In The Class

Documentation to UITableViewCell <https://developer.apple.com/documentation/uikit/uitableviewcell>

Documentation to UITableViewDataSource <https://developer.apple.com/documentation/uikit/uitableviewdatasource>

# Glossary - iOS Intermediate

Here are some terms that you should understand by the end of this section. Remember that Apple has really good documentation. Just search any term and add "apple documentation" and you will probably find something. Try searching [UITableView apple docs](https://www.google.com/search?q=UITableView+apple+docs" \t "_blank).

**UITableView** - A scrollable view that presents data using rows arranged in a single column. Most modern apps are based around dynamic table views (the data changes based on the input data, most often some API). Examples include Instagram (each image is a row in the table), Facebook (each post is a row in the table) etc.

**Prototype Cell** - A template cell for a TableView. One table may have many prototype cells.

**Cell Identifier** - A string name for a given cell, used as a parameter for the DequeueReusableCell method.

**DequeueReusableCell(withIdentifer:)** - This table view method returns a cell that we can use again. Remember to clear out any old information that may be left over from previous uses. To read more take a look at the [documentation](https://developer.apple.com/documentation/uikit/uitableview/1614891-dequeuereusablecell).

**Protocol** - A protocol declares a programmatic interface that any class may choose to implement. Protocols make it possible for two classes distantly related by inheritance to communicate with each other to accomplish a certain goal.

**Delegate** - Delegation is a design pattern that enables a class or structure to hand off (or delegate) some of its responsibilities to an instance of another type. The delegate is the instance that does the work for another instance (the worker)

**UINavigationController** -  A navigation controller is a container view controller that manages one or more child view controllers in a navigation interface.

**Segue** - An object that prepares for and performs the visual transition between two view controllers. This object contains a source and destination view controller, as well as an identifier.

**Show Segue** - A segue that will extend the existing navigation controller to the destination view controller.

**Modal Segue** - A segue that will NOT l extend the existing navigation controller.

**Prepare(for:sender:)** - A method of UIViewContrller that gets called whenever a segue is performed. This allows us to pass data to the segue destination. To read more take a look at the [documentation](https://developer.apple.com/documentation/uikit/uiviewcontroller/1621490-prepare).

**Unwind Segue** - A type of segue that lets you jump back multiple views and pass the information along.

# Table Views

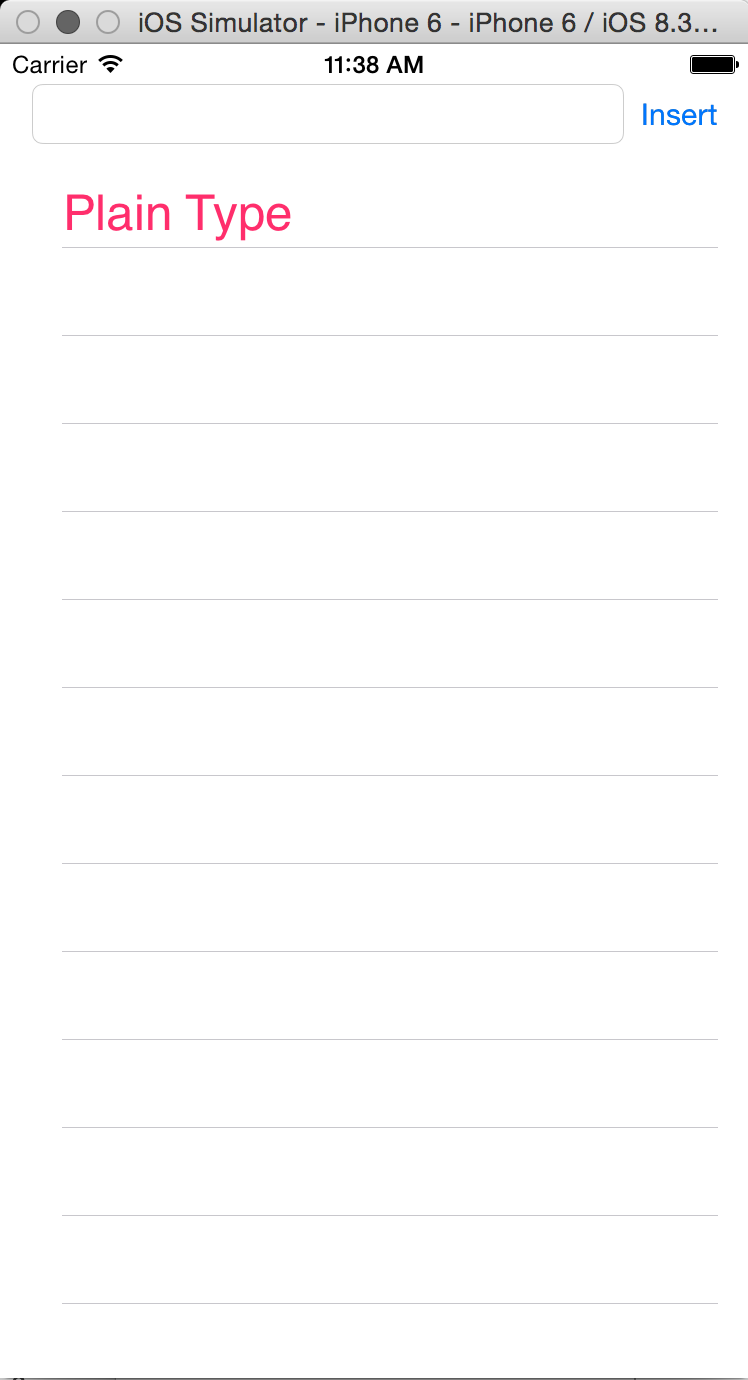
**Table Views are the UI View subclass behind some of the most popular apps on the App Store**. They are very versatile and this is one view we must master. The name 'Table' is a little bit misleading because yes, it is a table, but **a table with just one column**. In fact, it might be more accurate to **think of it as a list.**

## Content

First, we must **determine if our content is going to be dynamic or static**. **If we don't know how many cells we have to display until the application is running, we have to use Dynamic Prototypes to generate our cells**. Since we don't know how many cells to generate beforehand, **we create a prototype cell that will be used as a blueprint to create cells**. On the other hand, **if we know beforehand how many cells that we need, we can create Static Cells.**

## Two types

There are two main types of TableViews. **Plain type is for dynamic data** (when we don't know beforehand how long data is going to be). **Grouped type is for fixed data.** One example of a **Plain type is the Contacts App** and an example of a **Grouped type is the Settings App**.



صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

# Delegates & Protocols

In Swift, we use different controllers - Navigation Controllers, View Controllers, and the like. These controllers are classes, meaning that they (like just about everything in Swift) are using the paradigm of Object Oriented Programming. Our classes inherit properties and methods from other classes, which is very useful. It keeps us from having to rewrite commonly used code chunks and allows us to easily make use of and extend functionality created by others.

At runtime, each class is instantiated into an object, and these objects (including our controllers) are each almost entirely self-contained: they can't really talk to anything outside of themselves. This is useful -- if we want to change one controller, we shouldn't have to change our code hardly anywhere else. That in turn makes our code flexible and extensible. It does produce an interesting challenge, though: How can we pass data from one object to another?

### This is where delegation steps in.

Delegation is the process of setting up a pointer from one object to another, so we can call on specific methods to pass data back and forth between the two.

Table Views in particular require two delegates to be implemented in order to function. This is because a table view is designed to display data, and if we don't set up any place for it to get its data from, or tell it how to display that data, it won't be able to do its job.

These requirements particular classes have are known as ***protocols***. A protocol is just a set of rules our delegates must ***conform*** to in order to work with certain types of objects.

### Example protocol:

protocol UITableViewDataSource {

func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int

func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell

}

Pretty simple!  Declare what the function should be, and now any class that follows this protocol must include these functions.

### Let's put this in more concrete terms:

In order to embed a Table View into a view controller, our view controller must conform to the Table View's protocol, aka follow its requirements. In the case of table views, these requirements make sense because a table view won't do its job unless we both give it data to display and provide a way to customize how that data is displayed and behaves.

* **Object A (in this case, our ViewController) must conform to a protocol** that describes what methods it has that are relevant to Object B (the "delegate" protocol, in this case our UITableView)
* **Object B must have a reference** to a type that conforms to the protocol (this will be a reference to Object A)
* **Object A must implement the protocol methods**
* **Object B will call upon Object A's protocol methods** through its reference

### Implementing the Protocol:

Here's an example of how we set up the  delegates for a UITableView, adding them as extensions to a ViewController class:

extension ViewController: UITableViewDataSource {

func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

//logic

}

func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

//logic

}

}

extension ViewController: UITableViewDelegate {

}

Once we have these methods in place, all we need to do to update the data in our TableView is to modify the actual repository of data we're using to populate our cells, and then run the reloadData() method on the array or object that contains that data

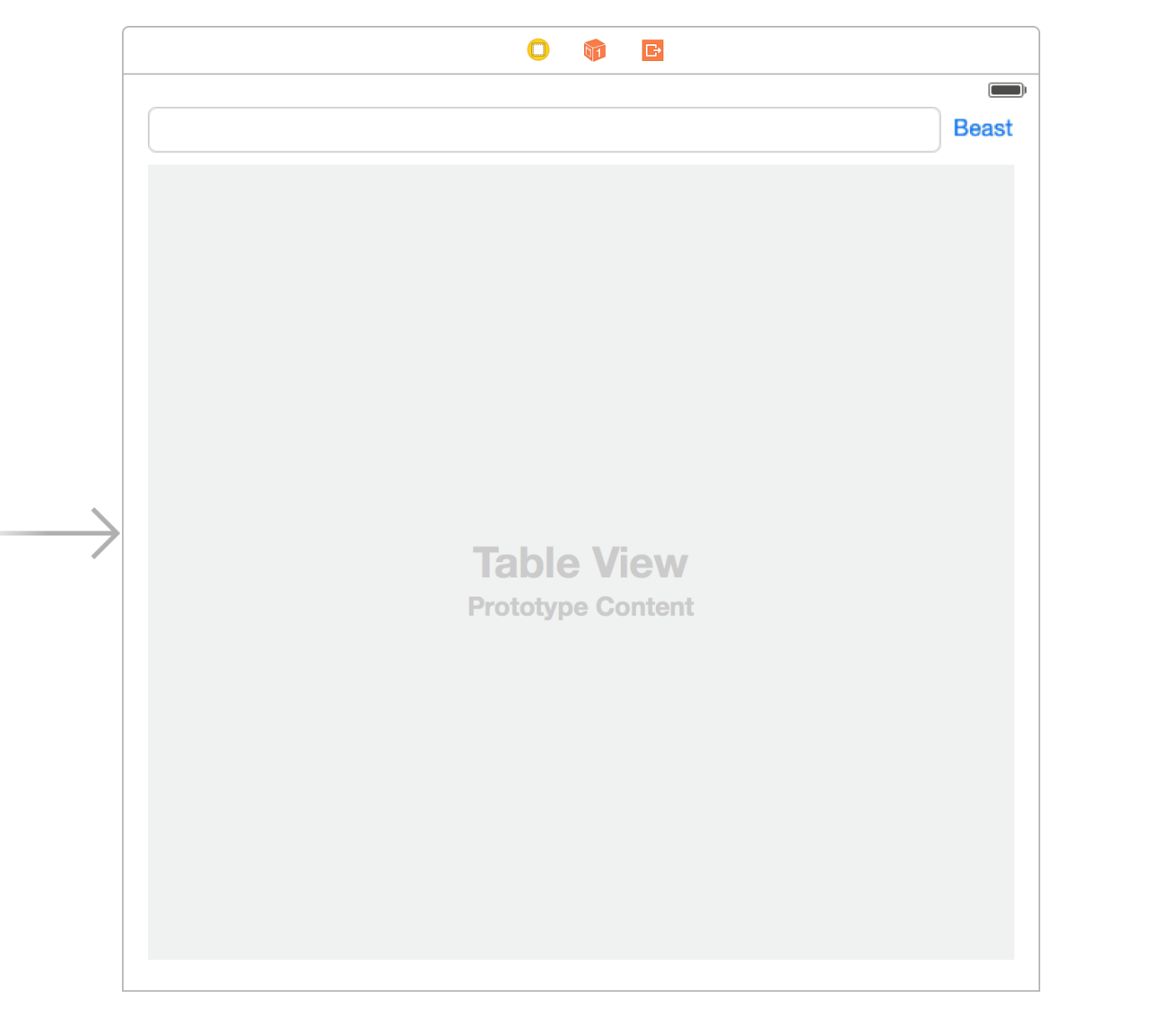
# Getting Started with Dynamic Cells

To get our feet wet with Table Views let's start by building a project using Dynamic Cells

We will be creating Beast List, a simple app that will help us stay focused and focus on a small collection of tasks that need to get done right away. **We will be using the Table View to display our tasks**.

## Step 1: Build our UI

Let's first drag and drop all the objects that we need. **It is crucial that we are using the blue guides when we lay out our UI because it helps XCode figure out (and us as well) who the nearest neighbor is**. When we pin something to the top, leading, trailing, or the bottom, **we are creating that constraint between the selected View and its closest neighbor**. We need a Text Field, Button, and Table View on our Storyboard. **Make sure we are using the Table View**. Our layout should look like the following picture:



Pin the Button to the top right corner. Also, create a horizontal constraint to the left with its closest neighbor (UITextField).

Now let's create a constraint between the UITextField and the UIButton. We want to say that their Center-Y should match. Then let's pin the UITextField to the super view on the left. We will get a warning saying that we need to set Content Hugging Priority.  **This priority determines how strongly should Auto Layout attempt to keep an object from expanding**. We can raise Content Hugging Priority of the UIButton to be higher than the UITextField so that in a situation where Auto Layout has to determine how to fill the entire horizontal place, **it will expand the text field to fill up the space and leave the button alone to be just as big as the content that it holds.**

Let's pin the Table View with a distance of 0 to its nearest neighbor (super view) while constraining to margins. Then let's pin it to the top and bottom with Standard Value away from its closest neighbor.

## Step 2: Make the Connections

**We are going to make two Outlets and one Action**.**We need to have an Outlet to our Text Field so that we can read what the user inputs into that text field**. **We also need an Outlet to the UITableView so that we assign our View Controller as its dataSource and delegate. We need an Action so that we can run the code once a user presses the Insert button.** Go ahead and write the following code and connect it to the appropriate UI.**Make sure you are selecting the right UI element.**

@IBOutlet weak var taskTextField: UITextField!

@IBOutlet weak var tableView: UITableView!

@IBAction func beastButtonPressed(\_ sender: UIButton) {

}

After making the connections let's implement the UITableViewDataSource protocol. First we have to **declare that our View Controller conforms to the UITableViewDataSource protocol. We place this code outside the ViewController class.**

extension ViewController: UITableViewDataSource {

}

**Then we need to set our UITableView's dataSource to self**when the viewDidLoad message gets sent to our View Controller

override func viewDidLoad() {

super.viewDidLoad()

tableView.dataSource = self

}

### Wait what's a Protocol?

In the above code we are "conforming" to the UITableViewDataSource Protocol. You should notice that conforming to a Protocol seems very similar to inheriting from a subclass. Although they are similar there is one key difference: **A Protocol defines the architecture of a particular type but not the implementation while a Parent Class defines both the architecture and a potential implementation that can be extended.**

In our case by conforming to the UITableViewDataSource Protocol we are promising to implement a couple of key methods to help provide the data for the TableView.

We will learn more about Protocols in the coming chapter.

**We need to implement UITableViewDataSource protocol whenever the data in the table is dynamic**. For example, our data might come from a database of records that change over time. There are two important methods in this protocol. The delegate has to answer these two questions: **How many cells are we going to need?,**and **How should I create each cell?**

Moreover in order to answer these two questions we need to implement 2 key methods:

##### How many cells are we going to need?

// MAKE SURE THESE ARE WITHIN UITableViewDataSource EXTENSION!

// How many cells are we going to need?

func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

// return an integer that indicates how many rows (cells) to draw

}

##### What cell should I draw in each section?

// How should I create each cell?

func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

// Get the UITableViewCell and create/populate it with data then return it

}

Now let's get back to the project and implement these methods.

### Back to Business

We are just going to have an Array of Strings for now. Go ahead and add the following property to our View Controller to hold our Model objects.

// Create this array at the top of your class as a property

var tasks = ["Exercise for 30 minutes", "Wireframe for some project", "Do laundry"]

Now that we have data to display we can implement the two required methods as follows:

// how many cells are we going to need?

func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

return tasks.count

}

// how should I create each cell?

func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

let cell = tableView.dequeueReusableCell(withIdentifier: "MyCell", for: indexPath)

// set text label to the model that is corresponding to the row in array

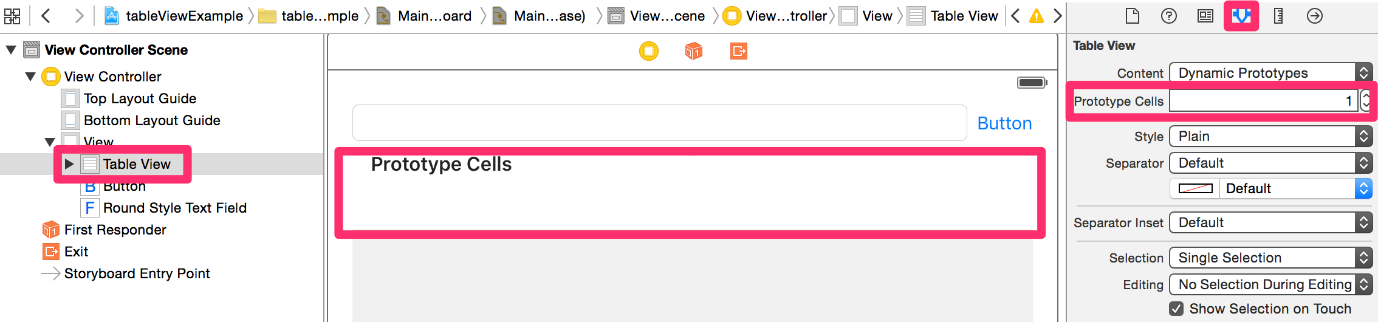
cell.textLabel?.text = tasks[indexPath.row]

// return cell so that Table View knows what to render in each row

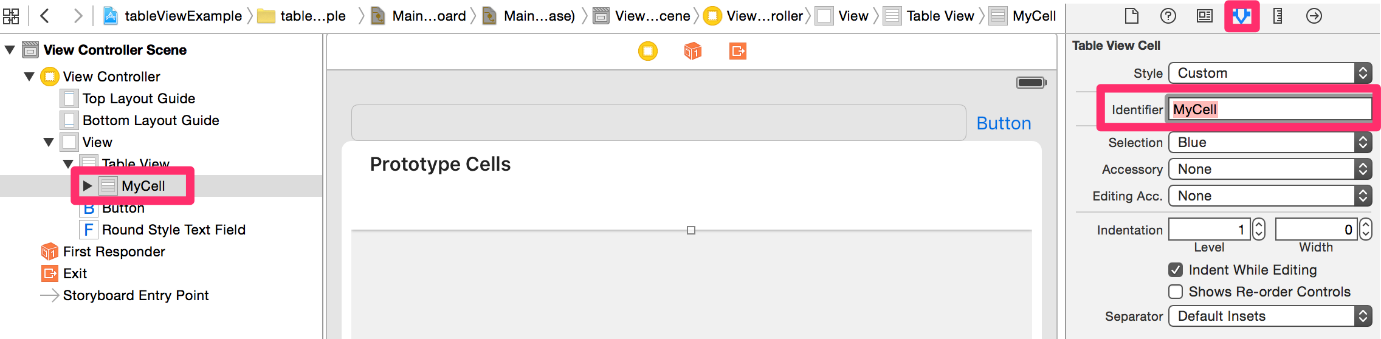
return cell

}

In our implementation of the UITableViewDataSource protocol, we are dequeuing a prototype cell with the name of MyCell. We haven't created this prototype cell yet. We can think of a prototype cell as a blueprint for all of the cells that we will be displaying in our Table View. First we have to specify that our Table View is going to display dynamic data in prototype cells.



Then, we can give a name for our prototype cell so that we can reference it in our implementation of the UITableViewDataSource protocol.



## Now run the application and see the data populate!

We'll explore how to interact with the list soon.