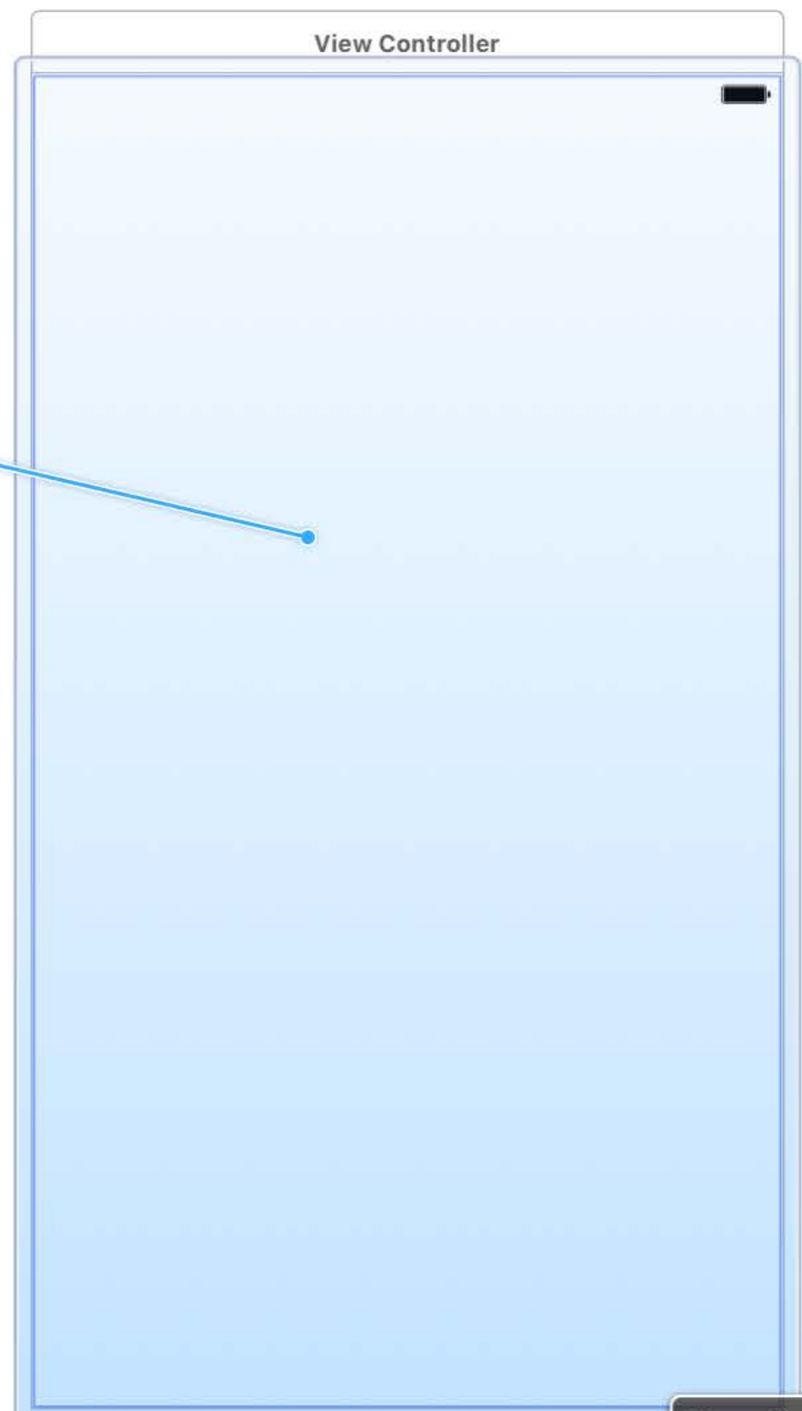
TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample







View Contr

Table View Cell

Style: Subtitle
Image: Image
Identifier: MyCell
Selection: Default
Accessory: Detail Disclosure
Editing Acc.: None
Focus Style: Default
Indentation: 0 (Level) 10 (Width)
☑ Indent While Editing
☐ Shows Re-order Controls
Separator: Default Insets

View

Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: ☑ User Interaction Enabled
☐ Multiple Touch
Alpha: 1
Background
Tint: Default
Drawing: ☑ Opaque
☐ Hidden
☑ Clears Graphics Context
☑ Clip To Bounds
☑ Autoresize Subviews
Stretching: 0 0

View as: iPhone 7 (wC hR)

75%

CS193p
Fall 2017-18

TVCExample > iPhone 7

TVCExample: **Ready**

TVCExample > TVCExample

Navigation Controller

PROTOTYPE CELLS

Title
Subtitle

View Controller

Navigation Controller

Just ctrl-drag as usual!

Table View
Prototype Content

Table View Cell

Style: Subtitle
Image: Image
Identifier: MyCell
Selection: Default
Accessory: Detail Disclosure
Editing Acc.: None
Focus Style: Default
Indentation: 0 (Level), 10 (Width)
- [x] Indent While Editing
- [ ] Shows Re-order Controls

Separator: Default Insets

View

Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction:
- [x] User Interaction Enabled
- [ ] Multiple Touch

Alpha: 1
Background
Tint: Default
Drawing:
- [x] Opaque
- [ ] Hidden
- [x] Clears Graphics Context
- [x] Clip To Bounds
- [x] Autoresize Subviews

Stretching: 0 0

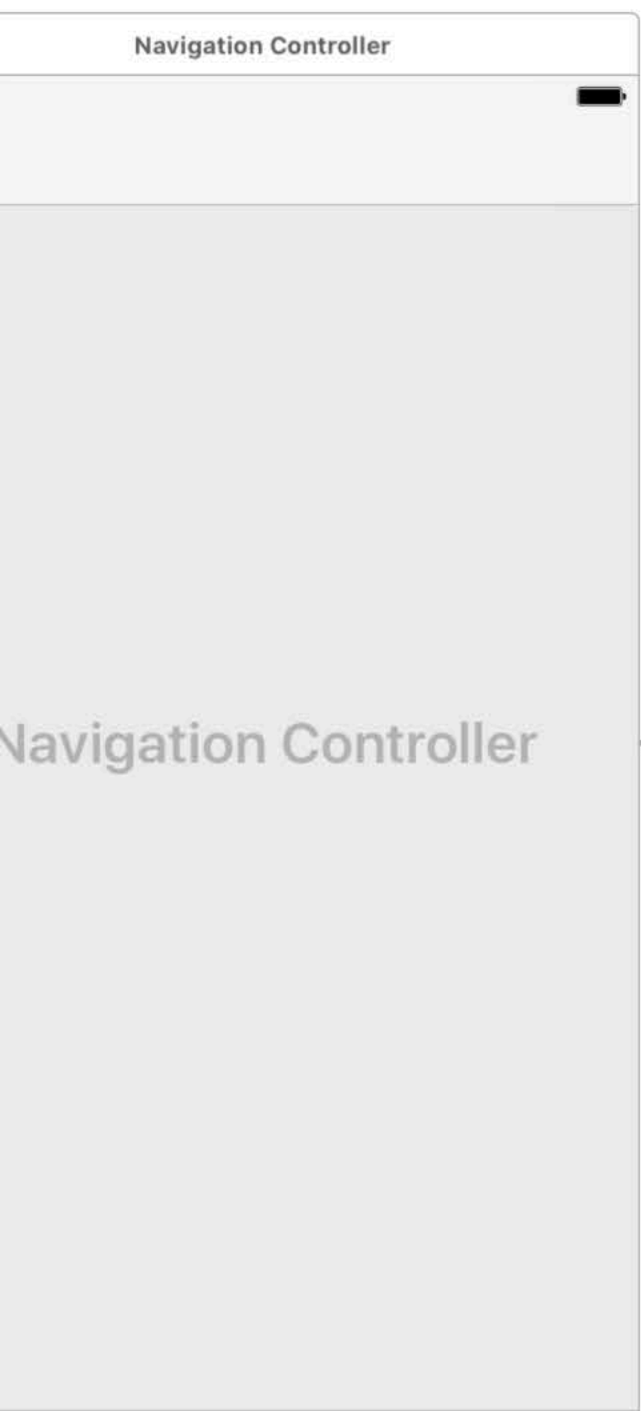
View as: iPhone 7 (wC hR)

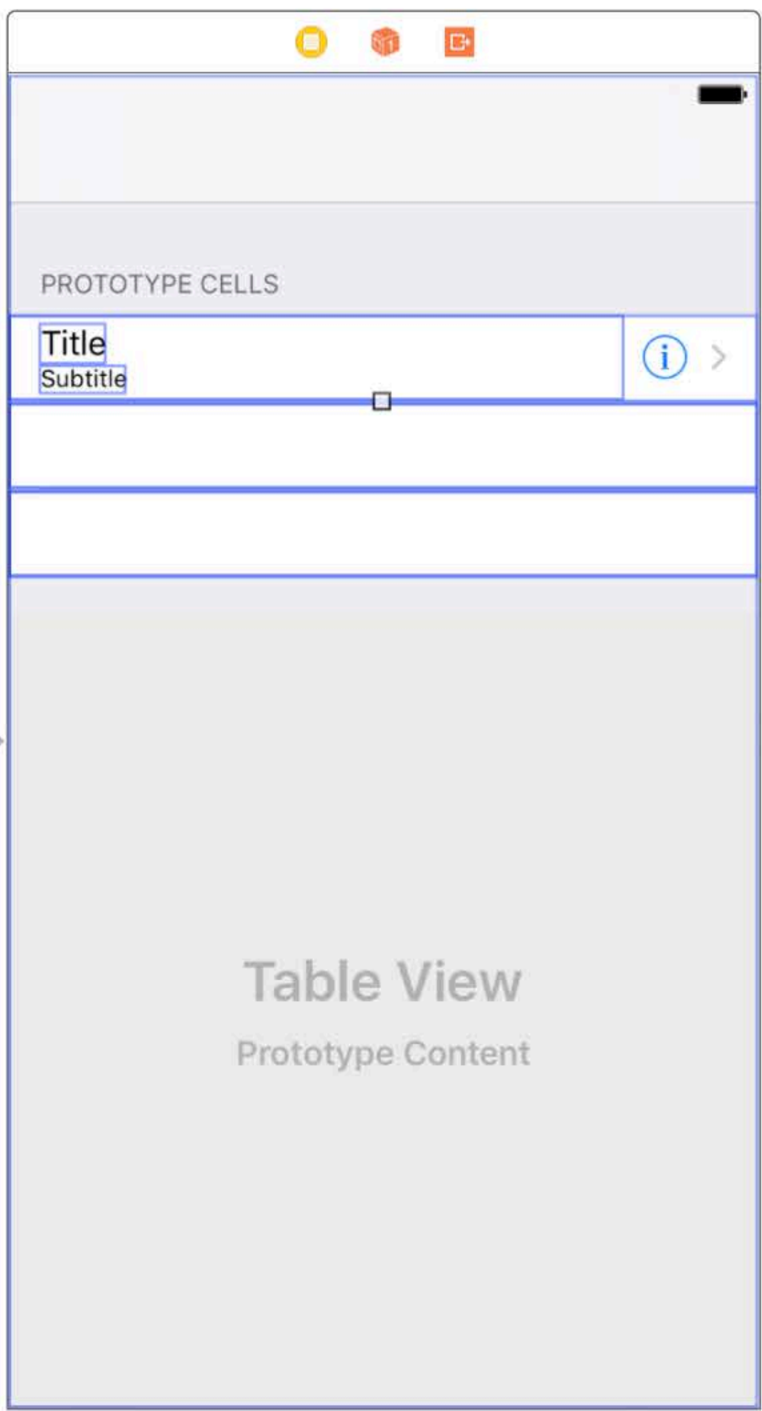
75%

TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample





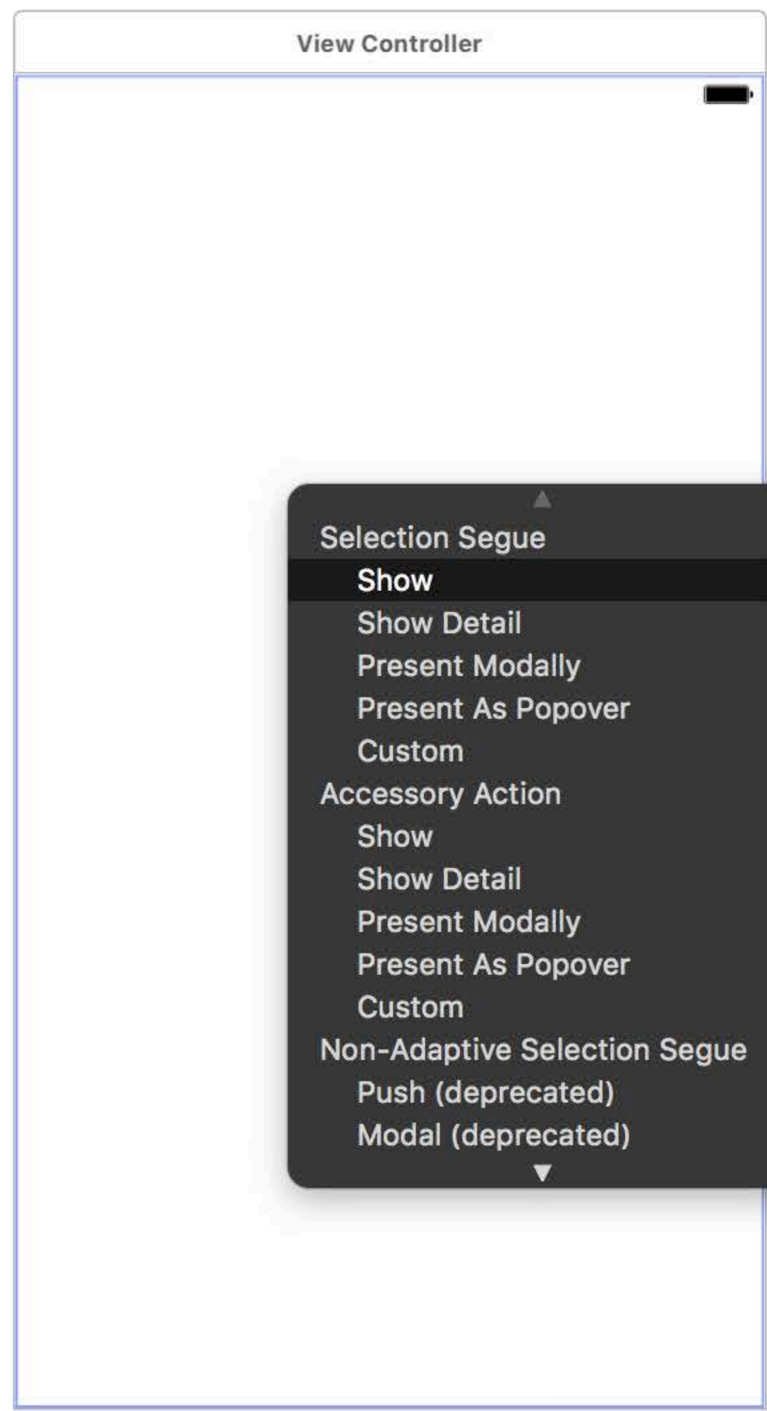


Table View Cell

Style: Subtitle
Image: Image
Identifier: MyCell
Selection: Default
Accessory: Detail Disclosure
Editing Acc.: None
Focus Style: Default
Indentation: 0 (Level) 10 (Width)
☑ Indent While Editing
☐ Shows Re-order Controls
Separator: Default Insets

View

Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: ☑ User Interaction Enabled
☐ Multiple Touch
Alpha: 1
Background
Tint: Default
Drawing: ☑ Opaque
☐ Hidden
☑ Clears Graphics Context
☑ Clip To Bounds
☑ Autoresize Subviews
Stretching: 0 0

View as: iPhone 7 (wC hR)

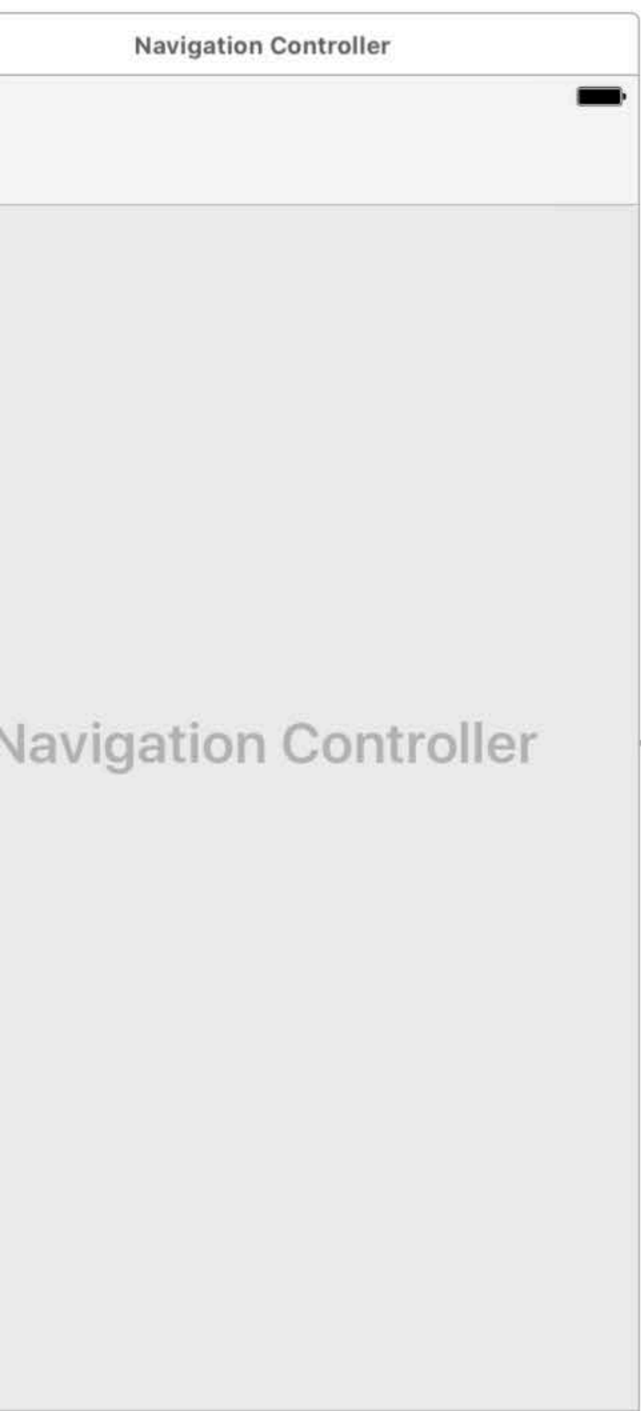
75%

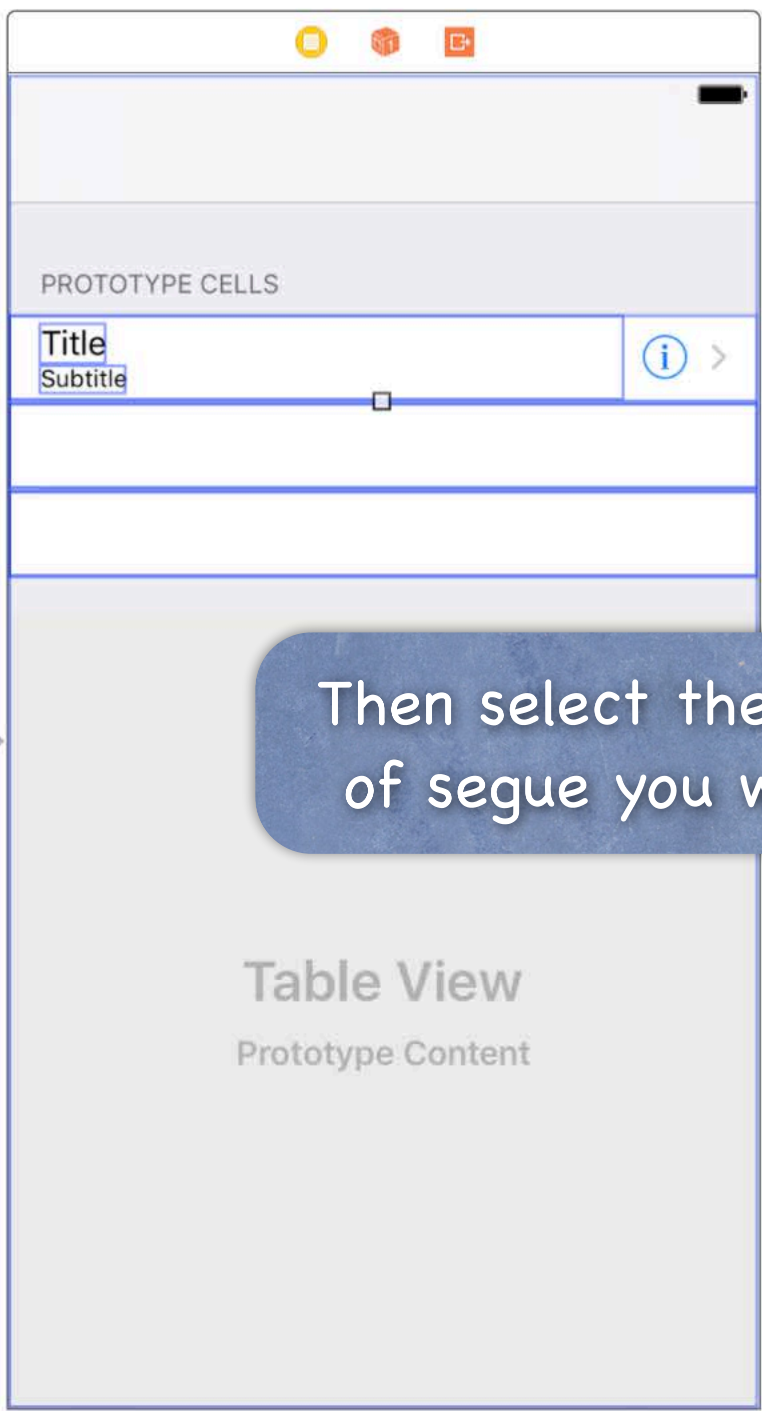
CS193p
Fall 2017-18

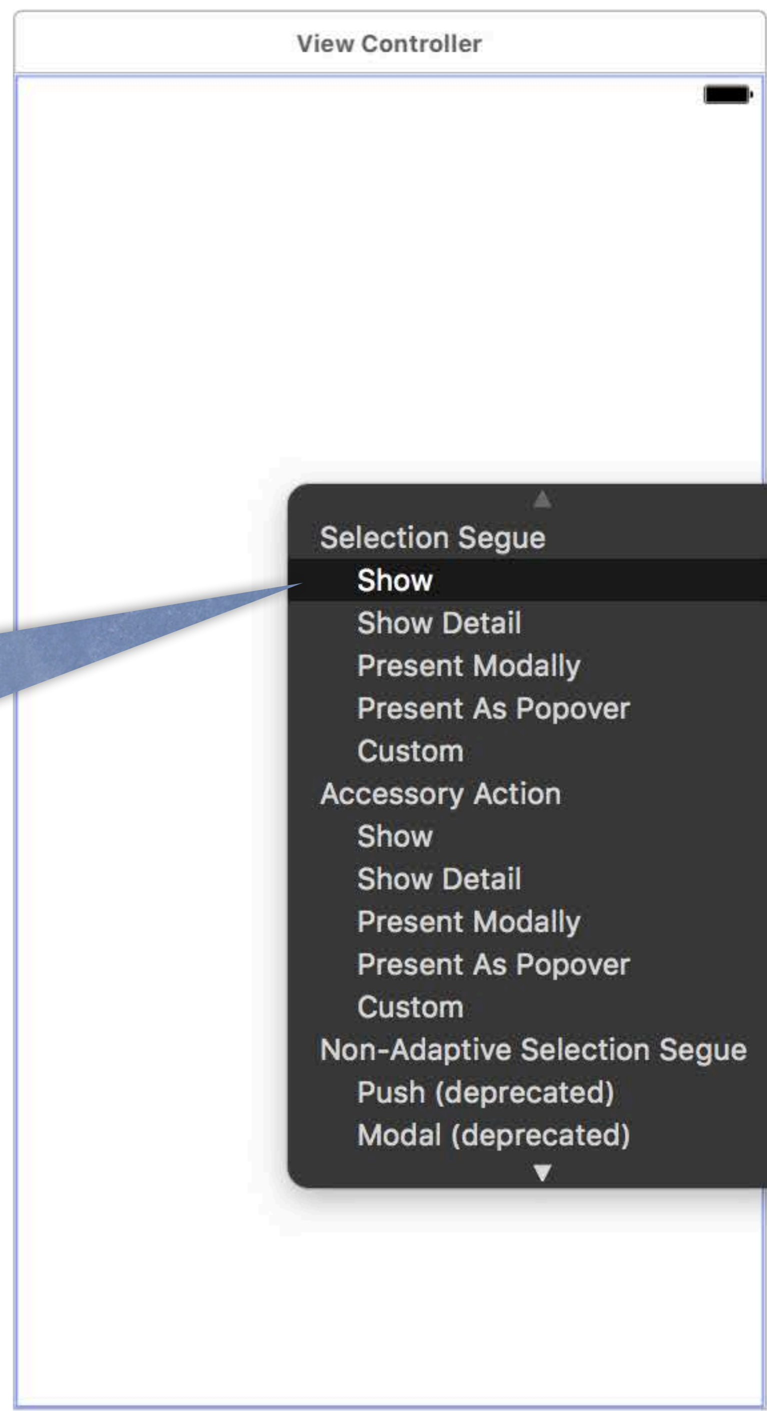
TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample







Then select the kind of segue you want.

Selection Segue
- Show
- Show Detail
- Present Modally
- Present As Popover
- Custom

Accessory Action
- Show
- Show Detail
- Present Modally
- Present As Popover
- Custom

Non-Adaptive Selection Segue
- Push (deprecated)
- Modal (deprecated)

Table View Cell

Style	Subtitle
Image	Image
Identifier	MyCell
Selection	Default
Accessory	Detail Disclosure
Editing Acc.	None
Focus Style	Default
Indentation	0 Level, 10 Width
	☑ Indent While Editing
	☐ Shows Re-order Controls
Separator	Default Insets

View

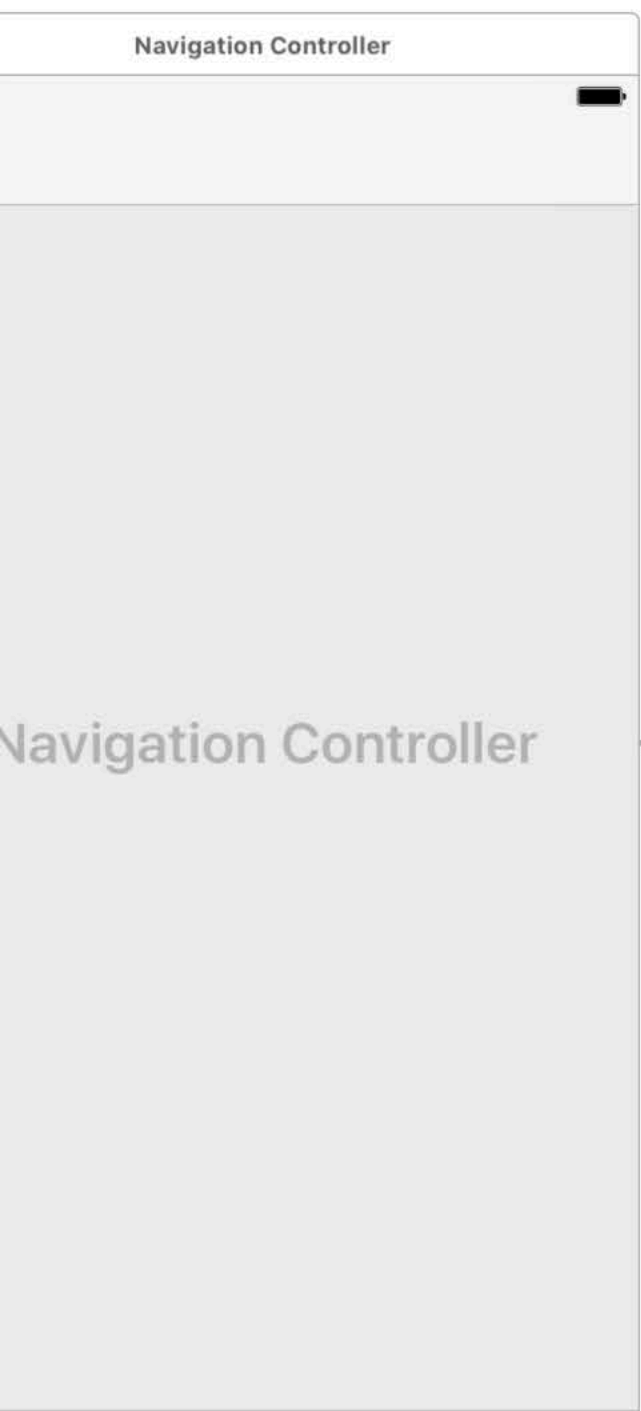
Content Mode	Scale To Fill
Semantic	Unspecified
Tag	0
Interaction	☑ User Interaction Enabled
	☐ Multiple Touch
Alpha	1
Background	
Tint	Default
Drawing	☑ Opaque
	☐ Hidden
	☑ Clears Graphics Context
	☑ Clip To Bounds
	☑ Autoresize Subviews
Stretching	0 0

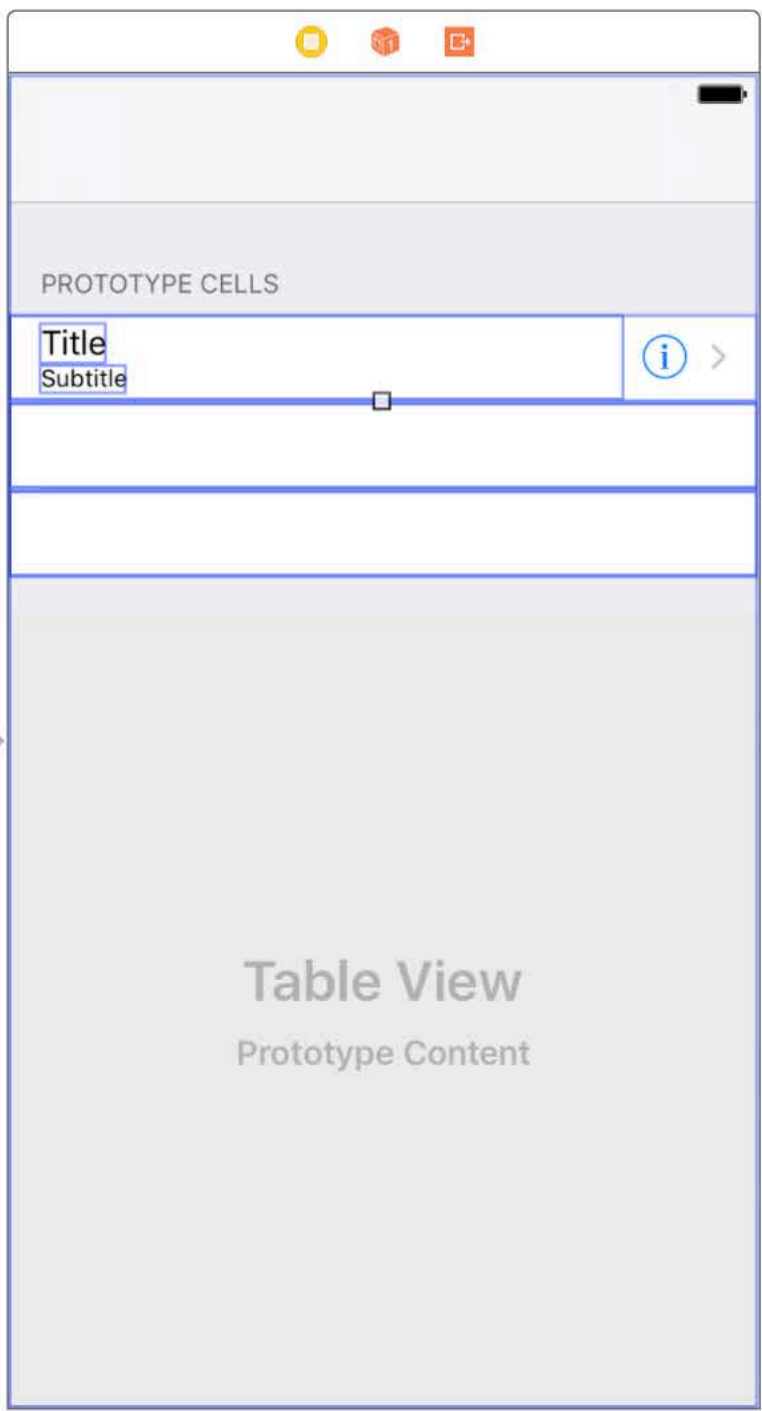
View as: iPhone 7 (wC hR)   75%

TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample





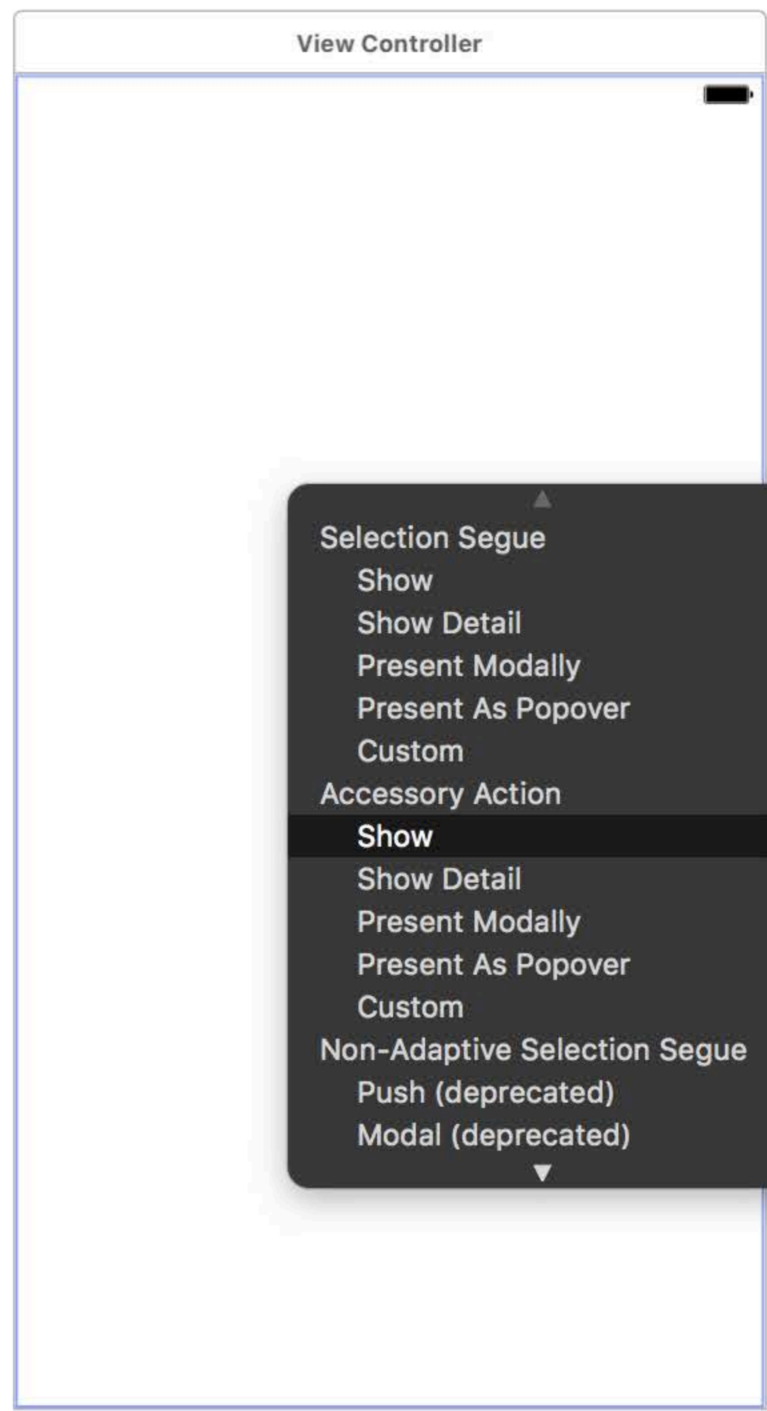


Table View Cell

Style: Subtitle
Image: Image
Identifier: MyCell
Selection: Default
Accessory: Detail Disclosure
Editing Acc.: None
Focus Style: Default
Indentation: 0 (Level) 10 (Width)
☑ Indent While Editing
☐ Shows Re-order Controls
Separator: Default Insets

View

Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: ☑ User Interaction Enabled
☐ Multiple Touch
Alpha: 1
Background
Tint: Default
Drawing: ☑ Opaque
☐ Hidden
☑ Clears Graphics Context
☑ Clip To Bounds
☑ Autoresize Subviews
Stretching: 0 0

View as: iPhone 7 (wC hR)

75%

CS193p
Fall 2017-18

TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample

Navigation Controller

Navigation Controller

PROTOTYPE CELLS

Title
Subtitle

Table View
Prototype Content

View Controller

Selection Segue
Show
Show Detail
Present Modally
Present As Popover
Custom
Accessory Action
Show
Show Detail
Present Modally
Present As Popover
Custom
Non-Adaptive Selection Segue
Push (deprecated)
Modal (deprecated)

You can select the segue for the Detail Disclosure Accessory too.

Table View Cell

Style: Subtitle
Image: Image
Identifier: MyCell
Selection: Default
Accessory: Detail Disclosure
Editing Acc.: None
Focus Style: Default
Indentation: 0 (Level) 10 (Width)
Indent While Editing
Shows Re-order Controls
Separator: Default Insets

View

Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: User Interaction Enabled
Multiple Touch
Alpha: 1
Background
Tint: Default
Drawing: Opaque
Hidden
Clears Graphics Context
Clip To Bounds
Autoresize Subviews
Stretching: 0 0

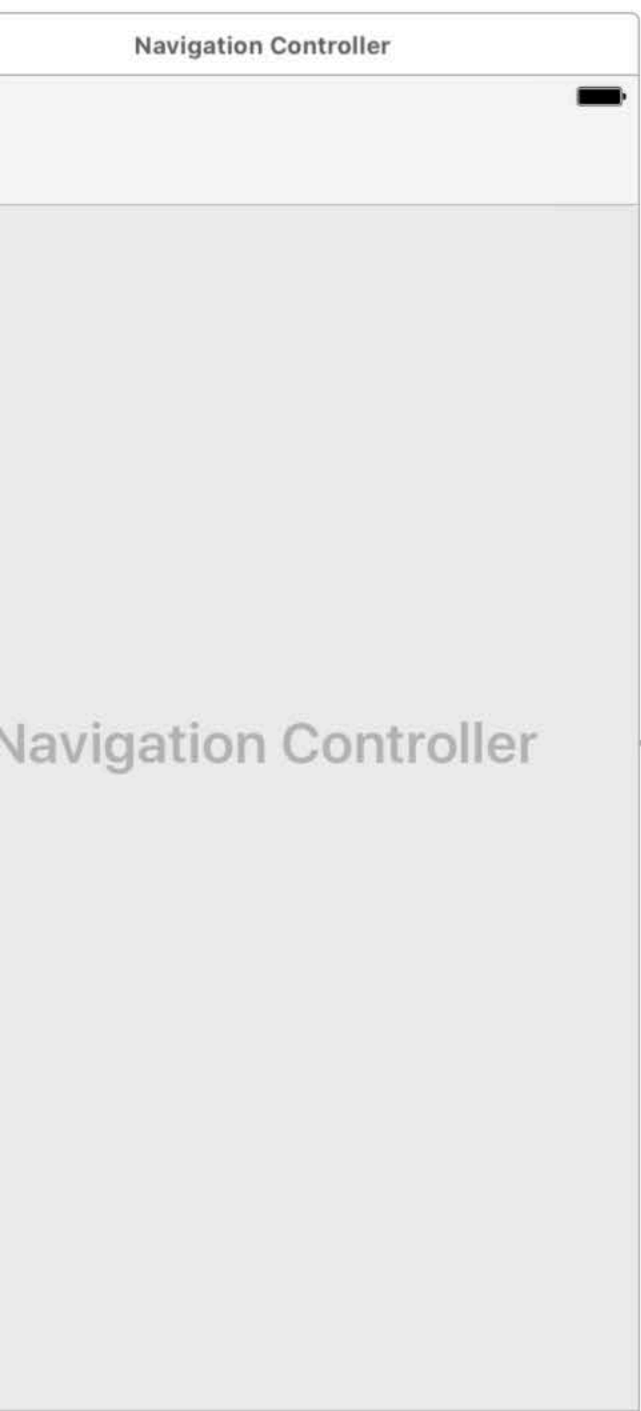
View as: iPhone 7 (wC hR)

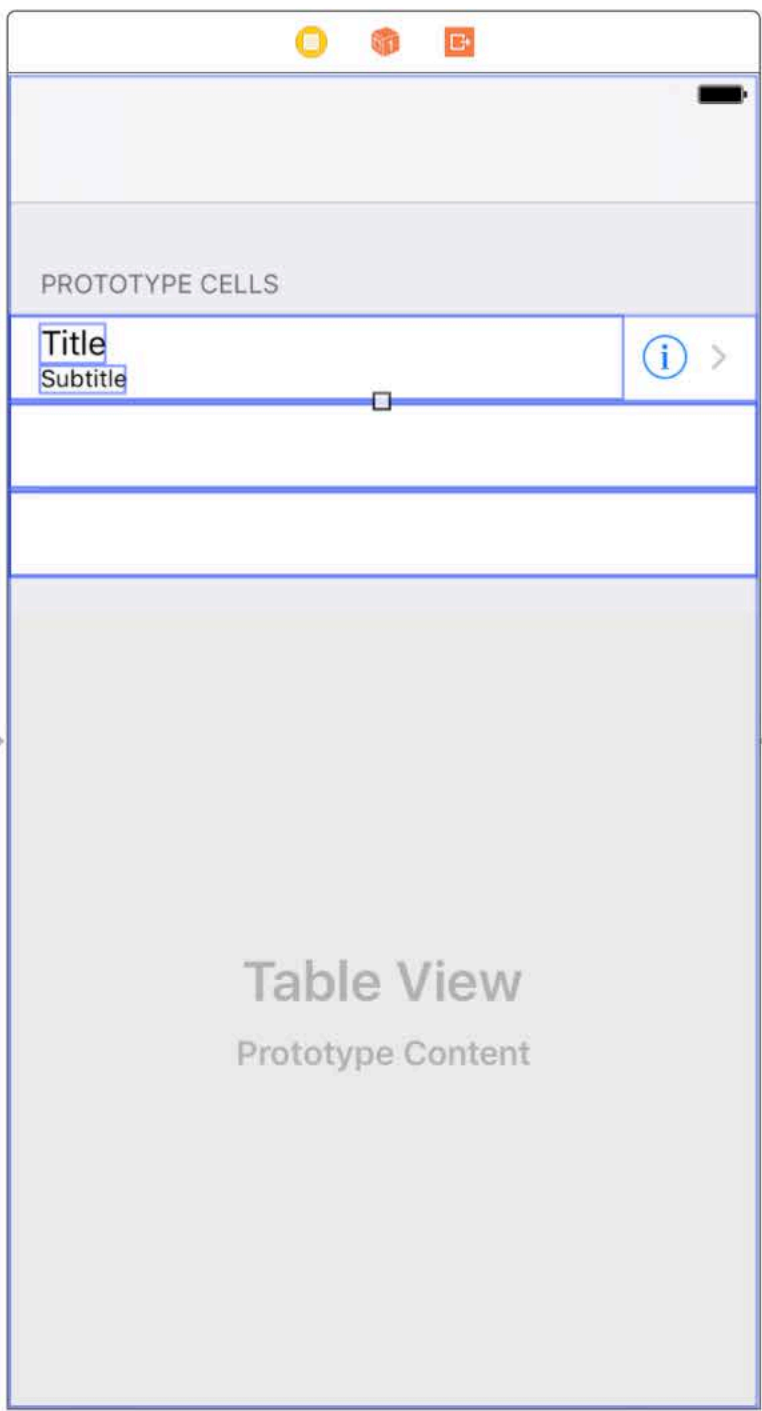
75%

TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample





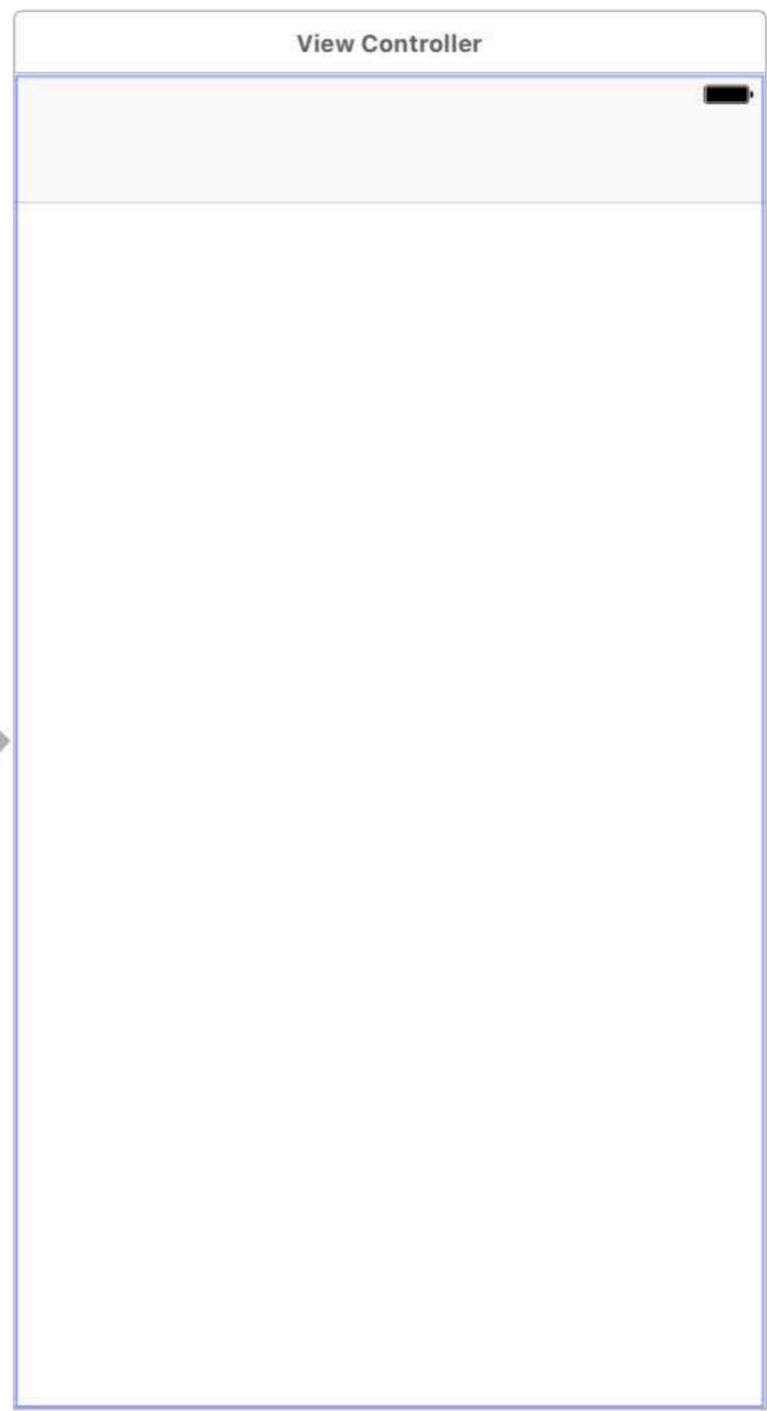


Table View Cell

Style: Subtitle
Image: Image
Identifier: MyCell
Selection: Default
Accessory: Detail Disclosure
Editing Acc.: None
Focus Style: Default
Indentation: 0 (Level) 10 (Width)
☑ Indent While Editing
☐ Shows Re-order Controls
Separator: Default Insets

View

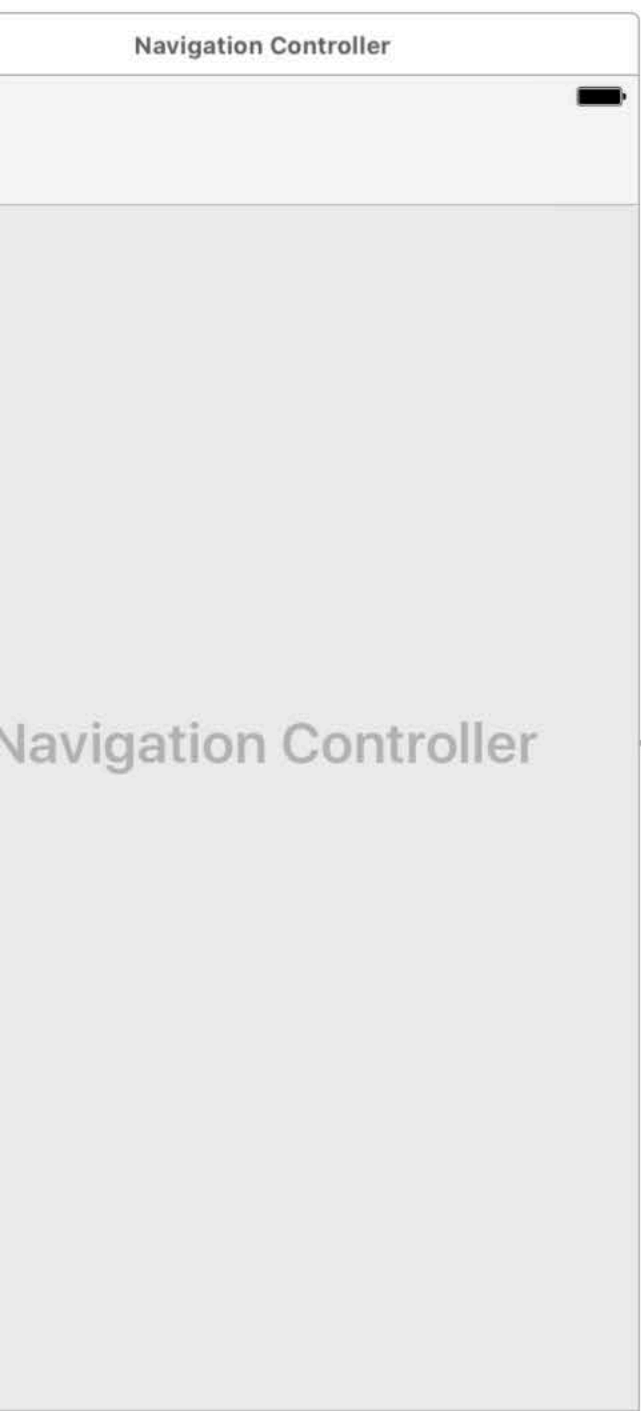
Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: ☑ User Interaction Enabled
☐ Multiple Touch
Alpha: 1
Background
Tint: Default
Drawing: ☑ Opaque
☐ Hidden
☑ Clears Graphics Context
☑ Clip To Bounds
☑ Autoresize Subviews
Stretching: 0 0

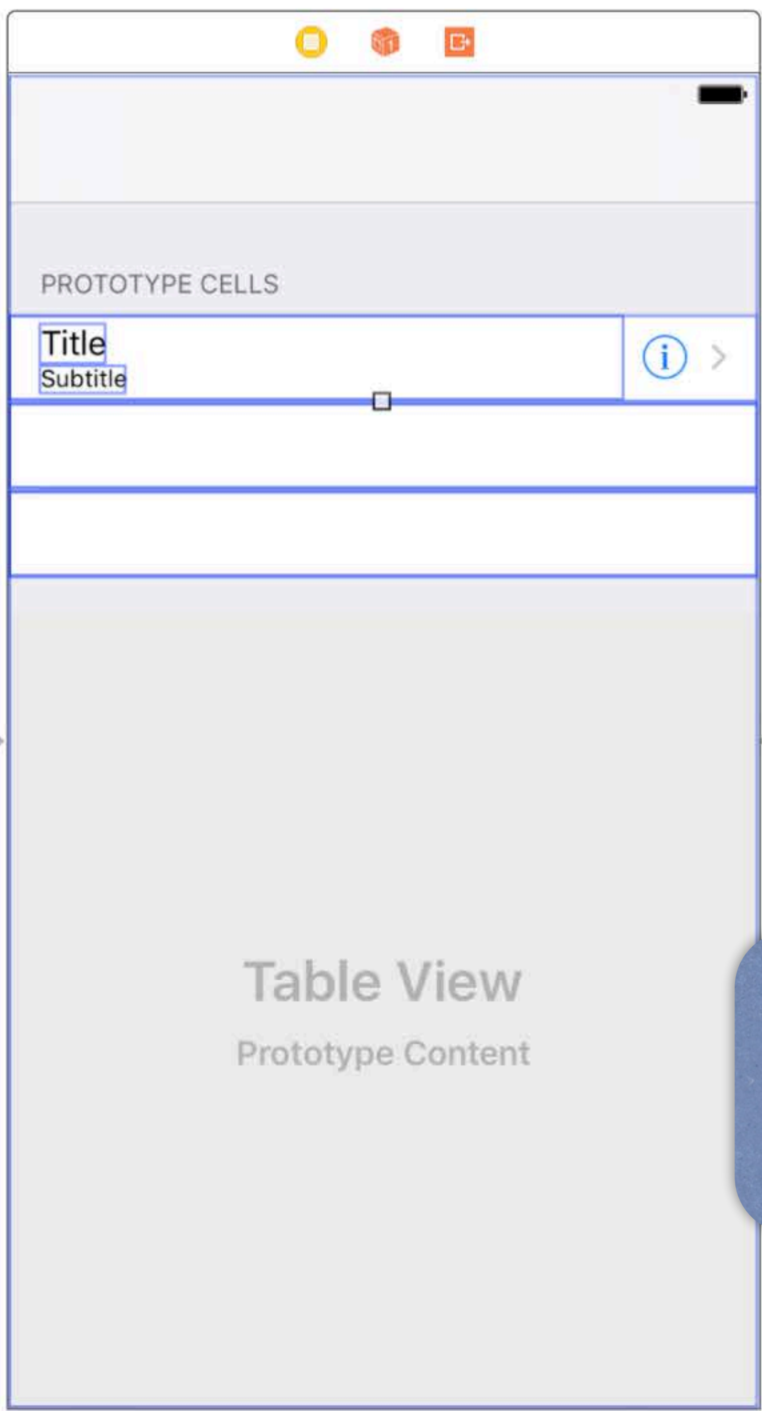
View as: iPhone 7 (wC hR)

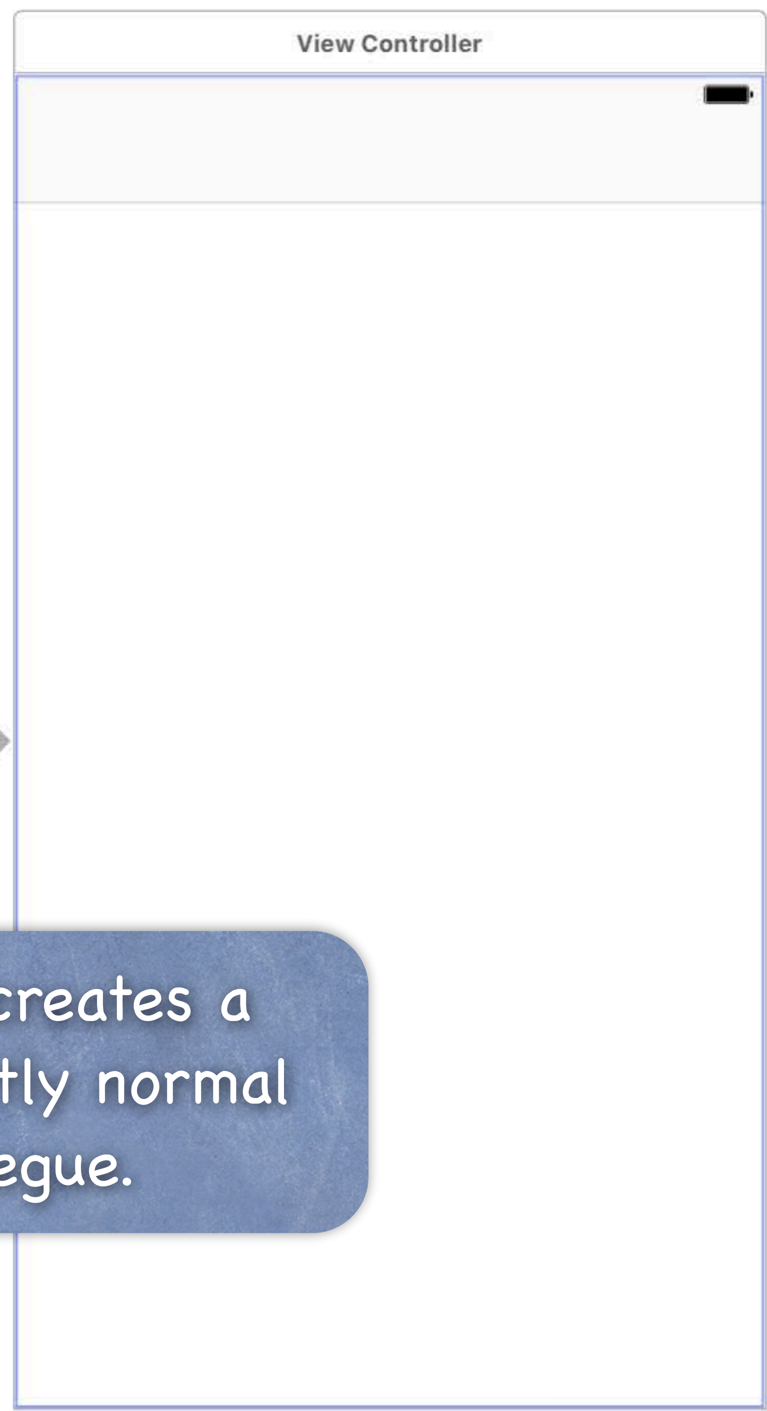
75%

TVCExample 〉 iPhone 7 TVCExample: **Ready**

TVCExample 〉 TVCExample







This creates a perfectly normal segue.

Table View Cell

Style	Subtitle
Image	Image
Identifier	MyCell
Selection	Default
Accessory	Detail Disclosure
Editing Acc.	None
Focus Style	Default
Indentation	0 (Level) 10 (Width)
	☑ Indent While Editing
	☐ Shows Re-order Controls
Separator	Default Insets

View

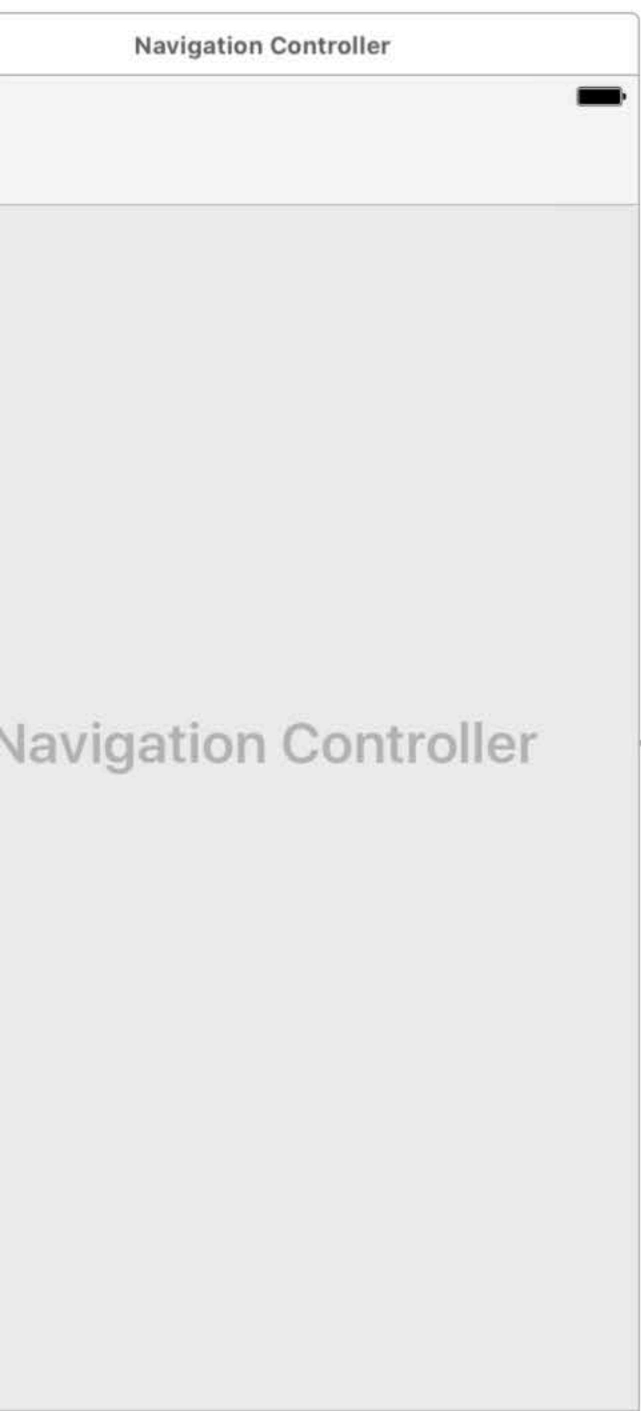
Content Mode	Scale To Fill
Semantic	Unspecified
Tag	0
Interaction	☑ User Interaction Enabled
	☐ Multiple Touch
Alpha	1
Background	
Tint	Default
Drawing	☑ Opaque
	☐ Hidden
	☑ Clears Graphics Context
	☑ Clip To Bounds
	☑ Autoresize Subviews
Stretching	0 0

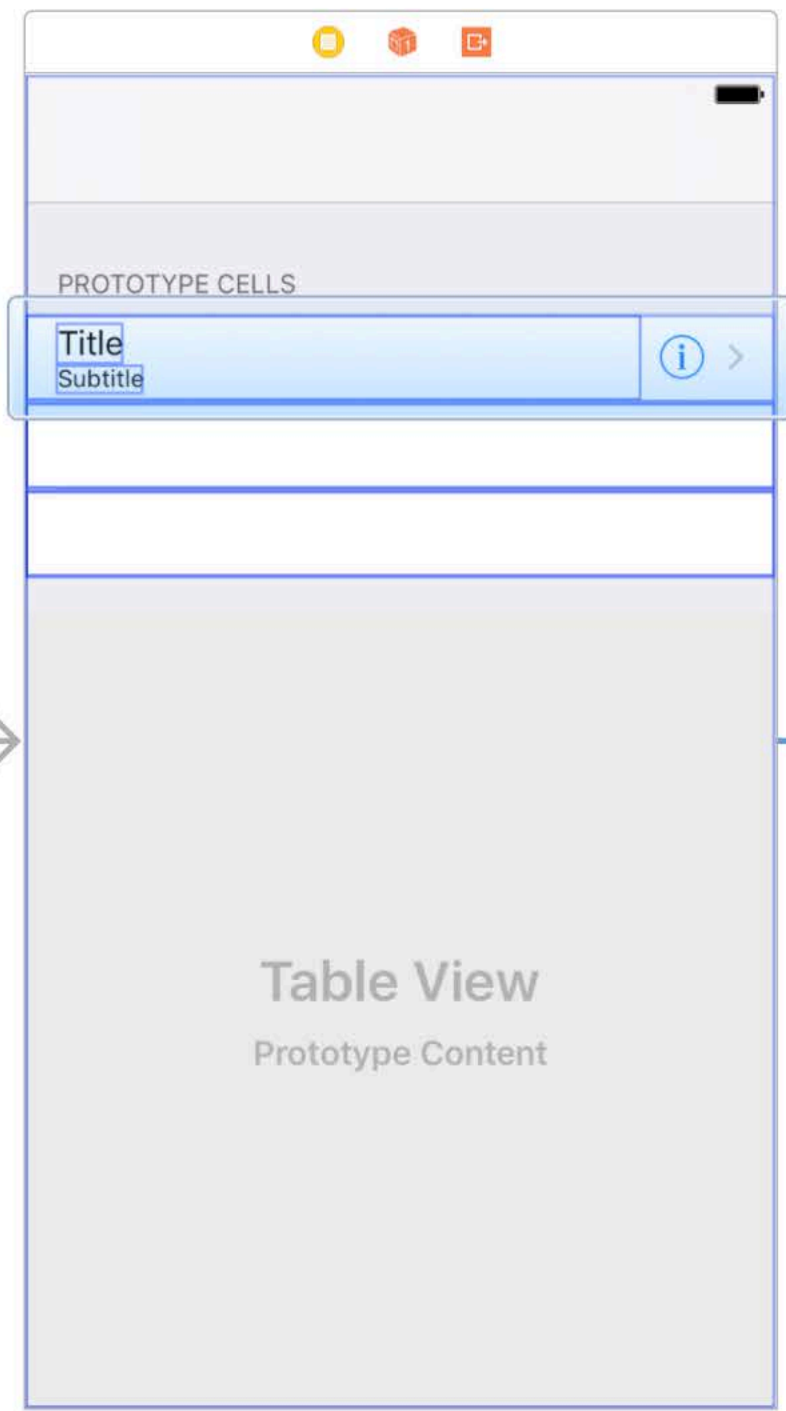
View as: iPhone 7 (wC hR) — 75% +

TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample





Storyboard Segue

Identifier: Identifier

Class: UIStoryboardSegue

Module: None

Kind: Show (e.g. Push)

☑ Animates

Peek & Pop ☐ Preview & Commit Segues

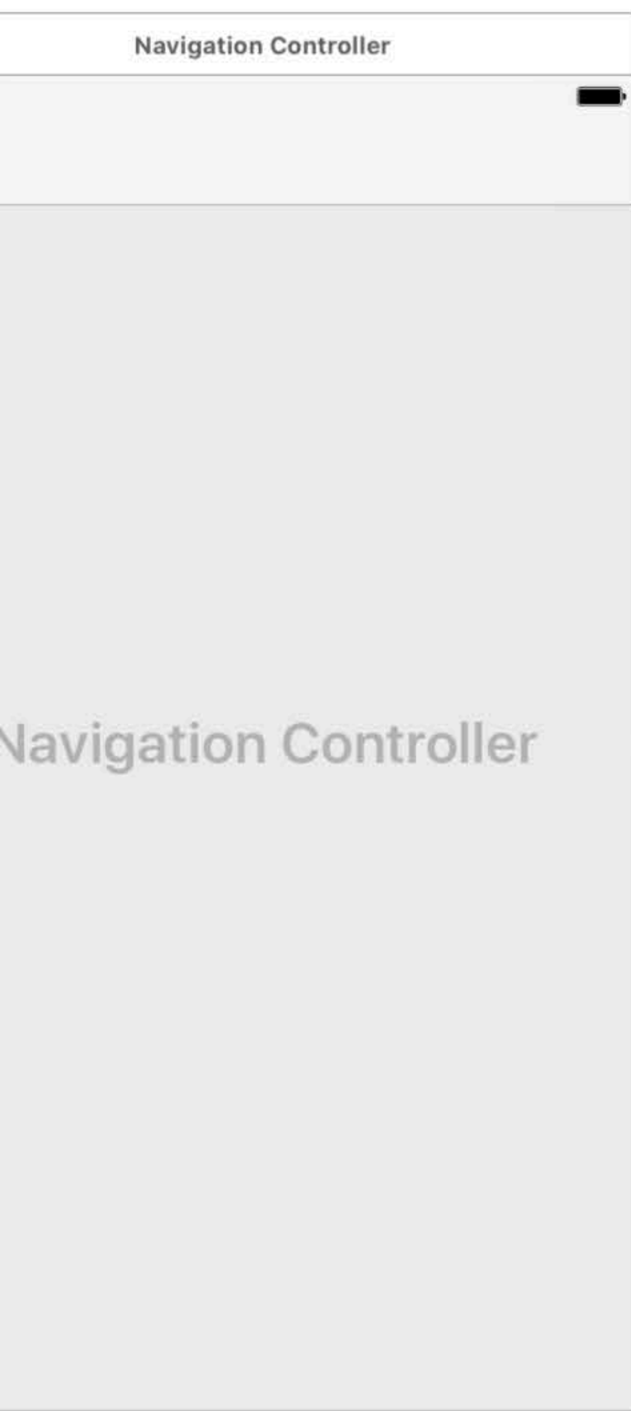
View as: iPhone 7 (wC hR)

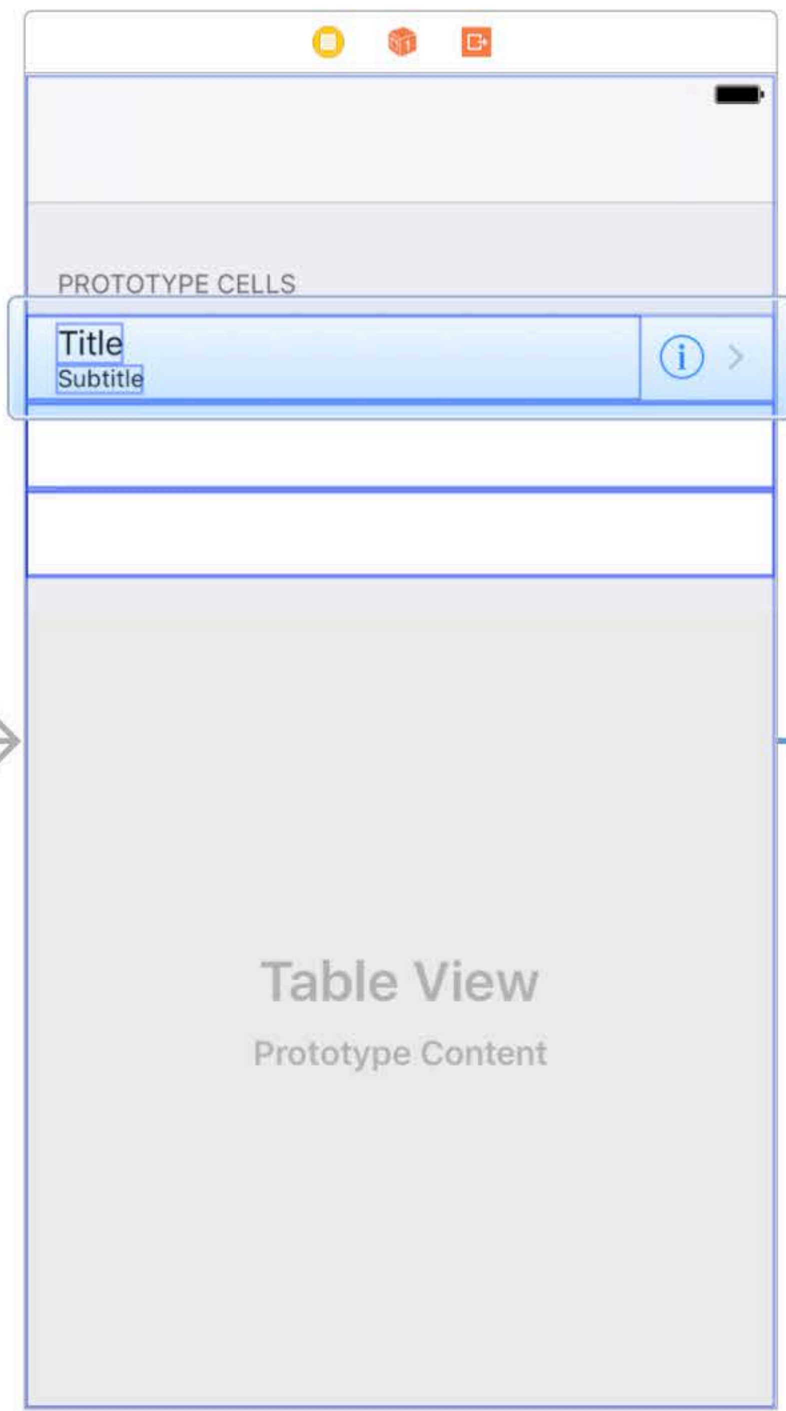
75%

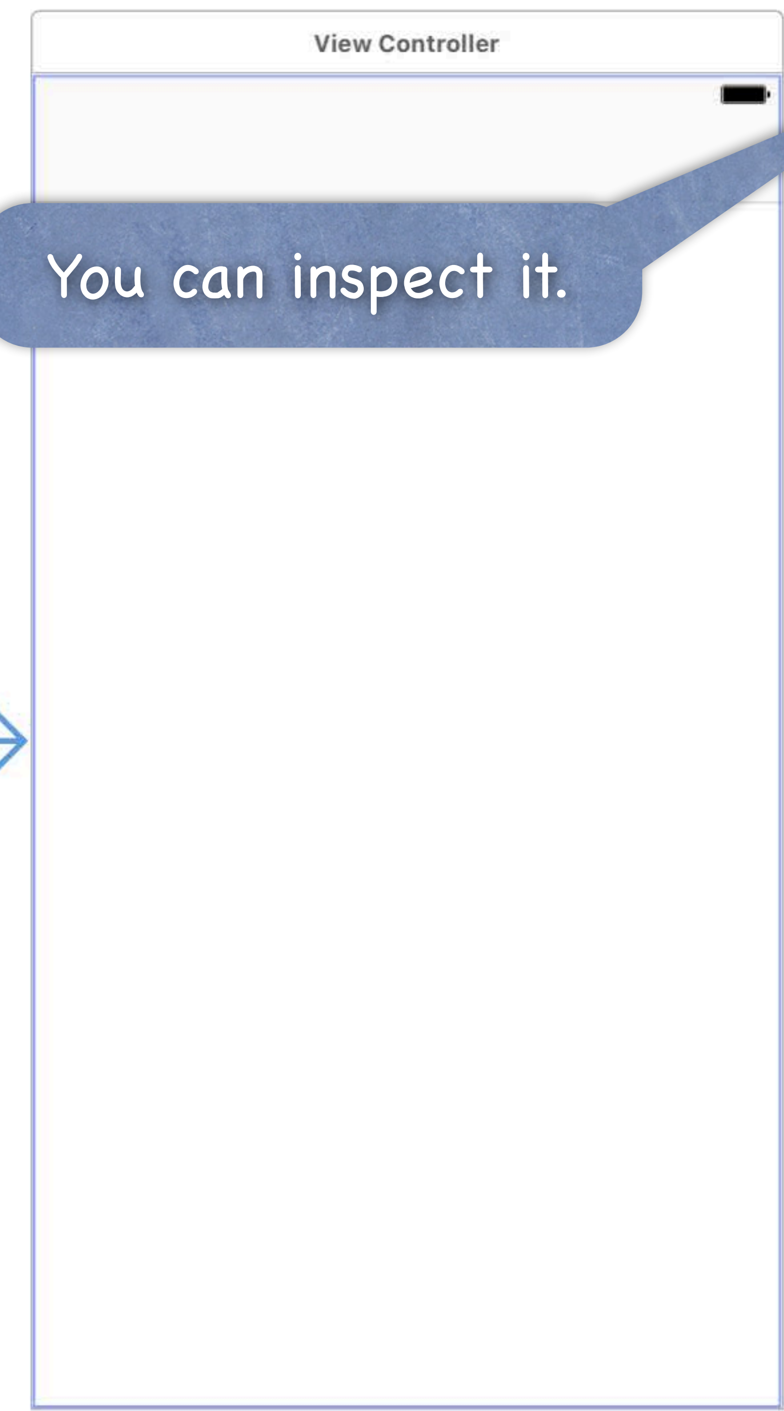
TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample







You can inspect it.

Storyboard Segue

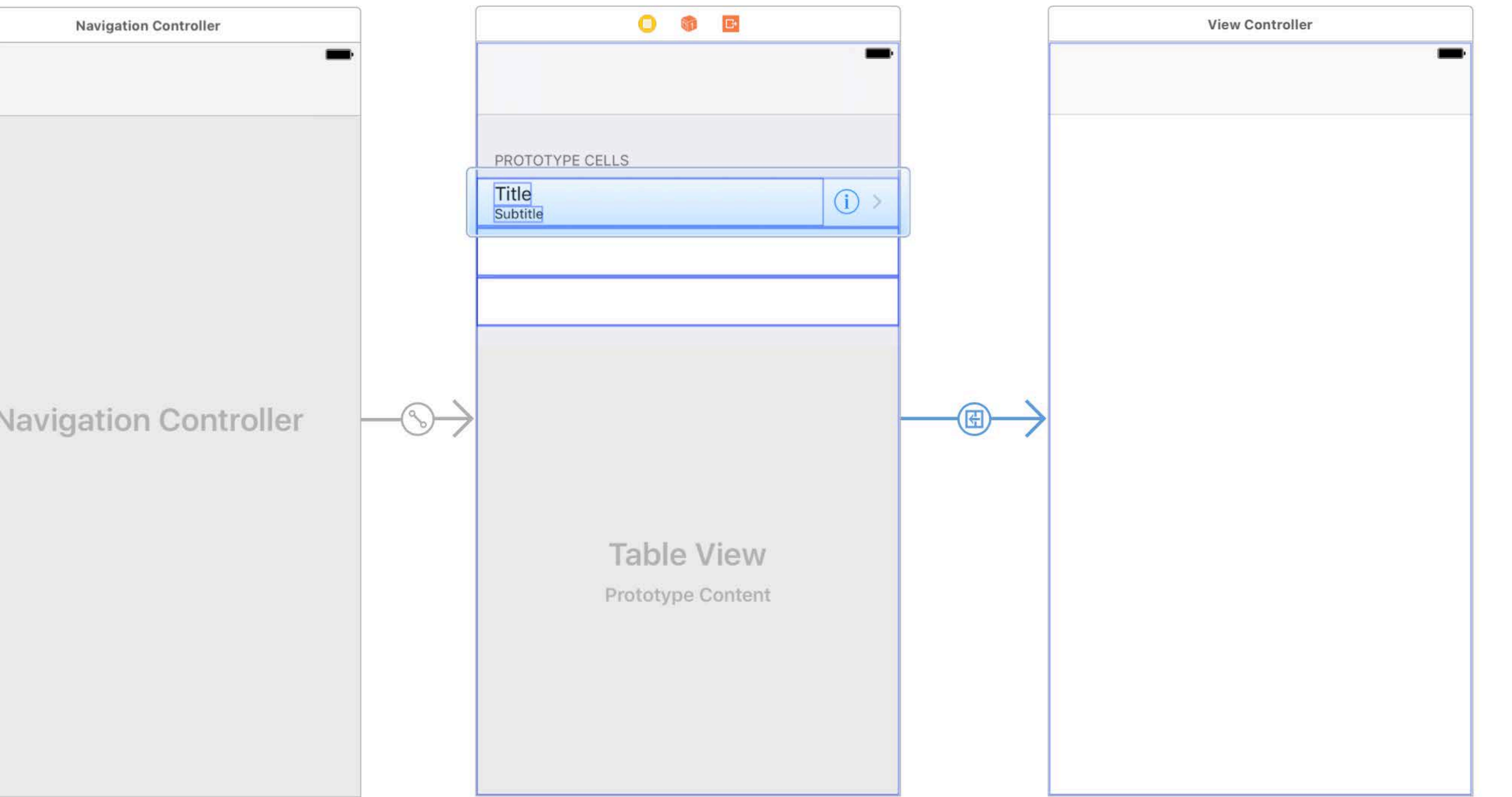
Identifier: Identifier
Class: UIStoryboardSegue
Module: None
Kind: Show (e.g. Push)
☑ Animates
Peek & Pop ☐ Preview & Commit Segues

View as: iPhone 7 (wC hR)   75%

TVCExample 〉 iPhone 7

TVCExample: **Ready**

TVCExample 〉 TVCExample



Storyboard Segue

Identifier: AbcSegue

Class: UIStoryboardSegue

Module: None

Kind: Show (e.g. Push)

☑ Animates

Peek & Pop ☐ Preview & Commit Segues

View as: iPhone 7 (wC hR)

75%

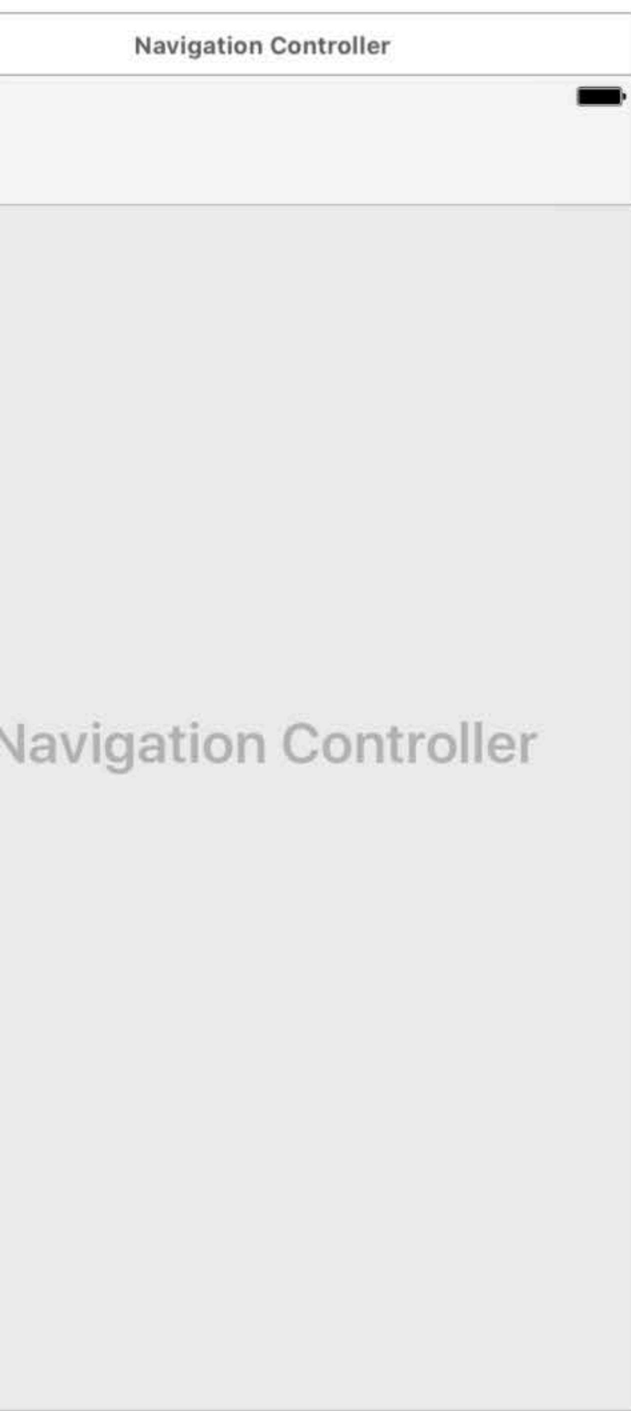
CS193p
Fall 2017-18

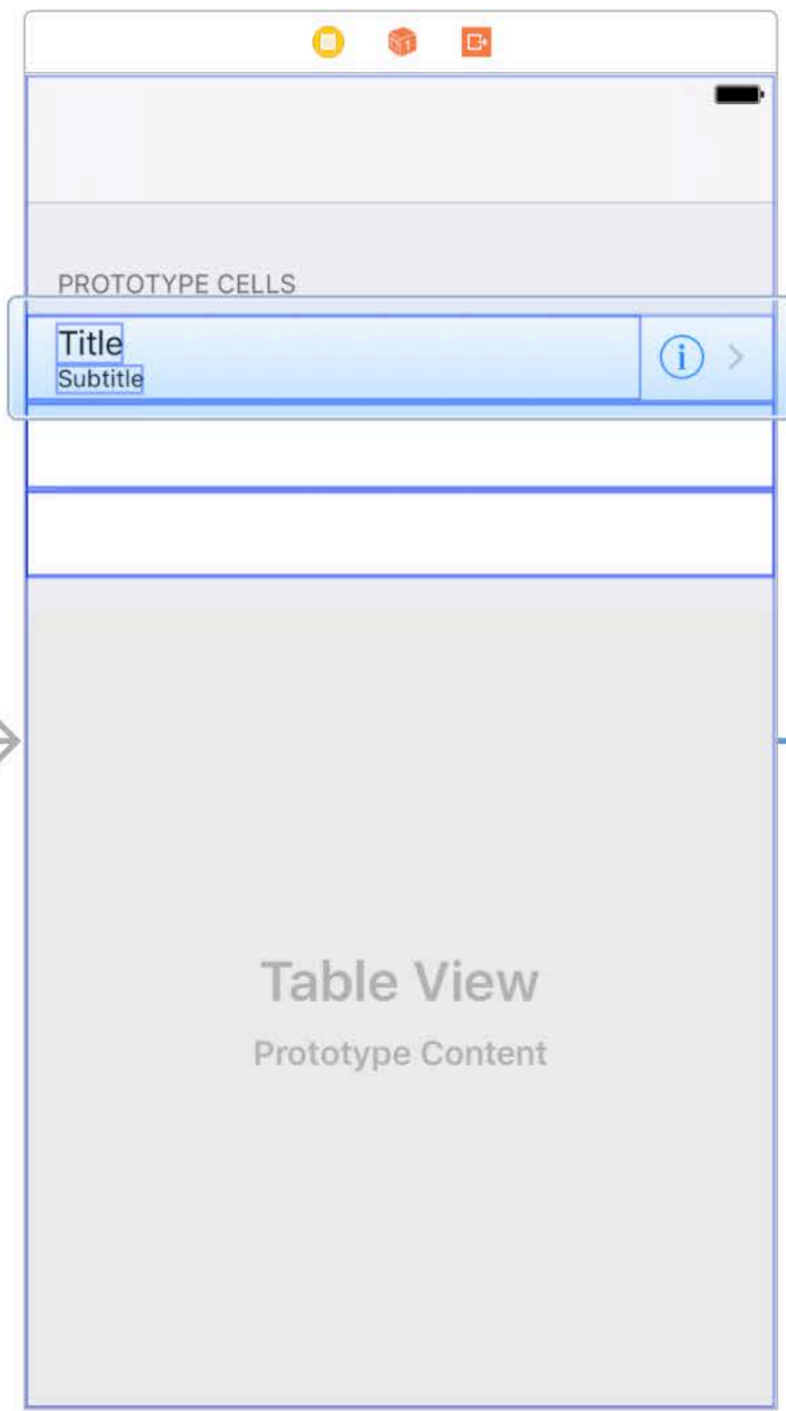
TVCExample: Ready

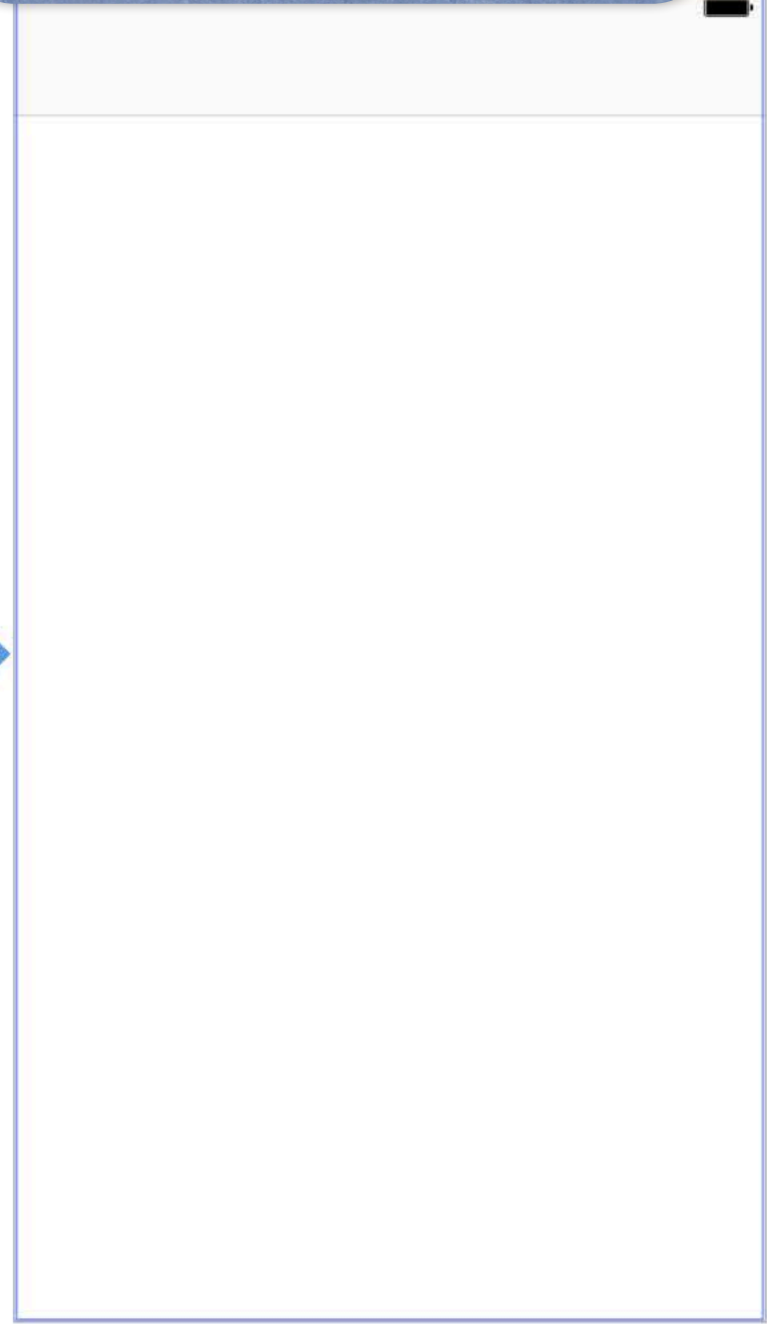
And set its identifier.

Storyboard Segue

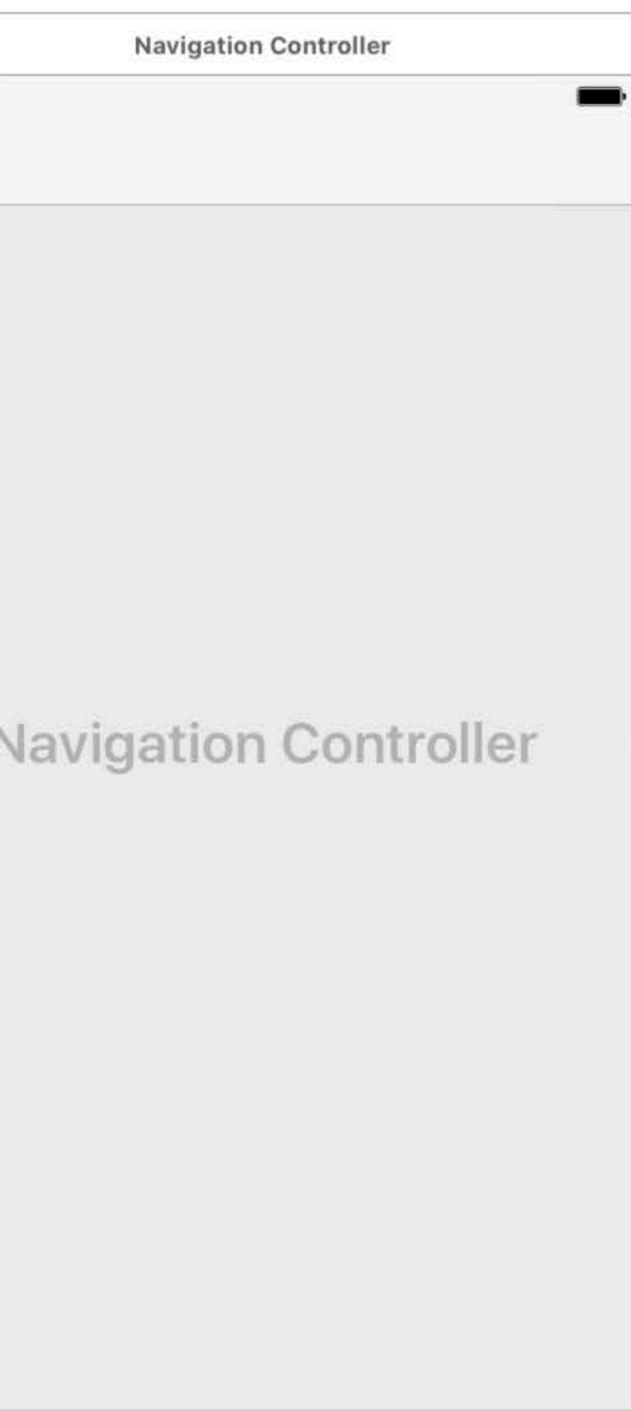
Identifier: AbcSegue
Class: UIStoryboardSegue
Module: None
Kind: Show (e.g. Push)
Animates
Peek & Pop: Preview & Commit Segues

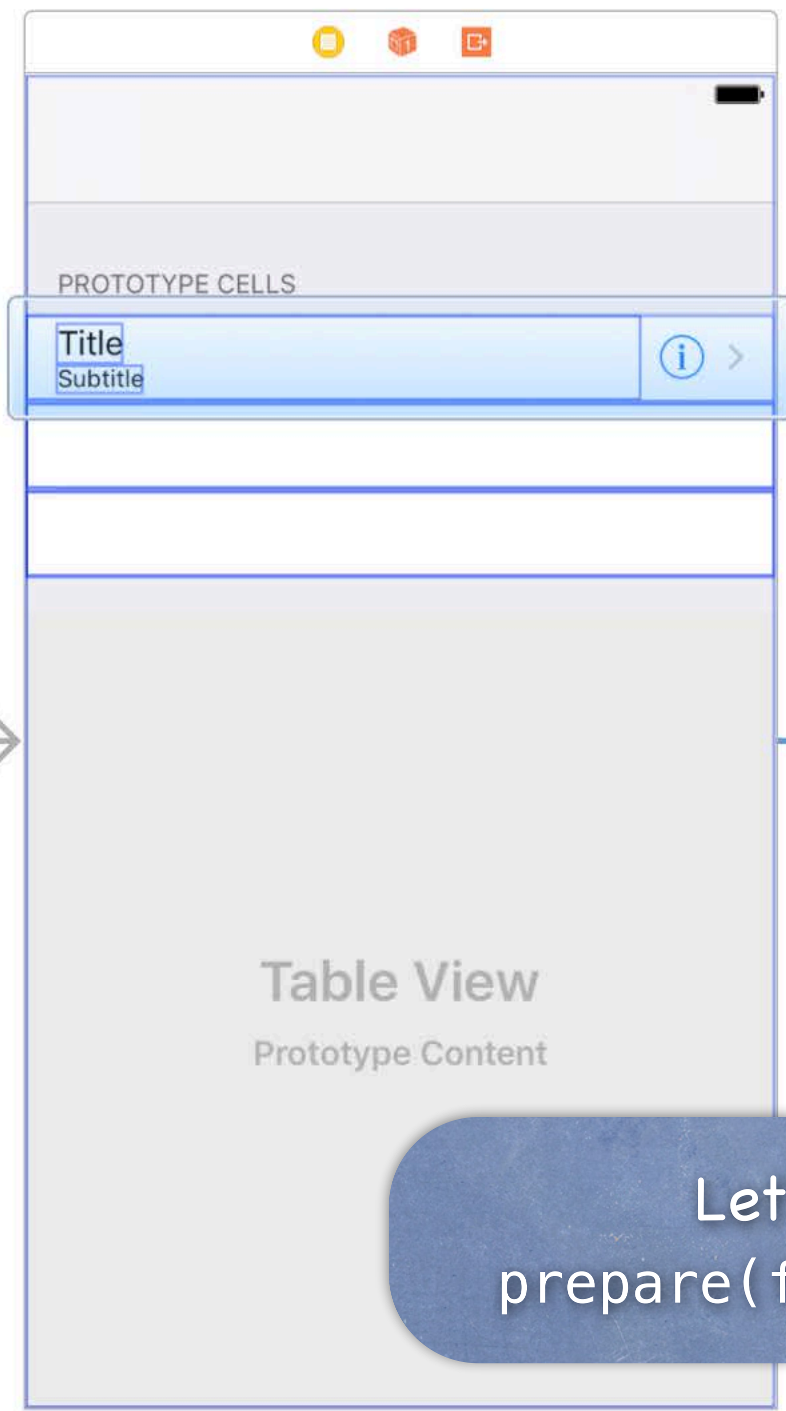






View as: iPhone 7 (wC hR) 75%





TVCExample: Ready

Storyboard Segue

Identifier AbcSegue

Class UIStoryboardSegue

Module None

Kind Show (e.g. Push)

Animates

Peek & Pop Preview & Commit Segues

And set its identifier.

Let's take a look at
prepare(for segue:sender:) …

View as: iPhone 7 (wC hR) 75%

Table View Segues

- Preparing to segue from a row in a table view

  The sender argument to prepareForSegue is the UITableViewCell of that row …

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case “XyzSegue”: // handle XyzSegue here
        case “AbcSegue”:




        default: break
        }
    }
}
```

  You can see now why sender is Any

  Sometimes it’s a UIButton, sometimes it’s a UITableViewCell

Table View Segues

- Preparing to segue from a row in a table view

  The sender argument to prepareForSegue is the UITableViewCell of that row …

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case “XyzSegue”: // handle XyzSegue here
        case “AbcSegue”:
            if let cell = sender as? MyTableViewCell {


            }
        default: break
        }
    }
}
```

  So you will need to cast sender with as? to turn it into a UITableViewCell

  If you have a custom UITableViewCell subclass, you can cast it to that if it matters

Table View Segues

- Preparing to segue from a row in a table view

The `sender` argument to `prepareForSegue` is the `UITableViewCell` of that row …

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case "XyzSegue": // handle XyzSegue here
        case "AbcSegue":
            if let cell = sender as? MyTableViewCell,
               let indexPath = tableView.indexPath(for: cell) {


            }
        default: break
        }
    }
}
```

> `indexPath(for cell:)` does not accept `Any`. It has to be a `UITableViewCell` of some sort.

Usually we will need the `IndexPath` of the `UITableViewCell`
Because we use that to index into our internal data structures

Table View Segues

- Preparing to segue from a row in a table view

The `sender` argument to `prepareForSegue` is the `UITableViewCell` of that row …

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case “XyzSegue”: // handle XyzSegue here
        case “AbcSegue”:
            if let cell = sender as? MyTableViewCell,
               let indexPath = tableView.indexPath(for: cell),
               let seguedToMVC = segue.destination as? MyVC {

            }
        default: break
        }
    }
}
```

Now we just get our destination MVC as the proper class as usual …

Table View Segues

- Preparing to segue from a row in a table view

  The sender argument to prepareForSegue is the UITableViewCell of that row …

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case “XyzSegue”: // handle XyzSegue here
        case “AbcSegue”:
            if let cell = sender as? MyTableViewCell,
               let indexPath = tableView.indexPath(for: cell),
               let seguedToMVC = segue.destination as? MyVC {
                  seguedToMVC.publicAPI = data[indexPath.section][indexPath.row]
            }
        default: break
        }
    }
}
```

  and then get data from our internal data structure using the IndexPath’s section and row

Table View Segues

- Preparing to segue from a row in a table view

The sender argument to prepareForSegue is the UITableViewCell of that row …

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case “XyzSegue”: // handle XyzSegue here
        case “AbcSegue”:
            if let cell = sender as? MyTableViewCell,
               let indexPath = tableView.indexPath(for: cell),
               let seguedToMVC = segue.destination as? MyVC {
                seguedToMVC.publicAPI = data[indexPath.section][indexPath.row]
            }
        default: break
        }
    }
}
```

and then get data from our internal data structure using the IndexPath’s section and row and use that information to prepare the segued-to API using its public API

Collection View Segue

- Seguing from Collection View cells

  Probably best done from this `UICollectionViewDelegate` method …

  ```
  func collectionView(collectionView: UICV, didSelectItemAtIndexPath indexPath: IndexPath)
  ```

  Use `performSegue(withIdentifier:)` from there.

  This strategy could also be used for `UITableView`.

Table and Collection View

- What if your Model changes?

```
func reloadData()
```

Causes it to call `numberOfSectionsInTableView` and `numberOfRows/ItemsInSection` all over again and then `cellForRow/ItemAt` on each visible row or item

Relatively heavyweight, but if your entire data structure changes, that's what you need

If only part of your Model changes, there are lighter-weight reloaders, for example …

```
func reloadRows(at indexPaths: [IndexPath], with animation: UITableViewRowAnimation)
```

… among others and of course similar methods for Collection View.

Table and Collection View

- Controlling the height of rows in a Table View

  Row height can be fixed (UITableView's var rowHeight: CGFloat)

  Or it can be determined using autolayout (rowHeight = UITableViewAutomaticDimension)

  If you do automatic, help the table view out by setting estimatedRowHeight to something

  The UITableView's delegate can also control row heights …

  ```
  func tableView(UITableView, {estimated}heightForRowAt indexPath: IndexPath) -> CGFloat
  ```

  Beware: the non-estimated version of this could get called A LOT if you have a big table

- Controlling the size of cells in a Collection View

  Cell size can be fixed in the storyboard.

  You can also drive it from autolayout similar to table view.

  Or you can return the size from this delegate method …

  ```
  func collectionView(_ collectionView: UICollectionView,
       layout collectionViewLayout: UICollectionViewLayout,
       sizeForItemAt indexPath: IndexPath
  ) -> CGSize
  ```

Table View Headers

- Setting a header for each section

  If you have a multiple-section table view, you can set a header (or footer) for each.

  There are methods to set this to be a custom UIView.

  But usually we just supply a `String` for the header using this method …

```
func tableView(_ tv: UITV, titleForHeaderInSection section: Int) -> String?
```

Collection View Headers

- Headers and footers are a bit more difficult in Collection View

  You can't just specify them as Strings.

  First you have to "turn them on" in the storyboard.

  They are reusable (like cells are), so you have to make a UICollectionReusableView subclass.

  You put your UILabel or whatever for your header, then hook up an outlet.

  Then you implement this dataSource method to dequeue and provide a header.

```
func collectionView(_ collectionView: UICollectionView,
    viewForSupplementaryElementOfKind kind: String,
    at indexPath: IndexPath
) -> UICollectionReusableView
```

  Use dequeueReusableSupplementaryView(ofKind:withReuseIdentifier:for:) in there.

  kind will be UICollectionElementKindSectionHeader or Footer.

Other Methods

- There are dozens of other methods in these classes
  - Controlling the look (separator style and color, default row height, etc.).
  - Getting cell information (cell for index path, index path for cell, visible cells, etc.).
  - Scrolling to a row (`UITableView`/`UICollectionView` are subclasses of `UIScrollView`).
  - Selection management (allows multiple selection, getting the selected row, etc.).
  - Moving, inserting and deleting rows, etc.
  - As always, part of learning the material in this course is studying the documentation

Example Code

- FoodForThought

  Example code doing most of what has been described will be posted to the class website.

  It's in an app called FoodForThought.

  You'll see all these things in action.

  And, of course, we will have an extensive demo of all this …

Demo Code

Download the demo code from today's lecture.