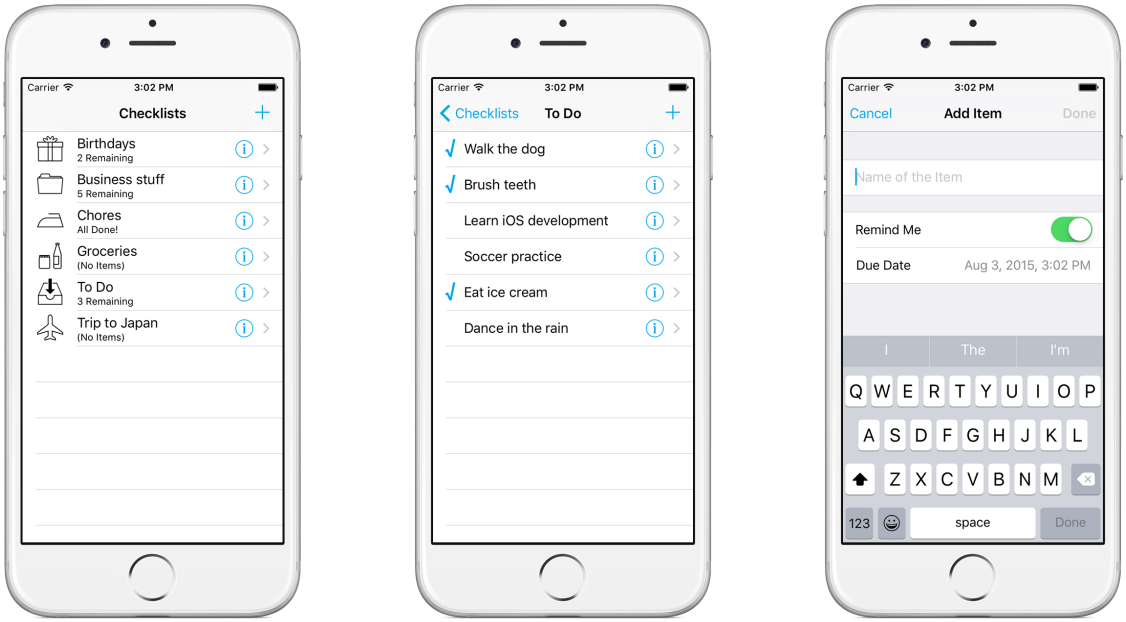
Chapter 9: Table Views

Ready to get started on your next app? Let's go!

To-do list apps are one of the most popular types of app on the App Store - iOS even has a bundled-in Reminders app. Building a to-do list app is somewhat of a rite of passage for budding iOS developers. So, it makes sense that you create one as well.

Your own to-do list app, ***Checklists***, will look like this when you're finished:



The finished Checklists app

The app lets you organize to-do items into lists and then check off these items once you've completed them. You can also set a reminder on a to-do item that will make the iPhone pop up an alert on the due date, even when the app isn't running.

As far as to-do list apps go, *Checklists* is very basic, but don't let that fool you. Even a simple app such as this already has five different screens and a lot of complexity behind the scenes.

This chapter covers the following:

- **Table views and navigation controllers:** A basic introduction to navigation controllers and table views.
- **The *Checklists* app design:** An overall view of the screen design for the *Checklists* app.
- **Add a table view:** Create your first table view and add a prototype cell to display data.
- **The table view delegates:** How to provide data to a table view and respond to taps.

Table views and navigation controllers

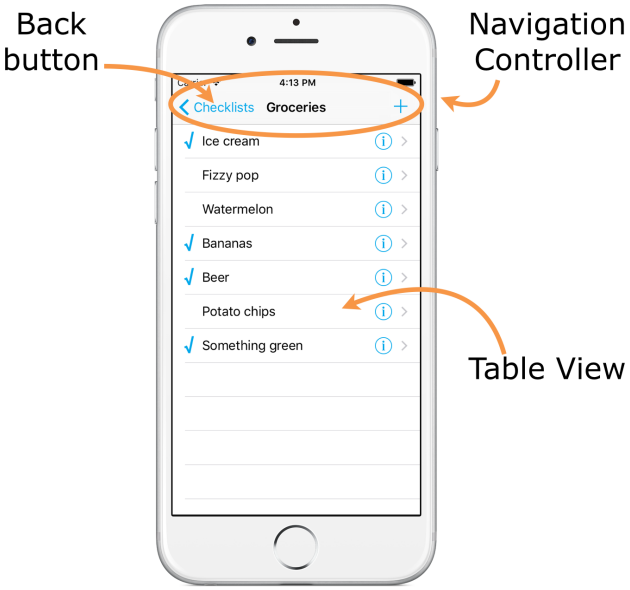
Checklists will introduce you to two of the most commonly used UI (user interface) elements in iOS apps: the table view and the navigation controller.

A **table view** shows a list of things. The three screens above all use a table view. In fact, all of this app's screens use table views. This component is extremely versatile and the most important one to master in iOS development.

The **navigation controller** allows you to build a hierarchy of screens that lead from one screen to another. It adds a navigation bar at the top with a title and a back button.

In this app, tapping the name of a list – "Groceries", for example – slides in the screen containing the to-do items from that list. The button in the upper-left corner takes you back to the previous screen with a smooth animation. Moving between those screens is the job of the navigation controller.

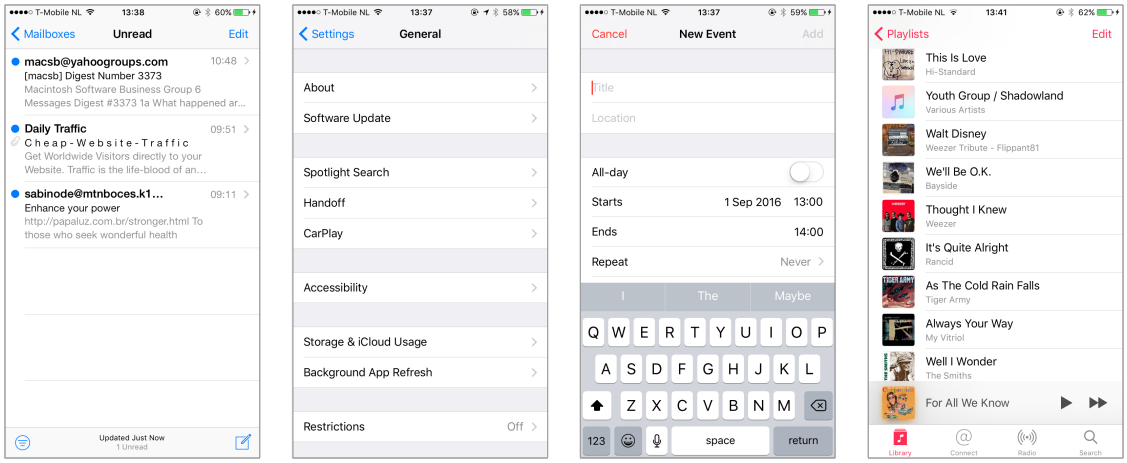
Navigation controllers and table views are often used together.



The grey bar at the top is the navigation bar. The list of items is the table view.

Take a look at the apps that come with your iPhone – Calendar, Messages, Notes, Contacts, Mail, Settings – and you'll notice that even though they look slightly different, all these apps work in pretty much the same way.

That's because they all use table views and navigation controllers:



These are all table views inside navigation controllers

(The Music app also has a *tab bar* at the bottom, something you'll learn about later on.)

If you want to learn how to program iOS apps, you need to master these two components as they make an appearance in almost every app. That's exactly what you'll focus on in this section of the book. You'll also learn how to pass data from one screen to another, a very important topic that often puzzles beginners.

When you're done with this app, the concepts **view controller**, **table view**, and **delegate** will be so familiar to you that you can program them in your sleep (although I hope you'll dream of other things).

This is a very long read with a lot of source code, so take your time to let it all sink in. I encourage you to experiment with the code that you'll be writing. Change stuff and see what it does, even if it breaks the app.

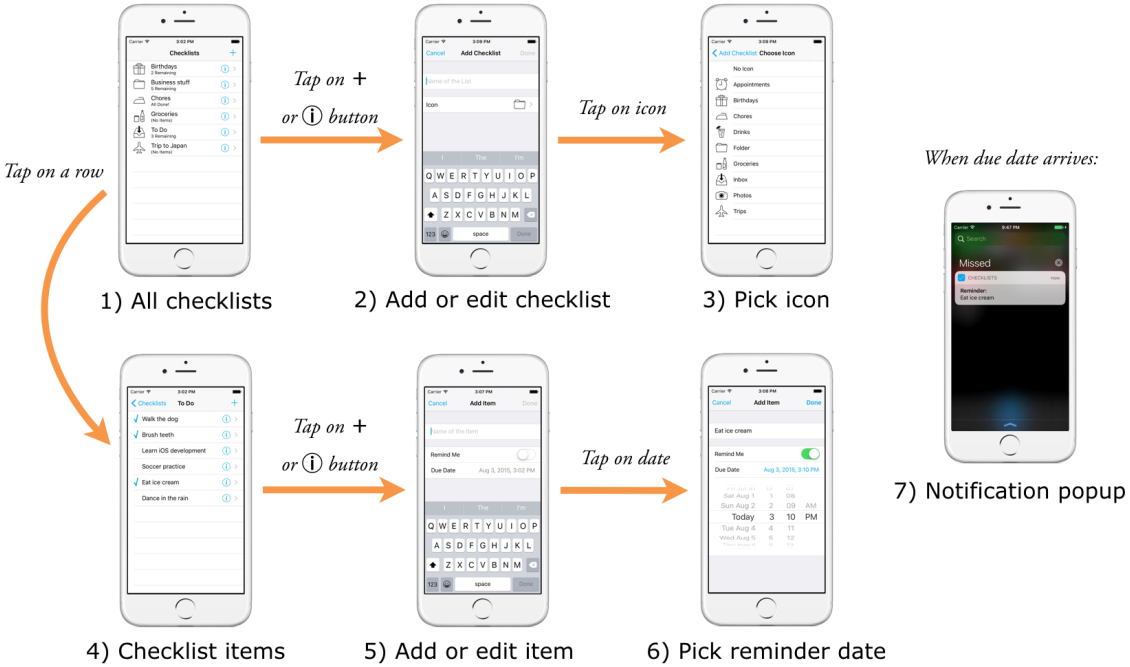
Making mistakes that result in bugs, tearing your hair out in frustration, the light bulb moment when you realize what's wrong, the satisfaction of fixing the bug – they're all essential parts of the developer learning process :]

There's no doubt: playing with code is the quickest way to learn!

By the way, if something is unclear to you – for example, you may wonder why method names in Swift look so funny – then don't panic! Have some faith and keep going... everything will be explained in due course.

The Checklists app design

Just so you know what you're in for, here is an overview of how the *Checklists* app will work:



All the screens of the Checklists app

The main screen of the app shows all your "checklists" (1). You can create multiple lists to organize your to-do items.

A checklist has a name, an icon, and zero or more to-do items. You can edit the name and icon of a checklist in the Add/Edit Checklist screen (2) and (3).

You tap on the checklist's name to view its to-do items (4).

A to-do item has a description, a checkmark to indicate that the item is done, and an optional due date. You can edit the item in the Add/Edit Item screen (5).

iOS will automatically notify the user of checklist items that have their "remind me" option set (6), even if the app isn't running (7). That's a pretty advanced feature, but I think you'll be up for the task.

You can find the full source code of this app in the Source Code folder, so have a play with it to get a feel for how it works.

Done playing? Then let's get started!

> **Important:** The *iOS Apprentice* projects are for **Xcode 9.0** and better only. If you're still using an older version of Xcode, please update to the latest version of Xcode from the Mac App Store.
>
> But don't get carried away either – often Apple makes beta versions available of upcoming Xcode releases. Please do *not* use an Xcode beta to follow along. Often, the beta versions break things in unexpected ways and you'll only end up confused. Stick to the official versions for now!

Add a table view

Seeing as table views are so important, you will start out by examining how they work. Making lists has never been this much fun!

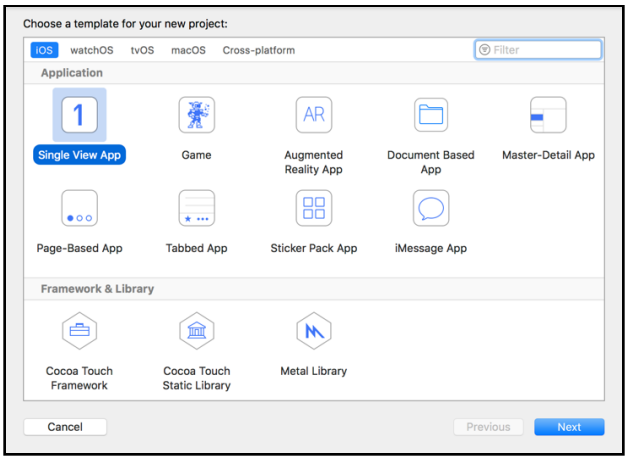
Because smart developers split up the workload into small, simple steps, this is what you're going to do in this chapter:

1. Put a table view on the app's screen.
2. Put data into that table view.
3. Allow the user to tap a row in the table to toggle a checkmark on and off.

Once you have these basics up and running, you'll keep adding new functionality over the next few chapters until you end up with a full-blown app.

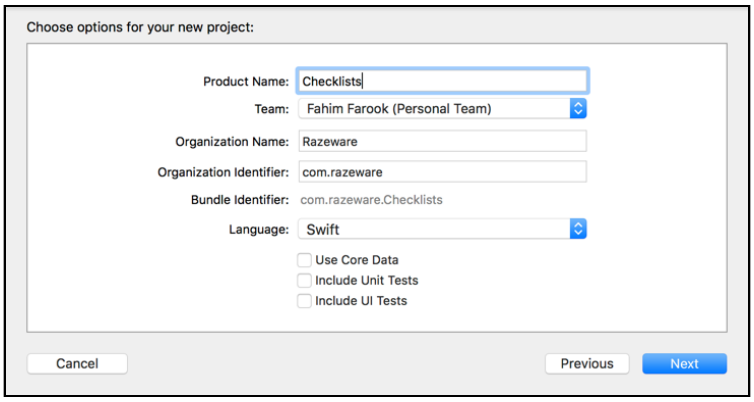
Create the project

➤ Launch Xcode and start a new project. Choose the **Single View Application** template:



Choosing the Xcode template

Xcode will ask you to fill out a few options:



Choosing the template options

➤ Fill out these options as follows:

- Product Name: **Checklists**
- Team: Since you already set up your developer account for the previous app (you did, didn't you?) you can select your team here - or, you can just leave this at the default setting.
- Organization Name: Your name or the name of your company

- Organization Identifier: Use your own identifier here, using reverse domain name notation
- Language: **Swift**
- Use Core Data, Include Unit Tests, Include UI Tests: these should be off.

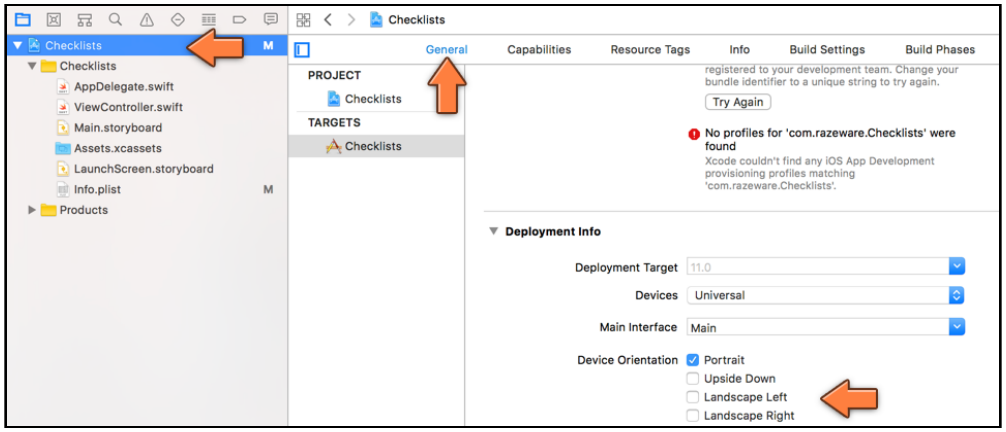
➤ Press **Next** and choose a location for the project.

You can run the app if you want, but as you might remember from the *Bull's Eye* app, at this point it is just a white screen.

Set the app orientation

Checklists will run in portrait orientation only. However, the default project that Xcode just generated also includes landscape support.

➤ Click on the Checklists project item at the top of the project navigator and go to the **General** tab. Under **Deployment Info**, **Device Orientation**, make sure that only **Portrait** is selected.



The Device Orientation setting

With the landscape options disabled, rotating the device will no longer have any effect. The app always stays in portrait orientation.

> **Upside down**
>
> There is also an Upside Down orientation but you typically won't use it.
>
> If your app supports Upside Down, users are able to rotate their iPhone so that the home button is at the top of the screen instead of at the bottom.
>
> That may be confusing, especially when the user receives a phone call: the microphone is at the wrong end with the phone upside down.

> iPad apps, on the other hand, are supposed to support all four orientations including upside-down.

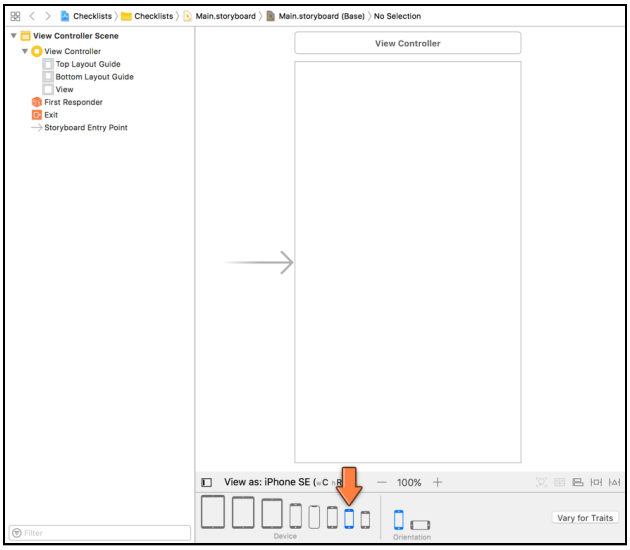
Edit the storyboard

Xcode created a basic app that consists of a single view controller. Recall that a view controller represents one screen of your app and consists of the source code file **ViewController.swift** and a user interface design in **Main.storyboard**.

The storyboard contains the designs of all your app's view controllers inside a single document, with arrows showing the flow between them. In storyboard terminology, each view controller is named a *scene*.

You already used a storyboard in *Bull's Eye* but in this app you will unlock the full power of storyboarding.

➤ Click on **Main.storyboard** to open Interface Builder.

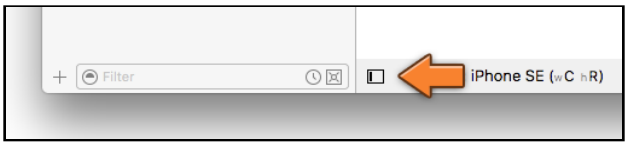


The storyboard editor with the app's only scene

By default, the scene will have the dimensions of a 5.5" iPhone. I used the **View as:** panel at the bottom to switch to the slightly smaller **iPhone SE** because that takes up less room in the book. However, it does not matter which device size you choose to edit the storyboard: the app will automatically resize to fit all iPhone models.

➤ Select **View Controller** in the Document Outline on the left.

Tip: Recall that the Document Outline shows the view hierarchy of all the scenes in the storyboard. If you cannot see the Document Outline, then click the small square button at the bottom of the Interface Builder window to toggle its visibility.



This button shows and hides the Document Outline

➤ Press **delete** on your keyboard to remove the **View Controller Scene** from the storyboard. The canvas should be empty and the Document Outline say "No Scenes".

You do this because you don't want a regular view controller but a **table view controller**. This is a special type of view controller that makes working with table views a little easier.

The view controller code

But remember, the scene on the storyboard is just half the equation - there's also the Swift code file. And the type specified in code has to match the scene's type. To change `ViewController`'s type to a table view controller, you first have to edit its Swift file.

➤ Click on **ViewController.swift** to open it in the source code editor. Change the following line from this:

```
class ViewController: UIViewController {
```

To this:

```
class ChecklistViewController: UITableViewController {
```

With this change you tell the Swift compiler that your own view controller is now a `UITableViewController` object instead of a regular `UIViewController`.

Remember that everything starting with "UI" is part of UIKit. These pre-fabricated components serve as the building blocks for your own app.

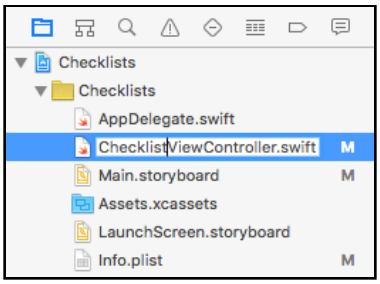
When Xcode made the project, it assumed you wanted the `ViewController` object to be built on top of a basic `UIViewController`, but here you're changing it to use the `UITableViewController` building block instead.

You also renamed `ViewController` to `ChecklistViewController` to give it a more descriptive name. This is your own object – you can tell because its name *doesn't* start with UI.

Over the course of this app, you will add data and functionality to the `ChecklistViewController` object to make the app actually do things. You'll also add several new view controllers to the app.

➤ In the Project navigator on the left, click once to select **ViewController.swift**, and then click again to edit its name. (Don't double-click too fast or you'll open the Swift file inside a new source code editor window.)

Change the filename to **ChecklistViewController.swift**:

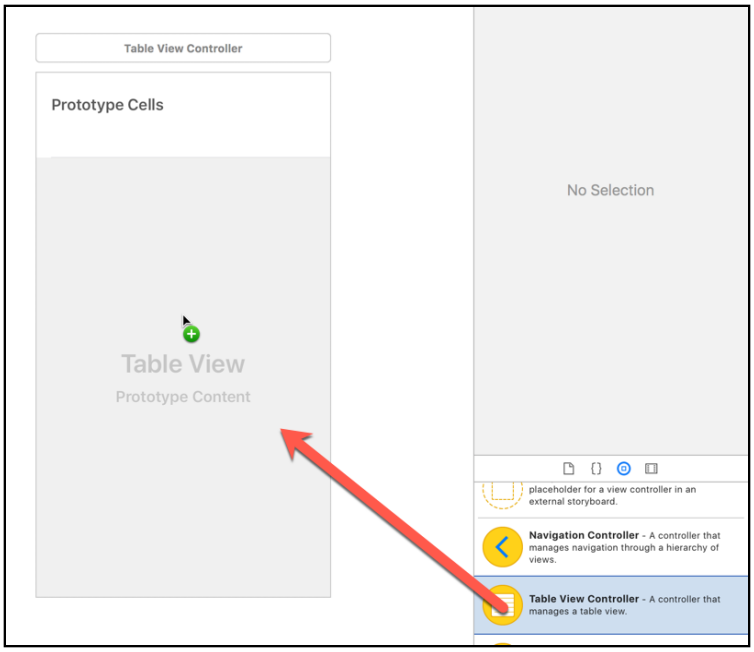


Renaming the Swift file

You might get a warning: "The document could not be saved. The file has been changed by another application." Click **Save Anyway** to make it go away.

Set the view controller class in the storyboard

➤ Go back to the storyboard and drag a **Table View Controller** from the Object Library (bottom-right corner) on to the canvas:

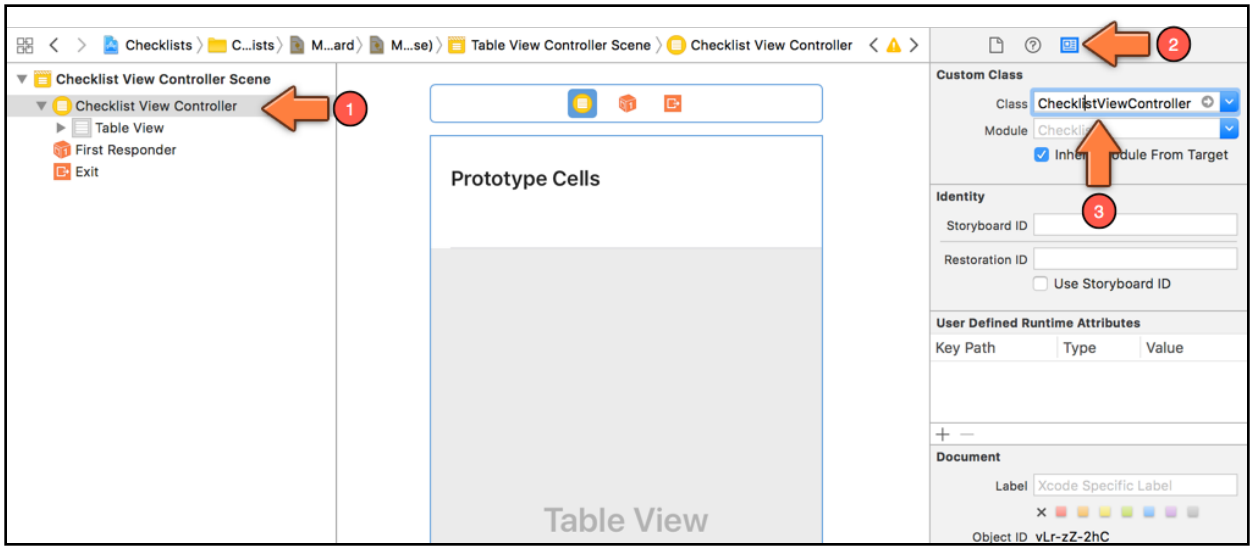


Dragging a Table View Controller into the storyboard

This adds a new Table View Controller scene to the storyboard.

➤ Go to the **Identity inspector** (the third tab in the inspectors pane on the right of the Xcode window) and under **Custom Class** type **ChecklistViewController** (or choose it using the dropdown list).

Tip: When you do this, make sure the actual Table View Controller is selected, not the Table View inside it. There should be a thin blue border around the scene.



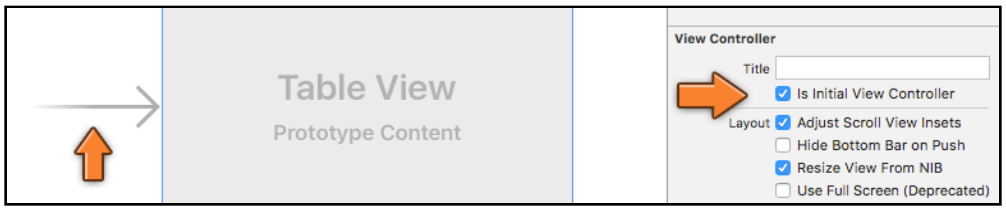
Changing the Custom Class of the Table View Controller

The name of the scene in the Document Outline on the left should change to "Checklist View Controller Scene". You have successfully changed `ChecklistViewController` from a regular view controller object into a table view controller.

As its name implies, and as you can see in the storyboard, the view controller contains a Table View object. We'll go into the difference between controllers and views soon, but for now, remember that the controller is the whole screen while the table view is the object that actually draws the list.

Set the initial view controller

If there is no big arrow pointing towards your new table view controller, then go to the **Attributes inspector** and check **Is Initial View Controller**.

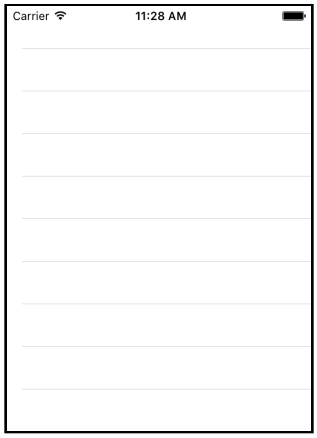


The arrow points at the initial view controller

The initial view controller is the first screen that your users will see. Without one of these, iOS won't know which view controller to load from your storyboard when the app starts up and you'll end up staring at a black screen.

➤ Run the app on the Simulator.

You should see an empty list. This is the table view. You can drag the list up and down but it doesn't contain any data yet.



The app now uses a table view controller

By the way, it doesn't really matter which Simulator you use. Table views resize themselves to the dimensions of the device, and the app will work equally well on the small iPhone SE or the huge iPhone X.

Personally, I use the iPhone SE Simulator because it's compact, but remember that you can open any of the simulators and then simply resize the simulator window by dragging on the corners, just like you resize any macOS window.

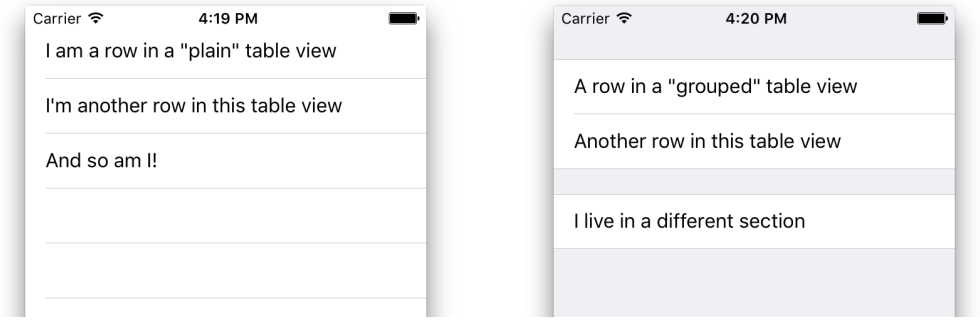
> **Note:** When you build the app, Xcode gives the warning "Prototype table cells must have reuse identifiers". Don't worry about this for now, we'll fix it soon.

The anatomy of a table view

First, let's talk a bit more about table views. A `UITableView` object displays a list of items.

> **Note:** I'm not sure why it's named a *table*, because a table is commonly thought of as a spreadsheet-type object that has multiple rows and multiple columns, whereas the `UITableView` only has rows. It's more of a list than a table, but I guess we're stuck with the name now. UIKit also provides a `UICollectionView` object that works similar to a `UITableView` but allows for multiple columns.

There are two styles of tables: "plain" and "grouped". They work mostly the same, but there are a few small differences. The most visible difference is that rows in the grouped style table are placed into boxes (the groups) on a light gray background.



A plain-style table (left) and a grouped table (right)

The plain style is used for rows that all represent something similar, such as contacts in an address book where each row contains the name of one person.

The grouped style is used when the items in the list can be organized by a particular attribute, like book categories for a list of books. The grouped style table could also be used to show related information which doesn't necessarily have to stand together - like the address information, contact information, and e-mail information for a contact.

You will use both table styles in the *Checklists* app.

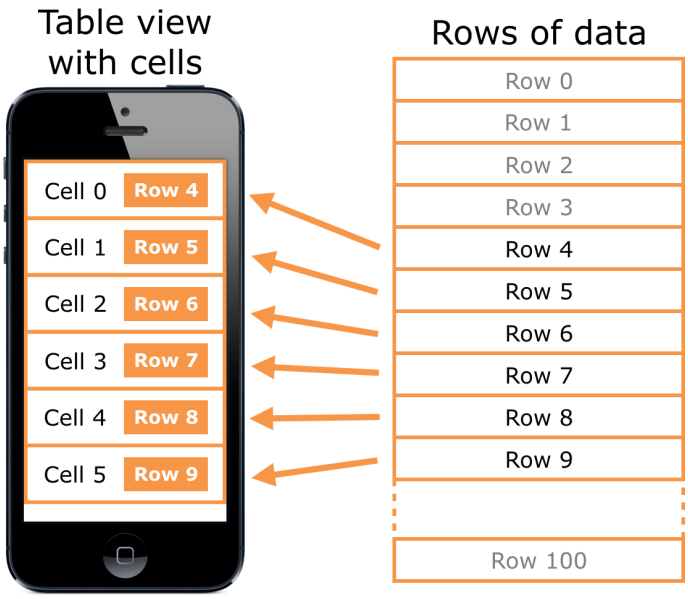
The data for a table comes in the form of **rows**. In the first version of *Checklists*, each row will correspond to a to-do item that you can check off when you're done with it.

You can potentially have many rows (even tens of thousands) but that kind of design isn't recommended. Most users will find it incredibly annoying to scroll through ten thousand rows to find the one they want. And who can blame them?

Tables display their data in **cells**. A cell is related to a row but it's not exactly the same.

A cell is a view that shows a row of data that happens to be visible at that moment. If your table can show 10 rows at a time on the screen, then it only has 10 cells, even though there may be hundreds of rows of actual data.

Whenever a row scrolls off the screen and becomes invisible, its cell will be re-used for a new row that becomes visible.

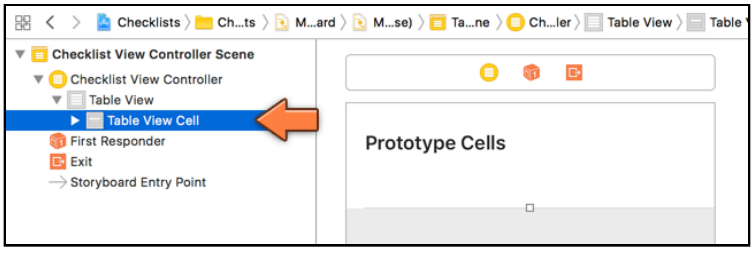


Cells display the contents of rows

Add a prototype cell

In the past, you had to put in quite a bit of effort to create cells for your tables. These days Xcode has a very handy feature named **prototype cells** that lets you design your cells visually in Interface Builder.

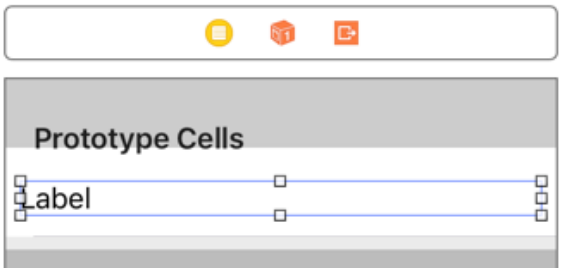
➤ Open the storyboard and click the empty cell (the white row below the Prototype Cells label) to select it.



Selecting the prototype cell

Sometimes it can be hard to see exactly what is selected, so keep an eye on the Document Outline to make sure you've picked the right thing.

➤ Drag a **Label** from the Object Library on to the white area in the table view representing the cell. Make sure the label spans the entire width of the cell (but leave a small margin on the sides).

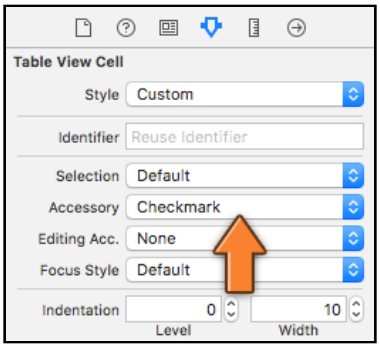


Adding the label to the prototype cell

> **Note:** If you simply drag the label on to the table view, it might not work. You need to drag the label on to the cell itself. You can check where the label ended up on the Document Outline. It has to be inside the Content View for the table view cell.

Besides the label you will also add a checkmark to the cell's design. The checkmark is provided by something called the **accessory**, a built-in view that appears on the right side of the cell. You can choose from a few standard accessory controls or provide your own.

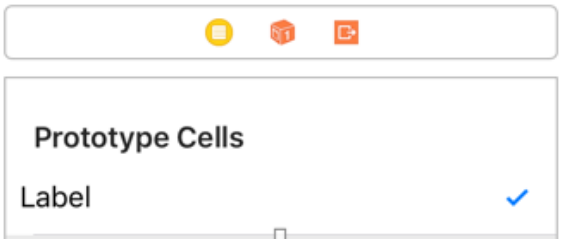
➤ Select the **Table View Cell** again. In the **Attributes inspector**, set the **Accessory** field to **Checkmark**:



Changing the accessory to get a checkmark

(If you don't see this option, then make sure you selected the Table View Cell, not the Content View or Label below it.)

Your design should now look something like this:



The design of the prototype cell: a label and a checkmark

Note: You may want to resize the label a bit so that it doesn't overlap the checkmark.

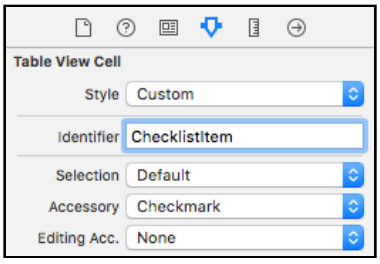
You also need to set a **reuse identifier** on the cell. This is an internal name that the table view uses to find free cells to reuse when rows scroll off the screen and new rows must become visible.

The table needs to assign cells for those new rows, and recycling existing cells is more efficient than creating new cells. This technique is what makes table views scroll smoothly.

Reuse identifiers are also important for when you want to display different types of cells in the same table. For example, one type of cell could have an image and a label and another could have a label and a button. You would give each cell type its own identifier, so the table view can assign the right cell for a given row type.

Checklists has only one type of cell but you still need to give it an identifier.

➤ Type **ChecklistItem** into the Table View Cell's **Identifier** field (you can find this in the **Attributes inspector**).



Giving the table view cell a reuse identifier

➤ Run the app and you'll see... zip, zilch, nada - exactly the same as before :] The table is still empty.

This is because you only added a cell design to the table, not actual data. Remember that the cell is just the visual representation of the row, not the actual data. To add data to the table, you have to write some code.

The table view delegates

➤ Switch to **ChecklistViewController.swift** and add the following methods just before the closing bracket at the bottom of the file:

```
override func tableView(_ tableView: UITableView,
      numberOfRowsInSection section: Int) -> Int {
  return 1
}

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

  let cell = tableView.dequeueReusableCell(
                        withIdentifier: "ChecklistItem",
                                  for: indexPath)
  return cell
}
```

These methods look a bit more complicated than the ones you've seen in *Bull's Eye*, but that's because each takes two parameters and returns a value to the caller. Other than that, they work the same way as the methods you've dealt with before.

Protocols

The above two methods are part of `UITableView`'s **data source** protocol.

What's a protocol, you ask? Well, its a standard set of methods that a class must adhere to - a protocol to be followed, so to speak. It allows code to be written in such a way that you know that a given class would implement certain methods (with specific parameters of a given type) but where you don't need to know all the implementation details of the class - such as all it's methods. A protocol usually allows you to add functionality for a certain type of operation to a class - for example, handling data for a table view.

The data source is the link between your data and the table view. Usually, the view controller plays the role of data source and implements the necessary methods. So, essentially, the view controller is acting as a delegate on behalf of the table view. (This is the delegate pattern that we've talked about before - where an object does some work on behalf of another object.)

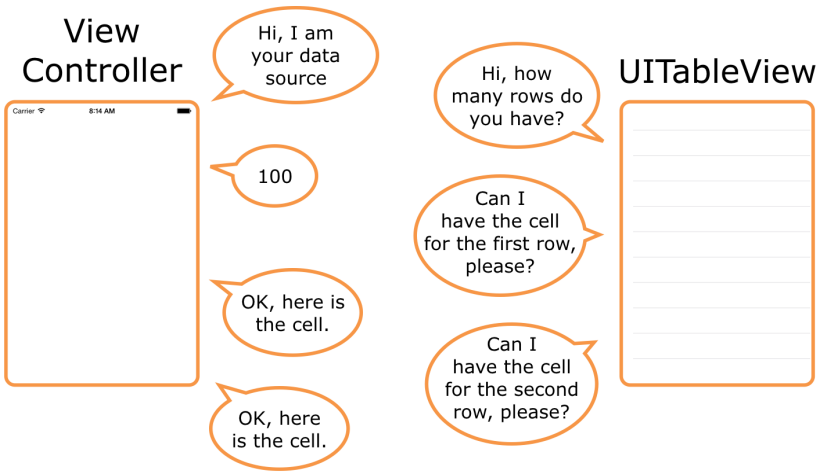
The table view needs to know how many rows of data it has and how it should display each of those rows. But you can't simply dump that data into the table view's lap and be done with it. You don't say: "Dear table view, here are my 100 rows, now go show them on the screen."

Instead, you say to the table view: "This view controller is now your data source. You can ask it questions about the data anytime you feel like it."

Once it is hooked up to a data source – i.e. your view controller – the table view sends a `numberOfRowsInSection` message to find out how many data rows there are.

And when the table view needs to draw a particular row on the screen it sends a `cellForRowAt` message to ask the data source for a cell.

You see this pattern all the time in iOS: one object does something on behalf of another object. In this case, the `ChecklistViewController` works to provide the data to the table view, but only when the table view asks for it.



The dating ritual of a data source and a table view

Your implementation of `tableView(_:numberOfRowsInSection:)` – the first method that you added – returns the value 1. This tells the table view that you have just one row of data.

The `return` statement is very important in Swift. It allows a method to send data back to its caller. In the case of `tableView(_:numberOfRowsInSection:)`, the caller is the `UITableView` object and it wants to know how many rows are in the table.

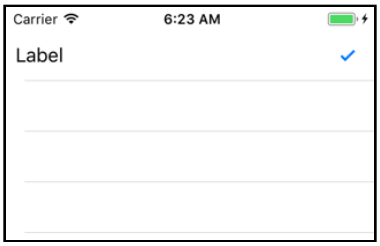
The statements inside a method usually perform some kind of computation using instance variables and any data received through the method's parameters. When the method is done, `return` says, "Hey, I'm done. Here is the answer I came up with." The return value is often called the *result* of the method.

For `tableView(_:numberOfRowsInSection:)` the answer is really simple: there is only one row, so `return 1`.

Now that the table view knows it has one row to display, it calls the second method you added – `tableView(_:cellForRowA:t)` – to obtain a cell for that row. This method grabs a copy of the prototype cell and gives that back to the table view, again with a `return` statement.

Inside `tableView(_:cellForRowAt:)` is also where you would normally put the row data into the cell, but the app doesn't have any row data yet.

➤ Run the app and you'll see there is a single cell in the table:



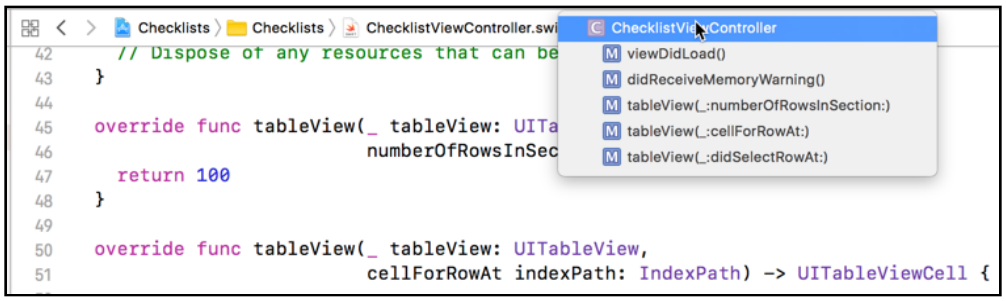
The table now has one row

Method signatures

In the above text, you might have noticed some special notation for the method names, like `tableView(_:numberOfRowsInSection:)` or `tableView(_:cellForRowAt:)`. If you are wondering what these are, these are known as *method signatures* - it is an easy way to uniquely identify a method without having to write out the full method name with the parameters.

The method signature identifies where each parameter would be (and the parameter name, where necessary) by separating out the parameters with a colon. In the method for `tableView(_:numberOfRowsInSection:)` for example, you might notice an underscore for the first parameter - that means that that method does not need to have the parameter name specified when calling the method - it is simply a convenience in Swift where the parameter can generally be inferred from the method name. (You might have more questions about this - but we'll come back to that later.)

If you are not sure about the signature for a method, take a look at the Xcode **Jump bar** (the tiny toolbar right above the source editor) and click on the last item of the file path elements to get a list of methods (and properties) in the current source file.



The Jump Bar shows the method signatures

Also, do note that in the above examples, `tableView` is not the method name - or rather, `tableView` by itself is not the method name. The method name is the `tableView` plus the parameter list - everything up to the closing bracket for the parameter list. That's how you get multiple unique methods such as `tableView(_:numberOfRowsInSection:)` and `tableView(_:cellForRowAt:)` eventhough they all look as if they are methods called `tableName` - the complete signature uniquely identifies the method, if that makes sense?

> **Exercise:** Modify the app so that it shows five rows.

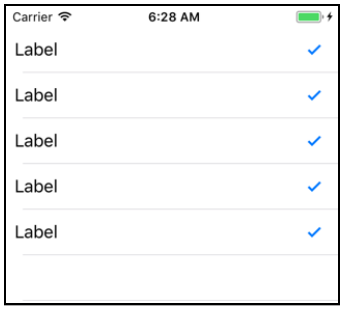
That shouldn't have been too hard:

```
override func tableView(_ tableView: UITableView,
      numberOfRowsInSection section: Int) -> Int {
  return 5
}
```

If you were tempted to go into the storyboard and duplicate the prototype cell five times, then you were confusing cells with rows :]

When you make `tableView(_:numberOfRowsInSection:)` return the number 5, you tell the table view that there will be five rows.

The table view then sends the `cellForRowAt` message five times, once for each row. Because `tableView(_:cellForRowAt:)` currently just returns a copy of the prototype cell, your table view will show five identical rows:



The table now has five identical rows

There are several ways to create cells in `tableView(_:cellForRowAt:)`, but by far the easiest approach is what you've done here:

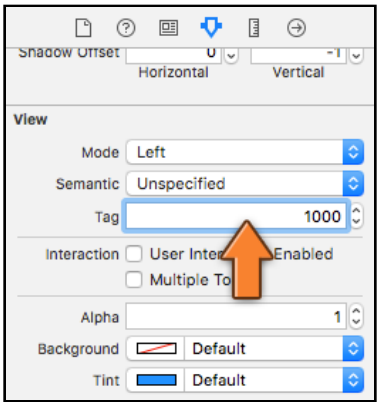
1. Add a prototype cell to the table view in the storyboard.
2. Set a reuse identifier on the prototype cell.
3. Call `tableView.dequeueReusableCell(withIdentifier:for:)`. This makes a new copy of the prototype cell if necessary, or, recycles an existing cell that is no longer in use.

Once you have a cell, you should fill it up with the data from the corresponding row and give it back to the table view. That's what you'll do in the next section.

Putting row data into the cells

Currently, the rows (or rather the cells) all contain the placeholder text "Label". Let's add some unique text for each row.

➤ Open the storyboard and select the **Label** inside the table view cell. Go to the **Attributes inspector** and set the **Tag** field to 1000.



Set the label's tag to 1000

A *tag* is a numeric identifier that you can give to a user interface control in order to uniquely identify it later. Why the number 1000? No particular reason. It should be something other than 0, as that is the default value for all tags. 1000 is as good a number as any.

Double-check to make sure you set the tag on the *Label*, not on the Table View Cell or its Content View. It's a common mistake to set the tag on the wrong view and then the results won't be what you expected!

➤ In **ChecklistViewController.swift**, change `tableView(_:cellForRowAt:)` to the following:

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

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

  // Add the following code
  let label = cell.viewWithTag(1000) as! UILabel

  if indexPath.row == 0 {
    label.text = "Walk the dog"
  } else if indexPath.row == 1 {
    label.text = "Brush my teeth"
  } else if indexPath.row == 2 {
    label.text = "Learn iOS development"
  } else if indexPath.row == 3 {
    label.text = "Soccer practice"
  } else if indexPath.row == 4 {
    label.text = "Eat ice cream"
  }
  // End of new code block

  return cell
}
```

You've already seen the first line. This gets a copy of the prototype cell – either a new one or a recycled one – and puts it into a local constant named `cell`:

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

(Recall that this is a constant because it's declared with `let`, not `var`. It is local because it's defined inside a method.)

But what is this `indexPath` thing?

`IndexPath` is simply an object that points to a specific row in the table. When the table view asks the data source for a cell, you can look at the row number inside the `indexPath.row` property to find out the row for which the cell is intended.

> **Note**: As I mentioned before, it is also possible for tables to group rows into sections. In an address book app you might sort contacts by last name. All contacts whose last name starts with "A" are grouped into their own section, all contacts whose last name starts with "B" are in another section, and so on.

To find out which section a row belongs to, you'd look at the `indexPath.section` property. The *Checklists* app has no need for this kind of grouping, so you'll ignore the `section` property of `IndexPath` for now.

The first new line that you've just added is:

```
    let label = cell.viewWithTag(1000) as! UILabel
```

Here you ask the table view cell for the view with tag 1000. That is the tag you just set on the label in the storyboard. So, this returns a reference to the corresponding `UILabel` object.

Using tags is a handy trick to get a reference to a UI element without having to make an `@IBOutlet` variable for it.

Exercise: Why can't you simply add an `@IBOutlet` variable to the view controller and connect the cell's label to that outlet in the storyboard? After all, that's how you created references to the labels in *Bull's Eye*... so why won't that work here?

Answer: There will be more than one cell in the table and each cell will have its own label. If you connected the label from the prototype cell to an outlet on the view controller, that outlet could only refer to the label from *one* of these cells, not all of them. Since the label belongs to the cell and not to the view controller as a whole, you can't make an outlet for it on the view controller. Confused? We'll circle around to this topic soon, so don't worry about it for now.

Back to the code. The next bit shouldn't give you too much trouble:

```
    if indexPath.row == 0 {
      label.text = "Walk the dog"
    } else if indexPath.row == 1 {
      label.text = "Brush my teeth"
    } else if indexPath.row == 2 {
      label.text = "Learn iOS development"
    } else if indexPath.row == 3 {
      label.text = "Soccer practice"
    } else if indexPath.row == 4 {
      label.text = "Eat ice cream"
    }
```

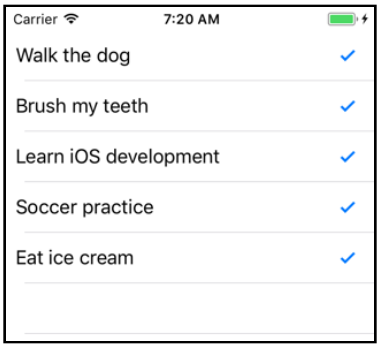
You have seen this `if - else if - else` structure before. It simply looks at the value of `indexPath.row`, which contains the row number, and changes the label's text accordingly. The cell for the first row gets the text "Walk the dog", the cell for the second row gets the text "Brush my teeth", and so on.

Note: Computers generally start counting at 0 for lists of items. If you have a list of 4 items, they are counted as 0, 1, 2 and 3. It may seem a little silly at first, but that's just the way programmers do things.

For the first row in the first section, `indexPath.row` is 0. The second row has row number 1, the third row is row 2, and so on.

Counting from 0 may take some getting used to, but after a while it becomes second nature and you'll start counting at 0 even when you're out for groceries :]

➤ Run the app - it now has five rows, each with its own text:



The rows in the table now have their own text

That is how you write the `tableView(_:cellForRowAt:)` method to provide data to the table. You first get a `UITableViewCell` object and then change the contents of that cell based on the row number of the `indexPath`.

Just for the heck of it, let's put 100 rows into the table.

➤ Make `tableView(_:numberOfRowsInSection:)` return 100.

➤ Also, change the code you added earlier to the following:

```
if indexPath.row % 5 == 0 {
  label.text = "Walk the dog"
} else if indexPath.row % 5 == 1 {
  label.text = "Brush my teeth"
} else if indexPath.row % 5 == 2 {
  label.text = "Learn iOS development"
} else if indexPath.row % 5 == 3 {
  label.text = "Soccer practice"
} else if indexPath.row % 5 == 4 {
  label.text = "Eat ice cream"
}
```

This uses the **remainder operator** (also known as the **modulo operator**), represented by the `%` sign, to determine what row you're on.

The `%` operator returns the remainder of a division. You may remember this from doing math in school. For example 13 % 4 = 1, because four goes into thirteen 3 times with a remainder of 1. However, 12 % 4 is 0 because there is no remainder.

The first row, as well as the sixth, eleventh, sixteenth and so on, will show the text "Walk the dog". The second, seventh and twelfth row will show "Brush my teeth". The third, eight and thirteenth row will show "Learn iOS development". And so on...

I think you get the picture: every five rows these lines repeat. Rather than typing in all the possibilities all the way up to a hundred, you let the computer calculate this for you (afterall, that is what they are good at):

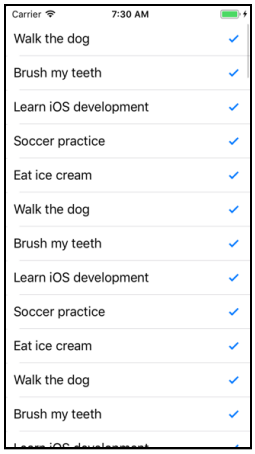
```
First row:     0 % 5 = 0
Second row:    1 % 5 = 1
Third row:     2 % 5 = 2
Fourth row:    3 % 5 = 3
Fifth row:     4 % 5 = 4

Sixth row:     5 % 5 = 0  (same as first row)  *** The sequence
Seventh row:   6 % 5 = 1  (same as second row)     repeats here
Eighth row:    7 % 5 = 2  (same as third row)
Ninth row:     8 % 5 = 3  (same as fourth row)
Tenth row:     9 % 5 = 4  (same as fifth row)

Eleventh row: 10 % 5 = 0  (same as first row)  *** The sequence
Twelfth row:  11 % 5 = 1  (same as second row)     repeats again
and so on...
```

If this makes no sense to you at all, then feel free to ignore it. You're just using this trick to quickly fill up a large table with data.

➤ Run the app and you should see this:



The table now has 100 rows

Note: To scroll through this table view on the Simulator, you have to pretend you're using an actual iPhone. Click the mouse to "grab" the table view and then drag up or down. Simply swiping without clicking first – the way you'd normally scroll things on the Mac – doesn't work.

Exercise: How many cells do you think this table view uses?

Answer: There are a 100 rows, but only about 14 (or more, depending on the device screen height) fit on the screen at a time. If you count the number of visible rows in the screenshot above you'll get up to 13, but it's possible to scroll the table in such a way that the top cell is still visible while a new cell is pulled in from below. So that makes at least 14 cells.

If you scroll really fast, then I guess it is possible that the table view needs to make a few more temporary cells, but I'm not sure about that. Is this important to know? Not really. You should let the table view take care of juggling the cells behind the scenes. All you have to do is give the table view a cell when it asks for it and fill it up with the data for the corresponding row.

You'll usually have fewer cells than rows. If the app always made a cell for each row, iOS would run out of memory really fast, especially on large tables. Because not all rows can be visible at once, that would be very wasteful and slow. iOS is a good citizen and recycles cells whenever it can.

Now you know why `UITableView` makes the distinction between rows – the data, of which you'll usually have lots – and cells – the visible representation of that data on the screen, of which there are only about a dozen.

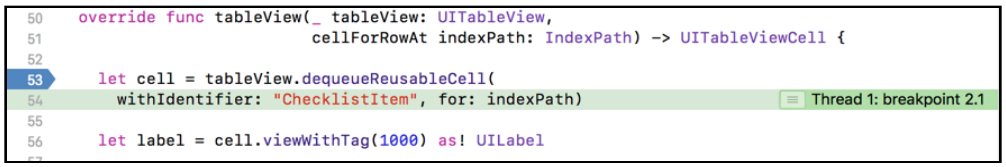
As the song goes, "Rows and cells, rows and cells, tables all the way. Oh what fun it is to learn about new things every day!"

Strange crashes?

A common question on the *iOS Apprentice* forums is, "I'm just following along with the book and suddenly my app crashes… What went wrong?"

If that happens to you, then make sure you haven't set a *breakpoint* on your code by accident. A breakpoint is a debugging tool that stops your program at a specific line and jumps into the Xcode debugger. It may appear like a crash, but your program simply paused.

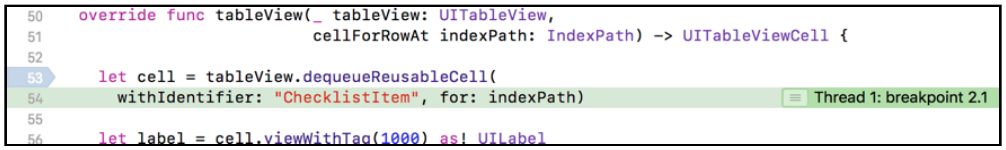
A breakpoint looks like a blue arrow in the left-hand margin (also known as the **gutter**) of the source editor:



The blue arrow sets a breakpoint

If your app suddenly pauses and the source editor shows a blue arrow on a particular line, then you simply hit a breakpoint. Sometimes people click in the margin by mistake and set a breakpoint without even realizing it (I've certainly done that!).

To remove the breakpoint, drag it out of the Xcode window. Or, you can deactivate a breakpoint by simply clicking on it - it will still be there, ready to be activated again by a click, but will not pause code execution. A deactivated breakpoint is indicated by a faded blue arrow.

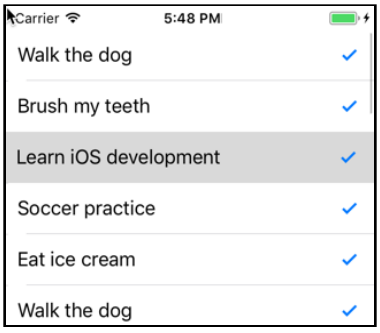


A deactivated breakpoint

By the way, the forums for this book are at forums.raywenderlich.com, so drop by if you have any questions.

Tap on the rows

When you tap on a row , the cell color changes to indicate it is selected. The cell remains selected till you tap another row. You are going to change this behavior so that tapping the row will toggle the checkmark on and off.



A tapped row stays gray

Taps on rows are handled by the table view's **delegate**. Remember I said before that in iOS you often find objects doing something on behalf of other objects? The data source is one example of this, but the table view also depends on another little helper, the table view delegate.

The concept of delegation is very common in iOS. An object will often rely on another object to help it out with certain tasks. This *separation of concerns* keeps the system simple, as each object does only what it is good at and lets other objects take care of the rest. The table view offers a great example of this.

Because every app has its own requirements for what its data looks like, the table view must be able to deal with lots of different types of data. Instead of making the table view very complex, or requiring that you modify it to suit your own apps, the UIKit designers have chosen to delegate the duty of providing the cells to display to another object, the data source.

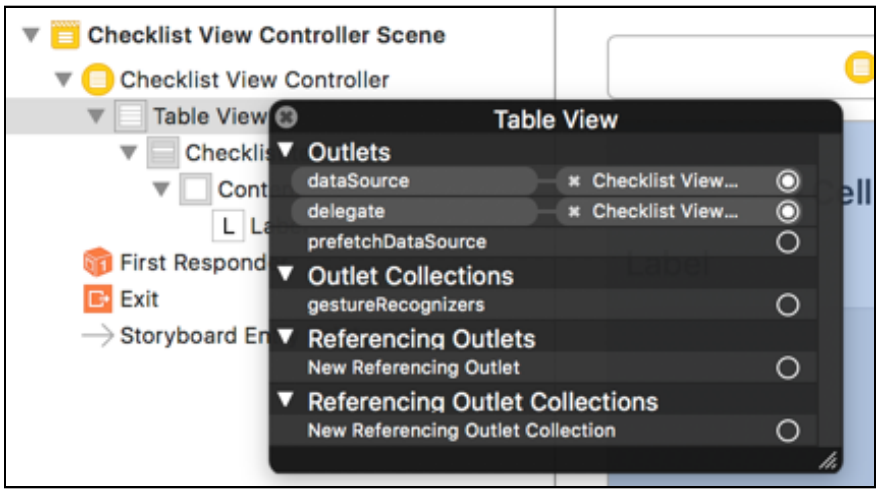
The table view doesn't really care who its data source is or what kind of data your app deals with, just that it can send the `cellForRowAt` message and receive a cell in return. This keeps the table view component simple and moves the responsibility for handling the data to where it belongs: in your code.

Likewise, the table view knows how to recognize when the user taps a row, but what it should do in response depends on the app. In this app, you'll make it toggle the checkmark; another app will likely do something totally different.

Using the delegation system, the table view can simply send a message that a tap occurred and let the delegate sort it out.

Usually, components will have just one delegate. But the table view splits up its delegate duties into two separate helpers: the `UITableViewDataSource` for putting rows into the table, and the `UITableViewDelegate` for handling taps on the rows and several other tasks.

➤ To see this, open the storyboard and **Control-click** on the table view to bring up its connections.



The table's data source and delegate are hooked up to the view controller

You can see that the table view's data source and delegate are both connected to the view controller. That is standard practice for a `UITableViewController`. (You can also use table views in a basic `UIViewController` but then you'll have to connect the data source and delegate manually.)

➤ Add the following method to **ChecklistViewController.swift**:

```
override func tableView(_ tableView: UITableView,
            didSelectRowAt indexPath: IndexPath) {

  tableView.deselectRow(at: indexPath, animated: true)
}
```

The `tableView(_:didSelectRowAt:)` method is one of the table view delegate methods and gets called whenever the user taps on a cell. Run the app and tap a row – the cell briefly turns gray and then becomes de-selected again.

➤ Let's make `tableView(_:didSelectRowAt:)` toggle the checkmark. Change the method to the following:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {

  if let cell = tableView.cellForRow(at: indexPath) {
    if cell.accessoryType == .none {
      cell.accessoryType = .checkmark
    } else {
      cell.accessoryType = .none
    }
  }

  tableView.deselectRow(at: indexPath, animated: true)
}
```

The checkmark is part of the cell (the accessory, remember?). So, you first need to find the `UITableViewCell` object for the tapped row. You simply ask the table view: what is the cell at this `indexPath` you've given me?

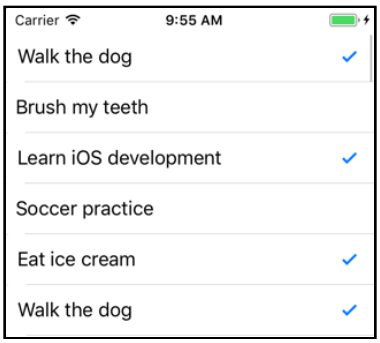
It is theoretically possible that there is no cell at the specified index-path, for example if that row isn't visible. So, you need to use the special `if let` statement.

The `if let` tells Swift that you only want to perform the the code inside the `if` condition only if there really is a `UITableViewCell` object. In this app there always will be one – after all, that's what the user just tapped – but Swift doesn't know that.

Once you have the `UITableViewCell` object, you look at the cell's accessory type, which you can access via the `accessoryType` property. If it is "none", then you change the accessory to a checkmark; if it is already a checkmark, you change it back to none.

> **Note:** In the above code, to find the cell you call `tableView.cellForRow(at:)`.
>
> It's important to realize this is not the same method as the data source method `tableView(_:cellForRowAt:)` that you added earlier.
>
> Despite the similar names they are different methods in different objects, performing different tasks. Tricky, eh?
>
> The purpose of your data source method is to deliver a new (or recycled) cell object to the table view when a row becomes visible. You never call this method yourself; only the `UITableView` may call its data source methods.
>
> The purpose of `tableView.cellForRow(at:)` is also to return a cell object, but this is an existing cell for a row that is currently being displayed. It won't create any new cells. If there is no cell for that row yet, it will return the special value `nil`, meaning that no cell could be found. (You use the `if let` statement to "catch" such `nil` values.)
>
> Remember how I said methods should have clear, descriptive names? UIKit is generally pretty good with its names, but this is a case where a very similar name used in two different places can lead to confusion and despair. Beware this pitfall!

➤ Run the app and try it out. You should be able to toggle the checkmarks on the rows. Sweet!



You can now tap on a row to toggle the checkmark

> **Note:** If the checkmark does not appear or disappear right away but only after you select *another* row, then make sure the method name is not `tableview(_:didDeselectRowAt:)`! You want `didSelect`, not `didDeselect`. Xcode's autocompletion may have fooled you into picking the wrong method name.

Unfortunately, the app has a bug. Here's how to reproduce it:

➤ Tap a row to remove the checkmark. Scroll that row off the screen and scroll back again (try scrolling really fast). The checkmark has reappeared!

In addition, the checkmark seems to spontaneously disappear from other rows. What is going on here?

Again, it's a matter of cells vs. rows: you have toggled the checkmark on the cell but the cell may be reused for another row when you're scrolling. Whether a checkmark is set or not should be a property of a given row (or rather, the data underlying that row), not the cell.

Instead of using the cell's accessory to remember to show a checkmark or not, you need some way to keep track of the checked status for each row. That means it's time to expand the data source and make it use a proper *data model*, which is the topic of the next section.

Methods with multiple parameters

Most of the methods you used in the *Bull's Eye* app took only one parameter or did not have any parameters at all, but these new table view data source and delegate methods take two:

```
override func tableView(
          _ tableView: UITableView,              // parameter 1
          numberOfRowsInSection section: Int)    // parameter 2
          -> Int {                               // return value
```

```
  . . .
}
override func tableView(
          _ tableView: UITableView,              // parameter 1
          cellForRowAt indexPath: IndexPath)     // parameter 2
          -> UITableViewCell {                   // return value
  . . .
}
override func tableView(
        _ tableView: UITableView,                // parameter 1
        didSelectRowAt indexPath: IndexPath) {   // parameter 2
  . . .
}
```

The first parameter is the `UITableView` object on whose behalf these methods are invoked. This is done for convenience, so you won't have to make an `@IBOutlet` in order to send messages back to the table view.

For `numberOfRowsInSection` the second parameter is the section number. For `cellForRowAt` and `didSelectRowAt` it is the index-path.

Methods are not limited to just one or two parameters, they can have many. But for practical reasons two or three is usually more than enough, and you won't see many methods with more than five parameters.

In other programming languages a method typically looks like this:

```
Int numberOfRowsInSection(UITableView tableView, Int section) {
  . . .
}
```

In Swift we do it a little bit differently, mostly to be compatible with the iOS frameworks, which are all written in the Objective-C programming language.

Let's take a look again at `numberOfRowsInSection`:

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

The method signature for the above method, as discussed before, is `tableView(_:numberOfRowsInSection:)`. If you say that out loud (without the underscores and colons, of course), it actually makes sense. It asks for the number of rows in a particular section of a particular table view.

The first parameter looks like this:

```
    _ tableView: UITableView
```

The name of this parameter is `tableView`. The name is followed by a colon and the parameter's type, `UITableView`.

The second parameter looks like this:

```
    numberOfRowsInSection section: Int
```

This one has two names, `numberOfRowsInSection` and `section`.

The first name, `numberOfRowsInSection`, is used when calling the method. This is known as the *external* parameter name. Inside the method itself you use the second name, `section`, known as the *local* parameter name. The data type of this parameter is `Int`.

The _ underscore is used when you don't want a parameter to have an external name. You'll often see the _ on the first parameter of methods that come from Objective-C frameworks. With such methods the first parameter only has one name but the other parameters have two. Strange? Yes.

It makes sense if you've ever programmed in Objective-C but no doubt it looks weird if you're coming from another language. Once you get used to it, you'll find that this notation is actually quite readable.

Sometimes people with experience in other languages get confused because they think that ChecklistViewController.swift contains three functions that are all named `tableView()`. But that's not how it works in Swift: the names of the parameters are part of the full method name. That's why these three methods are actually named:

```
    tableView(_:numberOfRowsInSection:)
    tableView(_:cellForRowAt:)
    tableView(_:didSelectRowAt:)
```

By the way, the return type of the method is at the end, after the –> arrow. If there is no arrow, as in `tableView(_:didSelectRowAt:)`, then the method is not supposed to return a value.

Phew! That was a lot of new stuff to take in, so I hope you're still with me. If not, then take a break and start at the beginning again. You're being introduced to a whole bunch of new concepts all at once and that can be overwhelming.

But don't worry, it's OK if everything doesn't make perfect sense yet. As long as you get the gist of what's going on, you're good to go.

If you want to check your work up to this point, you can find the project files for the app under **09 - Table Views** in the Source Code folder.