CS275
Fall 2020

iOS: `UITableView` and Controller

BNRG CHAPTER 10

Topics

- `UITableView` for presenting data
  - prototype cells
  - the data source
  - the reuse pool
- AppDelegate
- Discardable results
- Singleton model

General App Description

The purpose of many apps is to let the user see and manipulate data

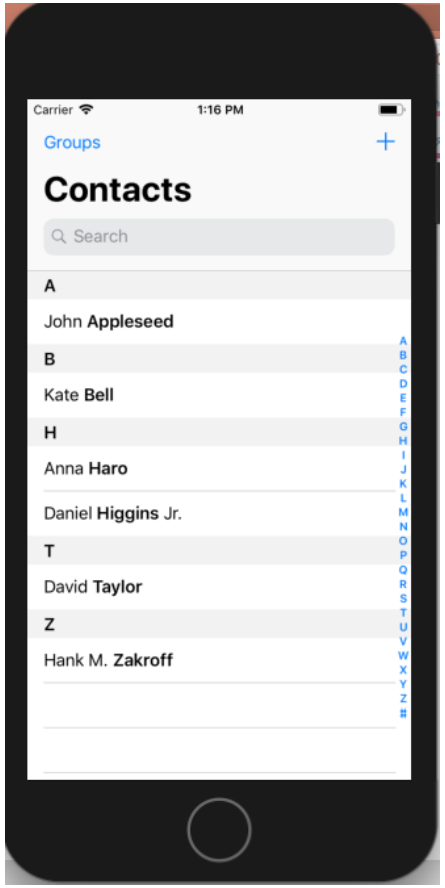
Frequently, the data can be presented in list form

A natural UI interface for this kind of data is a table

UITableView

A `UITableView` displays a single column of data with a variable number of rows

- here's the Contacts app



Design Patterns

A design pattern is a solution to a software-engineering problem

- it could be an abstract structure, such as Model-View-Controller
- or it could be a common architectural approach, such as a RESTful API

Design Patterns: Delegation

Delegation: one object delegates certain responsibilities to another object

For example, Some kinds of UI interactions (such as a change in a text field) are handled using delegation

- the delegate (a method you implement) is notified when some interaction with the UI occurs

Design Patterns: Data Source

Data Source: responsible for providing data to another object when requested

This is how we will provide the data to the `UITableView`

At a minimum, the data source needs to provide two kinds of information:

1. "How many rows should I display?"
2. "What data should I display at each index path (i.e., in each row of the table)?"

New Project: Homepwner

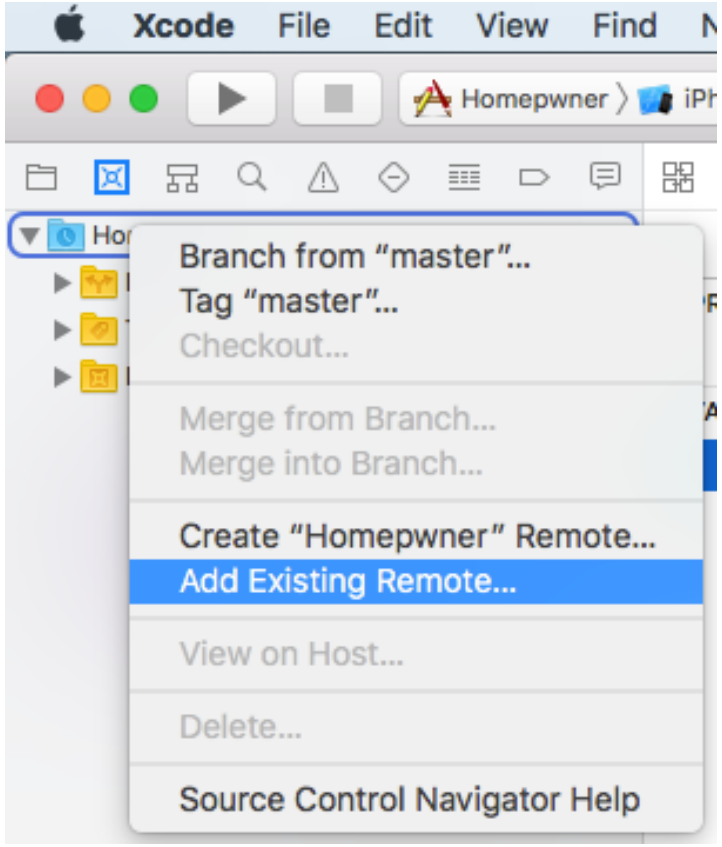
- An app to let us keep track of our possessions
- It will use the camera, master/detail views, more navigation, persistent data
- The project spans eight chapters in the book
- For practice, try creating a Git repository for the project
- Then create a remote on github or gitlab

Remote Repo

Create a repo "Homepwner" on gitlab or github

Then connect your Xcode project to that remote

- right-click on the project in the "Source Control" navigator
- paste in the address of your remote repo
- and if you haven't already, you'll have to create an ssh key for your laptop and put the public key on github or gitlab



UITableView

A `UITableView` is a view object

In the Model-View-Controller (MVC) design pattern:

- model: holds data and knows nothing about the UI
- view: what the users sees and interacts with; knows nothing about the model objects
- controller: keeps the UI and the model in sync and controls the flow of the application

`UITableView` is part of the view component

- it shows, in list form, elements from the model

UITableView

`UITableView` needs the following

- a view controller to manage its appearance on screen
- a data source: `UITableView` asks the data source for the number of rows to display and for the data to be shown in the rows; the data can be any type object so long as it conforms to the `UITableViewDataSource` protocol
- a delegate that can inform other objects of events involving the `UITableView`; can be any object so long as it conforms to the `UITableViewDelegate` protocol

An instance of `UITableViewController` can fill all three of these roles

- this is a subclass of `UIViewController`

UITableView

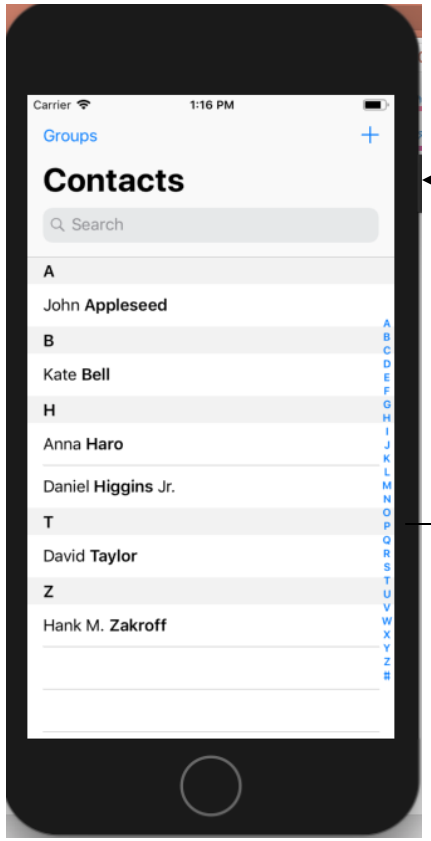
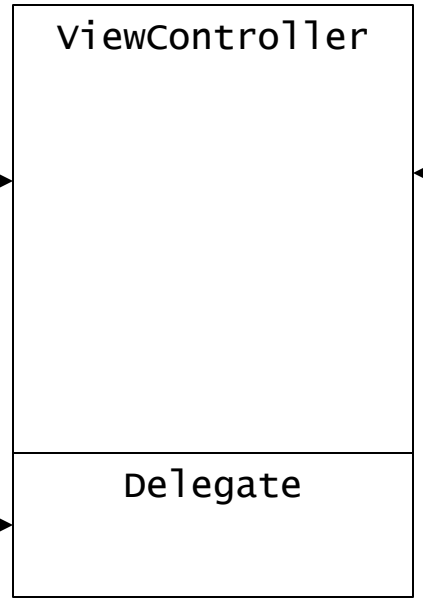


table view



name	phone number	phone number	image
name	phone number	phone number	image
name	phone number	phone number	image
name	phone number	phone number	image
name	phone number	phone number	image
name	phone number	phone number	image
name	phone number	phone number	image
name	phone number	phone number	image

data source

Subclassing `UITableViewController`

To create a view with a `UITableView`, create a view controller that subclasses `UITableViewController`

```
class ItemsViewController: UITableViewController {

}
```

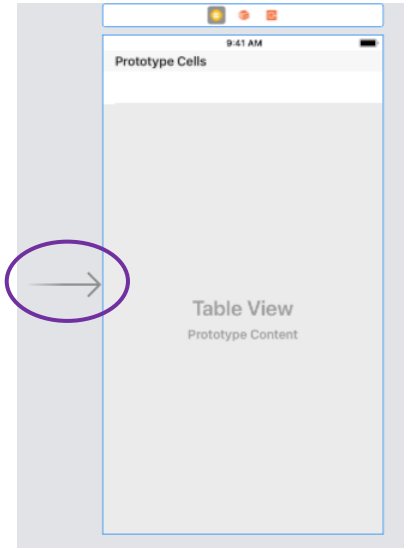
`UITableViewController` is a subclass of `UIViewController`

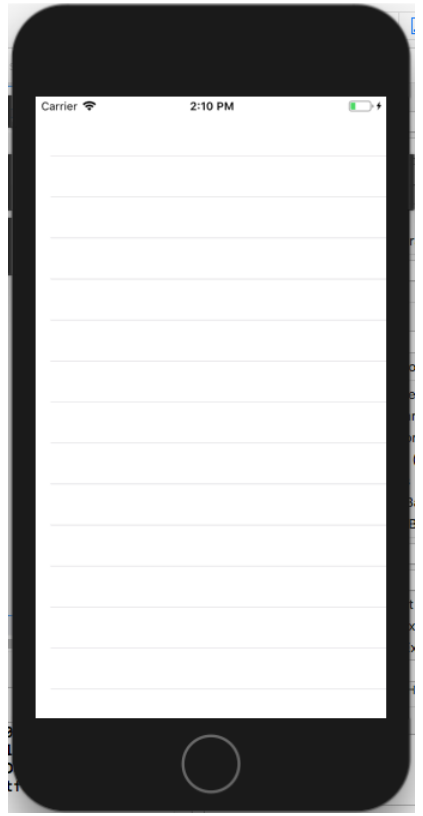
- so it inherits the `view` property
- and in this case, the view will be an instance of `UITableView`

ItemsViewController.swift

- Delete the existing ViewController that Xcode creates for a new project
- Create a new Swift class file: `ItemsViewController.swift`
- Drag a Table View Controller from the object library to the canvas
- Change the class for the Table View Controller to `ItemsViewController`
- Make `ItemsViewController` the initial view controller

this arrow points to the initial view controller for the app





empty table view

The Model Layer

We need a class to represent the data that the app will manipulate

Some aspect of each model instance (such as a name) will be shown in a table row

The Model Layer: `Item.swift`

`Item` is the class that will be the basis of the model

It should be a subclass of `NSObject`

- `NSObject` is the base class that most Objective-C classes inherit from
- this includes the various UI classes we've used, including `UIView`, `UITextField`, `UIViewController`
- `NSObject` has various things that let an object interact with the runtime system

```
class Item: NSObject {
    var name: String
    var valueInDollars: Int
    var serialNumber: String?
    let dateCreated: Date
}
```

Designated and Convenience Initializers

A class gets a free initializer when all of its properties have default values

```
init()
```

If some properties don't have default values, then we have to supply a designated initializer

- designated initialize: an initializer that does give all of the properties initial values

We can also provide other initializers

- convenience initializers
- they must call through to a designated initializer

Designated Initializer

```
init(name: String, serialNumber: String?, valueInDollars: Int) {
    self.name = name
    self.valueInDollars = valueInDollars
    self.serialNumber = serialNumber
    self.dateCreated = Date()

    super.init()
}
```

Convenience Initializer for `Item`

```
convenience init(random: Bool = false) {
    if random {
        let adjectives = ["Fluffy", "Rusty", "Shiny"]
        let nouns = ["Bear", "Spork", "Mac"]

        var idx = arc4random_uniform(UInt32(adjectives.count))
        let randomAdjective = adjectives[Int(idx)]

        idx = arc4random_uniform(UInt32(nouns.count))
        let randomNoun = nouns[Int(idx)]

        let randomName = "\(randomAdjective) \(randomNoun)"
        let randomValue = Int(arc4random_uniform(100))
        let randomSerialNumber = UUID().uuidString.components(separatedBy: "-").first!

        self.init(name: randomName, serialNumber: randomSerialNumber, valueInDollars: randomValue)
    } else {
        self.init(name: "", serialNumber: nil, valueInDollars: 0)
    }
}
```

`UITableView`'s Data Source

Cocoa Touch (the iOS UI system) displays a row of data in a `UITableView` by querying the data source

- it asks questions in order to know what to display on the screen—especially as the user scrolls

Here are the questions it asks:

- "how many elements are there in total?"
- "what's the information to display in row #1?"
- "what's the information to display in row #2?"
- etc.

`UITableView`'s Data Source

Here, the `UITableViewController` itself is the data source

For the model, create an array of model objects (here, an array of `Item` objects)

- and then encapsulate that in a new class called `ItemStore`
- and then give `ItemsViewController` a reference to the `ItemStore`

This is general good practice

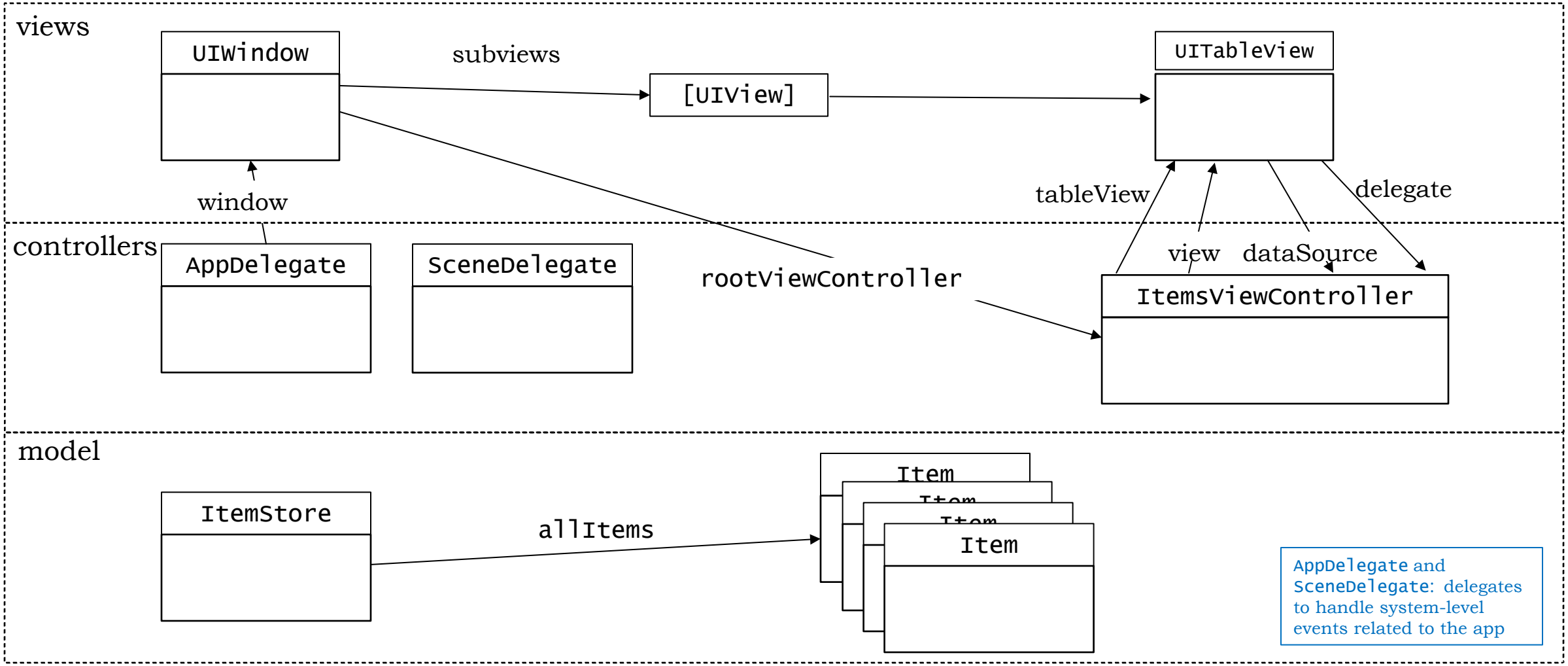
- a class `XYZ` to represent each model object
- a class `XYZStore` that contains an array of `XYZ` objects—rather than having a bare array of `XYZ`

ItemStore Class

```
import UIKit


class ItemStore {
    var allItems = [Item]()
}
```

This holds all of the items (rather than having a bare array)

Homepwner Object Diagram



Creating Item Instances

```
@discardableResult func createItem() -> Item
```

- this is an interesting software-engineering concept
- it will suppress a warning from the compiler if the return value of the function is not actually used
- otherwise, we would get this:

```
init() {
    for _ in 0..<5 {
        createItem()
    }
}
```

⚠ Result of call to 'createItem()' is unused

Creating Item Instances

```
@discardableResult
func createItem() -> Item {
    let newItem = Item(random: true)
    allItems.append(newItem)
    return newItem
}
```

Giving the Controller Access to the Store

We need to give the `UITableViewController` access to the list of items (the ItemStore)

```
class ItemsViewController: UITableViewController {
    var itemStore: ItemStore!
}
```

- Note the implicitly unwrapped optional
- This allows us to perform the initialization of `ItemStore` in a different place in the code (more on this later)
- it's saying "create this property—its value will be set later, but not during initialization"

AppDelegate

When an application first launches, this method in the `AppDelegate` is called:

```
application(_:didFinishLaunchingWithOptions:)
```

The `AppDelegate` class is declared in `AppDelegate.swift`

- it's created automatically when you create an iOS project in Xcode

More about application states in Chapter 16

SceneDelegate

Starting in iOS13, there's a new mechanism to do initialization on app startup

The primary motivation is to allow an app to display two different windows

- such as on a large-format screen
- or two running instances of a single app

See https://stackoverflow.com/questions/56498099/difference-between-scenedelegate-and-appdelegate

SceneDelegate

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options connectionOptions:
UIScene.ConnectionOptions) {
    // Use this method to optionally configure and attach the UIWindow `window` to the
    // provided UIWindowScene `scene`.
    // If using a storyboard, the `window` property will automatically be initialized and
    // attached to the scene.
    // This delegate does not imply the connecting scene or session are new (see
    // application:configurationForConnectingSceneSession` instead).

    guard let _ = (scene as? UIWindowScene) else { return }

    // create an ItemStore
    let itemStore = ItemStore()

    // access the ItemsViewController and set its item store
    let itemsController = window!.rootViewController as! ItemsViewController
    itemsController.itemStore = itemStore
}
```

Dependency Inversion

Why was `ItemStore` in `ItemViewController` set externally (i.e., not in `ItemViewController`)?

- why not just have the controller create and initializer the model (`ItemStore`)?

Dependency inversion: decouple objects in an application by inverting certain dependencies between them

- high-level objects should not depend on low-level objects; both should depend on abstractions
- abstractions should not depend on details; details should depend on abstraction

Dependency Inversion

Here, the store (`ItemStore`) is the lower-level object that handles the Item instances through details that are only known to that class

- `ItemStore` knows the details of Item

The `ItemViewController` is a higher-level object that knows it will be interacting with the store

- it will be provided with a utility object (the store) from which it can obtain a list of `Item` instances to which it can pass new or updated Item instances
- however, only `Item` and `ItemStore` will know the details of how to manage Items
- `ItemViewController` should not need to know any details of Item

So in this way, we could even replace `ItemStore` with a different object that fetches Items in a different fashion, and the `ItemViewController` would not have to change

Data-Source Methods

When a `UITableView` wants to know what to display, it calls methods from the set of methods declared in the `UITableViewDataSource` protocol

`UITableViewController` conforms to the `UITableViewDataSource` protocol

That means that `ItemViewController`, which is a subclass of `UITableViewController`, must implement all required methods from `UITableViewDataSource`

`UITableViewDataSource` Protocol

Configuring a Table View

`func tableView(UITableView, cellForRowAt: IndexPath) -> UITableViewCell`

Asks the data source for a cell to insert in a particular location of the table view.

Required.

`func numberOfSections(in: UITableView) -> Int`

Asks the data source to return the number of sections in the table view.

`func tableView(UITableView, numberOfRowsInSection: Int) -> Int`

Tells the data source to return the number of rows in a given section of a table view.

Required.

`func tableView(UITableView, titleForHeaderInSection: Int) -> String?`

Asks the data source for the title of the header of the specified section of the table view.

`func tableView(UITableView, titleForFooterInSection: Int) -> String?`

Asks the data source for the title of the footer of the specified section of the table view.

UITableViewDataSource

When the `UITableView` wants to display data, it needs to call methods on its dataSource

`tableView(_:numberOfRowsInSection) -> Int`

- this returns an integer corresponding to the total number of rows that the `UITableView` will be displaying
- we'll skip the section concept for now

`tableView(_:cellForRowAt:) -> UITableViewCell`

- this will tell the dataSource to return the data to be displayed for the row at the given position in the `UITableView`

UITableViewDataSource

In ItemViewController:

```
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
    return itemStore.allItems.count
}
```

This is straightforward: the total number of items in the table is the total number of items in the data store

Technically, this should be encapsulated by `ItemStore`

- the user of `ItemStore` should not need to know about the underlying representation of the data inside of `ItemStore`

UITableViewCell

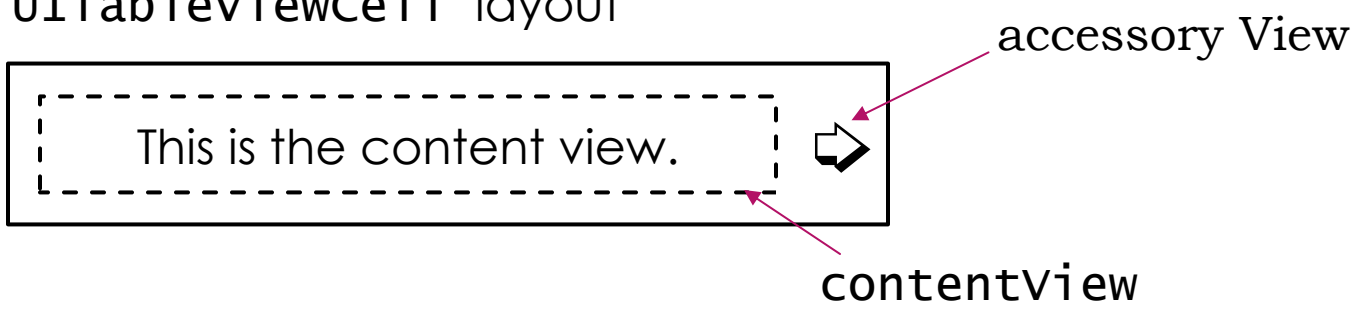
Each row of a `UITableView` is a cell

- it will be an instance of `UITableViewCell`

A cell has a single subview: its `contentView`

- the `contentView` is a superview for the contents of the cell
- the cell can also have an accessory view, for an action-oriented icon (such as ✓)

`UITableViewCell` layout

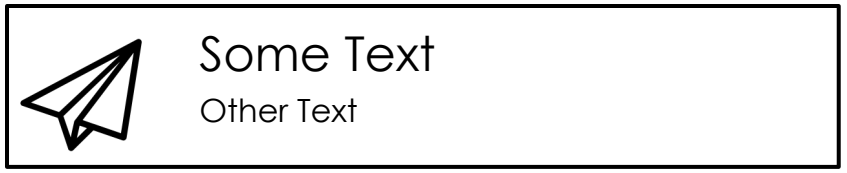


UITableViewCell

- The `contentView` has three subviews of its own
  - two `UILabel` instances that are properties of `UITableViewCell`
  - and a `UIImageView` that is also a property

- Each cell also has a `UITableViewCellStyle` that determines which subviews are used and what their position is within the `contentView`

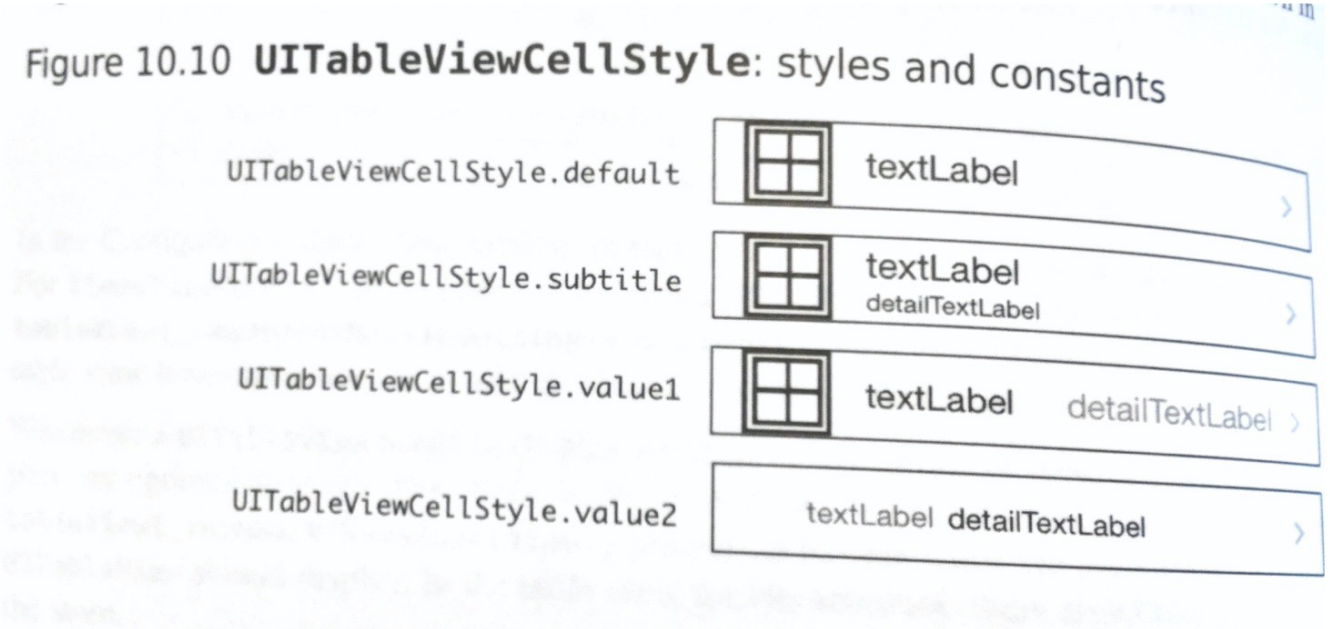
- Figures 10.9 and 10.10 show this



- The basic idea is that the UIKit provides a template for building the rows that will appear in the `UITableViewCell`

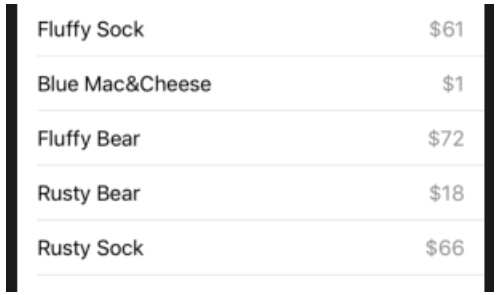
UITableViewCell Details

Figure 10.10 **UITableViewCellStyle**: styles and constants



Creating and Retrieving `UITableViewCell`s

Here's what the rows in the UITableView for this app will look like:



Each row will display an item's name and price

The `UITabelViewController` will call `tableView(_:cellForRowAt:)` for a particular row number

- we implement this function to fill in the name of the item and the `valueInDollars` for the item

Specifying a Row for the `UITableView`

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // creatd an instance of UITableViewCell, with default appearance
    let cell = UITableViewCell(style: .value1, reuseIdentifier: "UITableViewCell")

    // set the text on the cell with the description of the item
    // that is at the nth index of items, where n = row this cell
    // will appear in on the tableview
    let item = itemStore.allItems[indexPath.row]
    cell.textLabel?.text = item.name
    cell.detailTextLabel?.text = "$\(item.valueInDollars)"

    return cell
}
```

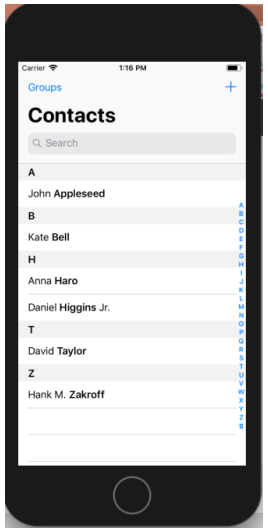
IndexPath

A `UITableView` can use two levels of hierarchy to organize the data that it's presenting

1. sections
2. rows within each section

Example: the Contacts app

- a section for each starting letter
- entries within each section



IndexPath

`IndexPath` is a struct in the Foundation framework that has two properties:

```
int section
int row
```

For a `UITableView` that has only a single section, the value of `section` will be zero

Reusing `UITableViewCells`

As we scroll up and down in a `UITableView`, cells move off of the screen and come onto the screen

We can reuse the cells that move off the screen for the rows that are going to move onto the screen—the cells are like reusable containers

- imagine a list that has 1000s of elements—it would not make sense to save a `UITableCell` for each element
- we only need a `UITableCell` for each element that is displayed

The cells that scroll off the screen end up in a *reuse pool*

Reusing `UITableViewCells`

The slight complication is that a `UITableView` can display different types of cells

- we might want to subclass `UITableViewCell` to create a different look/behavior for some of the cells (i.e., rows)
- so we could have different kinds of cells in the reuse pool

We have to make sure that a cell we want to reuse is the right kind of cell for the data we want to put in it

- every cell has a `reuseIdentifier`, which is a `String`
- when the dataSource asks the table view for a reusable cell, it passes a string that says "I want a cell with this reuse identifier"
- by convention, the reuse identifier is typically the name of the cell class

Prototype Cells

To reuse cells:

- we need to register with the table view either a prototype cell or else a class for a specific reuse identifier

Simplest: register the default `UITableViewCell` class

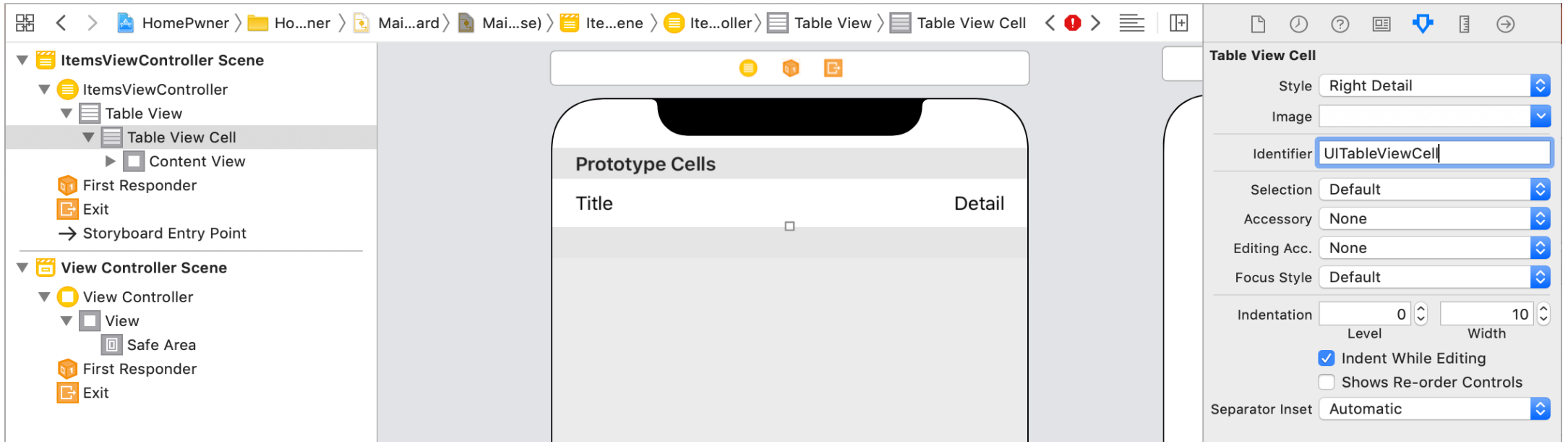
- this will tell the table view "any time I ask for a cell with this this reuse identifier, give me back a cell that is of this specific class"
- and then the table view will either give you back a cell of the desired type from the reuse pool or will create a new one if there are none available in the reuse pool

dequeueReusableCell(withIdentifier:for:)

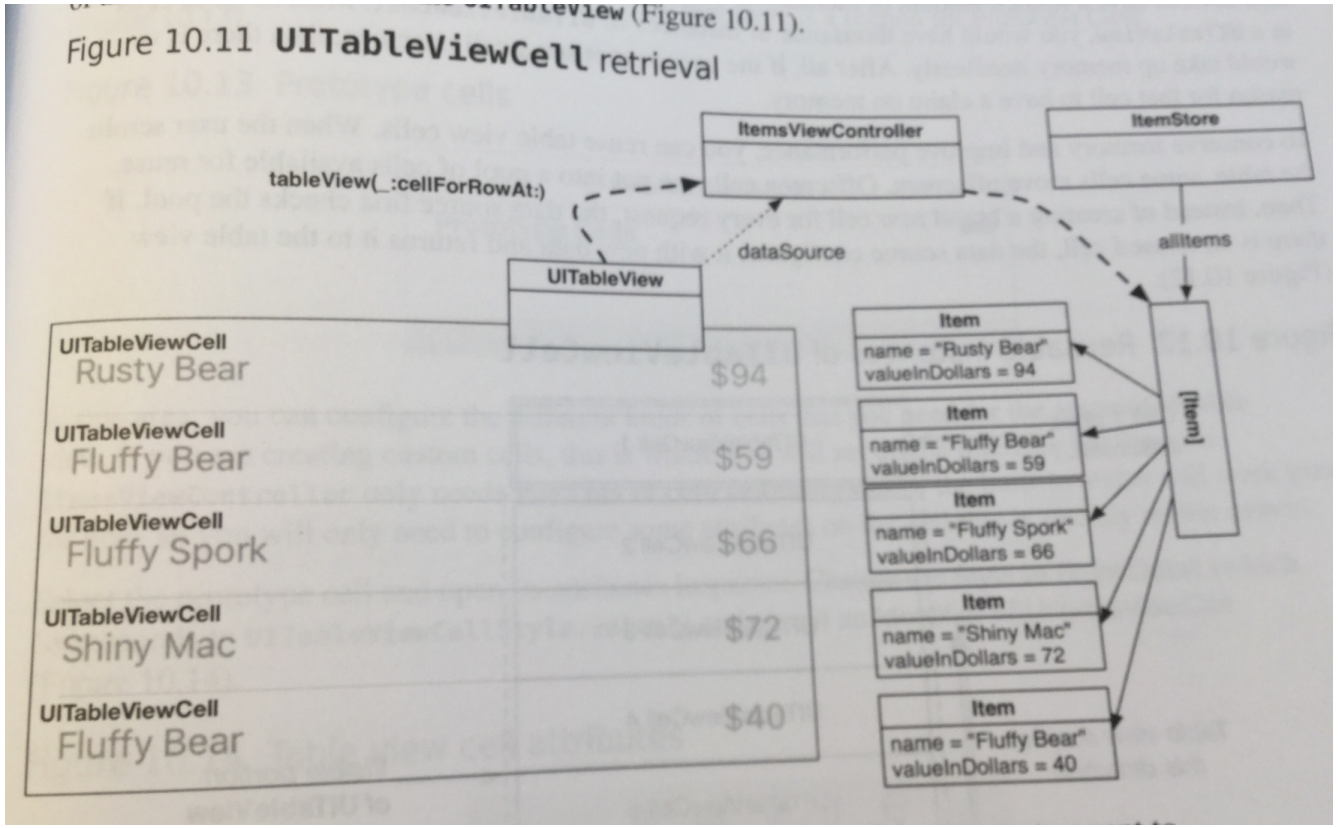
```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // create an instance of UITableViewCell, with default appearance
    // let cell = UITableViewCell(style: .value1, reuseIdentifier: "UITableViewCell")

    // get a new or recycled cell
    let cell = tableView.dequeueReusableCell(withIdentifer: "UITableViewCell", for: indexPath)

    // set the text on the cell with the description of the item
    // that is at the nth index of items, where n = row this cell
    // will appear in on the tableview
    let item = itemStore.allItems[indexPath.row]
    cell.textLabel?.text = item.name
    cell.detailTextLabel?.text = "$\(item.valueInDollars)"

    return cell
}
```

Configuring the Prototype Cell

- The Table View has a prototype cell
- Change its style to "Right Detail" (which corresponds to UITableViewCellStyle.value1
- And give it an identifier of UITableViewCell



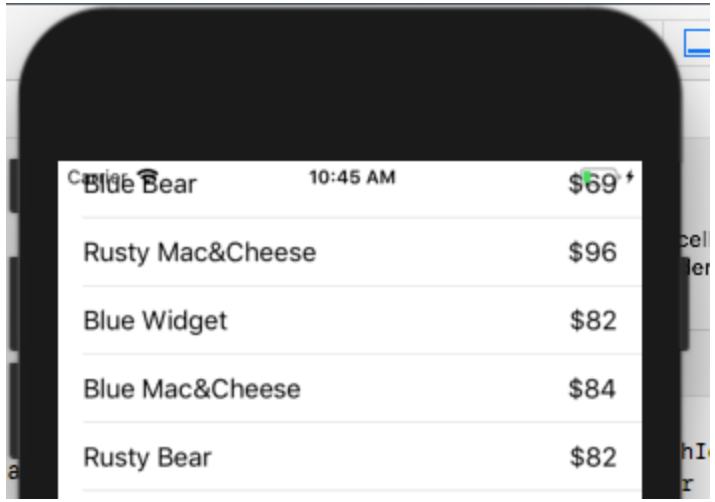
UITableView Processing



Content Insets

Currently, the `UITableView` fills the whole screen

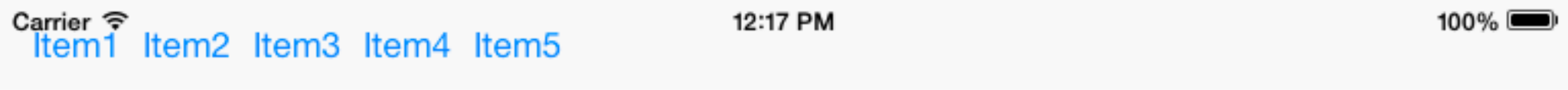
This means that it underlaps the status bar at the top of the screen:



Content Insets

- The book (on p. 193) says to change the `contentInset` property of the `UITableView`
- But Apple has changed the default behavior of this UI element:

"Beginning with iOS 7, view controllers are displayed full screen, by default, as shown in Figure 1. This means they will cover the entire screen including the area underneath the status bar."



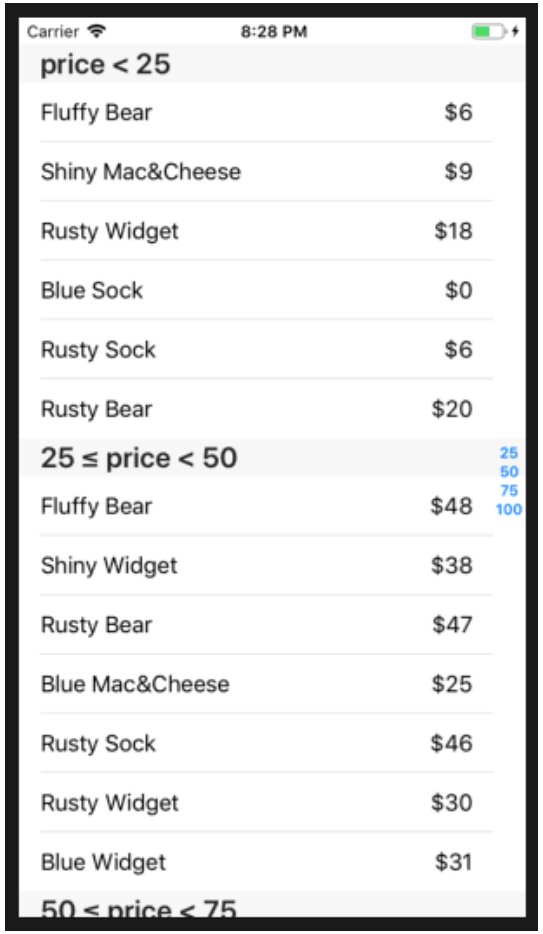
- It's more complex to change this behavior—it's OK to leave it this way if you want
  - have to define additional views, I think; still not sure how to change this behavior!
  - see https://developer.apple.com/library/archive/qa/qa1797/_index.html

Section Headers

For tables with more than one section, it's easy to add section headers

- a section header is just a string you specify

```
override func tableView(_ tableView: UITableView,
            titleForHeaderInSection section: Int) -> String?
```

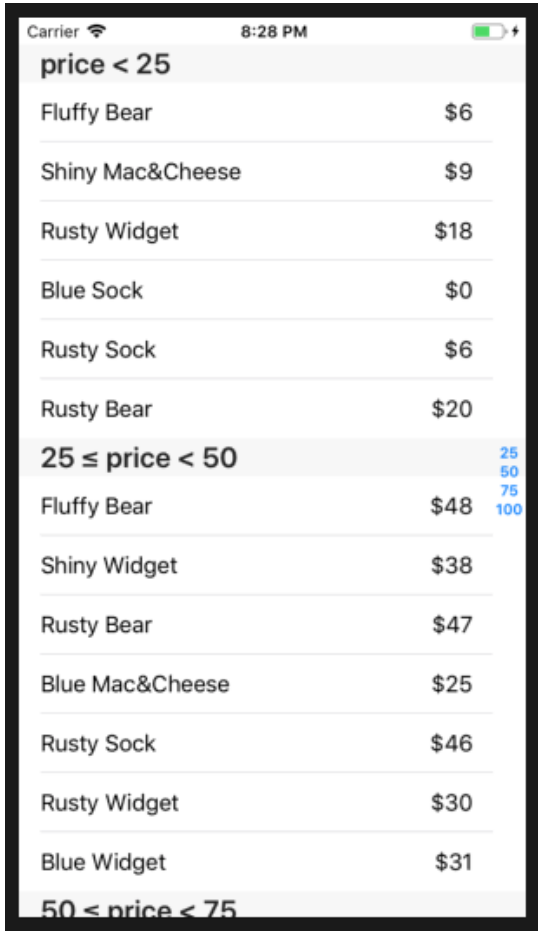


Section Index Titles

The little strings along the edge are Section Index Titles

- they let you jump (or scroll) from section to section

```
override func sectionIndexTitles(for tableView:
                        UITableView) -> [String]? {
    return itemStore.sectionIndexTitles
}
```



Challenges

Do the Bronze Challenge

- most of the changes you make will be in `ItemStore`
- you'll have to provide an implementation of `numberOfSections(in:)`
- and you'll have to use `IndexPath`

The Silver Challenge is also nice