GENERAL ASSEMBLY

MOBILE DEVELOPMENT

TABLE VIEWS

William Martin
Head of Product, Floored

LEARNING OBJECTIVES

- Deploy UITableViews in View Controllers.
- Know what it means to "meet a Protocol."
- Create "delegate" objects and describe the delegation pattern.
- Select between the default table cell templates.
- Implement passing data from master table to the detail View Controller.

UITABLEVIEWS

UITABLEVIEWS

- Table views display a one-dimensional, vertical list of units of information.
- They are one of the most common views you'll use in iOS.

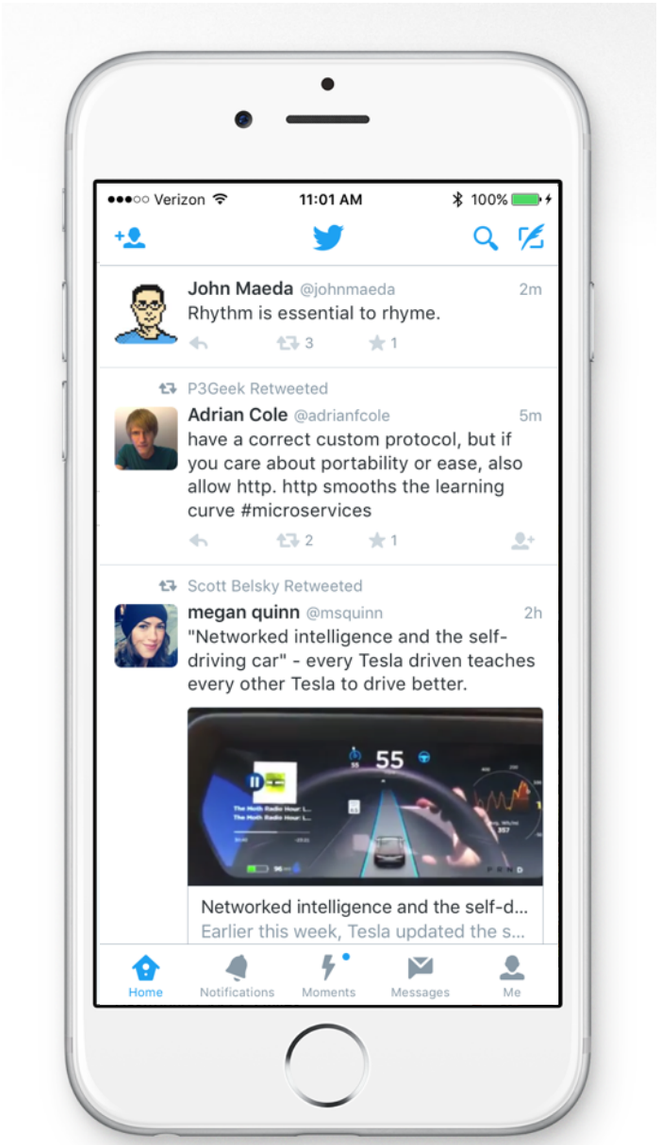
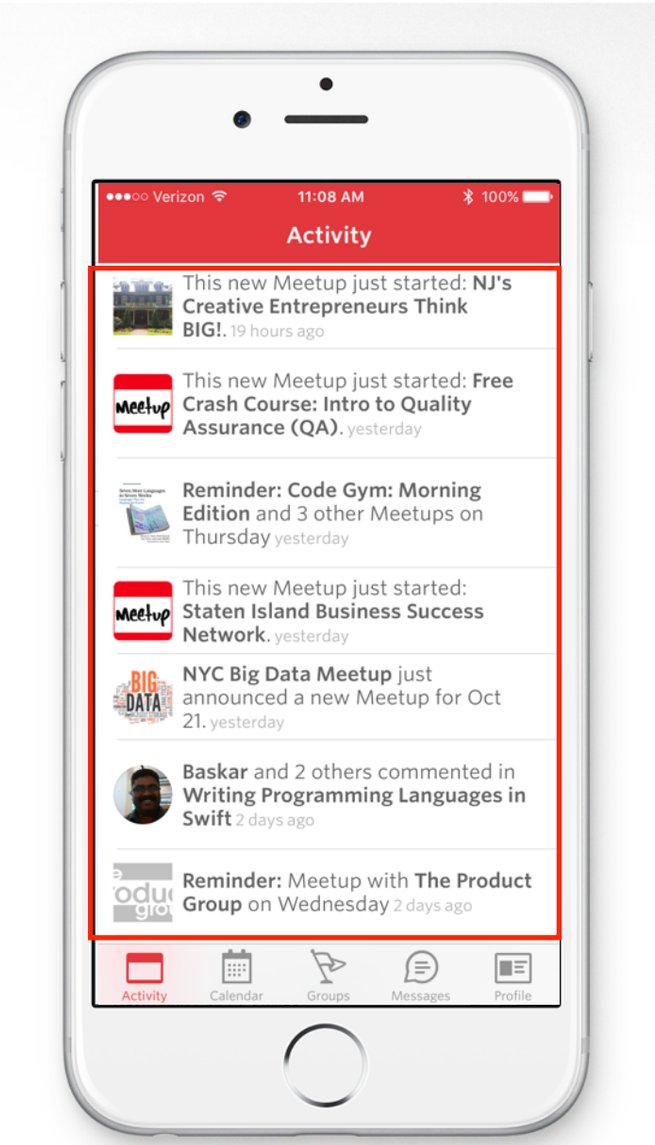
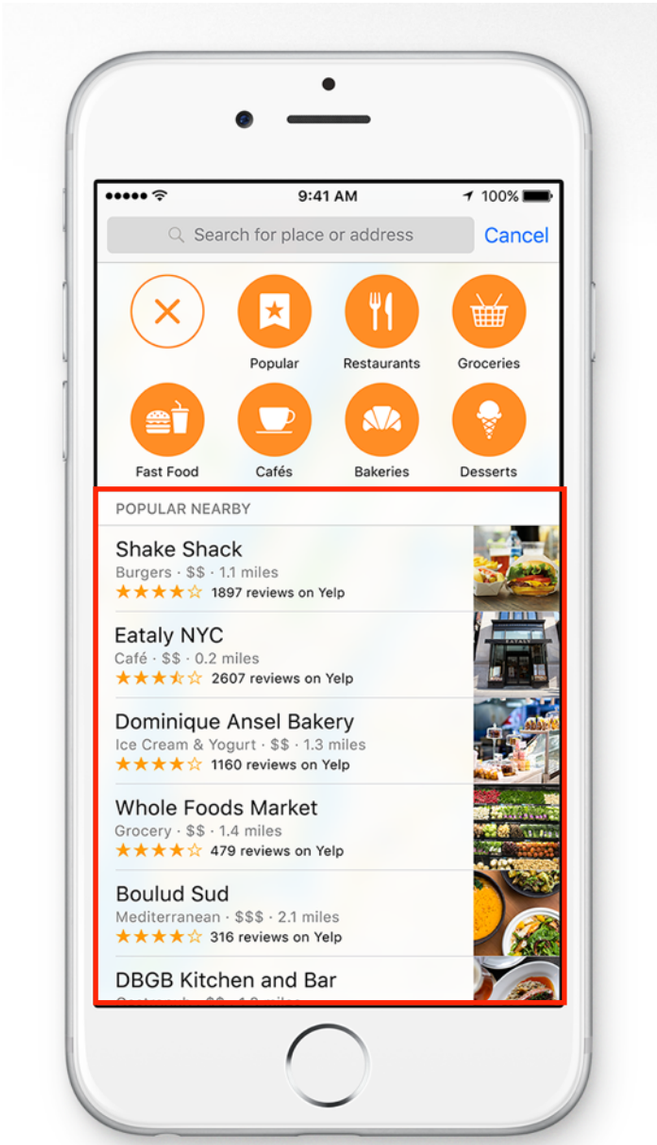


TABLE VIEWS


Verizon 11:08 AM 100%
Activity
This new Meetup just started: NJ's Creative Entrepreneurs Think BIG!. 19 hours ago
This new Meetup just started: Free Crash Course: Intro to Quality Assurance (QA). yesterday
Reminder: Code Gym: Morning Edition and 3 other Meetups on Thursday yesterday
This new Meetup just started: Staten Island Business Success Network. yesterday
NYC Big Data Meetup just announced a new Meetup for Oct 21. yesterday
Baskar and 2 others commented in Writing Programming Languages in Swift 2 days ago
Reminder: Meetup with The Product Group on Wednesday 2 days ago
Activity
Calendar
Groups
Messages
Profile


9:41 AM 100%
Search for place or address
Cancel
Popular
Restaurants
Groceries
Fast Food
Cafés
Bakeries
Desserts
POPULAR NEARBY
Shake Shack
Burgers · $$ · 1.1 miles
1897 reviews on Yelp
Eataly NYC
Café · $$ · 0.2 miles
2607 reviews on Yelp
Dominique Ansel Bakery
Ice Cream & Yogurt · $$ · 1.3 miles
1160 reviews on Yelp
Whole Foods Market
Grocery · $$ · 1.4 miles
479 reviews on Yelp
Boulud Sud
Mediterranean · $$$ · 2.1 miles
316 reviews on Yelp
DBGB Kitchen and Bar

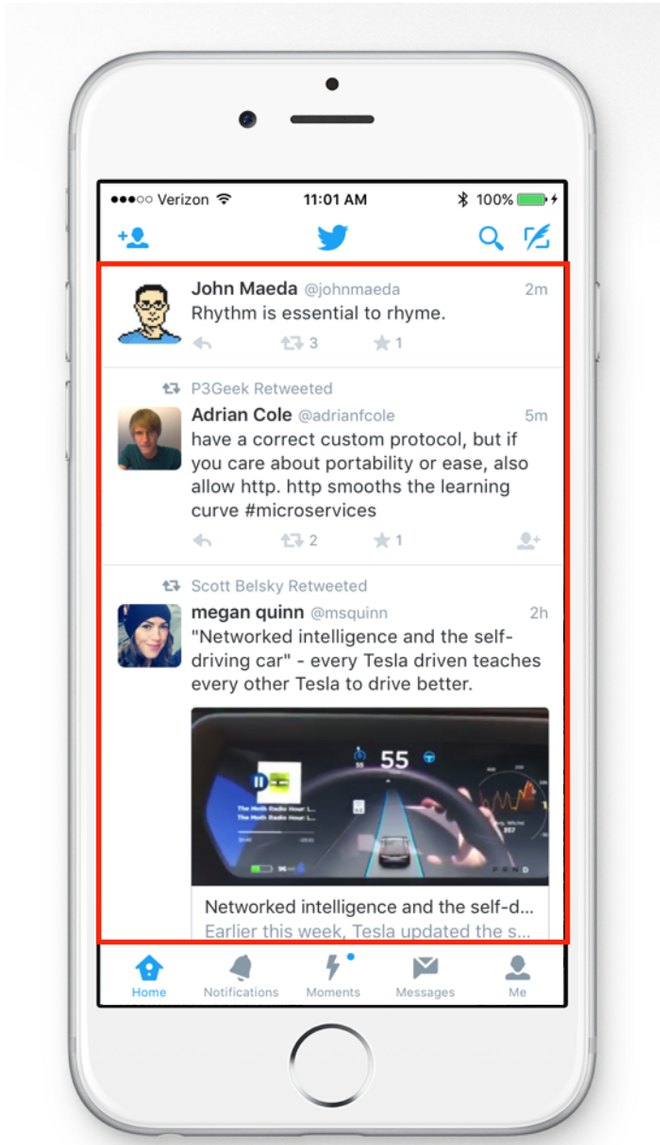

Verizon 11:01 AM 100%
John Maeda @johnmaeda 2m
Rhythm is essential to rhyme.
P3Geek Retweeted
Adrian Cole @adrianfcole 5m
have a correct custom protocol, but if you care about portability or ease, also allow http. http smooths the learning curve #microservices
Scott Belsky Retweeted
megan quinn @msquinn 2h
"Networked intelligence and the self-driving car" - every Tesla driven teaches every other Tesla to drive better.
Networked intelligence and the self-d...
Earlier this week, Tesla updated the s...
Home
Notifications
Moments
Messages
Me

TABLE VIEWS

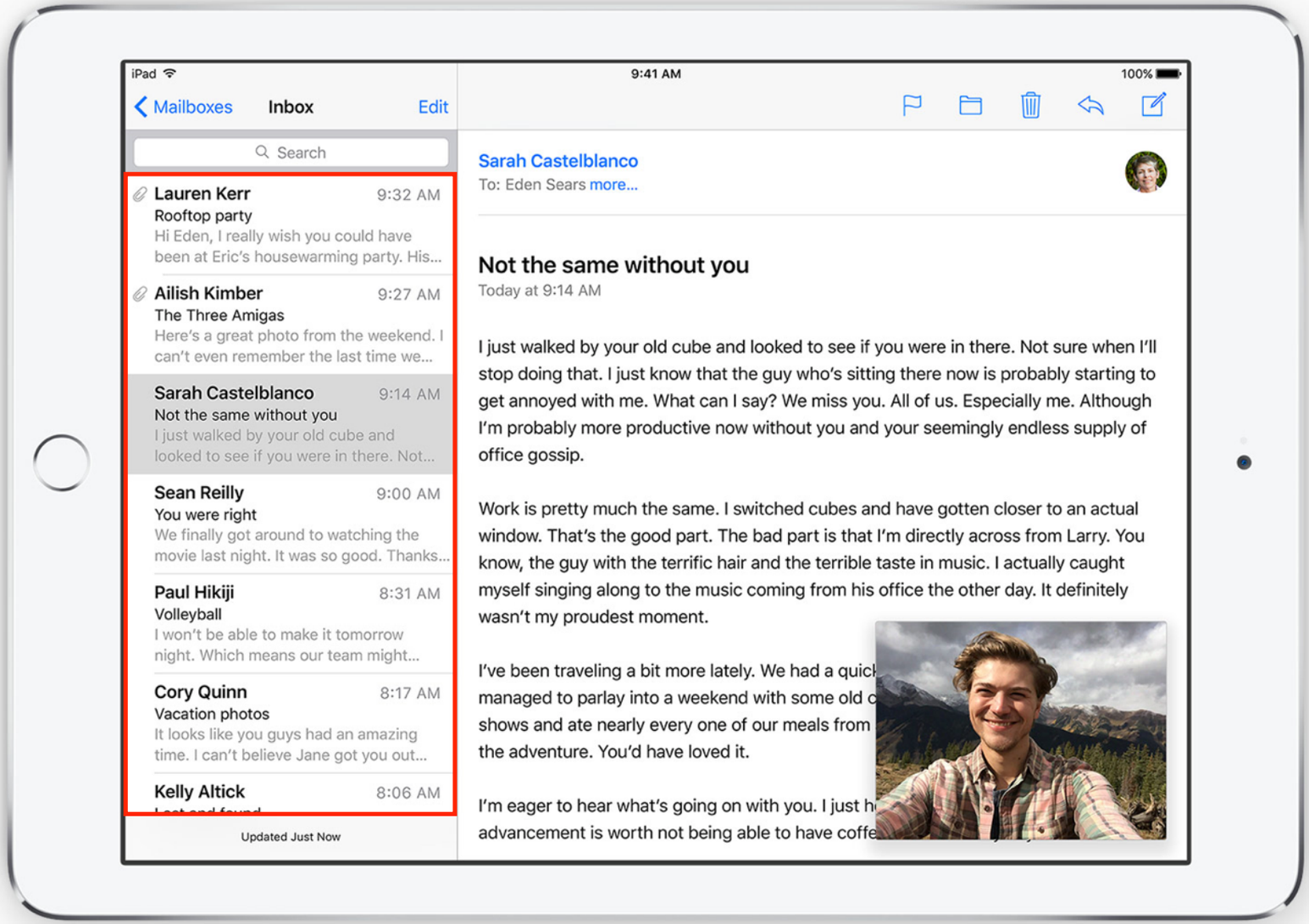
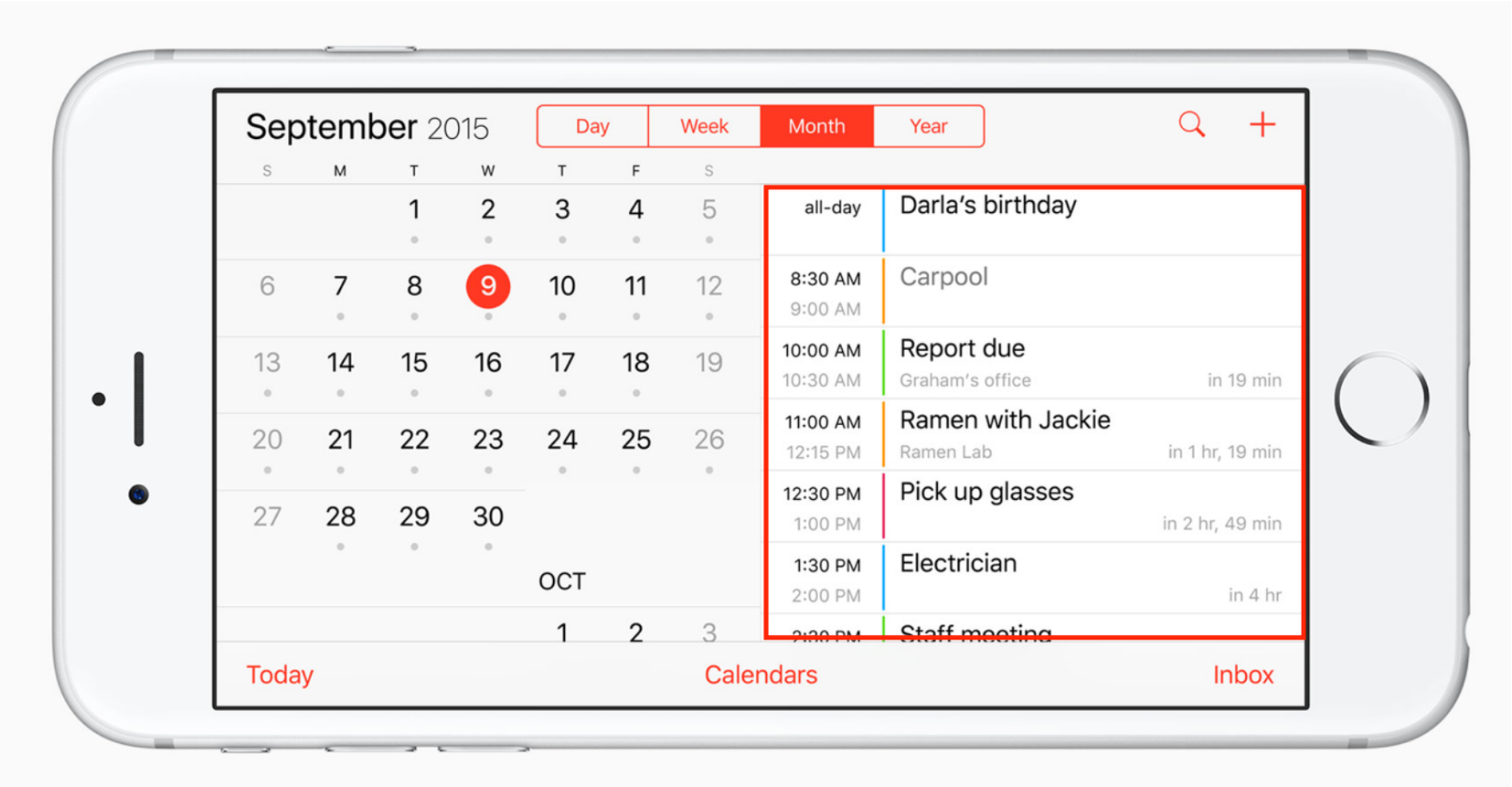

iPad
9:41 AM
100%
Mailboxes
Inbox
Edit
Search
Lauren Kerr 9:32 AM
Rooftop party
Hi Eden, I really wish you could have been at Eric's housewarming party. His...
Ailish Kimber 9:27 AM
The Three Amigas
Here's a great photo from the weekend. I can't even remember the last time we...
Sarah Castelblanco 9:14 AM
Not the same without you
I just walked by your old cube and looked to see if you were in there. Not...
Sean Reilly 9:00 AM
You were right
We finally got around to watching the movie last night. It was so good. Thanks...
Paul Hikiji 8:31 AM
Volleyball
I won't be able to make it tomorrow night. Which means our team might...
Cory Quinn 8:17 AM
Vacation photos
It looks like you guys had an amazing time. I can't believe Jane got you out...
Kelly Altick 8:06 AM
Updated Just Now
Sarah Castelblanco
To: Eden Sears more...
Not the same without you
Today at 9:14 AM
I just walked by your old cube and looked to see if you were in there. Not sure when I'll stop doing that. I just know that the guy who's sitting there now is probably starting to get annoyed with me. What can I say? We miss you. All of us. Especially me. Although I'm probably more productive now without you and your seemingly endless supply of office gossip.
Work is pretty much the same. I switched cubes and have gotten closer to an actual window. That's the good part. The bad part is that I'm directly across from Larry. You know, the guy with the terrific hair and the terrible taste in music. I actually caught myself singing along to the music coming from his office the other day. It definitely wasn't my proudest moment.
I've been traveling a bit more lately. We had a quick
managed to parlay into a weekend with some old c
shows and ate nearly every one of our meals from
the adventure. You'd have loved it.
I'm eager to hear what's going on with you. I just h
advancement is worth not being able to have coffe

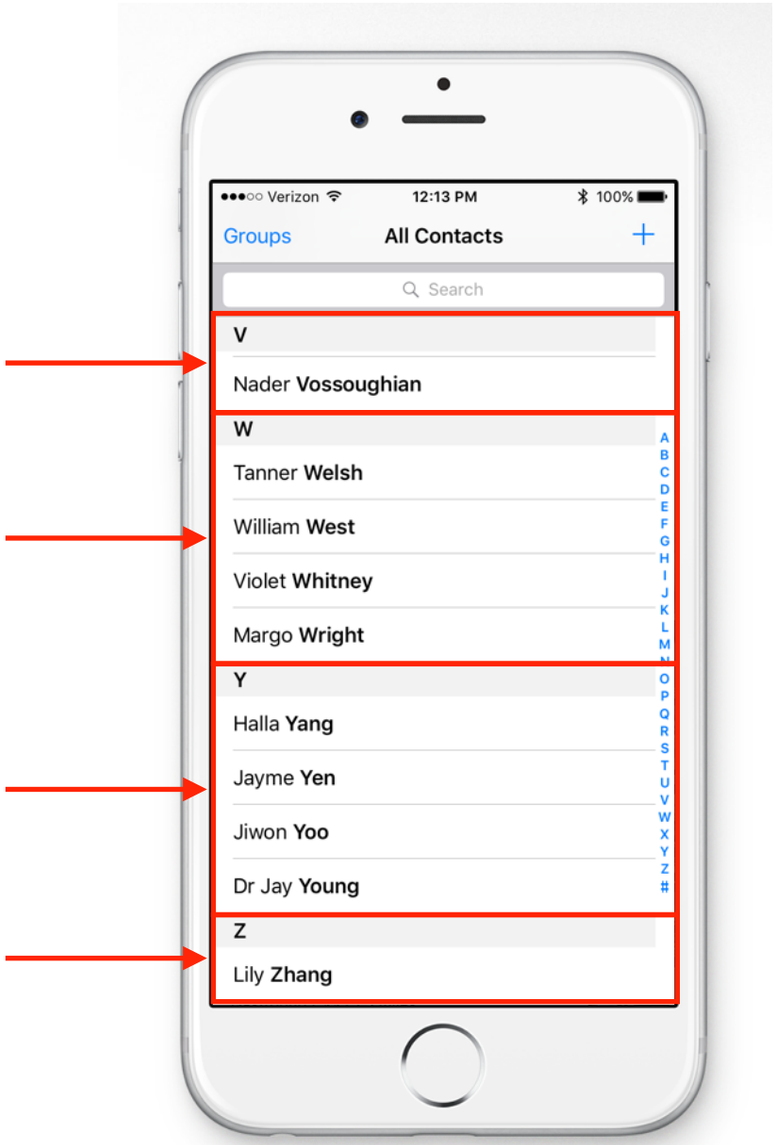
TABLE VIEWS


September 2015
Day
Week
Month
Year
S M T W T F S
all-day
Darla's birthday
8:30 AM
9:00 AM
Carpool
10:00 AM
10:30 AM
Report due
Graham's office
in 19 min
11:00 AM
12:15 PM
Ramen with Jackie
Ramen Lab
in 1 hr, 19 min
12:30 PM
1:00 PM
Pick up glasses
in 2 hr, 49 min
1:30 PM
2:00 PM
Electrician
in 4 hr
OCT
Today
Calendars
Inbox

UITABLEVIEWS

Table Views have a two-level structure:

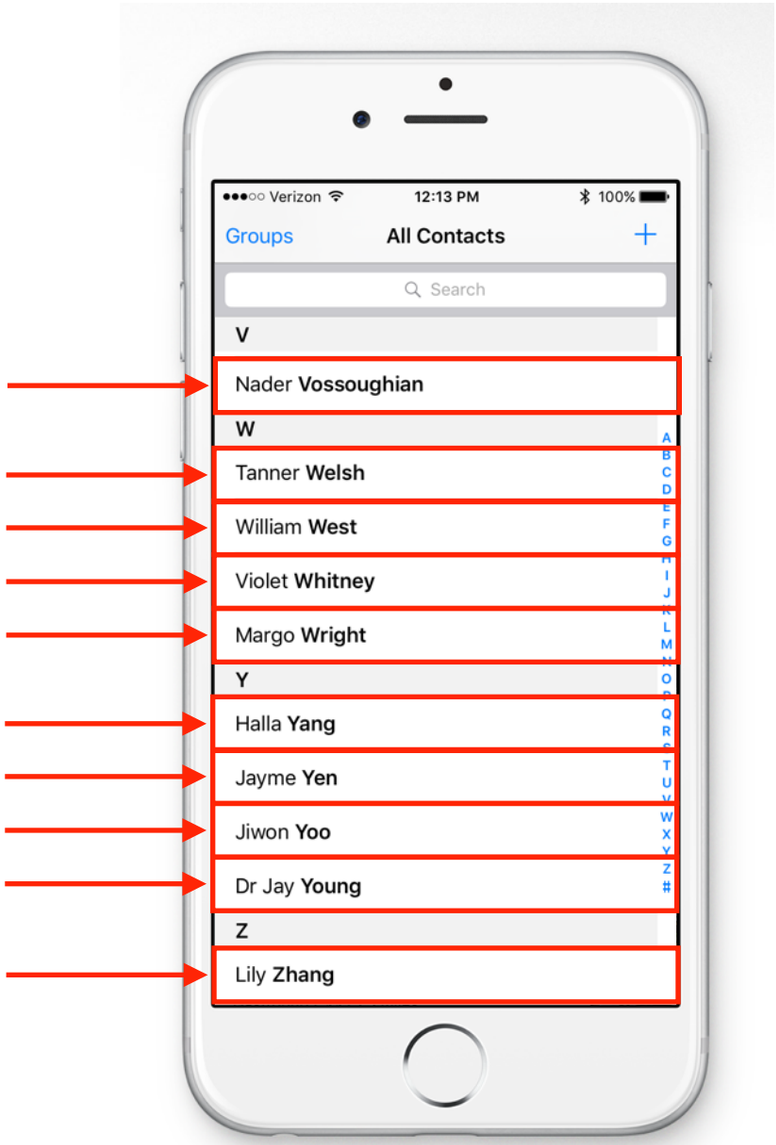
- Sections, which divide the information into related groups.
- …



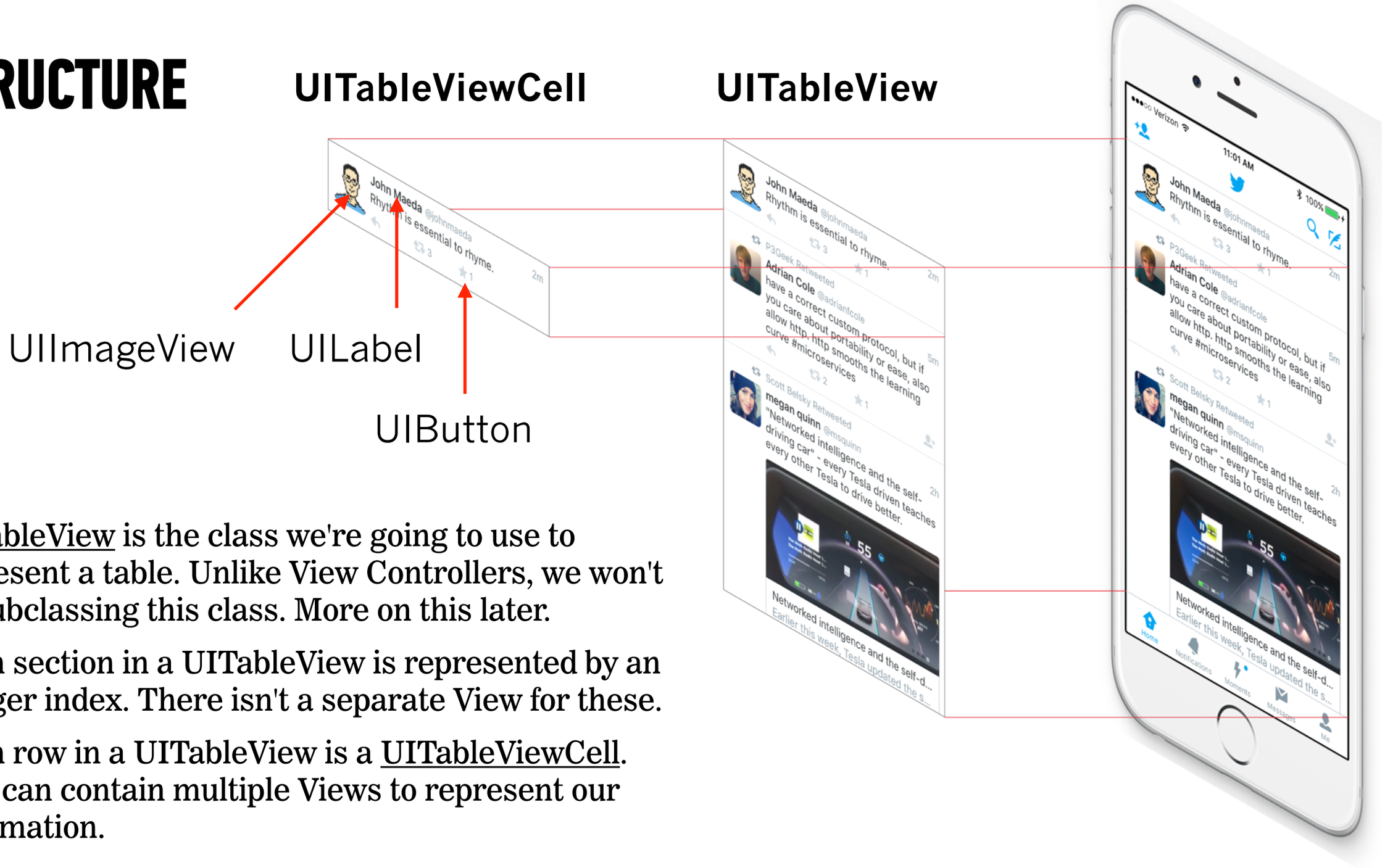
UITABLEVIEWS

Table Views have a two-level structure:

- Sections, which divide the information into related groups.
- Rows, which represent individual items in the list, each contained by a single section.



STRUCTURE



UITableView is the class we're going to use to represent a table. Unlike View Controllers, we won't be subclassing this class. More on this later.

Each section in a UITableView is represented by an Integer index. There isn't a separate View for these.

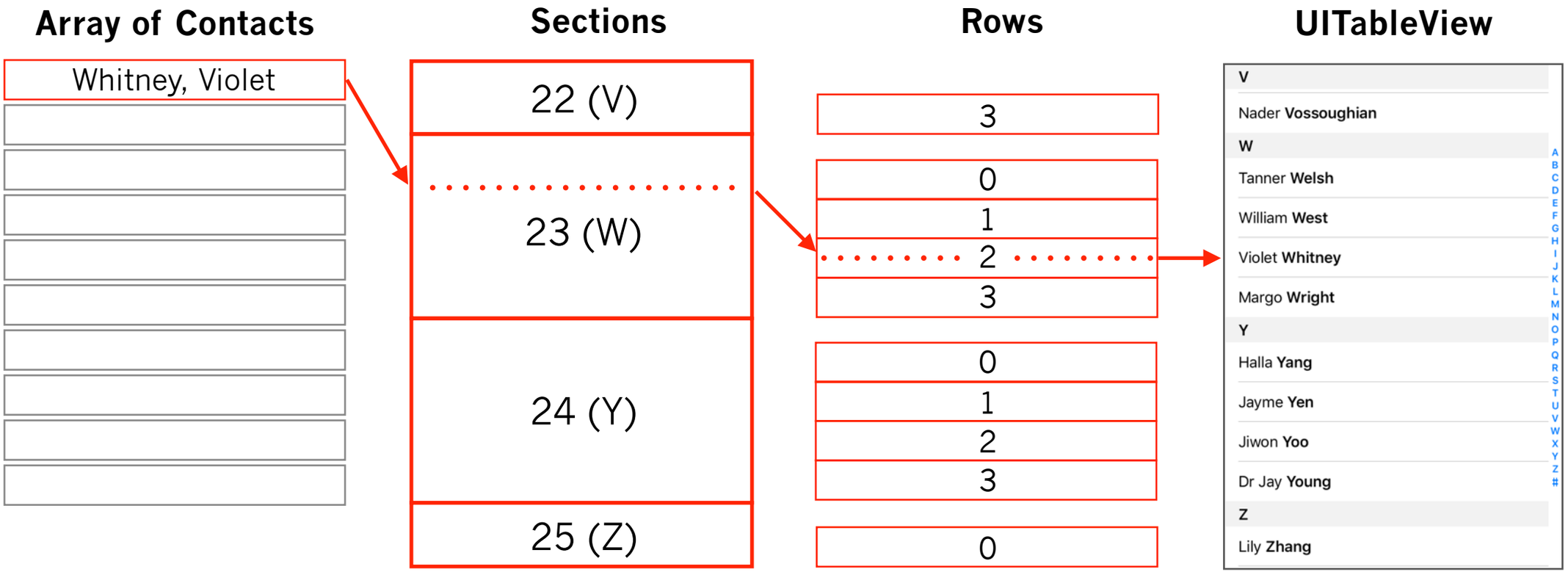
Each row in a UITableView is a UITableViewCell. This can contain multiple Views to represent our information.

HOW DO TABLE VIEWS WORK?

- Table Views essentially need a source for their data, usually held by an Array somewhere in your app.
- They use an object called NSIndexPath to represent:
  - which section a piece of data should be in (by an Integer), and
  - which row it should be placed in within the section (by another Integer)
- We'll use these objects to refer to various specific row locations (i.e. cells), like coordinates on a map, in the Table View.

HOW DO TABLE VIEWS WORK?

- To populate a Table View, we need to associate our Array's elements with a section and row.
- For the example below, we would use: `NSIndexPath(forRow: 2, inSection: 23)`



HOW DO WE USE A TABLE VIEW?

There are several things we can do to a Table View:

- Adjust its attributes in Interface Builder, like any other View.
- Pick from a set of pre-defined templates for Cells.
- Customize a Cell by adjusting its attributes.
- Enable a swipe gesture that reveals a remove button.
- Provide what to do when a cell is selected.
- Provide rules for reordering table rows.
- *and more…*

HOW DO WE USE A TABLE VIEW?

We're *required* to provide the following:

- Given the index of a section, provide the number of rows in that section.
- Given an NSIndexPath (i.e. a section index and row index), produce an instance of UITableViewCell and populate its Views with values to show the user.

How do we do these required things?

HOW DO WE USE A TABLE VIEW?

To provide Table Views with these rules and behaviors, they use a design pattern called, "delegation."

Essentially, we need to provide one or two instances of some class that have these behaviors defined.

- One is called the "data source."
- The other, the "delegate."

The following section describes delegation and why Table Views use this pattern.

DELEGATION PATTERN

CUSTOMIZING CLASSES

One of the major tasks in object-oriented programming is customizing existing classes for more specific uses.

This occurs a lot in iOS development, because UIKit provides a bunch of classes (UIViewController, UITableView, etc.) that we need to customize for our apps.

We have several mechanisms in Swift that empower us to do this:

- Subclassing
- Delegation
- Swift's "extension" keyword (which we haven't learned yet)

SUBCLASSING

Subclassing means a new class (i.e. "subclass") can "inherit" the methods and properties of another class (i.e. "superclass").

This has a number of effects that may be useful, depending on the situation.

E.g. *Polymorphism* - A subclass instance can masquerade as a superclass instance, because they share key methods and properties.

With the classes on the right, instances of all three classes can be contained in an Array of ListItem.

```
class ListItem {
    var title : String
    init(title:String) {
        self.title = title
    }
}

class Task : ListItem {
    var isComplete = false
    init(title:String) {
        super.init(title:title)
    }
}

class Email : ListItem {
    var sender : Person!
    // And so on...
}
```

COMPOSITION

Composition is the process of an object holding a reference to another object.

- For example, on the right, an instance of a Dog can hold a reference to its owner, a Human:

```
class Human {
    var name : String
    init(name:String) {
        self.name = name
    }
}

class Dog {
    var name : String
    var owner : Human!
    init(name:String) {
        self.name = name
    }
}

var dog = Dog(name: "Layla")
var em = Person(name: "Emily")
dog.owner = em
```

DELEGATION

Sometimes, subclassing doesn't do the job for us. We can use a special form of *composition* to solve scenarios like this:

A Mac OS X app that takes a folder of text files and:

- makes a website, PDF, or slideshow, plus
- publishes it to a folder, FTP server, or Amazon S3.

We might have a "renderer" object for the first, and a "publisher" object for the second. We can have a class that represents the app's content, and another to represent the app's logic.

DELEGATION - EXAMPLE

```
// A superclass for all objects that
// can "publish" somewhere.

class Publisher {
    func write(page:Page) {
        // Some code...
    }
}

class FTP : Publisher {
    // ...
}

class FileSystem : Publisher {
    // ...
}
```

```
// A superclass for all objects that
// can "render" pages.

class Renderer {
    func render(file:File) -> Page {
        // Some code...
    }
}

class Website : Renderer {
    // ...
}

class PDF : Renderer {
    // ...
}
```

DELEGATION - EXAMPLE

The process of providing instances that provide various functionality, but those instances can potentially fulfill that functionality in different ways, is called "delegation."

This provides flexibility in a way that doesn't require subclassing.

This satisfies "encapsulation" by making the App class responsible for core functionality, and the delegate objects are responsible for fulfilling interchangeable functionality.

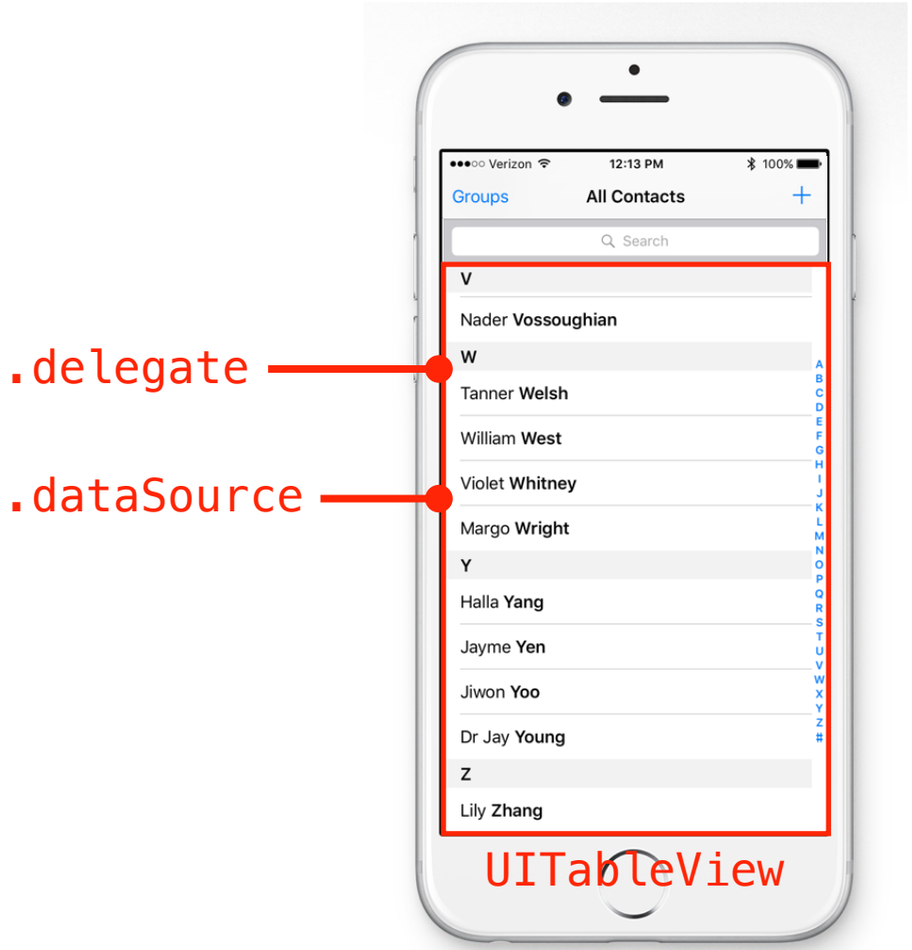
```
class App : UIApplication {
    var renderer : Renderer!
    var publisher : Publisher!
    var root : Folder

    init(root:Folder) {
        self.root = root
    }

    func run() {
        for file in self.root {
            let pg = renderer.render(file)
            publisher.write(pg)
        }
    }
}
// Delegate!
let app = App()
app.renderer = Website()
app.publisher = FTP()
```

UITABLEVIEW DELEGATION

Table Views use delegation to enable us to provide behaviors and data without subclassing UITableView.

The properties that hold the instances in question are "delegate" and "dataSource."

- delegate — Provides the ability to customize behaviors like managing selection, configuring properties of table rows (like height), managing the header and footer of the table, etc.
- dataSource — Provides the ability to customize UITableViewCells (given an NSIndexPath and our own Array of data), provide section titles, and manage other data-related tasks.



UITABLEVIEW DELEGATION

However!

Just like we don't want to subclass UITableView, the properties "delegate" and "dataSource" often need this flexibility as well.

This means we can add table functionality to instances that we're already creating, like to a subclass of UIViewController.

Thus, we're not limited to a specific class, but we still need a way to *specify what methods a Table View needs, and thus, what methods our delegate and data source need to provide.*

To achieve this, Swift provides a mechanism called a "Protocol."

SWIFT PROTOCOLS

PROTOCOLS

A protocol describes a *group of methods that a class should have defined* if it’s to be used properly by another class.

- It's like a contract that one class says another class should meet.
- An object is said to "meet a Protocol" if it includes all the required methods declared in the Protocol.
- Classes can meet as many Protocols as they’d like.
- Unlike subclassing, Protocols also enable us to declare which methods are "required" to meet the protocol.

DEFINING PROTOCOLS

Although we likely won't be writing our own Protocols, it's useful to know how they're defined, so it will make more sense how they're used.

- Protocols look like classes (syntax-wise), but they aren't.
- They don't define types; they're just a name for a collection of methods.

Here's a definition that enables other classes to act as an email filter:

```
protocol EmailFilter {
    func removeSpam(msgs:[Email]) -> [Email]
    func categorize(msgs:[Email]) -> [Email]
}
```

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html

USING PROTOCOLS

Using Protocols means adding them after the superclass when extending a class. This means you're committing to providing definitions for the methods required by the Protocol:

```
class EmailViewController : UIViewController, EmailFilter {
    override func viewDidLoad() {
        super.viewDidLoad()
    }

    func removeSpam(msgs:[Email]) -> [Email] {
        // Remove some spam messages here.
    }

    func categorize(msgs:[Email]) -> [Email] {
        // Categorization code here.
    }
}
```

IMPLEMENTING UITABLEVIEWS

PROTOCOLS FOR TABLE VIEWS

Now that we know about delegation and Protocols, we can start putting them together for Table Views.

We need to provide at a minimum, a single "dataSource" for a Table View. For an instance to serve properly as a data source, it must meet the UITableViewDataSource protocol.

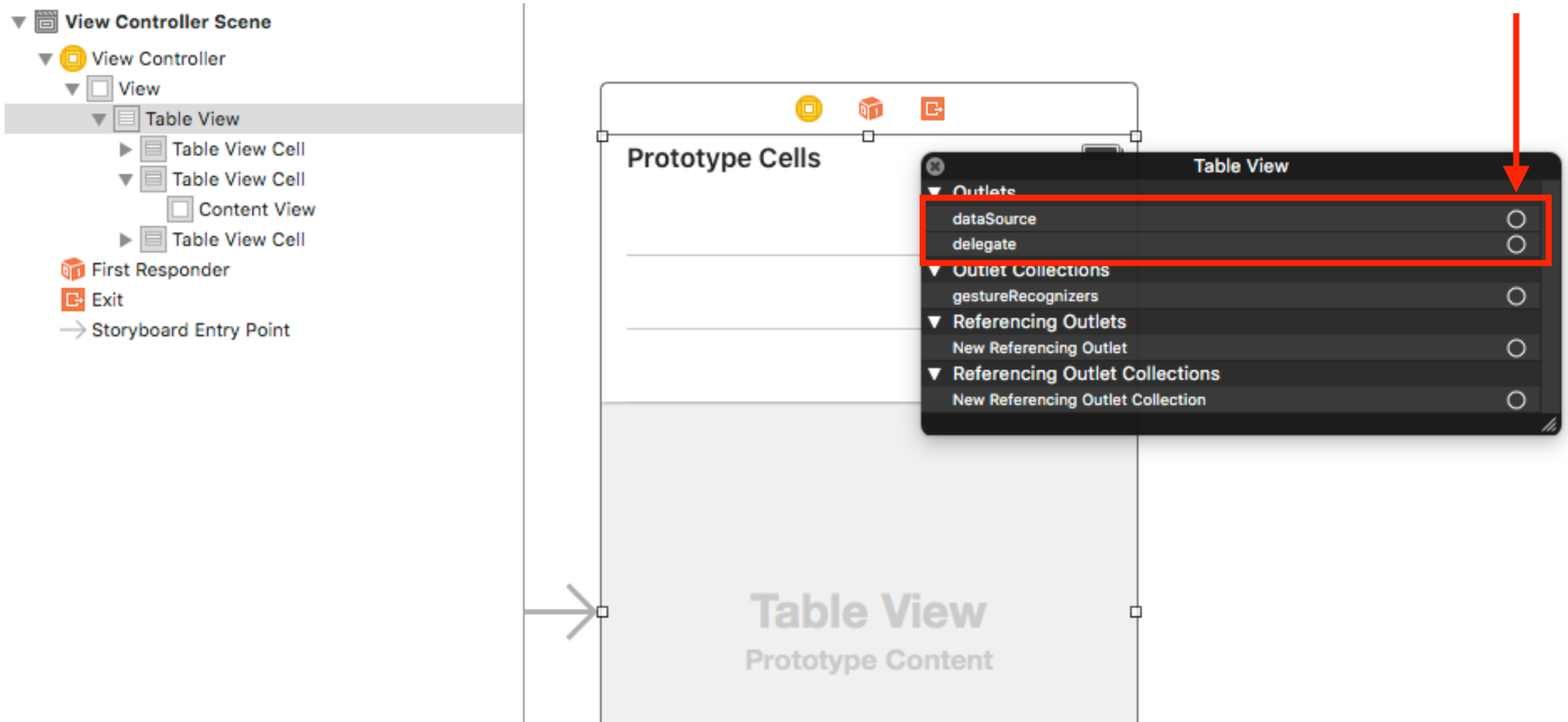
https://developer.apple.com/library/ios/documentation/UIKit/Reference/UITableViewDataSource_Protocol/

The "delegate" property is optional, but if we do use one, that instance needs to meet the UITableViewDelegate protocol.

https://developer.apple.com/library/ios/documentation/UIKit/Reference/UITableViewDelegate_Protocol/

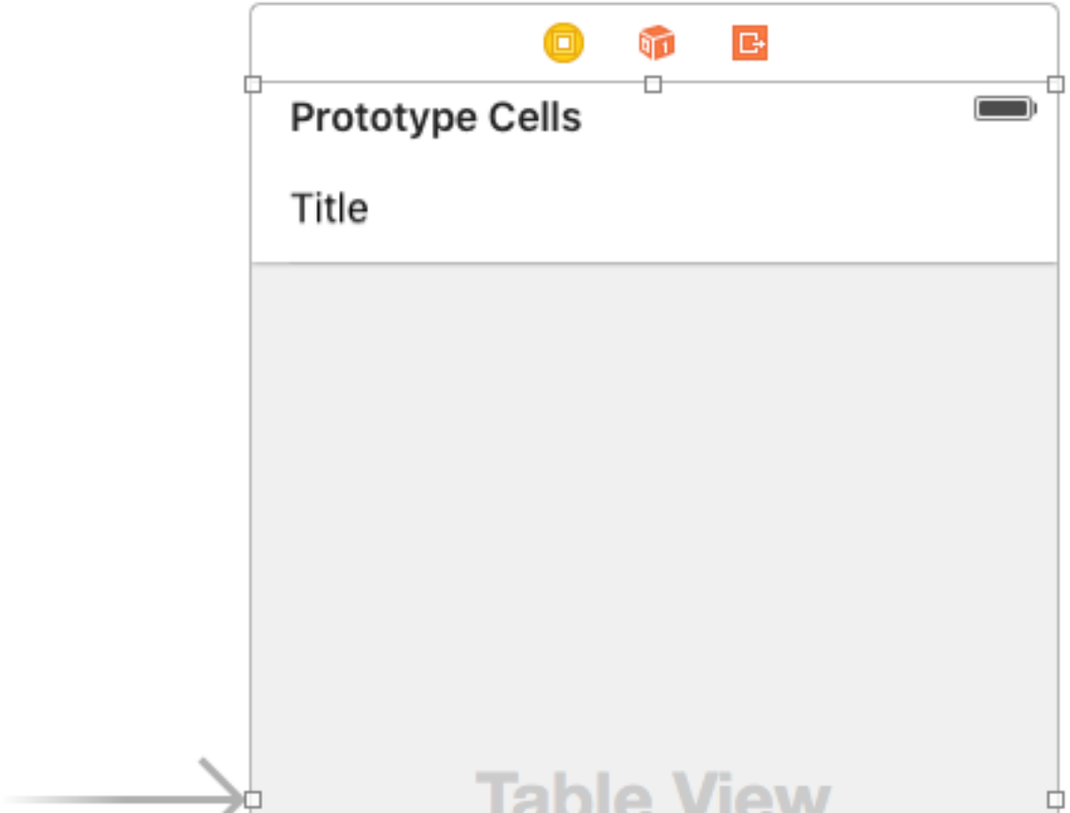
SPECIFYING DELEGATES FOR TABLE VIEWS

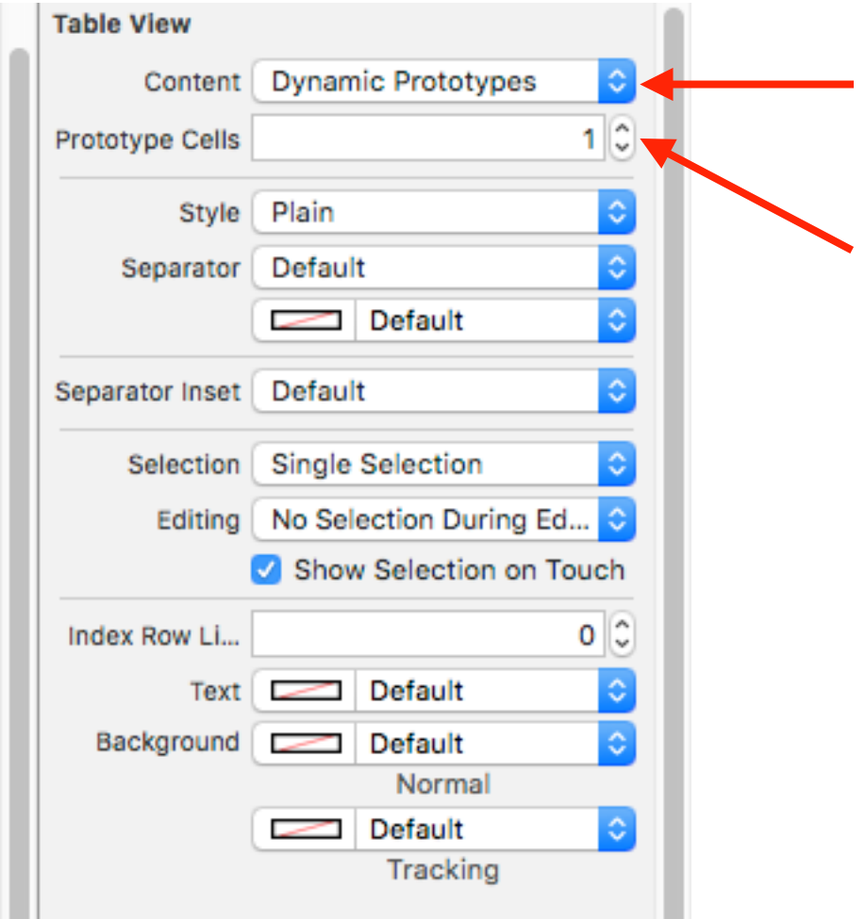
We can use Interface Builder to connect Table Views to other objects that meet these protocols.



SPECIFYING DELEGATES FOR TABLE VIEWS

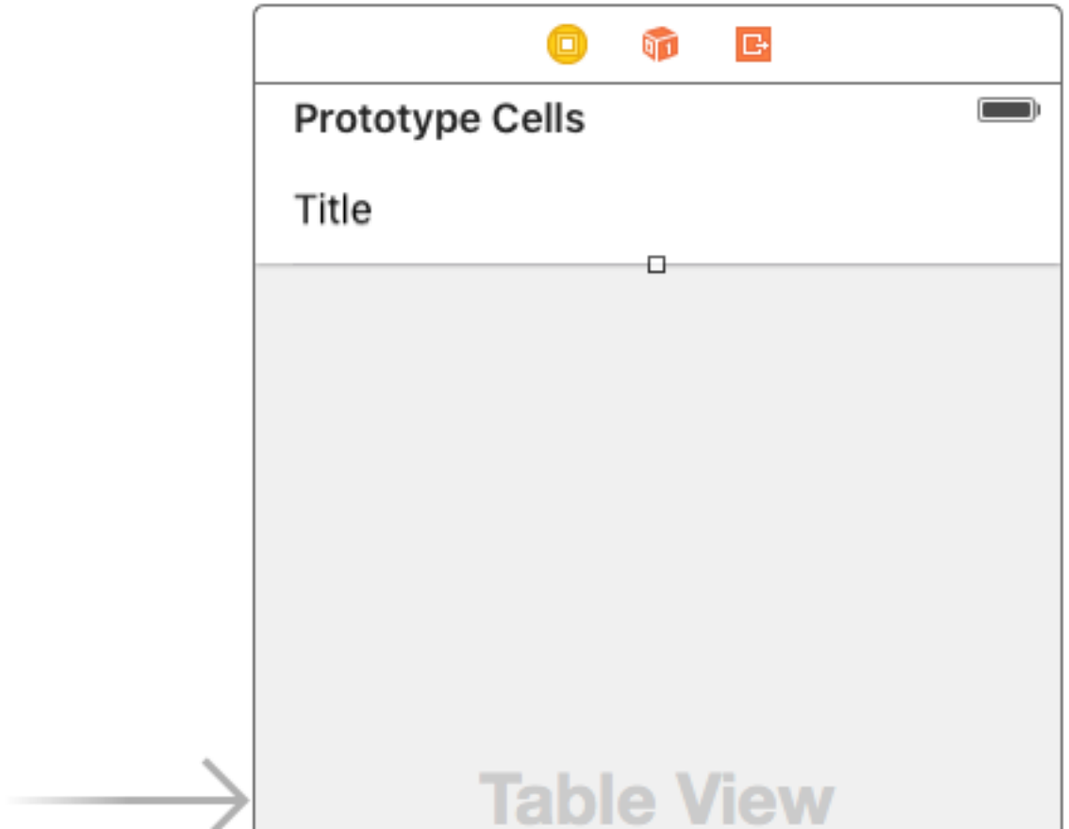
Select the UITableView, select "Dynamic Prototypes" and set Prototype Cells equal to 1.

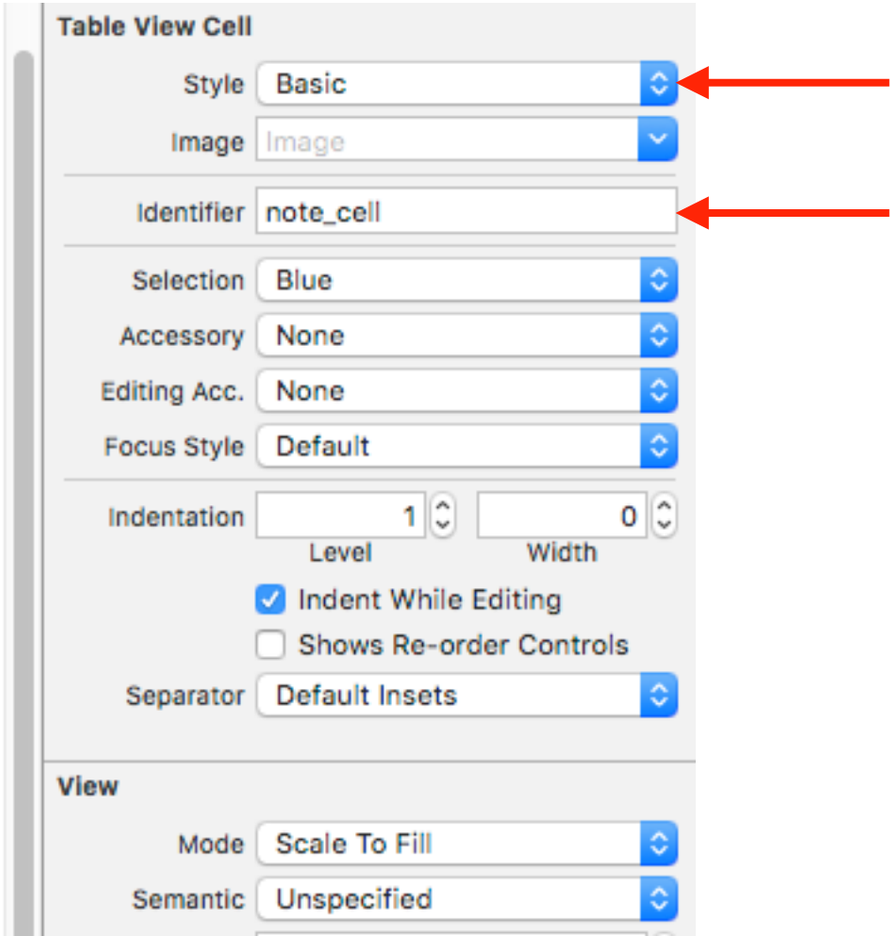




SPECIFYING DELEGATES FOR TABLE VIEWS

Select the cell, then change the style to "Basic" and the identifier to "note_cell".

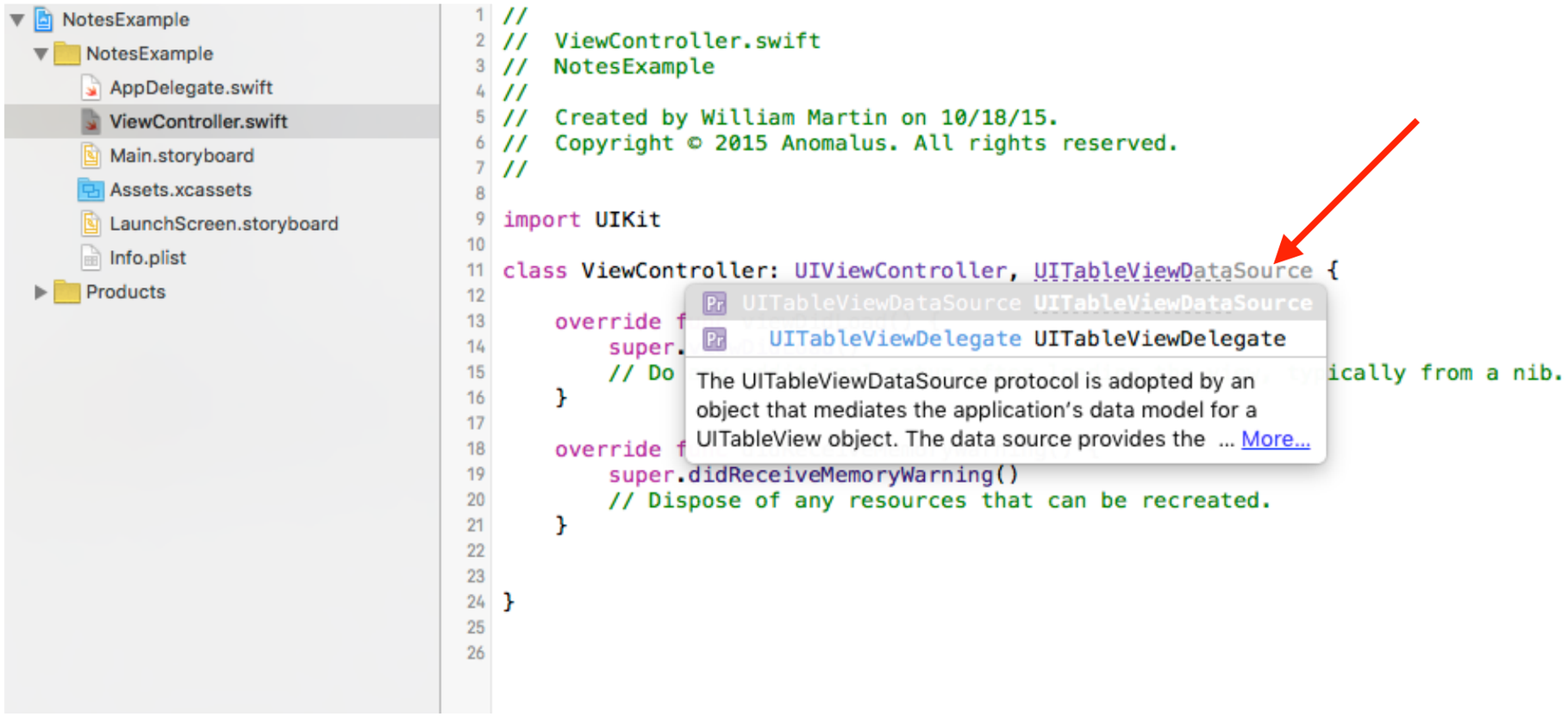




HOOKING UP TABLE VIEWS

One common pattern is to use a Table View's parent View Controller as the delegate and data source.

To start, add UITableViewDataSource to the class definition of ViewController.



HOOKING UP TABLE VIEWS

Add the two required methods, first cellForRowAtIndexPath. Use auto-complete to find it:

```
22
23      tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
24  M   tableView(tableView: UITableView, canEditRowAtIndexPath indexPath: NSIndexPath) -> Bool
25  M   tableView(tableView: UITableView, canMoveRowAtIndexPath indexPath: NSIndexPath) -> Bool
26  M   tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
27  M   tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle, forRo…
28  M   tableView(tableView: UITableView, moveRowAtIndexPath sourceIndexPath: NSIndexPath, toIndexPath destin…
29  M   tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int
30  M   tableView(tableView: UITableView, sectionForSectionIndexTitle title: String, atIndex index: Int) -> I…
31  M   tableView(tableView: UITableView, titleForFooterInSection section: Int) -> String?
32
33  Asks the data source for a cell to insert in a particular location of the table view. More...
34
35
36
37
38
39
```

HOOKING UP TABLE VIEWS

The other is numberOfRowsInSection.

```
22
23      func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
24
25      }
26
27      tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int
28  M   tableView(tableView: UITableView, canEditRowAtIndexPath indexPath: NSIndexPath) -> Bool
29  M   tableView(tableView: UITableView, canMoveRowAtIndexPath indexPath: NSIndexPath) -> Bool
30  M   tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle, forRo…
31  M   tableView(tableView: UITableView, moveRowAtIndexPath sourceIndexPath: NSIndexPath, toIndexPath destin…
32  M   tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int
33  M   tableView(tableView: UITableView, sectionForSectionIndexTitle title: String, atIndex index: Int) -> I…
34  M   tableView(tableView: UITableView, titleForFooterInSection section: Int) -> String?
35  M   tableView(tableView: UITableView, titleForHeaderInSection section: Int) -> String?
36
37  Tells the data source to return the number of rows in a given section of a table view. More...
38
39
40
41
42
43
```

HOOKING UP TABLE VIEWS

Add an Array of Strings that includes some notes to display in the table.

```
22
23      let items = [
24          "General Assembly notes",
25          "Swift 2.0 updates",
26          "Table view stuff"
27      ]
28
29      func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
30          return self.items.count
31      }
32
33      func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
34          let cell = tableView.dequeueReusableCellWithIdentifier("note_cell", forIndexPath: indexPath)
35          cell.textLabel!.text = self.items[indexPath.row]
36          return cell
37      }
38
39  }
40
```

HOOKING UP TABLE VIEWS

Return the count of the Array from numberOfRowsInSection.

```
22
23      let items = [
24          "General Assembly notes",
25          "Swift 2.0 updates",
26          "Table view stuff"
27      ]
28
29      func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
30          return self.items.count
31      }
32
33      func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
34          let cell = tableView.dequeueReusableCellWithIdentifier("note_cell", forIndexPath: indexPath)
35          cell.textLabel!.text = self.items[indexPath.row]
36          return cell
37      }
38
39  }
40
```

HOOKING UP TABLE VIEWS

Get an instance of UITableViewCell using dequeueReusableCellWithIdentifier and providing the String "note_cell" that we entered in Interface Builder.

```
22
23      let items = [
24          "General Assembly notes",
25          "Swift 2.0 updates",
26          "Table view stuff"
27      ]
28
29      func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
30          return self.items.count
31      }
32
33      func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
34          let cell = tableView.dequeueReusableCellWithIdentifier("note_cell", forIndexPath: indexPath)
35          cell.textLabel!.text = self.items[indexPath.row]
36          return cell
37      }
38
39  }
40
```

HOOKING UP TABLE VIEWS

Get the row number from indexPath.row, get the String to display from self.items, then set that String to the text of the cell's textLabel property. (Note the ! for the textLabel property.)

```
22
23      let items = [
24          "General Assembly notes",
25          "Swift 2.0 updates",
26          "Table view stuff"
27      ]
28
29      func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
30          return self.items.count
31      }
32
33      func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
34          let cell = tableView.dequeueReusableCellWithIdentifier("note_cell", forIndexPath: indexPath)
35          cell.textLabel!.text = self.items[indexPath.row]
36          return cell
37      }
38
39  }
40
```

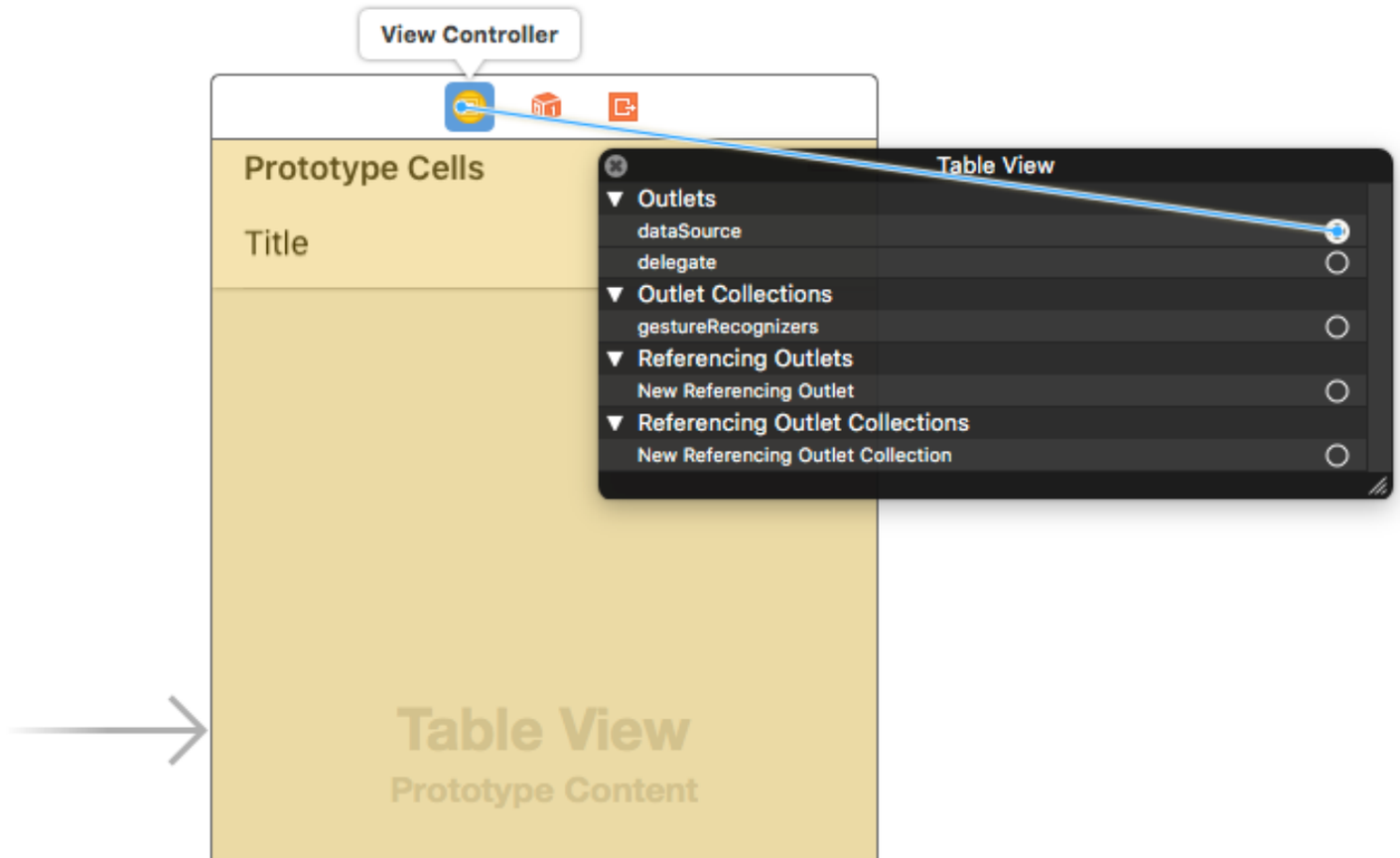
HOOKING UP TABLE VIEWS

Return the UITableViewCell.

```
22
23      let items = [
24          "General Assembly notes",
25          "Swift 2.0 updates",
26          "Table view stuff"
27      ]
28
29      func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
30          return self.items.count
31      }
32
33      func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
34          let cell = tableView.dequeueReusableCellWithIdentifier("note_cell", forIndexPath: indexPath)
35          cell.textLabel!.text = self.items[indexPath.row]
36          return cell
37      }
38
39  }
40
```

HOOKING UP TABLE VIEWS

In InterfaceBuilder, connect the dataSource outlet of the UITableView to the ViewController.



HOOKING UP TABLE VIEWS

Running the app should reveal a Table View showing the three values from the items Array.