Up to date for iOS 10, Xcode 8 & Swift 3

iOS Apprentice

FIFTH EDITION

Tutorial 2: Checklists

By Matthijs Hollemans

iOS Apprentice

Matthijs Hollemans

Notice of Rights

Notice of Liability

Trademarks

License

About the author

Matthijs Hollemans is a mystic who lives at the top of a mountain where he spends all of his days and nights coding up awesome apps. Actually he lives below sea level in the Netherlands and is pretty down-to-earth but he does spend too much time in Xcode. Check out his website at www.matthijshollemans.com.

About the cover

Striped dolphins live to about 55-60 years of age, can travel in pods numbering in the thousands and can dive to depths of 700 m to feed on fish, cephalopods and crustaceans. Baby dolphins don't sleep for a full a month after they're born. That puts two or three sleepless nights spent debugging code into perspective, doesn't it? :]

Table of Contents: Extended

Tutorial 2: Checklists 6

Your own to-do app 6
Playing with table views 10
Model-View-Controller 35
Adding new items to the checklist 59
The Add Item screen 70
Editing existing checklist items 109
Saving and loading the checklist items 126
Multiple checklists 146
Putting to-do items into the checklists 170
Using UserDefaults to remember stuff 190
Improving the user experience 202
Extra feature: local notifications 233
That's a wrap! 260

2 Tutorial 2: Checklists

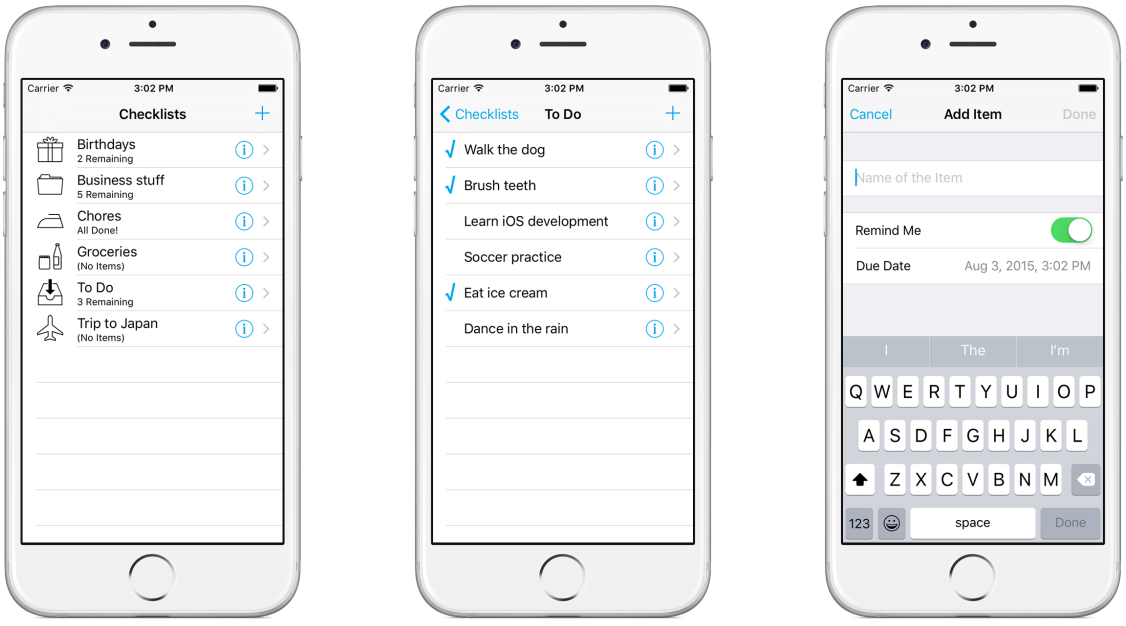
By Matthijs Hollemans

Your own to-do app

To-do list apps are one of the most popular types of app on the App Store, second only to fart apps. The iPhone even comes with the Reminders app (but fortunately no built-in fart 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're done with 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.

Table views and navigation controllers

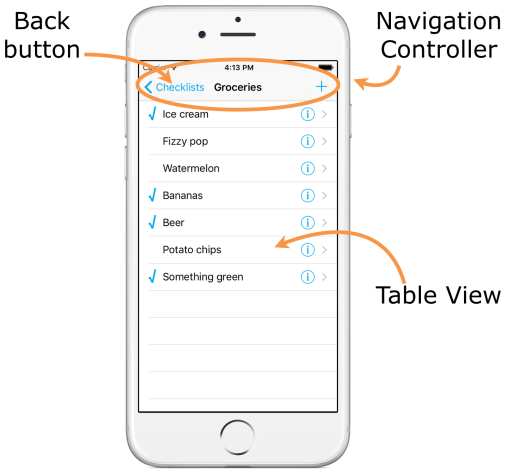
This tutorial will introduce you to two of the most commonly used UI (user interface) elements in iOS apps: the table view and the navigation controller.

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 are made with 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 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.

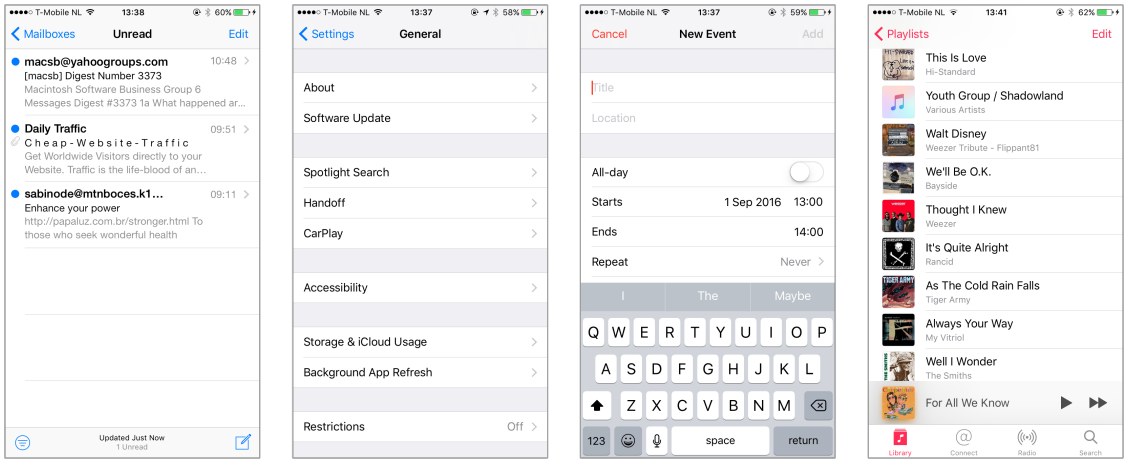
Navigation controllers and table views are often used together:



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 in the next tutorial.)

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 tutorial. 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 lesson, 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.

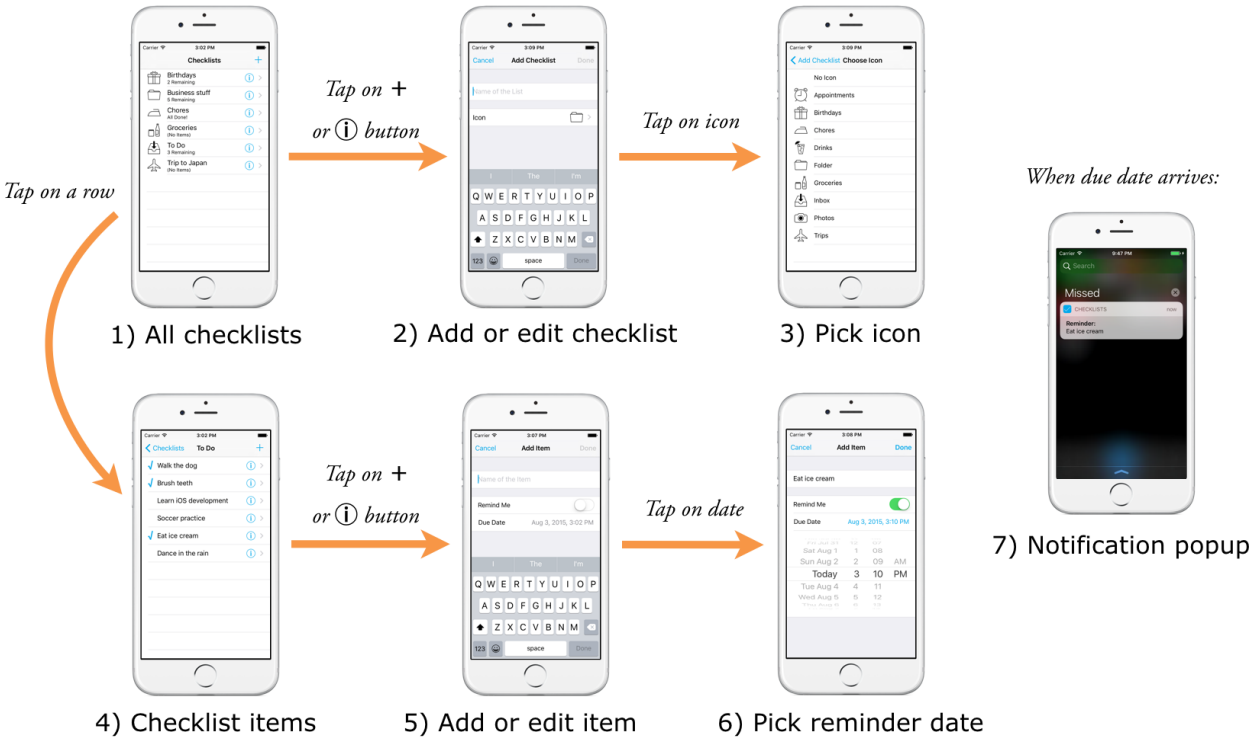
Making mistakes that result in bugs, tearing your hair out in frustration, the light bulb moment when you realize what's wrong, the satisfaction of fixing the bug – they're all essential parts of the learning process.

There's no doubt: playing with the 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 time.

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 mark the item as 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 this tutorial's Resources 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* tutorials are for **Xcode 8.0** and better only. If you're still using Xcode 7, 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 this tutorial. Often the beta versions break things in unexpected ways and you'll only end up confused. Stick to the official versions for now!

Playing with table views

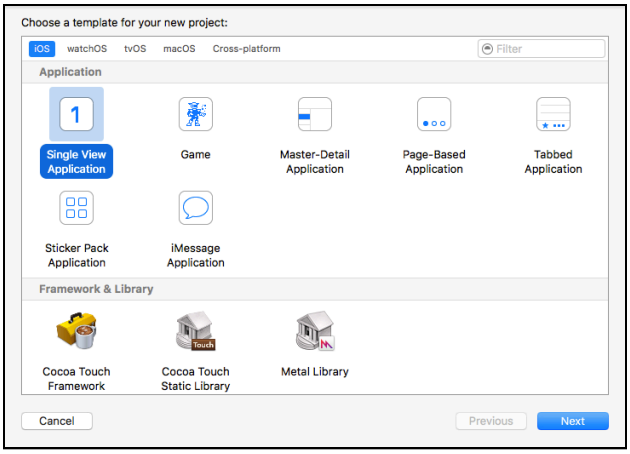
Seeing as table views are so important, you will start out by examining how they work. Making lists has never been this much fun!

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

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

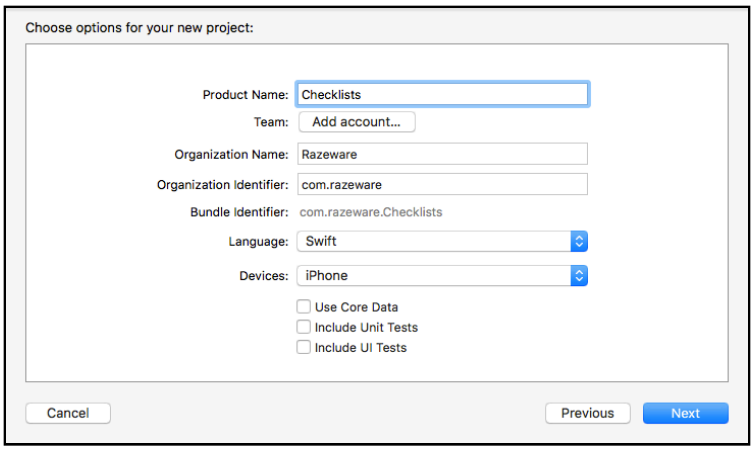
Once you have these basics up and running, you'll keep adding new functionality over the course of this tutorial until you end up with the full-blown app.

➤ 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:

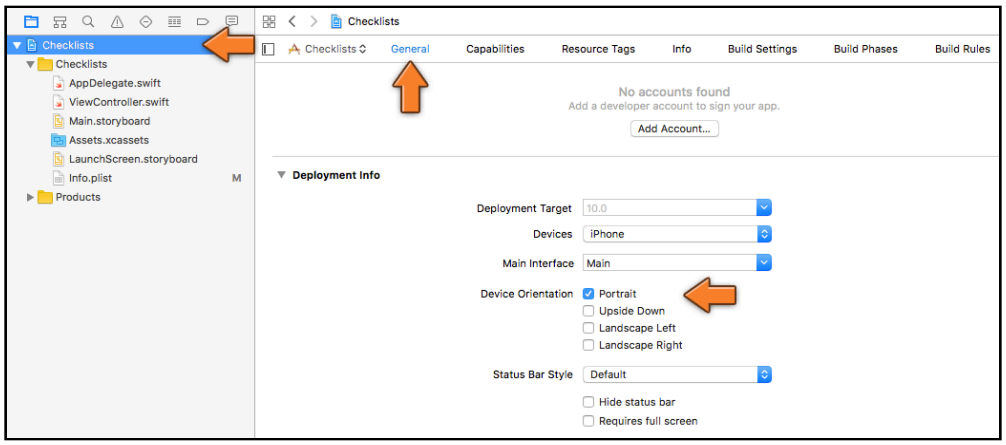
- Product Name: **Checklists**
- Team: Just leave this to the default setting
- Organization Name: Your name or the name of your company
- Organization Identifier: Use your own identifier here, using reverse domain name notation
- Language: **Swift**
- Devices: iPhone
- Use Core Data, Include Unit Tests, 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 at this point it just consists of a white screen.

Checklists will run in portrait orientation only but the project that Xcode just generated also includes the landscape orientation.

➤ 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.

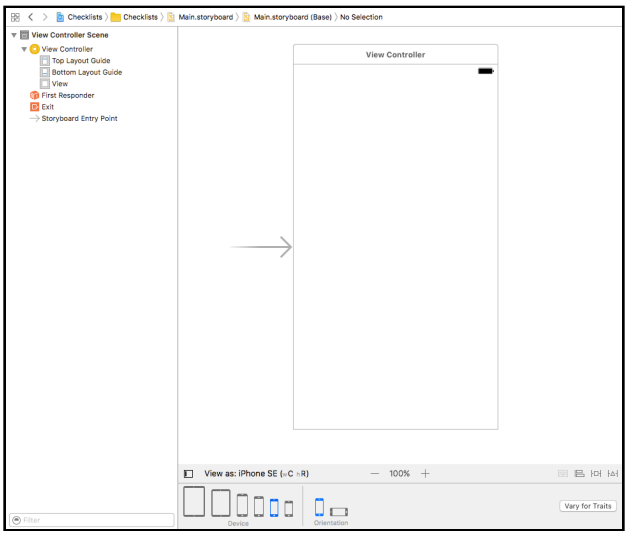
Editing the storyboard

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 tutorial you will unlock the full power of storyboarding.

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

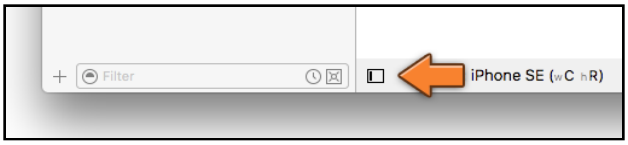


The storyboard editor with the app's only scene

The scene has the dimensions of the iPhone 6s and iPhone 7. 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 outline pane on the left.

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



This button shows and hides the outline pane

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

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

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 and change the following line from this:

```
class ViewController: UIViewController {
```

into 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.

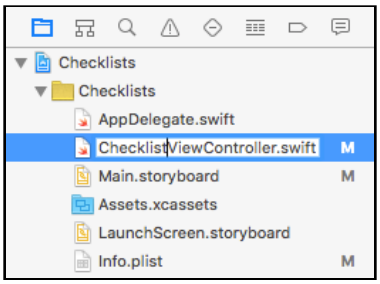
When Xcode made the project, it assumed you wanted the `ViewController` object to be built on top of a basic `UIViewController`, but here you're changing it to use the `UITableViewController` building block instead.

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 tutorial 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.)

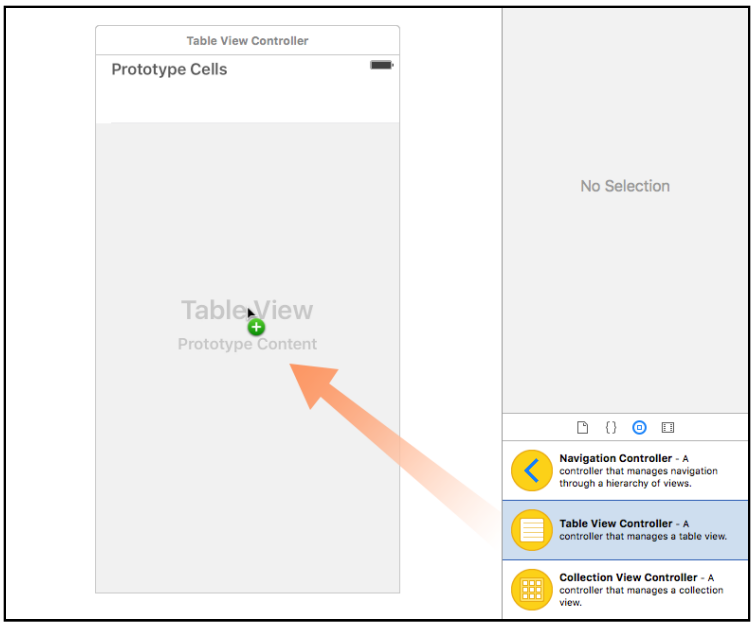
Change the filename to **ChecklistViewController.swift**:



Renaming the Swift file

You may now 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.

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

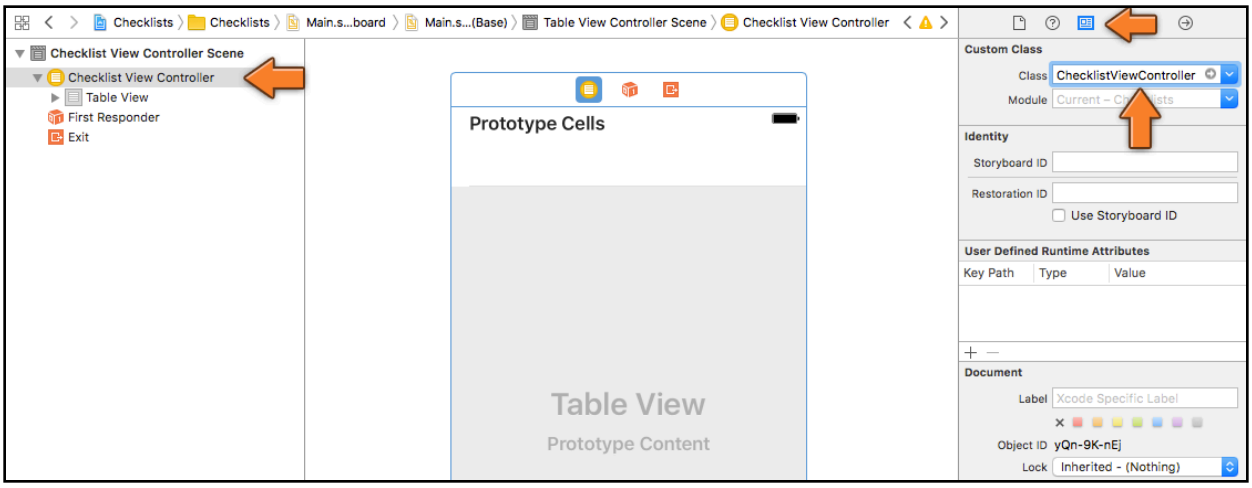


Dragging a Table View Controller into the storyboard

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 small arrow).

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 outline pane 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.

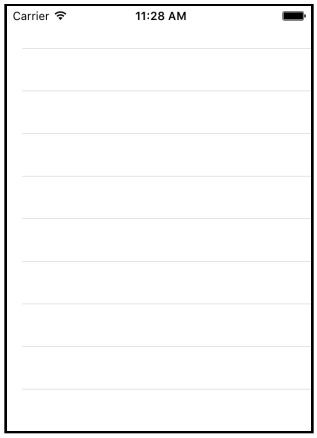
If there is no big arrow pointing towards your new table view controller, then go to the **Attributes inspector** and check **Is Initial View Controller**.

The arrow points at the initial view controller

The initial view controller is the first screen that your users will see. Without it, 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 and the huge iPhone 7 Plus.

Personally, I'm using the iPhone SE Simulator because that one still fits on my Mac's screen, if only barely! (Remember, you can use ⌘**1**, ⌘**2**, and ⌘**3** to zoom the Simulator window.)

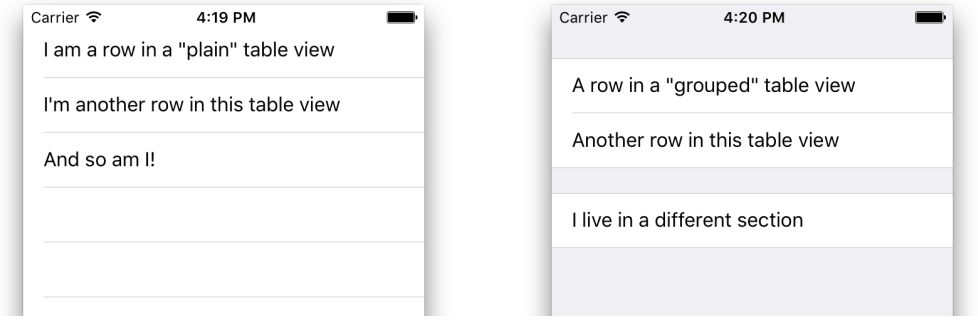
> **Note:** When you build the app, Xcode gives the warning "Prototype cells must have reuse identifiers". Don't worry about this for now, we'll fix it soon.

The anatomy of a table view

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

> **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 dissimilarity 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 each row represents something different, such as the various attributes of one of those contacts. The grouped style table would have a name row, an address row, a phone number row, and so on.

You will use both table styles in the Checklists app.

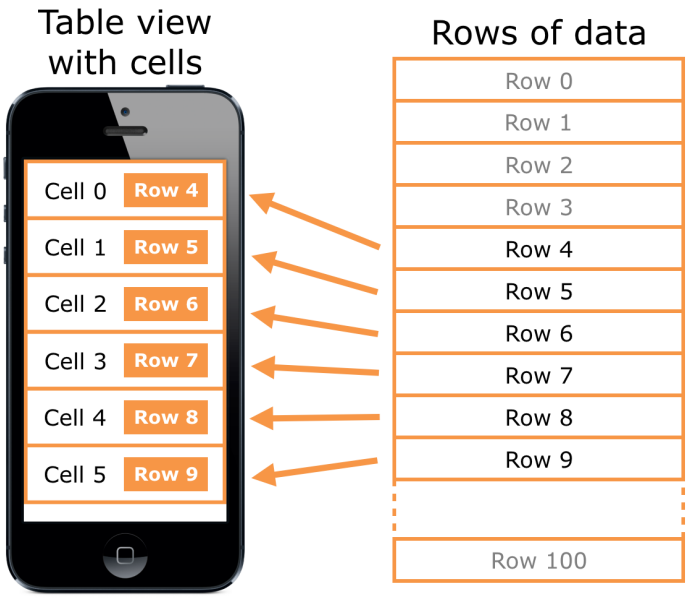
The data for a table comes in the form of **rows**. In the first version of Checklists, each row will correspond to a to-do item that you can check off when you're done with it.

You can potentially have many rows (tens of thousands) although 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 with actual data.

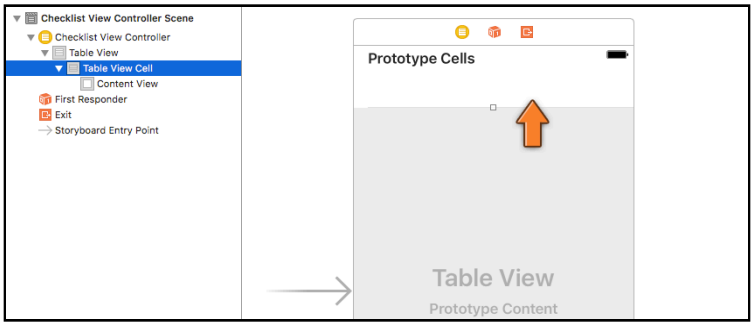
Whenever a row scrolls off the screen and becomes invisible, its cell will be re-used for a new row that scrolls into the screen.



Cells display the contents of rows

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

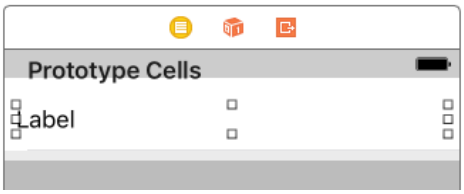
➤ Open the storyboard and click the empty cell to select it.



Selecting the prototype cell

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

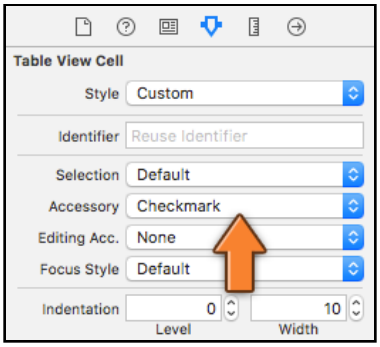
➤ Drag a **Label** from the Object Library into this cell. Make sure the label spans the entire width of the cell (but leave a small margin on the sides).



Adding the label to the prototype cell

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.

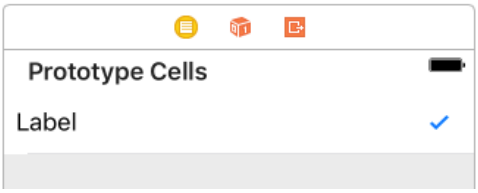
➤ Select the **Table View Cell** again. Inside the **Attributes inspector** set the **Accessory** field to **Checkmark**:



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 now looks like this:



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

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

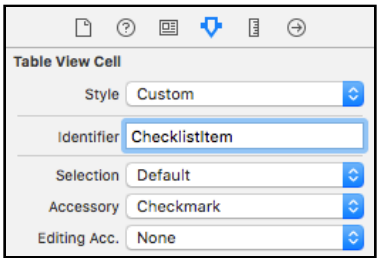
You also need to set a **reuse identifier** on the cell. This is an internal name that the table view uses to find free cells to reuse when rows scroll off the screen and new rows must become visible.

The table needs to assign cells to 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 to the right row.

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… exactly the same as before. The table is still empty.

You only added a cell design to the table, not actual rows. 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 data source

➤ Head on over 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 in the same fashion as the methods you've dealt with before.

These two particular methods are part of `UITableView`'s **data source** protocol.

The data source is the link between your data and the table view. Usually the view controller plays the role of data source and therefore implements these methods.

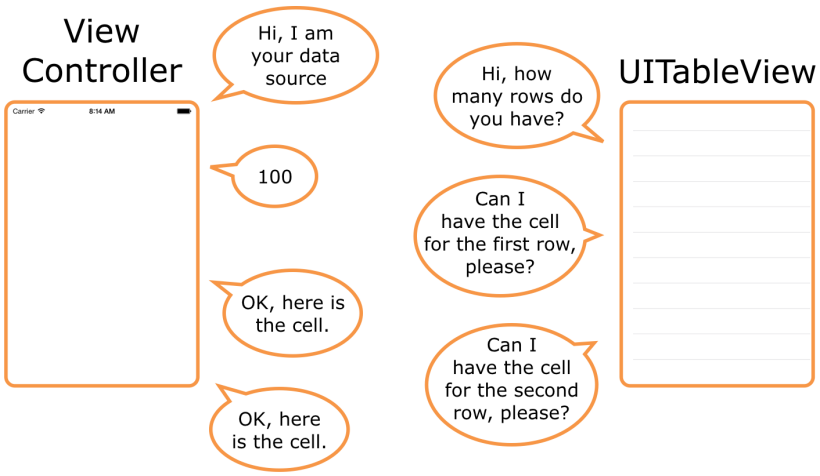
The table view needs to know how many rows of data it has and how it should display each of those rows. But you can't simply dump that data into the table view's lap and be done with it. You don't say: "Dear table view, here are my 100 rows, now go show them on the screen."

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 rows there are.

And when the table view needs to draw a particular row on the screen it sends the "`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 just have 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(cellForRowAt)` – 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

Notice how the iPhone's status bar partially overlaps the table view. The status bar does not have its own separate area but is simply drawn on top of everything. Later in this tutorial you will fix this small cosmetic problem by placing a navigation bar on top of the table view.

Exercise: Modify the app so now it shows five rows. ■

That shouldn't have been too hard:

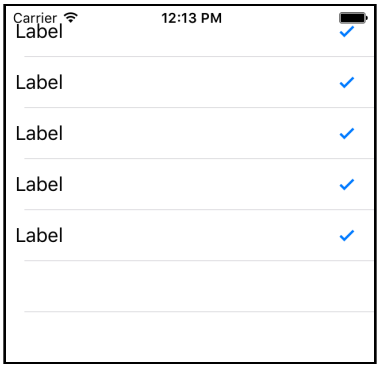
```
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
  return 5
}
```

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 shows 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:

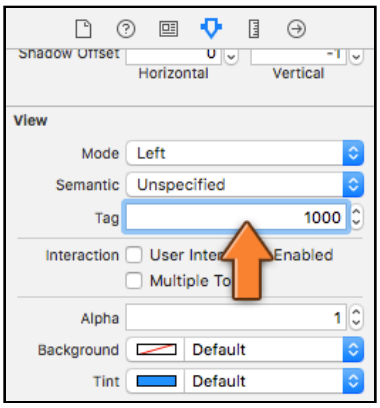
1. add a prototype cell to the table view in the storyboard;
2. set a reuse identifier on the prototype cell;
3. call `tableView.dequeueReusableCell(withIdentifier)`. This makes a new copy of the prototype cell if necessary or recycles an existing cell that is no longer in use.

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 give each row a different text.

➤ 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 easily look it up 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 expect!

➤ 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)

  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"
  }

  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 the local constant named `cell`:

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

(Recall that this is a constant because it's defined 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 for which row this cell is intended.

> **Note**: 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? Don't worry about if for now.

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

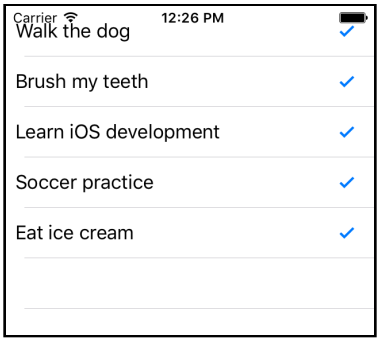
```
    if indexPath.row == 0 {
      label.text = "Walk the dog"
    } else if indexPath.row == 1 {
      label.text = "Brush my teeth"
```

```
    } 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 start counting at 0. 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 natural and you'll start counting at 0 even when you're out for groceries.

➤ Run the app and see that it 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 from `indexPath`.

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

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

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…

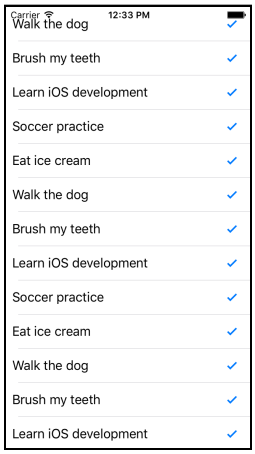
I think you get the picture: every five rows these lines repeat. Rather than typing in all the possibilities all the way up to a hundred, you let the computer calculate this for you (that is what they are good at):

```
  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 get a large table filled up.

➤ Also make `tableView(numberOfRowsInSection)` return 100.

➤ 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 100 rows but only about 14 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 (a few more on the larger iPhone 6s and 7).

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 from 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."

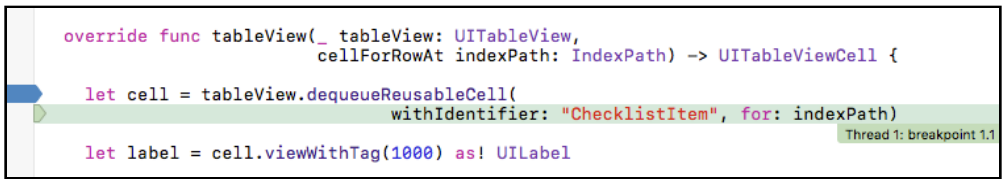
★ ★ ★

Strange crashes?

A common question on the iOS Apprentice forums is, "I'm just following along with the tutorial 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:



The blue arrow sets a breakpoint

If your app crashes and the line at which the error occurred – or the one right before it – has a blue arrow, then you simply hit a breakpoint. Sometimes people click in the margin by mistake and set a breakpoint without realizing it (I've certainly done that!).

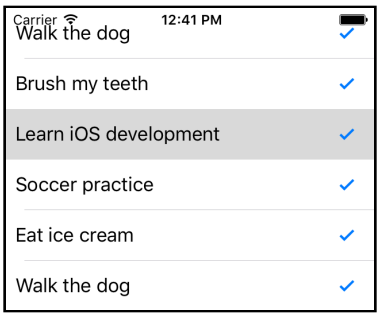
To remove the breakpoint, drag it out of the Xcode window.

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

★ ★ ★

Tapping on the rows

When you tap a row, the cell colors gray to indicate it is selected. But when you let go, the cell stays selected. You are going to change this 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 filling up the cells to another object, the data source.

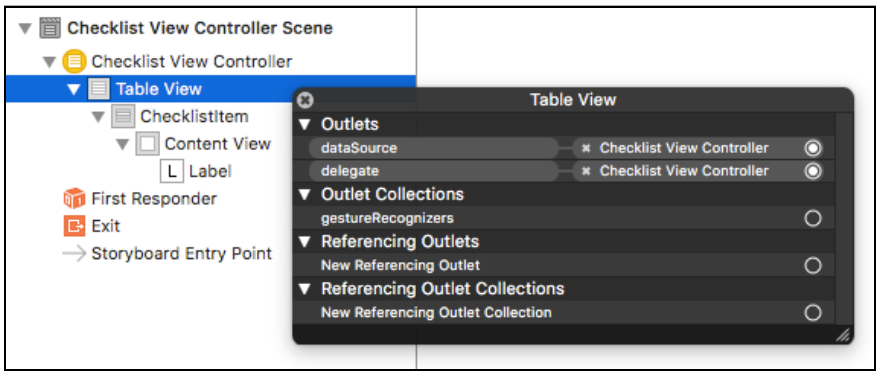
The table view doesn't really care who its data source is or what kind of data your app deals with, just that it can send the `cellForRowAt` message and receive a cell in return. This keeps the table view component simple and moves the responsibility for handling the data to where it belongs: in your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

Likewise, the table view knows how to recognize when the user taps a row, but what it should do in response completely 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 **Ctrl-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, so change it 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?

Because it is theoretically possible that there is no cell at the specified index-path,

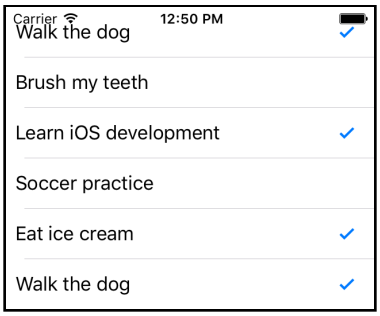
for example if that row isn't visible, you need to use the special `if let` statement.

The `if let` tells Swift that you only want to perform the rest of the code 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, which you can find with the `accessoryType` property. If it is "none", then you change the accessory to a checkmark; if it was a checkmark, you change it back to none.

> **Note:** 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 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 the story 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 the 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 have used in the Bull's Eye tutorial 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 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 full name of this method is officially `tableView(numberOfRowsInSection)`. If you pronounce that out loud, 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`. I'll tell you what the underscore _ is for in a second.

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)
```

Some developers also include the underscore and colons when referring to these methods, but we're not doing that in this book because it's harder to read:

```
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 fear, it's OK if not everything makes perfect sense yet. As long as you get the gist of what's going on, you're good to continue.

If you want to check your work, you can find the project files for the app up to this point under **01 - Table View** in the tutorial's Source Code folder.

Model-View-Controller

No tutorial on programming for iOS can escape an explanation of **Model-View-Controller**, or MVC for short.

MVC is one of the three fundamental design patterns of iOS. You've already seen the other two: *delegation*, making one object do something on behalf of another; and *target-action*, connecting events such as button taps to action methods.

Model-View-Controller means that the objects in your app can be split up into three

groups:

- **Model objects.** These objects contain your data and any operations on the data. For example, if you were writing a cookbook app, the model would consist of the recipes. In a game it would be the design of the levels, the score of the player, and the positions of the monsters.

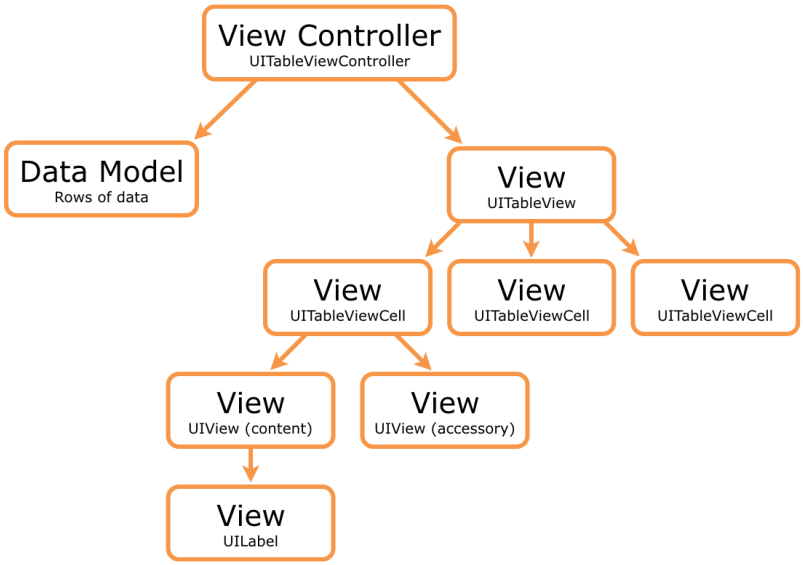
  The operations that the data model objects perform are sometimes called the *business rules* or the *domain logic*. For the app from this tutorial, the checklists and their to-do items form the data model.

- **View objects.** These objects make up the visual part of the app: images, buttons, labels, text fields, table view cells, and so on. In a game the views form the visual representation of the game world, such as the monster animations and a frag counter.

  A view can draw itself and responds to user input, but it typically does not handle any application logic. Many views, such as `UITableView`, can be re-used in many different apps because they are not tied to a specific data model.

- **Controller objects.** The controller is the object that connects your data model objects to the views. It listens to taps on the views, makes the data model objects do some calculations in response, and updates the views to reflect the new state of your model. The controller is in charge. On iOS, the controller is called the "view controller".

Conceptually, this is how these three building blocks fit together:



How Model-View-Controller works

The view controller has one main view, accessible through its `view` property, that contains a bunch of subviews. It is not uncommon for a screen to have dozens of views all at once. The top-level view usually fills the whole screen. You design the layout of the view controller's screen in the storyboard.

In the Checklists app, the main view is the `UITableView` and its subviews are the table view cells. Each cell also has several subviews of its own, namely the text label and the accessory.

A view controller handles one screen of the app. If your app has more than one screen, each of these is handled by its own view controller and has its own views. Your app flows from one view controller to the other.

You will often need to create your own view controllers but iOS also comes with ready-to-use view controllers, such as the image picker controller for photos, the mail compose controller that lets you write email, and the tweet sheet for sending Twitter messages.

Views vs. view controllers

Remember that a view and a view controller are two different things.

A view is an object that draws something on the screen, such as a button or a label. The view is what you see.

The view controller is what does the work behind the scenes. It is the bridge that sits between your data model and the views.

A lot of beginners give their view controllers names such as `FirstView` or `MainView`. That is very confusing! If something is a view controller, its name should end with "ViewController", not "View".

I sometimes wish Apple had left the word "view" out of "view controller" and just called it "controller" as that is a lot less misleading.

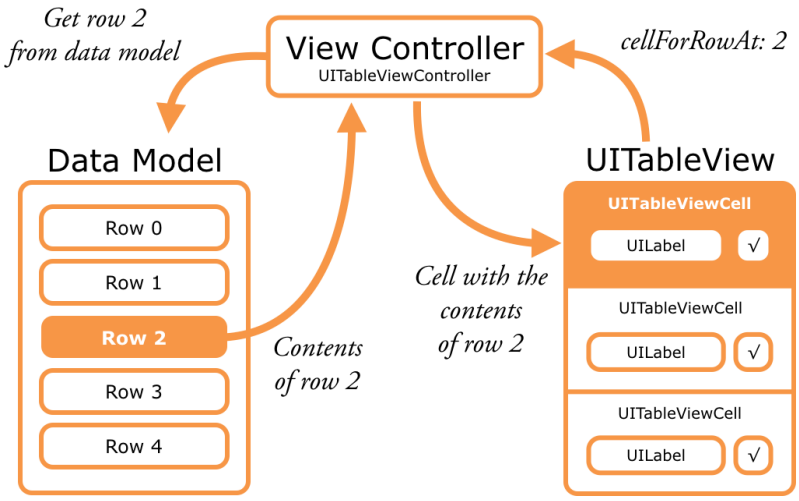
Creating the data model

So far you've put a bunch of fake data into the table view. The data consists of a text string and a checkmark that can be on or off.

As you saw, you cannot use the cells to remember the data as cells get re-used all the time and their old contents get overwritten.

Table view cells are part of the view. Their purpose is to display the app's data, but that data actually comes from somewhere else: the data model.

Remember this well: the rows are the data, the cells are the views. The table view controller is the thing that ties them together through the act of implementing the table view's data source and delegate methods.



The table view controller (data source) gets the data from the model and puts it into the cells

The data model for this app consists of a list of to-do items. Each of these items will get its own row in the table.

For each to-do item you need to store two pieces of information: the text ("Walk the dog", "Brush my teeth", "Eat ice cream") and whether the checkmark is set or not.

That is two pieces of information per row, so you need two variables for each row.

First I'll show you the cumbersome way to program this. It will work but it isn't very smart. Even though this is not the best approach, I'd still like you to follow along and copy-paste the code into Xcode and run the app.

You need to understand why this approach is problematic so you'll be able to appreciate the proper solution better.

➤ In **ChecklistViewController.swift**, add the following instance variables right after the class `ChecklistViewController` line:

```
class ChecklistViewController: UITableViewController {
  var row0text = "Walk the dog"
  var row1text = "Brush teeth"
  var row2text = "Learn iOS development"
  var row3text = "Soccer practice"
  var row4text = "Eat ice cream"
  . . .
```

These variables are defined outside of any method (they are not "local"), so they can be used by all of the methods from `ChecklistViewController`.

➤ Change the data source methods into:

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

override func tableView(_ tableView: UITableView,
                 cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(
                       withIdentifier: "ChecklistItem", for: indexPath)
  let label = cell.viewWithTag(1000) as! UILabel

  if indexPath.row == 0 {
    label.text = row0text
  } else if indexPath.row == 1 {
    label.text = row1text
  } else if indexPath.row == 2 {
    label.text = row2text
  } else if indexPath.row == 3 {
    label.text = row3text
  } else if indexPath.row == 4 {
    label.text = row4text
  }
  return cell
}
```

➤ Run the app. It still shows the same five rows as before.

What have you done here? For every row you have added an instance variable with the text for that row. Together, those five instance variables are your data model.

In `tableView(cellForRowAt)` you look at `indexPath.row` to figure out which row you're supposed to draw, and put the text from the corresponding instance variable into the cell.

Let's fix the checkmark toggling logic. You no longer want to toggle the checkmark on the cell but on the row. To do this, you add another five new instance variables to keep track of the "checked" state of each of the rows. These new variables also belong to your data model.

➤ Add the following instance variables:

```
var row0checked = false
var row1checked = false
var row2checked = false
var row3checked = false
var row4checked = false
```

These variables have the data type `Bool`. You've seen the data types `Int` (whole numbers), `Float` (numbers with decimals), and `String` (text) before. A `Bool` variable can hold only two possible values: `true` and `false`.

`Bool` is short for "boolean", after Englishman George Boole who long ago invented a kind of logic that forms the basis of all modern computing. The fact that computers talk in ones and zeros is largely due to him.

You use `Bool` variables to remember whether something is true (1) or not (0). The names of boolean variables often start with the verb "is" or "has", as in `isHungry` or `hasIceCream`.

The instance variable `row0checked` is `true` if the first row has its checkmark set and `false` if it hasn't. Likewise, `row1checked` reflects whether the second row has a checkmark or not. The same thing goes for the instance variables for the other rows.

> **Note:** How does the compiler know that the type of these variables is `Bool`? You never specified that anywhere.
>
> Swift uses a clever technique called *type inference* to determine the data type of a variable if you don't state it explicitly.
>
> Because you said "`var row0checked = false`", Swift assumes that you intended to make this a `Bool`, as `false` is valid only for `Bool` values.

The delegate method that handles the taps on the rows will now use these new instance variables to determine whether the checkmark for a row needs to be toggled on or off.

The code in `tableView(didSelectRowAt)` should be something like the following. Don't make these changes just yet! Just try to understand what happens.

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

  if let cell = tableView.cellForRow(at: indexPath) {
    if indexPath.row == 0 {
      row0checked = !row0checked
      if row0checked {
        cell.accessoryType = .checkmark
      } else {
        cell.accessoryType = .none
      }
    } else if indexPath.row == 1 {
      row1checked = !row1checked
      if row1checked {
        cell.accessoryType = .checkmark
      } else {
        cell.accessoryType = .none
      }
    } else if indexPath.row == 2 {
      row2checked = !row2checked
      if row2checked {
        cell.accessoryType = .checkmark
      } else {
        cell.accessoryType = .none
      }
    } else if indexPath.row == 3 {
      row3checked = !row3checked
      if row2checked {
```

```
        cell.accessoryType = .checkmark
      } else {
        cell.accessoryType = .none
      }
    } else if indexPath.row == 4 {
      row4checked = !row4checked
      if row4checked {
        cell.accessoryType = .checkmark
      } else {
        cell.accessoryType = .none
      }
    }
  }
  tableView.deselectRow(at: indexPath, animated: true)
}
```

It should be clear that the code looks at `indexPath.row` to find the row that was tapped, and then performs some logic with the corresponding "row checked" instance variable. But there's also some new stuff you may not have seen before.

Let's look at the first `if indexPath.row` statement in detail:

```
    if indexPath.row == 0 {
      row0checked = !row0checked
      if row0checked {
        cell.accessoryType = .checkmark
      } else {
        cell.accessoryType = .none
      }
    } . . .
```

If `indexPath.row` is 0, the user tapped on the very first row and the corresponding instance variable is `row0checked`.

You do the following to flip that boolean value around:

```
      row0checked = !row0checked
```

The `!` symbol is the **logical not** operator. There are a few other logical operators that work on `Bool` values, such as **and** and **or**, which you'll encounter soon enough.

What `!` does is simple: it reverses the meaning of the value. If `row0checked` is `true`, then `!` makes it `false`. Conversely, `!false` is `true`.

Think of `!` as "not": not yes is no and not no is yes. Yes?

Once you have the new value of `row0checked`, you can use it to show or hide the checkmark:

```
      if row0checked {
        cell.accessoryType = .checkmark
      } else {
        cell.accessoryType = .none
      }
```

The same logic is used for the other four rows.

In fact, the other rows use the *exact* same logic. The only thing that is different between each of these code blocks is the name of the "row checked" instance variable.

Because the code looks so familiar from one if-statement to the next, we can write this in a better way.

➤ Replace `tableView(didSelectRowAt)` with the following:

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

  if let cell = tableView.cellForRow(at: indexPath) {
    var isChecked = false

    if indexPath.row == 0 {
      row0checked = !row0checked
      isChecked = row0checked
    } else if indexPath.row == 1 {
      row1checked = !row1checked
      isChecked = row1checked
    } else if indexPath.row == 2 {
      row2checked = !row2checked
      isChecked = row2checked
    } else if indexPath.row == 3 {
      row3checked = !row3checked
      isChecked = row3checked
    } else if indexPath.row == 4 {
      row4checked = !row4checked
      isChecked = row4checked
    }

    if isChecked {
      cell.accessoryType = .checkmark
    } else {
      cell.accessoryType = .none
    }
  }
  tableView.deselectRow(at: indexPath, animated: true)
}
```

That's a lot shorter!

Notice how the logic that sets the checkmark on the cell has moved to the bottom of the method. There is now only one place where this happens.

To make this possible, you store the value of the "row checked" instance variable into the local variable `isChecked`. This temporary variable is just used to remember whether the selected row needs a checkmark or not.

By using a local variable you were able to remove a lot of duplicated code, which is a good thing. You've taken the logic that all rows had in common and moved it out of their if-statements into a single place.

Note: Code duplication makes programs a lot harder to read. Worse, it invites subtle mistakes that cause hard-to-find bugs. Always be on the lookout for opportunities to remove duplicate logic!

Exercise: There was actually a bug in the previous, longer version of this method – did you spot it? That's what happens when you use copy-paste to create duplicate code, like I did when I wrote that method. ■

➤ Run the app and observe… that it still doesn't work very well. You have to tap a few times on a row to actually make the checkmark go away.

What's wrong here? Simple: when you declared the `rowXchecked` variables you set their values to `false`.

So `row0checked` and the others think that there is no checkmark on their row, but the table draws one anyway. That's because you enabled the checkmark accessory on the prototype cell.

In other words: the data model (the "row checked" variables) and the views (the checkmarks inside the cells) are out-of-sync.

There are a few ways you could try to fix this: you could set the `Bool` variables to `true` to begin with, or you could remove the checkmark from the prototype cell in the storyboard.

Neither is a foolproof solution. What goes wrong here isn't so much that you initialized the "row checked" values wrong or designed the prototype cell wrong, but that you didn't set the cell's `accessoryType` property to the right value in `tableView(cellForRowAt)`.

When you are asked for a new cell, you always should configure all of its properties. The call to `tableView.dequeueReusableCell(withIdentifier)` could return a cell that was previously used for a row with a checkmark. If the new row shouldn't have a checkmark, then you have to remove it from the cell at this point (and vice versa).

Let's fix that.

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

```
func configureCheckmark(for cell: UITableViewCell,
                        at indexPath: IndexPath) {
  var isChecked = false

  if indexPath.row == 0 {
    isChecked = row0checked
  } else if indexPath.row == 1 {
    isChecked = row1checked
  } else if indexPath.row == 2 {
    isChecked = row2checked
  } else if indexPath.row == 3 {
    isChecked = row3checked
```

```
    } else if indexPath.row == 4 {
      isChecked = row4checked
    }

    if isChecked {
      cell.accessoryType = .checkmark
    } else {
      cell.accessoryType = .none
    }
  }
```

This new method looks at the cell for a certain row, specified as usual by `indexPath`, and makes the checkmark visible if the corresponding "row checked" variable is `true`, or hides the checkmark if the variable is `false`.

This logic should look very familiar! The only difference with before is that here you don't toggle the state of the "row checked" variable. You only read it and then set the cell's accessory.

You'll call this method from `tableView(cellForRowAt)`, just before you return the cell.

➤ Change that method to the following (recall that `. . .` means that the existing code at that spot doesn't change):

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

  configureCheckmark(for: cell, at: indexPath)
  return cell
}
```

➤ Run the app again.

Now the app works just fine. Initially all the rows are unchecked. Tapping a row checks it, tapping it again unchecks it. The rows and cells are now always in sync. This code guarantees that each cell always has the value that corresponds to its row.

External and internal parameter names

The new `configureCheckmark()` method has two parameters, `for` and `at`. Its full name is therefore `configureCheckmark(for:at:)`.

`for` and `at` are the so-called *external* names of these parameters.

Adding short prepositions such as "at", "with", or "for" is very common in Swift. It

makes the name of the method sound like a proper English phrase: "configure checkmark for this cell at that index-path". Doesn't it just roll off your tongue?

When you call the method, you always have to include those external parameter names:

```
configureCheckmark(for: someCell, at: someIndexPath)
```

Here, `someCell` is a variable that refers to a `UITableViewCell` object, and likewise, `someIndexPath` is a variable of type `IndexPath`.

You can't write the following:

```
configureCheckmark(someCell, someIndexPath)
```

This won't compile. The app doesn't have a `configureCheckmark()` method, only `configureCheckmark(for:at:)`. The `for` and `at` are an integral part of the method name!

Inside the method you use the *internal* labels `cell` and `indexPath` to refer to the parameters.

```
func configureCheckmark(for cell: UITableViewCell,
                        at indexPath: IndexPath) {
  if indexPath.row == 0 {
    . . .
  }

  cell.accessoryType = .checkmark
  . . .
}
```

You can't write `if at.row == 0` or `for.accessoryType = .checkmark`. That also sounds a little odd, doesn't it?

This split between external and internal labels is unique to Swift and Objective-C and takes some getting used to if you're familiar with other languages.

This naming convention primarily exists so that Swift can talk to older Objective-C code, which is a good thing as most of the iOS frameworks are still written in that language.

Why did you make `configureCheckmark(for:at:)` a method of its own anyway? Well, you can use it to simplify `tableView(didSelectRowAt)`.

Notice how similar these two methods currently are. That's another case of code

duplication that you can get rid of!

You can simplify "`didSelectRowAt`" by letting `configureCheckmark(for:at:)` do some of the work.

➤ Replace `tableView(didSelectRowAt)` with the following:

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

  if let cell = tableView.cellForRow(at: indexPath) {
    if indexPath.row == 0 {
      row0checked = !row0checked
    } else if indexPath.row == 1 {
      row1checked = !row1checked
    } else if indexPath.row == 2 {
      row2checked = !row2checked
    } else if indexPath.row == 3 {
      row3checked = !row3checked
    } else if indexPath.row == 4 {
      row4checked = !row4checked
    }

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

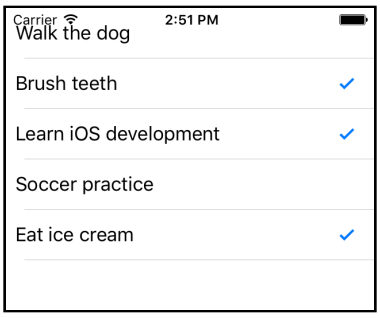
This method no longer sets or clears the checkmark from the cell, but only toggles the "checked" state in the data model and then calls `configureCheckmark(for:at:)` to update the view.

➤ Run the app again and it should still work.

➤ Change the declarations of the instance variables to the following and run the app again:

```
var row0checked = false
var row1checked = true
var row2checked = true
var row3checked = false
var row4checked = true
```

Now rows 1, 2 and 4 (the second, third and fifth rows) initially have a checkmark while the others don't.



The data model and the table view cells are now always in-sync

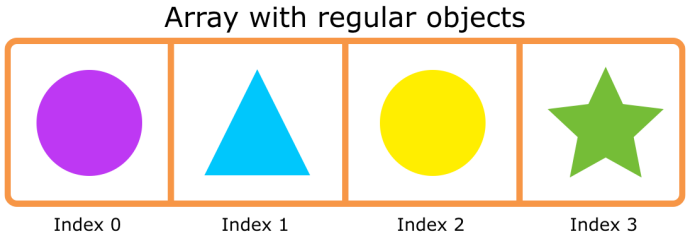
The approach that we've taken here to remember which rows are checked or not works just fine, but you'll have to agree that checking each index-path by hand seems like a lot of effort.

For only five rows it's doable, but what if you have 100 rows and they all need to be unique? Should you add another 95 "row text" and "row checked" variables to the view controller, as well as that many additional if-statements? I hope not!

There is a better way: arrays.

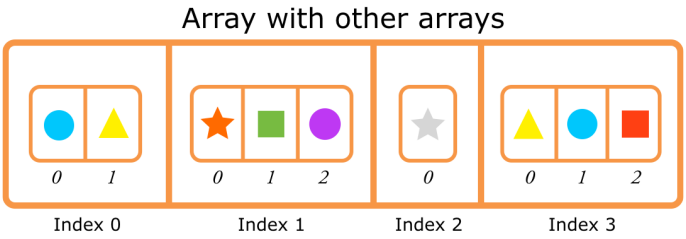
Arrays

An **array** is an ordered list of objects. If you think of a variable as a container of one value (or one object) then an array is a container for multiple objects.



Arrays are ordered lists containing multiple objects

Of course, the array itself is also an object (named `Array`) that you can put into a variable. And because arrays are objects, arrays can contain other arrays.



Arrays can also include other arrays

The objects inside an array are indexed by numbers, starting at 0 as usual. To ask the array for the first object, you write `array[0]`. The second object is at `array[1]`, and so on.

The array is *ordered*, meaning that the order of the objects it contains matters. The object at index 0 always comes before the object at index 1.

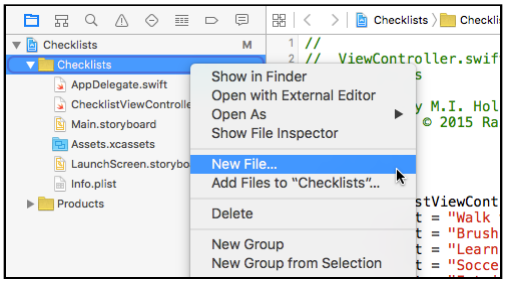
> **Note:** Array is a so-called *collection* object. There are several other collection objects and they all organize their objects in a different fashion. `Dictionary`, for example, contains so-called "key-value pairs", just like a real dictionary contains a list of words and a description for each of those words. You'll use some of these other collection types in the later tutorials.

The organization of an array is very similar to the rows from a table – they are both lists of objects in a particular order – so it makes sense to put your data model's rows into an array.

Arrays store one object per index, but your rows currently consist of two separate pieces of data: the text and the checked state. It would be easier if you made a single object for each row, because then the row number from the table simply becomes the index in the array.

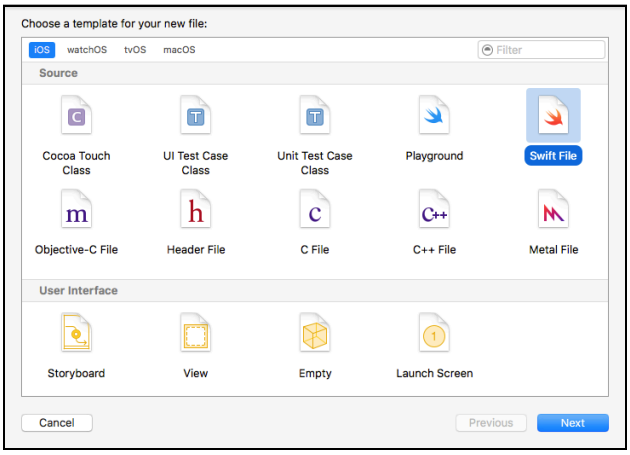
Let's combine the text and checkmark state into a new object of your own!

➤ Select the **Checklists** group in the project navigator and right click. Choose **New File...** from the popup menu:



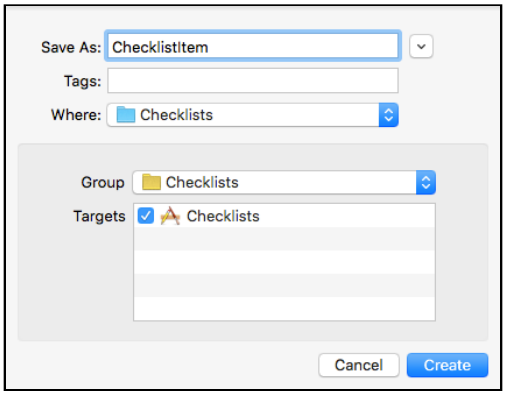
Adding a new file to the project

Under the **Source** section choose **Swift File**:



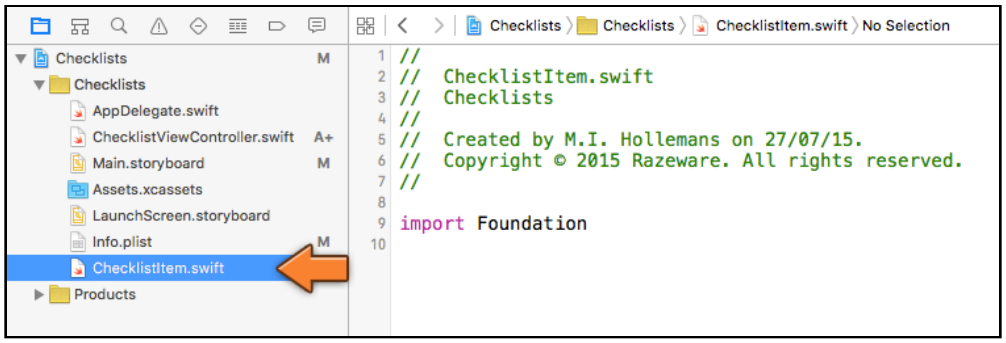
Choosing the Swift File class template

Click **Next** to continue. Save the new file as **ChecklistItem** (adding the **.swift** file extension is optional):



Saving the new Swift file

Press **Create** to add the new file to the project:



The new file is added to the project navigator

➤ Add the following to the new **ChecklistItem.swift** file, below the `import` line:

```
class ChecklistItem {
  var text = ""
  var checked = false
}
```

What you see here is the absolute minimum amount of stuff you need in order to make a new object. The `class` keyword names the object and the two lines with `var` add data items (instance variables) to it.

The `text` property will store the description of the checklist item (the text that will appear in the table view cell's label) and the `checked` property determines whether the cell gets a checkmark or not.

> **Note:** You may be wondering what the difference is between the terms *property* and *instance variable* – we've used both to refer to an object's data items. You'll be glad to hear that these two things mean the same thing.
>
> In Swift terminology, a property is a variable or constant that is used in the context of an object. That's exactly what an instance variable is. So you can use the terms property and instance variable interchangeably.
>
> (In Objective-C, properties and instance variables are closely related but not quite the same thing. In Swift they are the same.)

That's all for **ChecklistItem.swift** for now. The `ChecklistItem` object currently only serves to combine the `text` and the `checked` variables into one object. Later you'll add more to it.

Before you get around to using an array, let's replace the `String` and `Bool` instance variables in the view controller with these new `ChecklistItem` objects.

➤ In **ChecklistViewController.swift**, remove the old instance variables and replace them with `ChecklistItem` objects:

```
class ChecklistViewController: UITableViewController {
  var row0item: ChecklistItem
  var row1item: ChecklistItem
  var row2item: ChecklistItem
  var row3item: ChecklistItem
  var row4item: ChecklistItem
```

This replaces the `row0text`, `row0checked`, etc. instance variables.

Because some methods in the view controller still refer to these old variables, Xcode detects several errors. Before you can run the app again you need to fix these errors, so let's do that now.

Note: I generally encourage you to type in the code from this book by hand because that gives you a better feel for what you're doing, but in the following instances it's easier to just copy-paste from the PDF.

Unfortunately, copying from the PDF sometimes adds strange or invisible characters that confuse Xcode. It's best to first paste into a plain text editor such as TextWrangler and then copy that into Xcode.

Of course, if you're reading the print edition of this book, copying & pasting from the book isn't going to work, but you can still use copy-paste to save yourself some effort. Make the changes on one line and then copy them over to the other lines. Copy-paste is a programmer's best friend, but don't forget to update the lines you pasted to use the correct variable names!

➤ In `tableView(cellForRowAt)`, replace the if-statements with the following:

```
if indexPath.row == 0 {
  label.text = row0item.text
} else if indexPath.row == 1 {
  label.text = row1item.text
} else if indexPath.row == 2 {
  label.text = row2item.text
} else if indexPath.row == 3 {
  label.text = row3item.text
} else if indexPath.row == 4 {
  label.text = row4item.text
}
```

➤ In `tableView(didSelectRowAt)`, change the following lines:

```
  if indexPath.row == 0 {
    row0item.checked = !row0item.checked
  } else if indexPath.row == 1 {
    row1item.checked = !row1item.checked
  } else if indexPath.row == 2 {
    row2item.checked = !row2item.checked
  } else if indexPath.row == 3 {
    row3item.checked = !row3item.checked
  } else if indexPath.row == 4 {
    row4item.checked = !row4item.checked
  }
```

➤ And finally, in `configureCheckmark(for:at:)`, make these changes:

```
if indexPath.row == 0 {
  isChecked = row0item.checked
} else if indexPath.row == 1 {
  isChecked = row1item.checked
} else if indexPath.row == 2 {
  isChecked = row2item.checked
} else if indexPath.row == 3 {
  isChecked = row3item.checked
} else if indexPath.row == 4 {
```

```
    isChecked = row4item.checked
  }
```

Instead of using the separate `row0text` and `row0checked` variables, you now use `row0item.text` and `row0item.checked`. Likewise for the other rows.

That takes care of most of the errors, but not all of them. Xcode complains that "Class ChecklistViewController has no initializers." This was not a problem before, so what has gone wrong?

Previously you gave the "row text" and "row checked" variables a value when you declared them, like so:

```
var row0text = "Walk the dog"
var row0checked = false
```

With the new `ChecklistItem` object you can't do that because a `ChecklistItem` consists of more than one value.

Instead you used a so-called *type annotation* to tell Swift that `row0Item` is an object of type `ChecklistItem`:

```
var row0item: ChecklistItem
```

But at this point `row0item` doesn't have a value yet, it's just an empty container for a `ChecklistItem` object.

And that's a problem: in Swift programs, all variables should always have a value – the containers can never be empty.

If you can't give the variable a value right away when you declare it, then you have to give it a value inside a so-called *initializer* method.

➤ Add the following to **ChecklistViewController.swift**. This is a special type of method (which is why it doesn't start with the word `func`). It is customary to place it near the top of the file, just below the instance variables.

```
required init?(coder aDecoder: NSCoder) {
  row0item = ChecklistItem()
  row0item.text = "Walk the dog"
  row0item.checked = false

  row1item = ChecklistItem()
  row1item.text = "Brush my teeth"
  row1item.checked = true

  row2item = ChecklistItem()
  row2item.text = "Learn iOS development"
  row2item.checked = true

  row3item = ChecklistItem()
  row3item.text = "Soccer practice"
  row3item.checked = false
```

```
    row4item = ChecklistItem()
    row4item.text = "Eat ice cream"
    row4item.checked = true

    super.init(coder: aDecoder)
  }
```

Every object in Swift has an `init` method, or initializer. Some objects even have more than one.

The `init` method is called by Swift when the object comes into existence.

For the view controller that happens when it is loaded from the storyboard during app startup. At that point, its `init?(coder)` method is called.

That makes `init?(coder)` a great place for putting values into any variables that still need them (soon you'll learn more about what the "coder" parameter is for).

Inside `init?(coder)`, you first create a new `ChecklistItem` object:

```
    row0item = ChecklistItem()
```

and then set the properties:

```
    row0item.text = "Walk the dog"
    row0item.checked = false
```

You repeat this for the other four rows. Each row gets its own `ChecklistItem` object that you store in its own instance variable.

This is essentially doing the same thing as before, except that this time the `text` and `checked` variables are not separate instance variables of the view controller but properties of the `ChecklistItem` objects.

➤ Run the app just to make sure that everything works again.

Putting the `text` and `checked` properties into their own `ChecklistItem` object already improved the code, but it is still a bit unwieldy.

With the current approach, you need to keep around a `ChecklistItem` instance variable for each row. That's not ideal, especially not if you want more than just a handful of rows.

Time to put that array into action!

➤ In **ChecklistViewController.swift**, throw away all the instance variables and replace them with a single array variable named `items`:

```
class ChecklistViewController: UITableViewController {

  var items: [ChecklistItem]
```

Instead of five different instance variables, one for each row, you now have just one variable for the array.

This looks similar to how you declared the previous variables but this time there are square brackets around `ChecklistItem`. Those square brackets indicate that this is going to be an array.

➤ Make the following changes in `init?(coder)`:

```
required init?(coder aDecoder: NSCoder) {
  items = [ChecklistItem]()                  // add this line

  let row0item = ChecklistItem()             // let
  row0item.text = "Walk the dog"
  row0item.checked = false
  items.append(row0item)                     // add this line

  let row1item = ChecklistItem()             // let
  row1item.text = "Brush my teeth"
  row1item.checked = true
  items.append(row1item)                     // add this line

  let row2item = ChecklistItem()             // let
  row2item.text = "Learn iOS development"
  row2item.checked = true
  items.append(row2item)                     // add this line

  let row3item = ChecklistItem()             // let
  row3item.text = "Soccer practice"
  row3item.checked = false
  items.append(row3item)                     // add this line

  let row4item = ChecklistItem()             // let
  row4item.text = "Eat ice cream"
  row4item.checked = true
  items.append(row4item)                     // add this line

  super.init(coder: aDecoder)
}
```

This is not so different from before, except that you first create – or *instantiate* – the array object:

```
  items = [ChecklistItem]()
```

You've seen that the notation `[ChecklistItem]` means an array of `ChecklistItem` objects. But that is just the data type of the `items` variable; it is not the actual array object yet.

To get the array object you have to construct it first. That is what the parentheses `()` are for: they tell Swift to make the new array object.

The data type is like the brand name of a car. Just saying the words "Porsche 911" out loud doesn't magically get you a new car – you actually have to go to the dealer

to buy one.

The parentheses `()` behind the type name are like going to the object dealership to buy an object of that type. The parentheses tell Swift's object factory, "Build me an object of the type array-with-`ChecklistItems`."

It is important to remember that just declaring that you have a variable does not automatically make the corresponding object for you. The variable is just the container for the object. You still have to instantiate the object and put it into the container. The variable is the box and the object is the thing inside the box.

So until you order an actual array-of-`ChecklistItems` object from the factory and put that into `items`, the variable is empty. And empty variables are a big no-no in Swift.

Just to drive this point home:

```
// This declares that items will hold an array of ChecklistItem objects
// but it does not actually create that array.
// At this point, items does not have a value yet.
var items: [ChecklistItem]

// This instantiates the array. Now items contains a valid array object,
// but the array has no ChecklistItem objects inside it yet.
items = [ChecklistItem]()
```

Each time you make a `ChecklistItem` object, you also add it into the array:

```
// This instantiates a new ChecklistItem object. Notice the ().
let row0item = ChecklistItem()

// Give values to the data items inside the new ChecklistItem object.
row0item.text = "Walk the dog"
row0item.checked = false

// This adds the ChecklistItem object into the items array.
items.append(row0item)
```

Notice that you're also using the parentheses here to create each of the individual `ChecklistItem` objects.

It's also important that `row0item` and the others are now local to the `init` method. They are no longer valid instance variable names (because you removed those earlier). That's why you need to use the `let` keyword; without it, the app won't compile.

At the end of `init?(coder)`, the `items` array contains five `ChecklistItem` objects. This is your new data model.

Now that you have all your rows in the `items` array, you can simplify the table view data source and delegate methods once again.

➤ Change these methods:

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(
                         withIdentifier: "ChecklistItem", for: indexPath)

  let item = items[indexPath.row]

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

  configureCheckmark(for: cell, at: indexPath)
  return cell
}
```

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

  if let cell = tableView.cellForRow(at: indexPath) {
    let item = items[indexPath.row]
    item.checked = !item.checked

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

```
func configureCheckmark(for cell: UITableViewCell,
                         at indexPath: IndexPath) {
  let item = items[indexPath.row]

  if item.checked {
    cell.accessoryType = .checkmark
  } else {
    cell.accessoryType = .none
  }
}
```

That's a lot simpler than what you had before! Each method is now only a handful of lines long.

In each method, you do:

```
let item = items[indexPath.row]
```

This asks the array for the `ChecklistItem` object at the index that corresponds to the row number. Once you have that object, you can simply look at its `text` and `checked` properties and do whatever you need to do.

If the user were to add 100 to-do items to this list, then none of this code would need to change. It works equally well with five items as with a hundred (or a thousand).

Speaking of the number of items, you can now change "`numberOfRowsInSection`" to return the number of items in the array, instead of a hard-coded number.

➤ Change the `tableView(numberOfRowsInSection)` method to:

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

Not only is the code a lot shorter and easier to read, it can now also handle an arbitrary number of rows. That is the power of arrays.

➤ Run the app and see for yourself. It should still do exactly the same as before but its internal structure is much better.

Exercise: Add a few more rows to the table. You should only have to change `init?(coder)` for this to work. ■

Cleaning up the code

There are a few more things you can do to improve the source code.

➤ Replace the `configureCheckmark(for:at:)` method with this one:

```
func configureCheckmark(for cell: UITableViewCell,
                        with item: ChecklistItem) {
  if item.checked {
    cell.accessoryType = .checkmark
  } else {
    cell.accessoryType = .none
  }
}
```

Instead of an index-path, you now directly pass it the `ChecklistItem` object.

This again is an example of a parameter with an extra label, giving it the external name `with`. It makes the full name of this method `configureCheckmark(for:with:)` and that's how you will call it from other places in the app. Inside the method itself you use the local name for this parameter, `item`.

Why did you change this method? Previously it received an index-path and then did this to find the corresponding `ChecklistItem`:

```
let item = items[indexPath.row]
```

But in both "`cellForRowAt`" and "`didSelectRowAt`" you already do that as well. It is simpler to pass that `ChecklistItem` object directly to `configureCheckmark()` instead of making it do the same work twice. Anything that simplifies the code is good.

➤ Also add a new method:

```
func configureText(for cell: UITableViewCell,
                   with item: ChecklistItem) {
  let label = cell.viewWithTag(1000) as! UILabel
```

```
    label.text = item.text
  }
```

This sets the checklist item's text on the cell's label. Previously you did that in "`cellForRowAt`" but it's clearer to put that in its own method.

➤ Update `tableView(cellForRowAt)` so that it calls these new methods:

```
  override func tableView(_ tableView: UITableView,
                          cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
                         withIdentifier: "ChecklistItem", for: indexPath)

    let item = items[indexPath.row]

    configureText(for: cell, with: item)
    configureCheckmark(for: cell, with: item)
    return cell
  }
```

➤ Also update `tableView(didSelectRowAt)`:

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

    if let cell = tableView.cellForRow(at: indexPath) {
      let item = items[indexPath.row]
      item.toggleChecked()
      configureCheckmark(for: cell, with: item)
    }
    tableView.deselectRow(at: indexPath, animated: true)
  }
```

This no longer modifies the `ChecklistItem`'s `checked` property directly but calls a new method named `toggleChecked()` on the item object.

You still need to add this new method to the `ChecklistItem` object otherwise the app won't run.

➤ Open **ChecklistItem.swift** and add the following method:

```
  func toggleChecked() {
    checked = !checked
  }
```

Naturally, your own objects can also have methods. As you can see, this method does exactly what "`didSelectRowAt`" used to do, except that you've added this bit of functionality to `ChecklistItem` instead.

A good object-oriented design principle is that you should let objects change their own state as much as possible. Previously, the view controller implemented this toggling behavior but now `ChecklistItem` knows how to toggle itself on or off.

➤ Run the app, and well, it still should work exactly the same as before – but the

code is a lot better. You can now have lists with thousands of to-do items, for those especially industrious users. :-)

If you want to check your work, you can find the project files for the current version of the app in the folder **02 - Arrays** in the tutorial's Source Code folder.

Clean up that mess!

So what's the point of making all of these changes if the app still works exactly the same? For one, the code is much cleaner and that helps to avoid bugs. By using an array you've also made the code more flexible. The table view can now handle any number of rows.

You'll find that when you are programming you are constantly restructuring your code to make it better. It's impossible to do the whole thing 100% perfect right from the start.

So you write code until it becomes messy and then you clean it up. After a little while it becomes a big mess again and you clean it up again. The process for cleaning up code is called *refactoring* and it's a cycle that never ends.

There are a lot of programmers who never do clean up their code. The result is what we call "spaghetti code" and it's a horrible mess to maintain.

If you haven't looked at your code for several months but need to add a new feature or fix a bug, you may need some time to read it through to understand again how everything fits together.

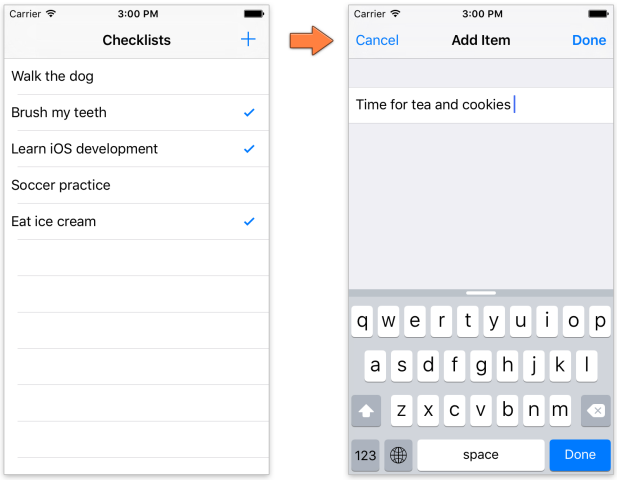
It's in your own best interest to write code that is as clean as possible, otherwise untangling that spaghetti mess is no fun.

Adding new items to the checklist

So far your table view contains a handful of fixed rows but the idea behind this app is that users can create their own lists. Therefore, you need to give the user the ability to add to-do items.

In this section you'll expand the app to have a so-called **navigation bar** at the top. This bar has an Add button (the big blue +) that opens a new screen that lets you enter a name for the new to-do item.

When you tap Done, the new item will be added to the list.



The + button in the navigation bar opens the Add Item screen

Presenting a new screen to add items is a common pattern in a lot of apps. Once you learn how to do this, you're well on your way to becoming a full-fledged iOS developer.

What you'll do in this section:

- Add a navigation controller
- Put the Add button into the navigation bar
- Add a fake item to the list when you press the Add button
- Delete items with swipe-to-delete
- Open the Add Item screen that lets the user type the text for the item

As always, we take it in small steps. After you've put the Add button on the screen, you'll first write the code to add a "fake" item to the list. Instead of writing all of the code for the Add Item screen at once, you simply pretend that some parts of it already exist.

Once you've learned how to add fake items, you can build the Add Item screen for real.

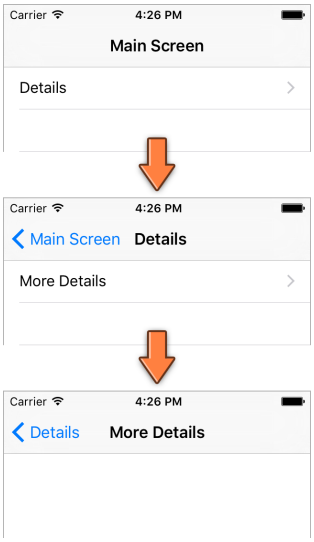
Navigation controllers

First, let's add the navigation bar. You may have seen in the Object Library that there is an object named Navigation Bar. You can drag this into your view and put it at the top. However, you won't do that here.

Instead, you will embed the view controller inside a **navigation controller**.

Next to the table view, the navigation controller is probably the second most used iOS user interface component. It is the thing that lets you go from one page to

another:



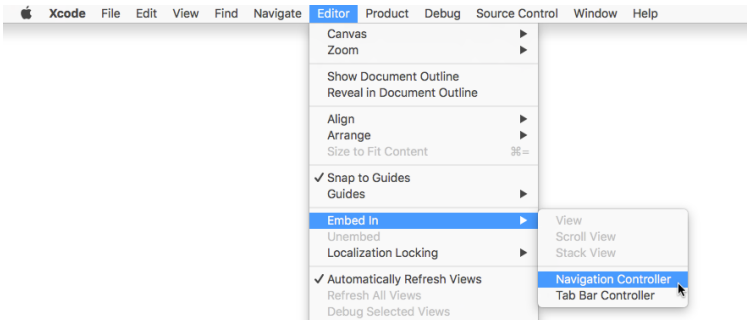
A navigation controller in action

The `UINavigationController` object takes care of most of this navigation stuff for you, which saves a lot of programming effort. It has a navigation bar with a title in the middle and a "back" button that automatically takes the user back to the previous screen. You can put a button of your own on the right.

Adding a navigation controller is really easy.

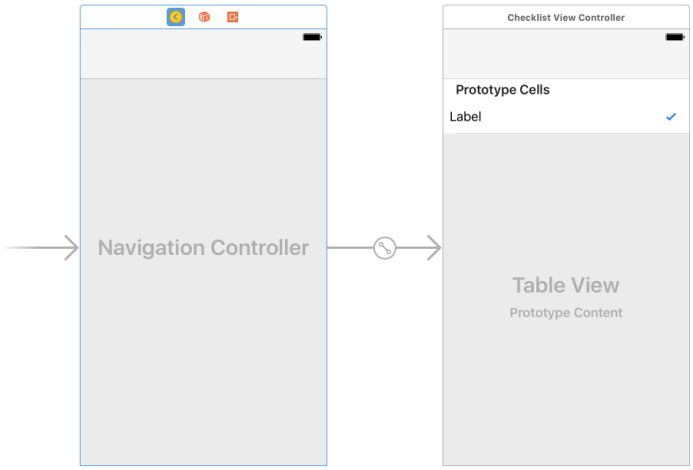
➤ Open **Main.storyboard** and select the **Checklist View Controller**.

➤ From the menu bar at the top of the screen, choose **Editor → Embed In → Navigation Controller**.



Putting the view controller inside a navigation controller

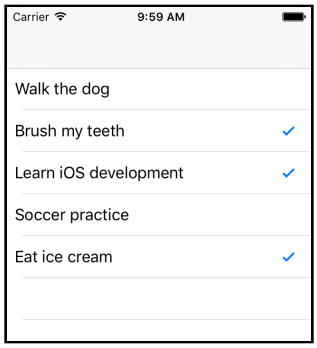
That's it. Interface Builder has now added a new Navigation Controller scene and made a relationship between it and your view controller.



The navigation controller is now linked with your view controller

When the app starts up, the Checklist View Controller is automatically put inside a navigation controller.

➤ Run the app and try it out.

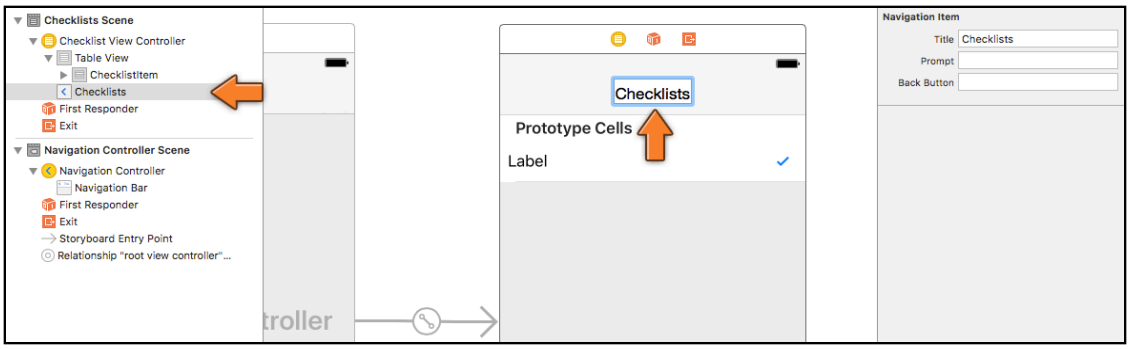


The app now has a navigation bar at the top

The only thing different (visually) is that the app now has a navigation bar at the top. Thanks to this, the status bar no longer overlaps the label from the first cell.

➤ Go back to the storyboard and double-click on the navigation bar inside the Checklist View Controller to make the title editable. (You need to double-click roughly in the center of the navigation bar for this to work.)

Give it the name **Checklists**.



Changing the title in the navigation bar

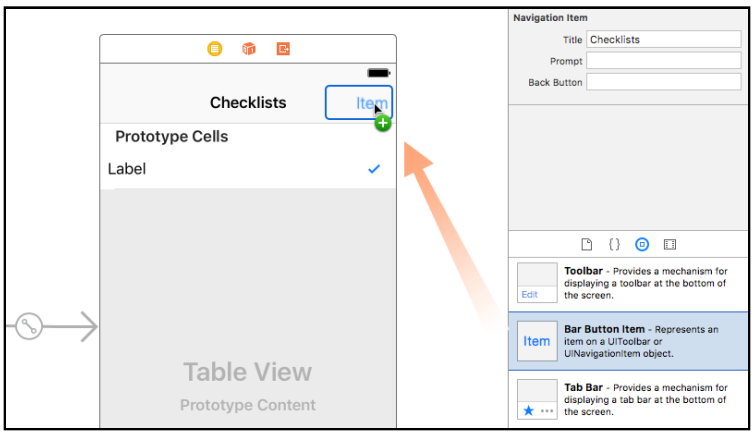
What you're doing here, is changing a **Navigation Item** object that was automatically added to the view controller when you chose the Embed In command.

The Navigation Item object contains the title and buttons that appear in the navigation bar when this view controller becomes active. Each embedded view controller has its own Navigation Item that it uses to configure what shows up inside the navigation bar.

When the navigation controller slides a new view controller into the screen, it replaces the contents of the navigation bar with that view controller's Navigation Item.

➤ Go to the Object Library and look for **Bar Button Item**. Drag it into the right-side slot of the navigation bar.

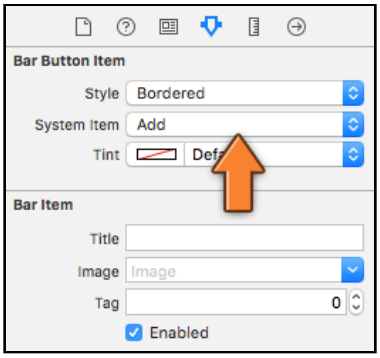
Be sure to use the navigation bar on the Checklist View Controller, not the one from the navigation controller!



Dragging a Bar Button Item into the navigation bar

By default this new button is named "Item" but for this app you want it to have a big + sign.

➤ In the **Attributes inspector** for the bar button item, choose **System Item: Add**.

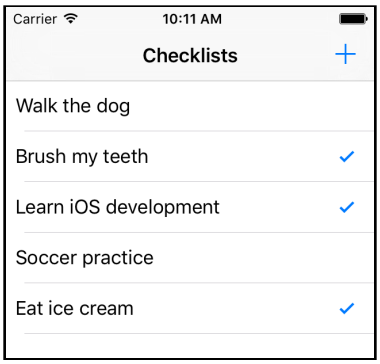


Bar Button Item attributes

If you look through that list you'll see a lot of predefined bar button types: Add, Compose, Reply, Camera, and so on. You can use these in your own apps but only for their intended purpose.

You shouldn't use the camera icon on a button that sends an email, for example. Improper use of these icons may lead Apple to reject your app from the App Store and that sucks.

OK, that gives us a button. If you run the app, it should look like this:



The app with the Add button

Of course, pressing the button doesn't actually do anything yet because you haven't hooked it up to an action. In a little while you will create a new screen, the "Add Item" screen, and show it when the button is tapped. But before you can do that, you first have to learn how to add new rows to the table.

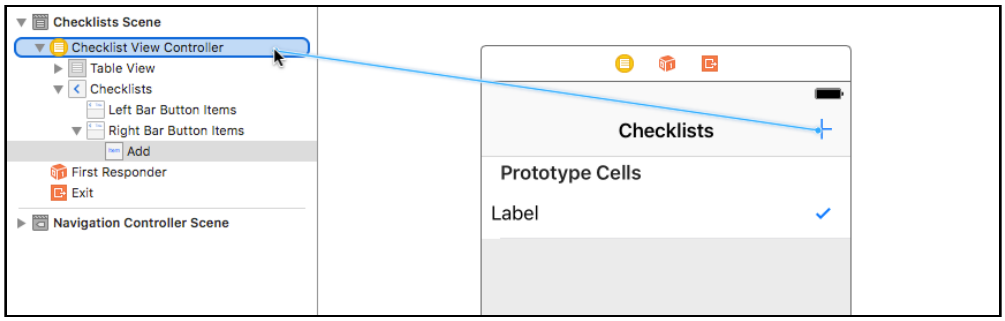
Let's hook up the Add button to an action. You got plenty of exercise with this in the previous tutorial, so this shouldn't be too much of a problem.

➤ Add a new action method to **ChecklistViewController.swift**:

```
@IBAction func addItem() {
}
```

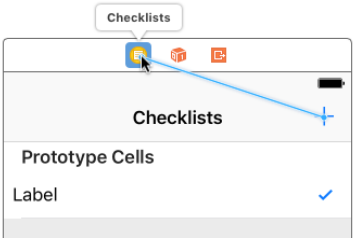
You're leaving this empty for the moment, but it needs to be there so you have something to connect the button to.

➤ Open the storyboard and hook up the Add button to this action. To do this, **Ctrl-drag** from the **+** button to the Checklist View Controller item in the sidebar:



Ctrl-drag from Add button to Checklist View Controller

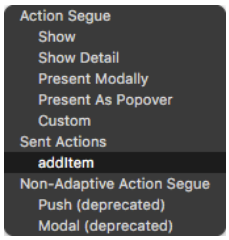
Or, even simpler, Ctrl-drag from the Add button to the yellow circle in the dock area above the scene:



Ctrl-drag from Add button to the view controller (alternative method)

In fact, you can Ctrl-drag from the Add button to almost anywhere into the same scene to make the connection (dragging onto the status bar area is a good spot).

➤ After dragging, pick **addItem** from the popup (under **Sent Actions**):



Connecting to the addItem action

➤ Let's give `addItem()` something to do. Back in **ChecklistViewController.swift**,

fill out the body of that method:

```
@IBAction func addItem() {
  let newRowIndex = items.count

  let item = ChecklistItem()
  item.text = "I am a new row"
  item.checked = false
  items.append(item)

  let indexPath = IndexPath(row: newRowIndex, section: 0)
  let indexPaths = [indexPath]
  tableView.insertRows(at: indexPaths, with: .automatic)
}
```

Inside this method you create a new `ChecklistItem` object and add it to the data model (the `items` array). You also have to tell the table view, "I've inserted a row at this index, please update yourself."

Let's take it section by section:

```
  let newRowIndex = items.count
```

You need to calculate what the index of the new row in your array should be. This is necessary in order to properly update the table view.

When you start the app there are 5 items in the array and 5 rows on the screen. Computers start counting at 0, so the existing rows have indexes 0, 1, 2, 3 and 4. To add the new row to the end of the array, the index for that new row must be 5.

In other words, when you're adding a row to the end of a table view, the index for the new row is always equal to the number of items currently in that table. Let that sink in for a second.

You put the index for the new row in the local constant `newRowIndex`. This can be a constant instead of a variable because it never has to change.

The following few lines should look familiar:

```
  let item = ChecklistItem()
  item.text = "I am a new row"
  item.checked = false
  items.append(item)
```

You have seen this code before in `init?(coder)`. It creates the new `ChecklistItem` object and adds it to the end of the array.

The data model now consists of 6 `ChecklistItem` objects inside the `items` array. Note that at this point `newRowIndex` is still 5 even though `items.count` is now 6. That's why you read the item count and stored this value in `newRowIndex` *before* you added the new item to the array.

Just adding the new `ChecklistItem` object to the data model's array isn't enough.

You also have to tell the table view about this new row so it can add a new cell for that row.

```
    let indexPath = IndexPath(row: newRowIndex, section: 0)
```

As you know by now, table views use index-paths to identify rows, so first you make an `IndexPath` object that points to the new row, using the row number from the `newRowIndex` variable. This index-path object now points to row 5 (in section 0).

The next line creates a new, temporary array holding just the one index-path item:

```
    let indexPaths = [indexPath]
```

You will use the table view method `insertRows(at:with:)` to tell the table view about the new row, but as its name implies this method actually lets you insert multiple rows at the same time.

Instead of a single `IndexPath` object, you need to give it an array of index-paths. Fortunately it is easy to create an array that contains a single index-path object by writing `[indexPath]`. The notation `[]` creates a new `Array` object that contains the objects between the brackets.

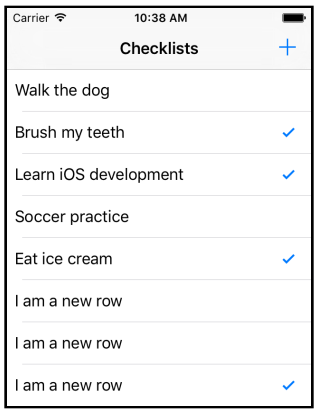
Finally, you tell the table view to insert this new row. The "`with: .automatic`" parameter makes the table view use a nice animation when it inserts the row:

```
    tableView.insertRows(at: indexPaths, with: .automatic)
```

To recap, you:

1. created a new `ChecklistItem` object
2. added it to the data model, and
3. inserted a new cell for it in the table view.

➤ Try it out. You can now add many new rows to the table. You can also tap these new rows to turn their checkmarks on and off again. When you scroll the table up and down, the checkmarks stay with the proper rows.



After adding new rows with the + button

Remember, the rows always have to be added to both your data model and the table view. When you send the `insertRows(at:with:)` message to the table view, you say: "Hey table, my data model has a bunch of new items added to it."

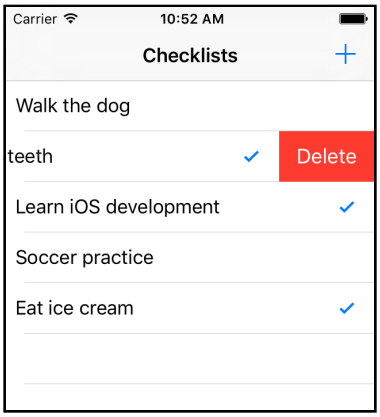
This is important! If you forget to tell the table view about your new items or if you tell the table view there are new items but you don't actually add them to your data model, then your app will crash. These two things always have to be in sync.

Exercise: Give the new items checkmarks by default. ■

Deleting rows

While you're at it, you might as well give users the ability to delete rows.

A common way to do this in iOS apps is "swipe-to-delete". You swipe your finger over a row and a Delete button slides into the screen. A tap on the Delete button confirms the removal, tapping anywhere else will cancel.



Swipe-to-delete in action

Swipe-to-delete is very easy to implement.

➤ Add the following method to **ChecklistViewController.swift**. Just to keep

things organized, I suggest you put this near the other table view methods.

```
override func tableView(_ tableView: UITableView,
                        commit editingStyle: UITableViewCellEditingStyle,
                        forRowAt indexPath: IndexPath) {
  // 1
  items.remove(at: indexPath.row)

  // 2
  let indexPaths = [indexPath]
  tableView.deleteRows(at: indexPaths, with: .automatic)
}
```

When the "`commitEditingStyle`" method is present in your view controller (it comes from the table view data source), the table view will automatically enable swipe-to-delete. All you have to do is:

1. remove the item from the data model, and
2. delete the corresponding row from the table view.

This mirrors what you did in `addItem()`. Again you make a temporary array with only one index-path object and then tell the table view to remove the rows with an animation.

➤ Run the app to try it out!

If at any point you got stuck, you can refer to the project files for the app from the **03 - Data Model** folder in the tutorial's Source Code folder.

Destroying objects

When you do `items.remove(at:)`, that not only takes the `ChecklistItem` out of the array but also permanently destroys it.

We'll talk more about this in the next tutorial, but if there are no more references to an object, it is automatically destroyed. As long as a `ChecklistItem` object sits inside an array, that array has a reference to it.

But when you pull that `ChecklistItem` out of the array, the reference goes away and the object is destroyed. Or in computer-speak, it is *deallocated*.

What does it mean for an object to be destroyed? Each object occupies a small section of the computer's memory. When you create an object instance, a chunk of memory is reserved to hold the object's data items.

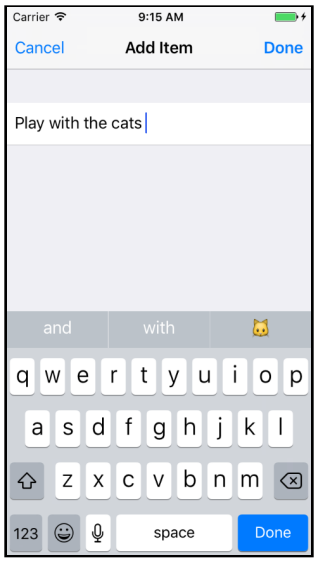
If the object is deallocated, that memory becomes available again and will eventually be occupied by new objects. After it has been deleted, the object does

not exist anymore and you can no longer use it.

On older versions of iOS you had to take care of this memory bookkeeping by hand. Fortunately times have changed for the better. Swift uses a mechanism called Automatic Reference Counting or ARC to manage the lifetime of the objects in your app, freeing you from having to worry about that bookkeeping. I like not having to worry about things!

The Add Item screen

You've learned how to add new rows to the table, but all of these rows get the same text. You will now change the `addItem()` action to open a new screen that lets the user enter his or her own text for those new `ChecklistItems`.



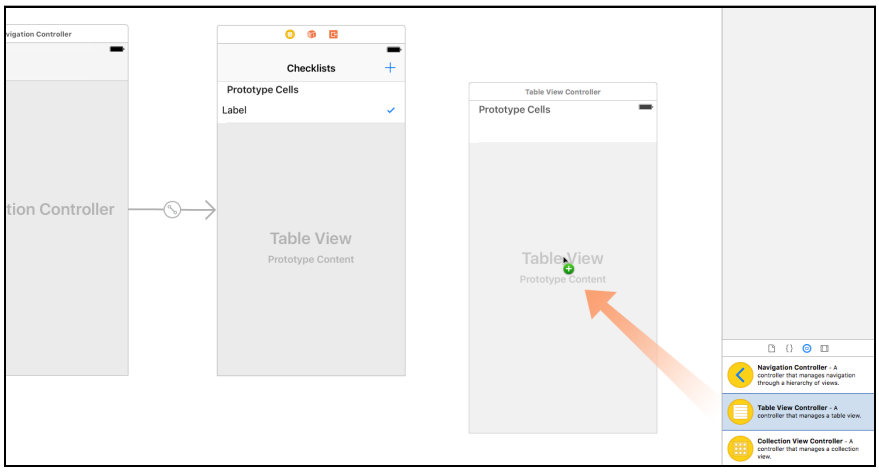
The Add Item screen

The plan for this section:

- Create the Add Item screen using the power of storyboarding
- Add a text field and allow the user to type into it using the on-screen keyboard
- Recognize when the user presses Cancel or Done on the Add Item screen
- Create a new `ChecklistItem` with the text from the text field
- Add the new `ChecklistItem` object to the table on the main screen

A new screen means a new view controller, so you begin by adding a new scene to the storyboard.

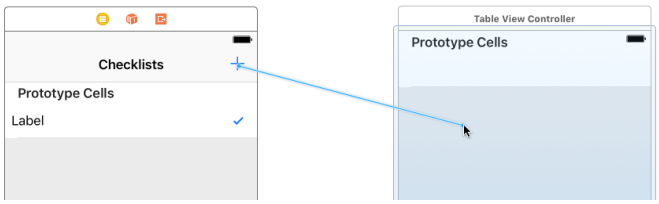
➤ Go to the Object Library and drag a new **Table View Controller** (not a regular view controller) into the storyboard canvas.



Dragging a new Table View Controller into the canvas

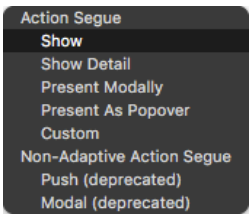
You may need to zoom out to fit everything properly. Right-click on the canvas to get a popup with zoom options, or use the **- 100% +** controls at the bottom of the Interface Builder canvas. (You can also double-click on an empty spot in the canvas to zoom in or out.)

➤ With the new view controller in place, select the **Add button** from the Checklist View Controller. **Ctrl-drag** to the new view controller.



Ctrl-drag from the Add button to the new table view controller

Let go of the mouse and a list of options pops up:



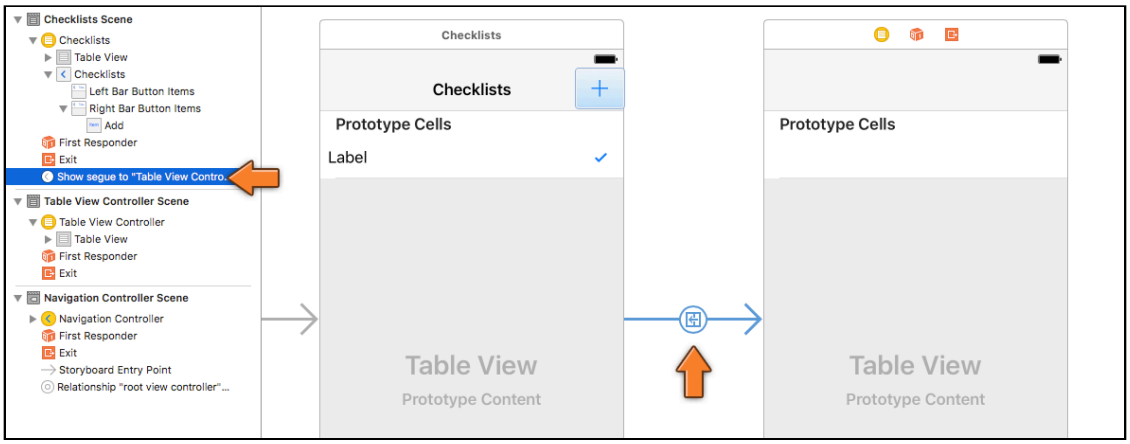
The Action Segue popup

The options in this menu are the different types of connections you can make between the Add button and the new screen.

➤ Choose **Show** from the menu.

This type of connection is named a **segue** (if you're not a native English speaker, that is pronounced "seg-way" like the strange scooters that you can stand on).

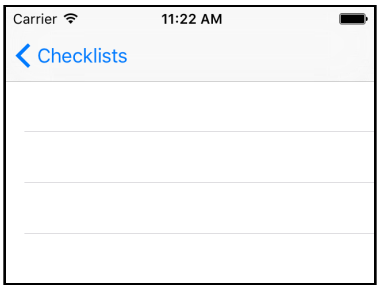
The segue is represented by the arrow between the two view controllers:



A new segue is added between the two view controllers

➤ Run the app to see what it does.

When you press the Add button, a new empty table slides in from the right. You can press the back button – the one that says "Checklists" – at the top to go back to the previous screen.

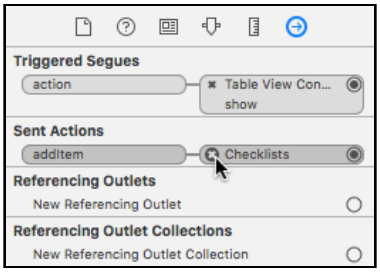


The screen that shows up after you press the Add button

You didn't even have to write any code and you have yourself a working navigation controller!

Note that the Add button no longer adds a new row to the table. That connection has been broken and is replaced by the segue. Just in case, you should remove the button's connection with the `addItem` action.

➤ Select the Add button, go to the **Connections inspector**, and press the small X next to **addItem**.



Removing the addItem action from the Add button

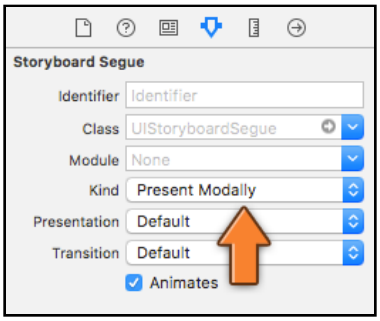
Notice that this inspector also shows the connection with the segue that you've just made (under **Triggered Segues**).

So now you have a new table view controller that slides into the screen when you press the Add button. This isn't actually what you want, though. For a screen that lets you add new items, it is better to use a so-called **modal** segue.

➤ Click the arrow between the two view controllers to select the segue.

A segue is an object like any other (remember, everything is an object!) and as such it has attributes that you can change.

➤ In the **Attributes inspector**, choose **Kind**: **Present Modally**.



Changing the segue style to Present Modally

The navigation bar now disappears from the new view controller. This new screen is no longer presented as part of the navigation hierarchy, but as a separate screen that lies on top of the existing one.

➤ Run the app to see the difference.

When you do, you'll notice that you no longer have a way to go back to the previous screen. Eek! Getting stuck is not something users appreciate…

Modal screens usually have a navigation bar with a Cancel button on the left and a Done button on the right. (In some apps the button on the right is called Save or

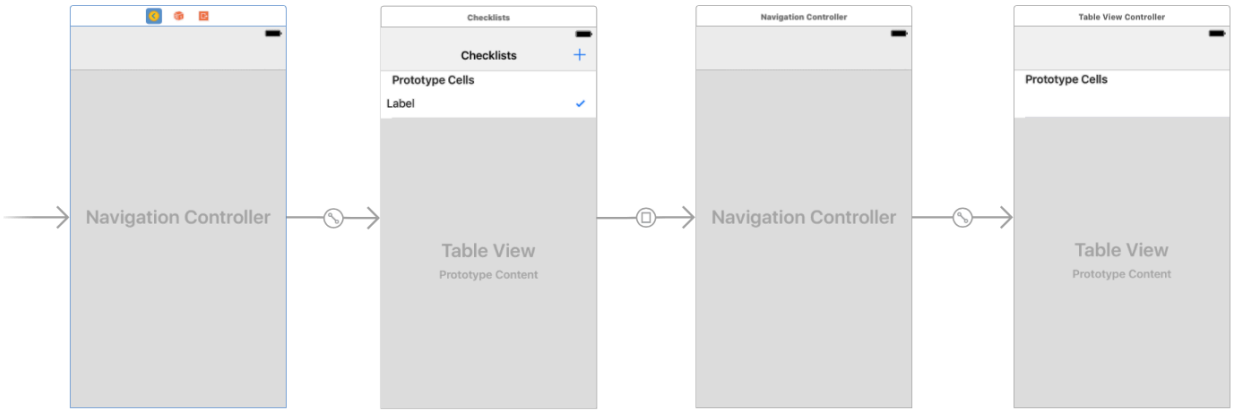
Send.)

Pressing either of these buttons will close the screen, but only Done will save your changes.

The easiest way to add a navigation bar and these two buttons is to wrap the view controller for the Add Item screen into a navigation controller of its own. The steps to do this are the same as before:

➤ Select the table view controller (the new one), choose **Editor → Embed In → Navigation Controller**.

Now the storyboard looks like this:

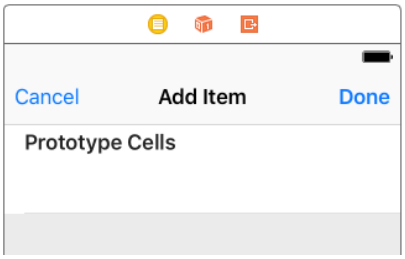


Two table view controllers that are both embedded in their own navigation controllers

The new navigation controller has been inserted in between the two table view controllers. The Add button now performs a modal segue to the new navigation controller.

➤ Double-click the navigation bar in the right-most table view controller to edit its title and change it to **Add Item**. (You can also change this in the Attributes inspector for the Navigation Item.)

➤ Drag two **Bar Button Items** into the navigation bar, one in the left slot and one in the right slot.



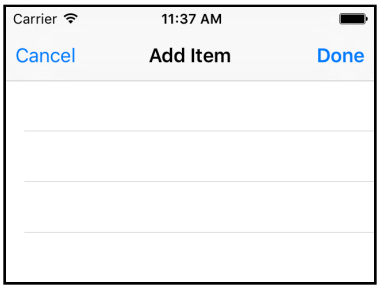
The navigation bar items for the new screen

➤ In the **Attributes inspector** for the left button choose **System Item**: **Cancel**.

➤ For the right button choose **Done** for both **System Item** and **Style** attributes.

Don't type anything into the button's Title field. The Cancel and Done buttons are built-in button types that automatically use the proper text. If your app runs on an iPhone where the language is set to something other than English, these predefined buttons are automatically translated into the user's language.

➤ Run the app and you'll see that your new screen has Cancel and Done buttons.



The Cancel and Done buttons in the app

The new buttons look good! But you still need to tell the app what to do when they get tapped…

> **Note:** Xcode may be giving you the warning, "Prototype table cells must have reuse identifiers". You will fix this issue soon.

Making your own view controller object

The Cancel and Done buttons ought to close the Add Item screen and return the app to the main screen, but tapping them has no effect yet.

In the next tutorial you will learn how to perform such a "backwards" segue directly in the storyboard, but here you will do it by writing code – in other words, you have to hook up these buttons to action methods.

Where do you put these action methods? Not in ChecklistViewController.swift because that is not the view controller you're dealing with here.

Instead, you have to make a new view controller source code file specifically for the Add Item screen and connect it to the scene that you've just designed in Interface Builder.

➤ Right-click on the Checklists group in the project navigator (the yellow icon) and choose **New File...** Choose the **Swift File** template.

➤ Save the file as **AddItemViewController.swift**. This adds the new file to the project, but apart from some comments and a single line of code, the file is empty.

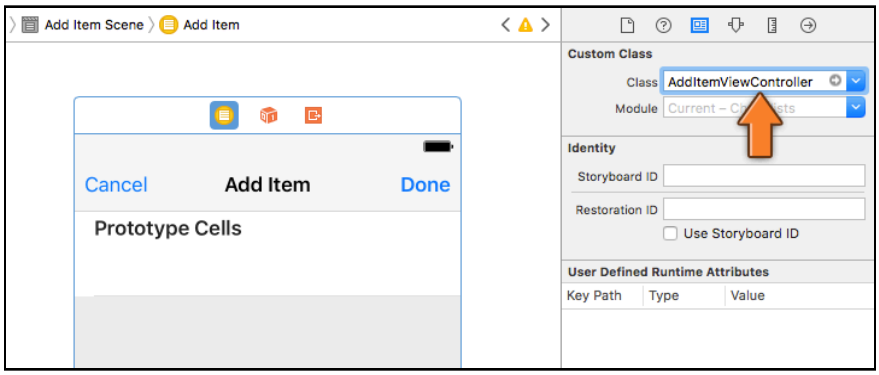
➤ Add the following lines to the new file:

```
import UIKit

class AddItemViewController: UITableViewController {

}
```

This tells Swift that you have a new object for a table view controller that goes by the name of `AddItemViewController`. You'll add the rest of the code soon. First, you have to let the storyboard know about this new view controller too.

➤ In the storyboard, select the table view controller and go to the **Identity inspector**. Under **Custom Class**, type **AddItemViewController**.

This tells the storyboard that the view controller from this scene is actually your new `AddItemViewController` object.



Changing the class name of the AddItemViewController

Don't forget this step! Without it, the Add Item screen will simply not work.

Make sure that it is really the view controller that is selected before you change the fields in the Identity inspector (the scene needs to have a blue border). A common mistake is to select the table view and change that.

You will now implement the action methods in **AddItemViewController.swift**.

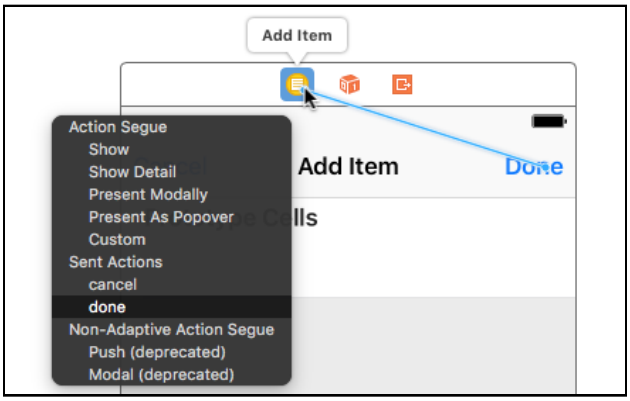
➤ Add the new `cancel()` and `done()` action methods:

```
@IBAction func cancel() {
  dismiss(animated: true, completion: nil)
}

@IBAction func done() {
  dismiss(animated: true, completion: nil)
}
```

This tells the app to close the Add Item screen with an animation.

You still need to hook up the Cancel bar button to the `cancel()` action and the Done bar button to the `done()` action.

➤ Open the storyboard and find the Add Item View Controller. **Ctrl-drag** from the bar buttons to the yellow icon and pick the proper action from the popup menu.



Ctrl-dragging from the bar button to the view controller

➤ Run the app to try it out. The Cancel and Done buttons now return the app to the main screen.

What do you think happens to the `AddItemViewController` object when you dismiss it? After the view controller disappears from the screen, its object is destroyed and the memory it was using is reclaimed by the system.

Every time the user opens the Add Item screen, the app makes a new instance for it. This means a view controller object is only alive for the duration that the user is interacting with it; there is no point in keeping it around afterwards.

★ ★ ★

Container view controllers

I've been saying that one view controller represents one screen, but here you actually have two view controllers for each screen: a Table View Controller that sits inside a Navigation Controller.

The Navigation Controller is a special type of view controller that acts as a container for other view controllers. It comes with a navigation bar and has the ability to easily go from one screen to another, by sliding them in and out of sight. The container essentially "wraps around" these screens.

The Navigation Controller is just the frame that contains the view controllers that do the real work, which are known as the "content" controllers. Here, the `ChecklistViewController` provides the content for the first screen; the content for

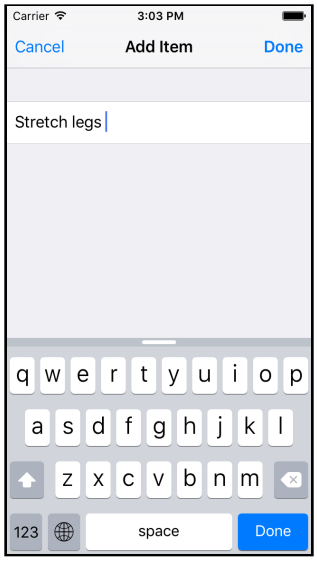
the second screen comes from the `AddItemViewController`.

Another often-used container is the Tab Bar Controller, which you'll see in the next tutorial.

On the iPad, container view controllers are even more commonplace. View controllers on the iPhone are full-screen but on the iPad they often occupy only a portion of the screen, such as the content of a popover or one of the panes in a split-view.

Static table cells

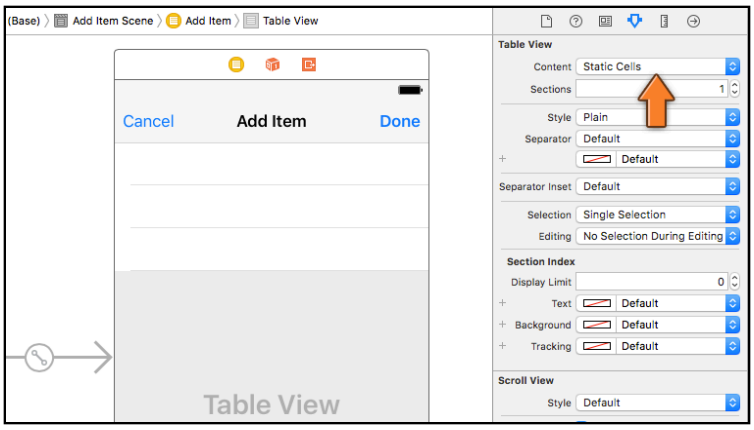
Let's change the look of the Add Item screen. Currently it is an empty table with a navigation bar on top, but I want it to look like this:



What the Add Item screen will look like when you're done

➤ Open the storyboard and select the **Table View** object inside the Add Item View Controller.

➤ In the **Attributes inspector**, change the **Content** setting from Dynamic Prototypes to **Static Cells**.

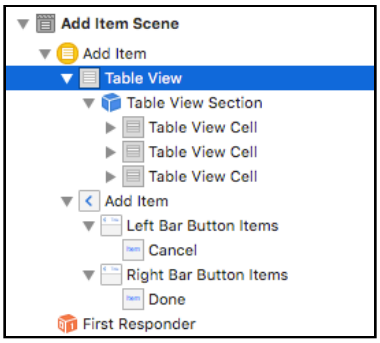


Changing the table view to static cells

You use static cells when you know beforehand how many sections and rows the table view will have. This is handy for screens that require the user to enter data, such as the one you're building here.

You can design the rows directly in the storyboard. For a table with static cells you don't need to provide a data source, and you can hook up the labels and other controls from the cells directly to outlets on the view controller.

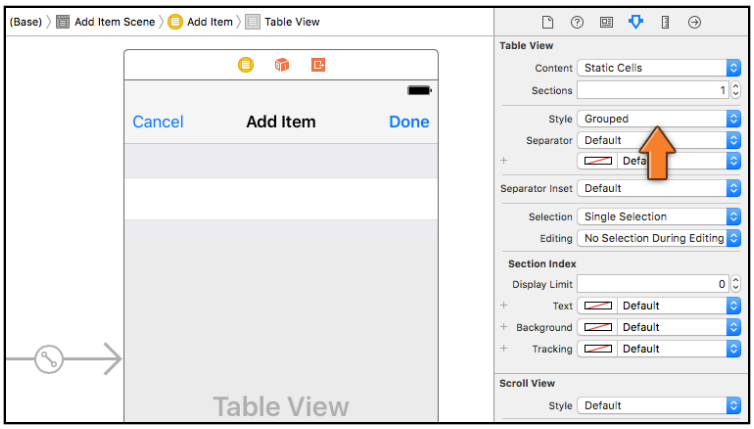
As you can see in the outline pane on the left, the table view now has a Table View Section object hanging under it, and three Table View Cells in that section. (You may need to expand the Table View item first by clicking the arrow next to it.)



The table view has a section with three static cells

➤ Click on the bottom two cells and delete them (press the **delete** key on your keyboard). You only need one cell for now.

➤ Select the Table View again and in the **Attributes inspector** set its **Style** to **Grouped**. That gives us the look we want.

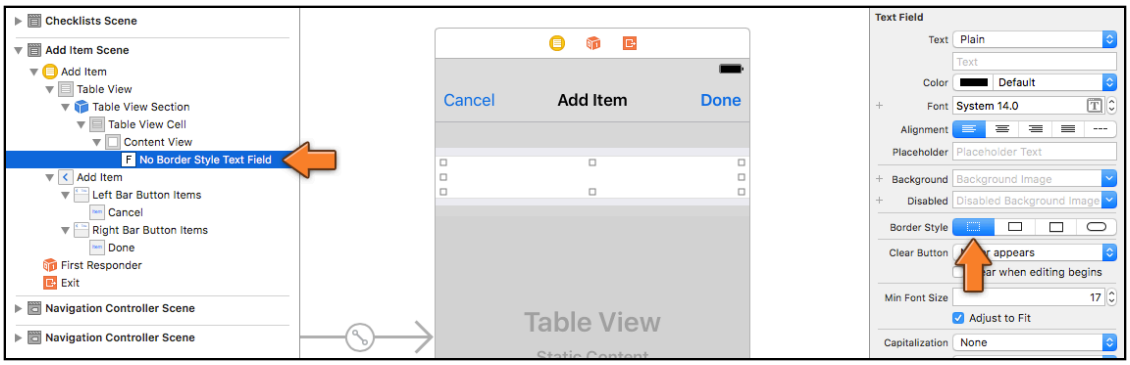


The table view with grouped style

Next up, you'll add a text field component inside the table view cell that lets the user type text.

➤ Drag a **Text Field** object into the cell and size it up nicely.

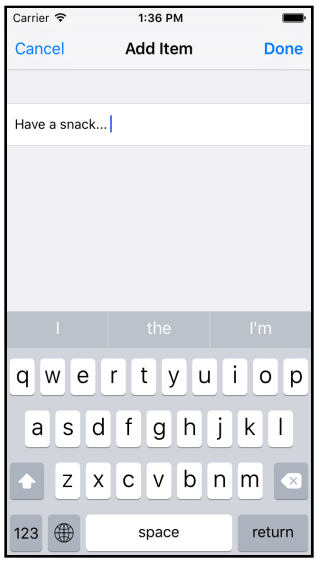
➤ In the **Attributes inspector** for the text field, set the **Border Style** to **no border** (select the dotted box):



Adding a text field to the table view cell

➤ Run the app and press the + button to open the Add Item screen. Tap on the cell and you'll see the keyboard slide in from the bottom of the screen.

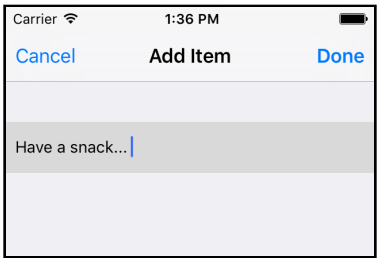
Any time you make a text field active, the keyboard automatically appears. You can type into the text field by tapping on the letters. (On the Simulator, you can simply type using your Mac's keyboard.)



You can now type text into the table view cell

> **Note:** If the keyboard does not appear in the Simulator, press ⌘**K** or use the **Hardware → Keyboard → Toggle Software Keyboard** menu option. You can also use your normal Mac keyboard to type into the text field, even if the on-screen keyboard is not visible. If that doesn't work, also select **Hardware → Keyboard → Connect Hardware Keyboard** from the menu.

Look what happens when you tap just outside the text field's area, but still in the cell (try tapping in the margins that surround the text field):



Whoops, that looks a little weird

The row turns gray because you selected it. That's not what you want, so you should disable selections for this row.

➤ In **AddItemViewController.swift***, add the following method:

```
override func tableView(_ tableView: UITableView,
                        willSelectRowAt indexPath: IndexPath) -> IndexPath? {
  return nil
}
```

This is another one of those table view delegate methods. When the user taps in a

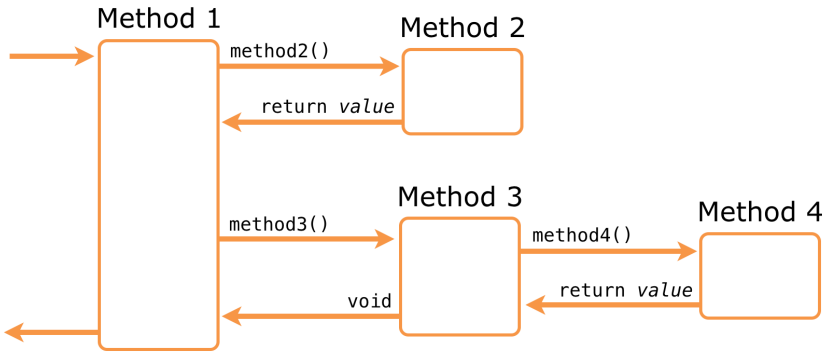
cell, the table view sends the delegate a "`willSelectRowAt`" message that says: "Hi delegate, I am about to select this particular row."

By returning the special value `nil`, the delegate answers: "Sorry, but you're not allowed to!"

Return to sender

You've seen the `return` statement a few times now. You use `return` to send a value from a method back to the method that called it.

Let's take a more detailed look at what it does.



Methods call other methods and receive values in return.

You cannot just return any value. The value you return must be of the data type that is specified after the –> arrow that follows the method name.

For example, `tableView(numberOfRowsInSection)` must return an `Int` value, which is any whole number:

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

If instead you were to write,

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

then the compiler would give an error message, as "1" is a string, not an Int. To a human reader they look similar and you can easily understand the intent, but Swift isn't that tolerant. Data types have to match or it just isn't allowed.

Your most recent version of this method looks like this:

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

That is also a valid return statement because items is an Array and the count property from Array also has the type Int.

The tableView(cellForRowAt) method is supposed to return a UITableViewCell object:

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

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

The local constant cell contains a UITableViewCell object, so it's OK to return the value of cell from the method.

The tableView(willSelectRowAt) method is supposed to return an IndexPath object. However, you can also make it return "nil", which means no object.

```
override func tableView(_ tableView: UITableView,
                  willSelectRowAt indexPath: IndexPath) -> IndexPath? {
  return nil
}
```

That's what the ? behind -> IndexPath? is for: The question mark tells Swift that you can also return nil from this method. That is only allowed if there is a question mark (or exclamation point) behind the return type.

The special value nil represents "no value" but it's used to mean different things throughout the iOS SDK. Sometimes it means "nothing found" or "don't do anything". Here it means that the row should not be selected when the user taps it.

How do you know what nil means for a certain method? You can find that in the documentation of the method in question.

In the case of "willSelectRowAt", the iOS documentation says:

> Return Value: An index-path object that confirms or alters the selected row.

> Return an IndexPath object other than indexPath if you want another cell to be selected. Return nil if you don't want the row selected.

This means you can either:

1. Return the same index-path you were given. This confirms that this row can be selected.
2. Return another index-path in order to select another row.
3. Return `nil` to prevent the row from being selected, which is what you did.

So remember, you need to use the `return` statement to exit a method that expects to return something. If you forget, then Xcode will give the following error: "Missing return in a function expect to return".

You've also seen methods that do not return anything:

```
@IBAction func addItem()
```

and:

```
func configureCheckmark(for cell: UITableViewCell,
                        with item: ChecklistItem)
```

These methods do not have an –> arrow. Such a method does not pass a value back to the caller and therefore does not need a `return` statement. (You can still use `return` to exit from such methods but it may not be followed by a value.)

Strictly speaking, even methods without a return type *do* return a value, the so-called *empty tuple*. Think of this as a special object that embodies the concept of "nothing". (Don't confuse this with *nil*, which is an actual value.)

You sometimes see this written as:

```
func methodThatDoesNotReturnValue() -> ()

func anotherMethodThatDoesNotReturnValue() -> Void
```

The notation for an empty tuple is `()`, so in this context the parentheses mean there is no return value. The term `Void` is a synonym for `()`.

But really, if a method does not return anything it's just as easy to leave out the –> arrow. It's a rule that `@IBAction` methods never return a value.

There is one more thing you need to do to prevent the row from going gray. It's already impossible to select the row, as you've just told the table view you won't allow it.

However, the cell also has a selection color property. Even if you make it impossible for the row to be selected, sometimes UIKit still briefly draws the cell gray when you tap it. Therefore it is best to also disable this selection color.

➤ In the storyboard, select the table view cell and go to the **Attributes inspector**. Set the **Selection** attribute to **None**.

Now if you run the app, it is impossible to select the row and make it turn gray. Try and prove me wrong! :-)

Reading from the text field

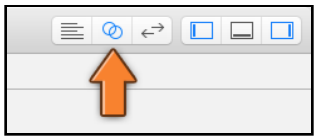
You have a text field in a table view cell that the user can type into, but how do you read the text that the user has typed?

When the user taps Done, you need to get that text and somehow put it into a new `ChecklistItem` and add it to the list of to-do items. This means the `done()` action needs to be able to refer to the text field.

You already know how to refer to controls from within your view controller: use an outlet. When you added outlets in the previous tutorial, I told you to type in the `@IBOutlet` declaration in the source file and make the connection in the storyboard.

I'm going to show you a trick now that will save you some typing. You can let Interface Builder do all of this automatically by Ctrl-dragging from the control in question directly into your source code file.

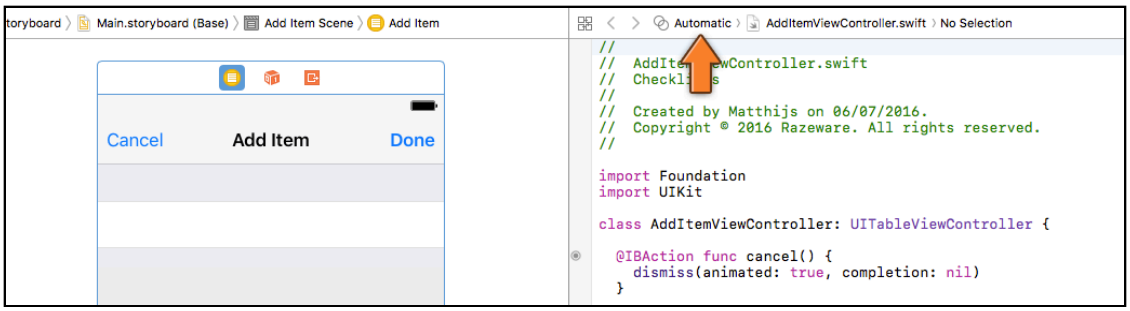
➤ First, go to the storyboard and select the **Add Item View Controller**. Then open the **Assistant editor** using the toolbar button. This button looks like two circles:



Click the toolbar button to open the Assistant editor

This may make the screen a little crowded – there are now five horizontal panels open. If you're running out of space you might want to close the project navigator and the utilities pane using the other toolbar buttons.

The Assistant editor opens a new pane on the right of the screen. In the Jump Bar (the bar below the toolbar) it should say **Automatic** and the Assistant editor should be displaying the **AddItemViewController.swift** file:

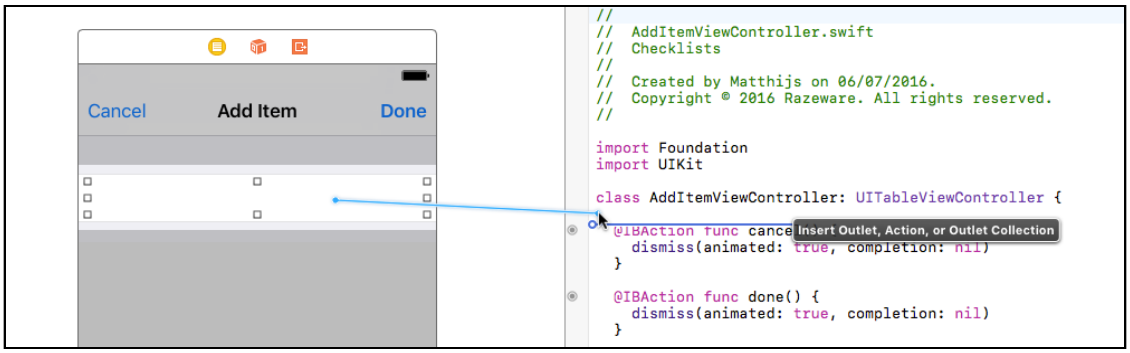


The Assistant editor

"Automatic" means the Assistant editor figures out what other file is related to the one you're currently editing. When you're editing the Storyboard, the related file is the selected view controller's Swift file.

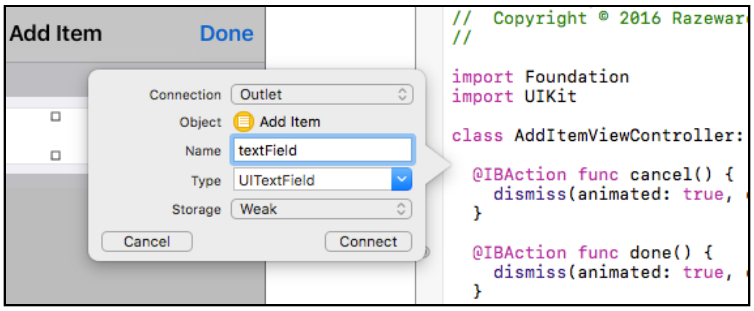
(Sometimes Xcode can be a little dodgy here. If it shows you something other than AddItemViewController.swift, then click in the Jump Bar to select that file.)

➤ With the storyboard and the Swift file side by side, select the text field. Then **Ctrl-drag** from the text field into the Swift file:



Ctrl-dragging from the text field into the Swift file

When you let go, a popup appears:



The popup that lets you add a new outlet

➤ Choose the following options:

- Connection: Outlet
- Name: **textField**
- Type: UITextField
- Storage: Weak

> **Note:** If "Type" does not say UITextField but UITableViewCell or UIView, then you selected the wrong thing.
>
> Make sure you're Ctrl-dragging from the text field inside the cell, not the cell itself. Granted, it's kinda hard to see, being white on white. If you're having trouble selecting the text field, click that area several times in succession.
>
> You can also Ctrl-drag from "No Border Style Text Field" in the outline pane.

➤ Press **Connect** and voila, Xcode has automatically inserted an `@IBOutlet` for you and connected it to the text field object.

In code it looks like this:

```
@IBOutlet weak var textField: UITextField!
```

Just by dragging you have successfully hooked up the text field object with a new property named `textField`. How easy was that!

Now you'll modify the `done()` action to write the contents of this text field to the Xcode debug area, the pane at the bottom of the screen where `print()` messages show up. This is a quick way to verify that you can actually read what the user typed.

➤ In **AddItemViewController.swift**, change `done()` to:

```
@IBAction func done() {
  print("Contents of the text field: \(textField.text!)")

  dismiss(animated: true, completion: nil)
}
```

You can make these changes directly inside the Assistant editor. It's very handy that you can edit the source code and the storyboard side-by-side.

➤ Run the app, press the + button and type something in the text field. When you press Done, the Add Item screen should close and Xcode should reveal the Debug pane with a message like this:

```
Contents of the text field: Hello, world!
```

Great, so that works. `print()` should be an old friend by now. It's my faithful debugging companion.

Recall that you can print the value of a variable by placing it inside `\( … )` in the string. Here you used `\(textField.text!)` to print out the contents of the text field's `text` property. (I'll explain what the exclamation point is for later.)

> **Note:** Because the iOS Simulator already outputs a lot of debug messages of its own, it may be a bit hard to find your `print()` messages in the Debug pane. Luckily there is a Filter box at the bottom that lets you search for your own messages.

Polishing it up

Before you'll write the code to take the text and insert it as a new item into the list, let's improve the design and workings of the Add Item screen a little.

For instance, it would be nice if you didn't have to tap into the text field in order to bring up the keyboard. It would be more convenient if the keyboard automatically appeared once the screen opens.

➤ To accomplish this, add a new method to **AddItemViewController.swift**, `viewWillAppear()`:

```
override func viewWillAppear(_ animated: Bool) {
  super.viewWillAppear(animated)
  textField.becomeFirstResponder()
}
```

The view controller receives the `viewWillAppear()` message just before it becomes visible. That is a perfect time to make the text field active. You do this by sending it the `becomeFirstResponder()` message.

If you've done programming on other platforms, this is often called "giving the control focus". In iOS terminology, the control becomes the *first responder*.

➤ Run the app and go to the Add Item screen; you can start typing right away.

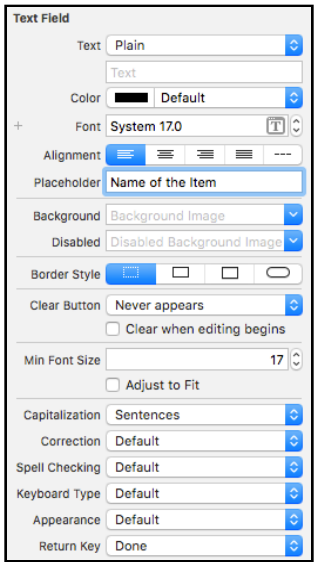
(Again note that the keyboard may not appear on the Simulator. Press **⌘+K** to bring it up. The keyboard will always appear when you run the app on an actual device, though.)

It's often little features like this that make apps a joy to use. Having to tap on the text field before you can start typing gets old really fast. In this fast-paced age, using their mobiles on the go, users don't have the patience for that. Such minor annoyances may be reason enough to switch to a competitor's app. I always put a lot of effort into making my apps as frictionless as possible.

With that in mind, let's style the input field a bit.

➤ Open the storyboard and select the text field. Go to the **Attributes inspector** and set the following attributes:

- Placeholder: **Name of the Item**
- Font: System 17
- Adjust to Fit: Uncheck this
- Capitalization: Sentences
- Return Key: Done



The text field attributes

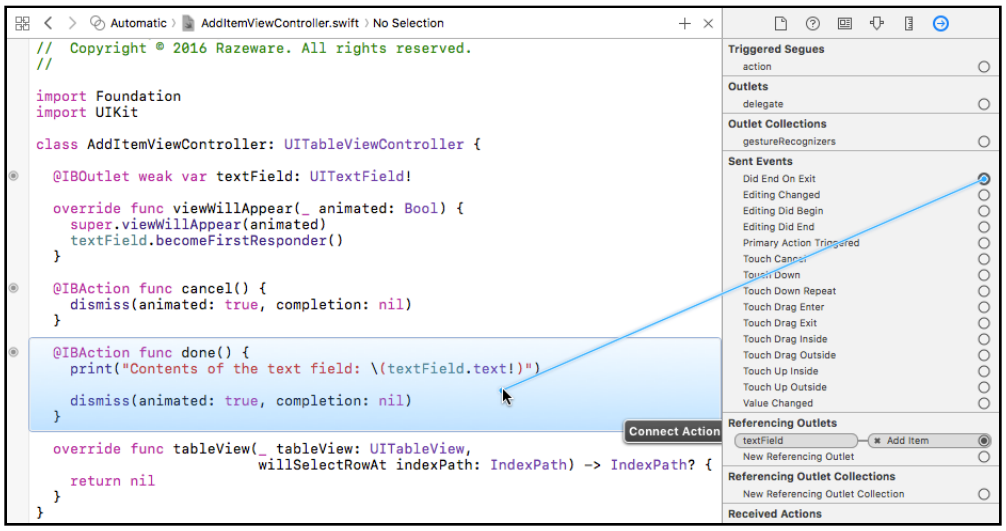
There are several options here that let you configure the keyboard that appears when the text field becomes active.

If this were a field that only allowed numbers, for example, you would set the Keyboard Type to Number Pad. If it were an email address field, you'd set it to E-mail Address. For our purposes, the Default keyboard is appropriate.

You can also change the text that is displayed on the keyboard's Return Key. By default it says "return" but you set it to "Done". This is just the text on the button; it doesn't automatically close the screen. You still have to make the keyboard's Done button trigger the same action as the Done button from the navigation bar.

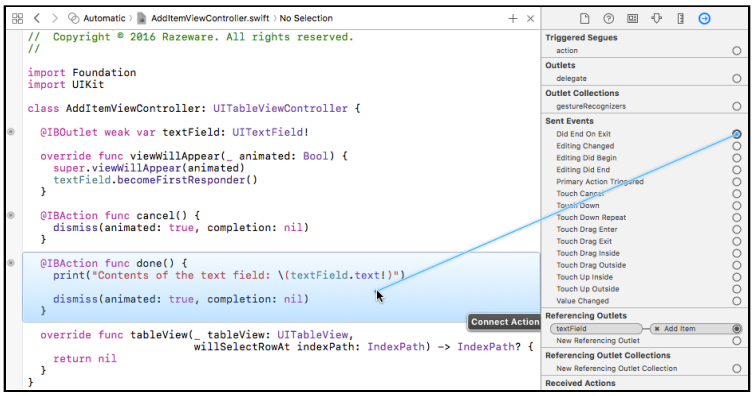
➤ Make sure the text field is selected and open the **Connections inspector**. Drag from the **Did End on Exit** event to the view controller and pick the **done** action.

If you still have the Assistant editor open, you can also drag directly to the source code for the `done()` method:



Connecting the text field to the done() action method

To see the connections for the `done` action, click on the circle in the margin next to the method name. The popup shows that `done()` is now connected to both the bar button and the text field:



Viewing the connections for the done() method

➤ Run the app. Pressing Done on the keyboard will now close the screen and print the text to the debug area.

The keyboard now has a big blue Done button

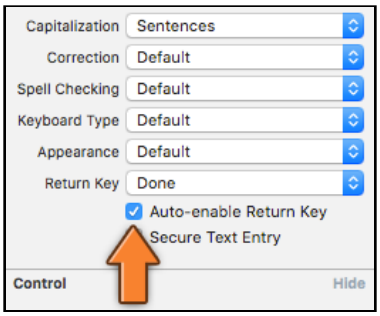
It's always good to validate the input from the user to make sure what they're entering is acceptable. For instance, what should happen if the user immediately taps the Done button on the Add Item screen without entering any text?

Adding a to-do item to the list that has no text is not very useful, so in order to prevent this you should disable the Done button when no text has been typed yet.

Of course, you have two Done buttons to take care of, one on the keyboard and one in the navigation bar. Let's start with the Done button from the keyboard as this is the simplest one to fix.

➤ On the **Attributes inspector** for the text field, check **Auto-enable Return Key**.

That's it. Now when you run the app the Done button on the keyboard automatically is disabled when there is no text in the text field. Try it out!



The Auto-enable Return Key option disables the return key when there is no text

For the Done button in the navigation bar you have to do a little more work. You have to check the contents of the text field after every keystroke to see if it is now empty or not. If it is, then you disable the button.

The user can always press Cancel, but Done only works when there is text.

In order to listen to changes to the text field – which may come from taps on the keyboard but also from cut/paste – you need to make the view controller a *delegate* for the text field.

The text field will send events to this delegate to let it know what is going on. The delegate, which will be the `AddItemViewController`, can then respond to these events and take appropriate actions.

A view controller is allowed to be the delegate for more than one object. The `AddItemViewController` is already a delegate (and data source) for the `UITableView` (because it is a `UITableViewController`). Now it will also become the delegate for the text field object, `UITextField`.

These are two different delegates and you make the view controller play both roles. Later in this tutorial you'll add even more delegates.

How to become a delegate

Delegates are used everywhere in the iOS SDK, so it's good to remember that it always takes three steps to become someone's delegate.

1. You declare yourself capable of being a delegate. To become the delegate for `UITextField` you need to include `UITextFieldDelegate` in the `class` line for the view controller. This tells the compiler that the view controller can actually handle the notification messages that the text field sends to it.
2. You let the object in question, in this case the `UITextField`, know that the view controller wishes to become its delegate. If you forget to tell the text field that it has a delegate, it will never send you any notifications.
3. Implement the delegate methods. It makes no sense to become a delegate if you're not responding to the messages you're being sent!

Often, delegate methods are optional, so you don't need to implement all of them. For example, `UITextFieldDelegate` actually declares seven different methods but you only care about `textField(shouldChangeCharactersIn, replacementString)` for this app.

➤ In **AddItemViewController.swift**, add `UITextFieldDelegate` to the class declaration:

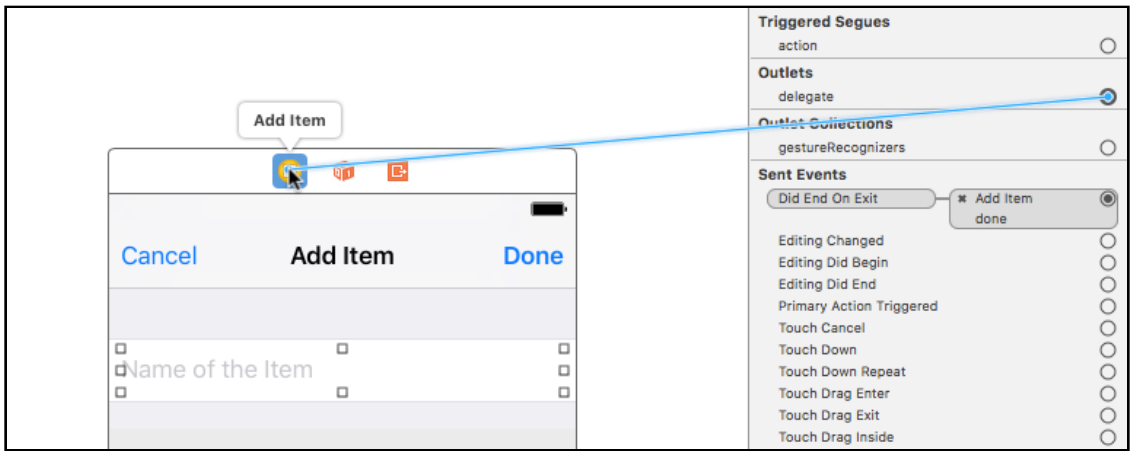
```
class AddItemViewController: UITableViewController, UITextFieldDelegate
```

The view controller now says, "I can be a delegate for text field objects."

You also have to let the text field know that you have a delegate for it.

➤ Go to the storyboard and select the text field.

There are several different ways in which you can hook up the text field's delegate outlet to the view controller. I prefer to go to its **Connections inspector** and drag from **delegate** to the view controller's little yellow icon:



Drag from the Connections inspector to connect the text field delegate

You also have to add an outlet for the Done bar button item, so you can send it messages from within the view controller in order to enable or disable it.

➤ Open the **Assistant editor** and make sure **AddItemViewController.swift** is visible in the assistant pane.

➤ **Ctrl-drag** from the Done bar button into the Swift file and let go. Name the new outlet `doneBarButton`.

This adds the following property:

```
@IBOutlet weak var doneBarButton: UIBarButtonItem!
```

➤ Add the following to **AddItemViewController.swift**, at the bottom:

```
func textField(_ textField: UITextField,
               shouldChangeCharactersIn range: NSRange,
               replacementString string: String) -> Bool {

  let oldText = textField.text! as NSString
  let newText = oldText.replacingCharacters(in: range, with: string)
                                          as NSString
  if newText.length > 0 {
    doneBarButton.isEnabled = true
  } else {
    doneBarButton.isEnabled = false
  }
  return true
}
```

This is one of the `UITextField` delegate methods. It is invoked every time the user changes the text, whether by tapping on the keyboard or by cut/paste.

First, you figure out what the new text will be:

```
  let oldText = textField.text! as NSString
  let newText = oldText.replacingCharacters(in: range, with: string)
```

```
                                                    as NSString
```

The `textField(shouldChangeCharactersIn, replacementString)` delegate method doesn't give you the new text, only which part of the text should be replaced (the range) and the text it should be replaced with (the replacement string).

You need to calculate what the new text will be by taking the text field's text and doing the replacement yourself. This gives you a new string object that you store in the `newText` constant.

NSString vs. String

Text strings in Swift are of the data type `String`. But in the above method you used something called `NSString`. What is the difference between the two?

`NSString` is the object that Objective-C programmers use for storing text. To be honest, it is more powerful and often easier to use than Swift's own `String`.

However, Swift has a trick up its sleeve: `String` and `NSString` are "bridged", meaning that you can use `NSString` in place of `String`. Here, you want to use `NSString`'s `replacingCharacters(in:with:)` method, so you let Swift know that it should treat the text as an `NSString`, not as a `String`.

The notation `let oldText = . . . as NSString` tells Swift that `oldText` should be a constant of type `NSString`. If you were to leave out the "`as NSString`" bit, Swift would use type inference to determine that it should be a regular Swift `String` instead, which isn't what you want here.

By the way, `String` isn't the only thing that is bridged to an Objective-C type. Another example is `Array` and its Objective-C counterpart `NSArray`. Because the iOS frameworks are written in different language than Swift, sometimes these little Objective-C holdovers creep in.

Once you have the new text, you check if it's empty by looking at its length, and enable or disable the Done button accordingly:

```
if newText.length > 0 {
  doneBarButton.isEnabled = true
} else {
  doneBarButton.isEnabled = false
}
```

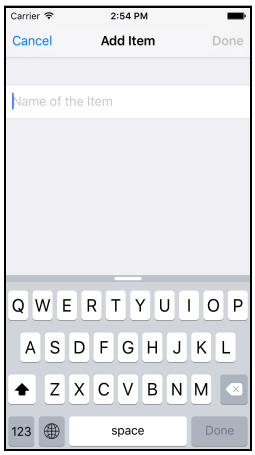
➤ Run the app and type some text into the text field. Now remove that text and you'll see that the Done button in the navigation bar properly gets disabled when the text field becomes empty.

One problem: The Done button is initially enabled when the Add Item screen opens, but there is no text in the text field at that point so it really should be disabled. This is simple enough to fix.

➤ In the storyboard, select the **Done** bar button and go to the **Attributes**

inspector. Uncheck the **Enabled** box.

The Done buttons are now properly disabled when there is no text in the text field:



You cannot press Done if there is no text

There is actually a slightly simpler way to write the above method.

➤ Replace the if-statement with just this line:

```
doneBarButton.isEnabled = (newText.length > 0)
```

The if-statement used to do this:

```
if newText.length > 0 {
   // you get here if the length is greater than 0
} else {
   // you get here if the length is equal to 0
}
```

You check the condition `newText.length > 0`. If that condition is true, i.e. the text length is greater than 0, you set `doneBarButton`'s `isEnabled` property to `true`. If the condition is false, you set the `isEnabled` property to `false`.

Notice that these sentences are basically saying: if the condition is true then `isEnabled` becomes `true` but if the condition is false then `isEnabled` becomes `false`. In other words, you always set the `isEnabled` property to the result of the condition: true or false.

That makes it possible to skip the `if`, and simply do,

```
doneBarButton.isEnabled = the result of the condition
```

which in Swift reads as follows:

```
doneBarButton.isEnabled = (newText.length > 0)
```

The ( ) parentheses are not really necessary. You can also write it like this:

```
doneBarButton.isEnabled = newText.length > 0
```

However, I find this slightly less readable, so I use the parentheses to make it clear beyond a doubt that `newText.length > 0` is evaluated first and that the assignment takes place after that.

To recap: If `newText.length` is greater than 0, `doneBarButton.isEnabled` becomes `true`; otherwise it becomes `false`.

You can fit this into a single statement because the comparison operators all return true or false depending on the condition. These are Swift's comparison operators:

>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal
!=	not equal

Remember this trick – whenever you see code like this,

```
if some condition {
  something = true
} else {
  something = false
}
```

you can write it simply as:

```
something = (some condition)
```

In practice it doesn't really matter which version you use. I prefer the shorter one; that's what the pros do. Just remember that comparison operators such as == and > always return `true` or `false`, so the extra `if` really isn't necessary.

Adding new ChecklistItems

You now have an Add Item screen with a keyboard that lets the user enter text. The app also properly validates the input so that you'll never end up with text that is empty.

But how do you get this text into a new `ChecklistItem` object that you can add to the array from the Checklists screen?

Somehow you'll have to make the Add Item screen let the Checklist View Controller

know that it is done. This is one of the fundamental tasks that every iOS app needs to do: sending messages from one view controller to another.

Sending a ChecklistItem object to the screen with the items array

Exercise: How would you tackle this problem? The `done()` method needs to create a new `ChecklistItem` object with the text from the text field (easy), then add it to the `items` array and the table view in `ChecklistViewController` (not so easy). ■

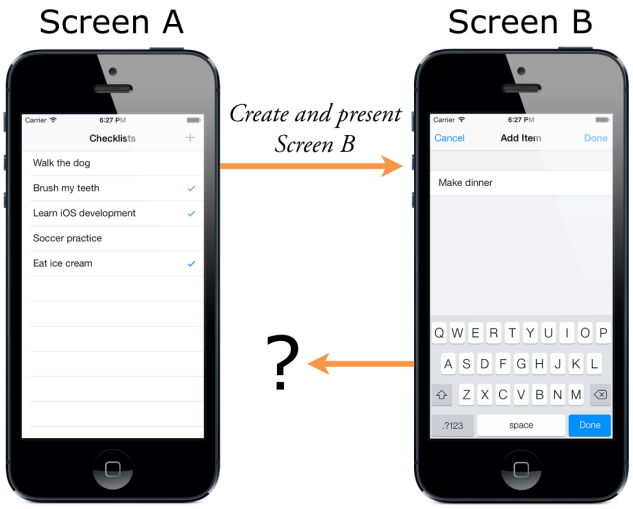
Maybe you came up with something like this:

```
class AddItemViewController: UITableViewController, . . . {

  // This variable refers to the other view controller
  var checklistViewController: ChecklistViewController

  @IBAction func done() {
    // Create the new checklist item object
    let item = ChecklistItem()
    item.text = textField.text!

    // Directly call a method from ChecklistViewController
    checklistViewController.add(item)
  }
}
```

In this scenario, `AddItemViewController` has a variable that refers to the `ChecklistViewController`, and `done()` calls its `add()` method with the new `ChecklistItem` object.

This will work, but it's not the iOS way. The big downside of this approach is that it shackles these two view controller objects together.

As a general principle, if screen A launches screen B then you don't want screen B to know too much about the screen that invoked it (A). The less B knows of A, the better.



Screen A knows all about screen B, but B knows nothing of A

Giving `AddItemViewController` a direct reference to `ChecklistViewController` prevents you from opening the Add Item screen from somewhere else in the app. It can only ever talk back to `ChecklistViewController`. That's a big disadvantage.

You won't actually need to do this in the Checklists app, but in many apps it's common for one screen to be accessible from multiple places. For example, a login screen that appears after the user has been logged out due to inactivity. Or a details screen that shows more information about a tapped item, no matter where that item is located in the app (you'll see an example of this in the next tutorial).

Therefore, it's best if `AddItemViewController` doesn't know anything about `ChecklistViewController`.

But if that's the case, then how can you make the two communicate?

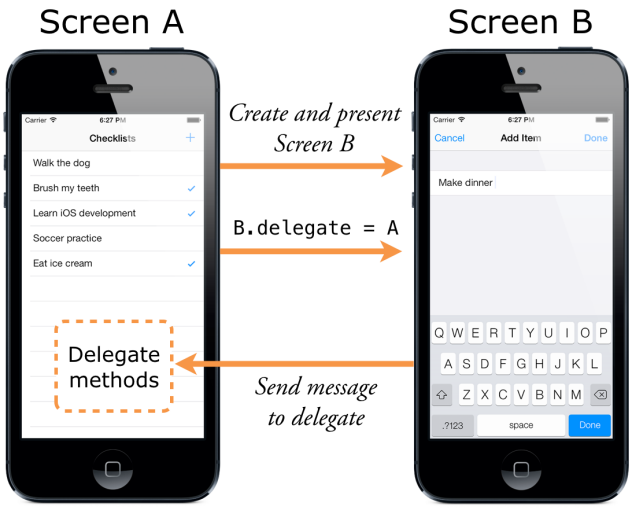
The solution is to make your own *delegate*.

You've already seen delegates in a few different places: The table view has a delegate that responds to taps on the rows. The text field has a delegate that you used to validate the length of the text. And the app also has something named the `AppDelegate` (see the project navigator).

You can't turn a corner in this place without bumping into a delegate…

The delegate pattern is commonly used to handle the situation you find yourself in: Screen A opens screen B. At some point screen B needs to communicate back to screen A, usually when it closes.

The solution is to make screen A the delegate of screen B, so that B can send its messages to A whenever it needs to.

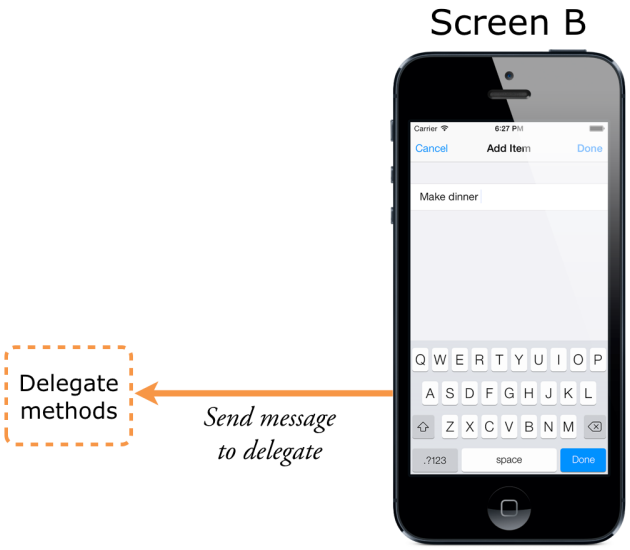


Screen A launches screen B and becomes its delegate

The cool thing about the delegate pattern is that screen B doesn't really know anything about screen A. It just knows that *some* object is its delegate. Other than that, screen B doesn't care who that is.

Just like `UITableView` doesn't really care about your view controller, only that it delivers table view cells when the table view asks for them.

This principle, where screen B is independent of screen A yet can still talk to it, is called *loose coupling* and is considered good software design practice.



This is what Screen B sees: only the delegate part, not the rest of screen A

You will use the delegate pattern to let the `AddItemViewController` send notifications back to the `ChecklistViewController`, without it having to know anything about this object.

Delegates go hand-in-hand with *protocols*, a prominent feature of the Swift language.

➤ At the top of **AddItemViewController.swift**, add this in between the `import` and `class` lines (it is not part of the `AddItemViewController` object):

```
protocol AddItemViewControllerDelegate: class {
func addItemViewControllerDidCancel(_ controller: AddItemViewController)
func addItemViewController(_ controller: AddItemViewController,
                           didFinishAdding item: ChecklistItem)
}
```

This defines the `AddItemViewControllerDelegate` protocol. You should recognize the lines inside the `protocol { ... }` block as method declarations, but unlike other methods they don't have any source code in them. The protocol just lists the names of the methods.

Think of the delegate protocol as a contract between screen B, in this case the Add Item View Controller, and any screens that wish to use it.

Protocols

In Swift, a *protocol* doesn't have anything to do with computer networks or meeting royalty. It is simply a name for a group of methods.

A protocol doesn't implement any of the methods it declares. It just says: any object that conforms to this protocol must implement methods X, Y and Z.

The two methods listed in the `AddItemViewControllerDelegate` protocol are:

- `addItemViewControllerDidCancel()`
- `addItemViewController(didFinishAdding)`

Delegates often have very long method names!

The first method is for when the user presses Cancel, the second is for when she presses Done. In that case, the `didFinishAdding` parameter passes along the new `ChecklistItem` object.

To make the `ChecklistViewController` conform to this protocol, it must provide implementations of these two methods. From then on you can refer to the `ChecklistViewController` using just the protocol name.

(If you've programmed in other languages before, you may recognize protocols as being very similar to "interfaces".)

Inside `AddItemViewController`, you can write the following to refer back to the `ChecklistViewController`:

```
var delegate: AddItemViewControllerDelegate
```

The variable `delegate` is nothing more than a reference to *some* object that implements the methods of this protocol. You can send messages to the object from the `delegate` variable, without knowing what kind of object it really is.

Of course, *you* know the object from `delegate` is the `ChecklistViewController` but `AddItemViewController` doesn't need to be aware of that. All it sees is some object that implements its delegate protocol.

If you wanted to, you could make some other object implement the protocol and `AddItemViewController` would be perfectly OK with that. That's the power of delegation: you have removed – or *abstracted* away – the dependency between the `AddItemViewController` and the rest of the app.

It may seem a little overkill for a simple app such as this, but delegates are one of the cornerstones of iOS development. The sooner you master them, the better!

You're not done yet in **AddItemViewController.swift**. The view controller must have a property that it can use to refer to the delegate.

➤ Add this inside the class `AddItemViewController`, below the outlets:

```
weak var delegate: AddItemViewControllerDelegate?
```

It looks like a regular instance variable declaration, with two differences: `weak` and the question mark.

Delegates are usually declared as being *weak* – not a statement of their moral character but a way to describe the relationship between the view controller and its delegate. Delegates are also *optional* (the question mark).

You'll learn more about those things in a moment.

➤ Replace the `cancel()` and `done()` actions with the following:

```
@IBAction func cancel() {
  delegate?.addItemViewControllerDidCancel(self)
}
```

```
@IBAction func done() {
  let item = ChecklistItem()
  item.text = textField.text!
  item.checked = false

  delegate?.addItemViewController(self, didFinishAdding: item)
}
```

Let's look at the changes you made. When the user taps the Cancel button, you send the `addItemViewControllerDidCancel()` message back to the delegate.

You do something similar for the Done button, except that the message is `addItemViewController(didFinishAdding)` and you pass along a new `ChecklistItem` object that has the text string from the text field.

Note: It is customary for the delegate methods to have a reference to their owner as the first (or only) parameter.

Doing this is not required but still a good idea. For example, in the case of table views it may happen that an object is the delegate or data source for more than one table view. In that case, you need to be able to distinguish between those two table views. To allow for this, the table view delegate methods have a parameter for the `UITableView` object that sent the notification. Having this reference also saves you from having to make an `@IBOutlet` for the table view.

That explains why you pass `self` to your delegate methods. Recall that `self` refers to the object itself, in this case `AddItemViewController`. It's also why the method names start with the term "`addItemViewController`".

➤ Run the app and try the Cancel and Done buttons. They no longer work!

I hope you're not too surprised… The Add Item screen now depends on a delegate to make it close, but you haven't told the Add Item screen yet who its delegate is.

That means the `delegate` property has no value and the messages aren't being sent to anyone – there is no one listening for them.

Optionals

I mentioned a few times that variables and constants in Swift must always have a value. In other programming languages the special symbol `nil` or `NULL` is often used to indicate that a variable has no value. This is not allowed in Swift for normal variables.

The problem with `nil` and `NULL` is that they are a frequent cause of crashing apps. If an app attempts to use a variable that is `nil` when you don't expect it to be, the app will crash. This is the dreaded "null pointer dereference".

Swift stop this from happening by preventing you from using `nil`.

However, sometimes a variable does need to have "no value". In that case you can make it an *optional*. You mark something as optional in Swift using either a question mark `?` or an exclamation point `!`.

Only variables that are made optional can have the value `nil`.

You've already seen the question mark used with `IndexPath?`, the return type of `tableView(willSelectRowAt)`. Returning `nil` from this method is a valid response; it means that the table should not select a particular row.

The question mark tells Swift that it's OK for the method to return `nil` instead of an actual `IndexPath` object.

The variable that refers to the delegate is usually marked as optional too. You can tell because there is a question mark behind its type:

```
weak var delegate: AddItemViewControllerDelegate?
```

Thanks to the `?` it's perfectly acceptable for delegate to be `nil`.

You may be wondering why delegate would ever be `nil`. Doesn't that negate the idea of having a delegate in the first place? There are two reasons.

Often delegates are truly optional; a `UITableView` works fine even if you don't implement any of its delegate methods (but you do need to provide at least some of its data source methods).

More importantly, when `AddItemViewController` is loaded from the storyboard and instantiated, it won't know right away who its delegate is. Between the time the view controller is loaded and the delegate is assigned, the `delegate` variable will be `nil`. And variables that can be `nil`, even if it is only temporary, must be optionals.

When `delegate` is `nil`, you don't want `cancel()` or `done()` to send any of the messages. Doing that would crash the app because there is no one to receive the messages.

Swift has a handy shorthand for skipping the work when `delegate` is not set:

```
delegate?.addItemViewControllerDidCancel(self)
```

Here the `?` tells Swift not to send the message if `delegate` is `nil`. You can read this as, "Is there a delegate? Then send the message." This practice is called *optional chaining* and it's used a lot in Swift.

In this app it should never happen that `delegate` is `nil` – that would get users stuck

on the Add Item screen – but Swift doesn't know that. So you'll have to pretend that it can happen anyway and use optional chaining to send messages to the delegate.

Optionals aren't common in other programming languages, so they may take some getting used to. I find that optionals do make programs clearer – most variables never have to be `nil`, so it's good to prevent them from becoming `nil` and avoid these potential sources of bugs.

Remember, if you see `?` or `!` in a Swift program, you're dealing with optionals. In the course of this tutorial I'll come back to this topic a few more times and explain the finer points of using optionals in more detail.

Before you can give `AddItemViewController` its delegate, you first need to make the `ChecklistViewController` suitable to play the delegate role.

➤ In **ChecklistViewController.swift**, change the class line to the following (this goes all on one line):

```
class ChecklistViewController: UITableViewController,
                                        AddItemViewControllerDelegate {
```

This tells the compiler that `ChecklistViewController` now promises to do the things from the `AddItemViewControllerDelegate` protocol.

If you try to run the app, Xcode gives an error: "Type ChecklistViewController does not conform to protocol AddItemViewControllerDelegate." That is correct: you still need to add the methods that are listed in `AddItemViewControllerDelegate`.

➤ Add the implementations of the protocol's methods to `ChecklistViewController`:

```
func addItemViewControllerDidCancel(
                                _ controller: AddItemViewController) {
  dismiss(animated: true, completion: nil)
}

func addItemViewController(_ controller: AddItemViewController,
                           didFinishAdding item: ChecklistItem) {
  dismiss(animated: true, completion: nil)
}
```

Currently these methods simply close the Add Item screen. This is what the `AddItemViewController` used to do in its `cancel()` and `done()` actions. You've simply moved that responsibility to the delegate.

The code that puts the new `ChecklistItem` object into the table view is left out for

now. You'll add this in a moment, but there's something else you need to do first.

Delegates in five easy steps

These are the steps for setting up the delegate pattern between two objects, where object A is the delegate for object B, and object B will send messages back to A. The steps are:

1 - Define a delegate `protocol` for object B.

2 - Give object B a `delegate` optional variable. This variable should be `weak`.

3 - Make object B send messages to its delegate when something interesting happens, such as the user pressing the Cancel or Done buttons, or when it needs a piece of information. You write `delegate?.methodName(self, . . .)`

4 - Make object A conform to the delegate protocol. It should put the name of the protocol in its `class` line and implement the methods from the protocol.

5 - Tell object B that object A is now its delegate.

You've done steps 1 to 4, so there is one more thing you need to do (step 5): tell `AddItemViewController` that the `ChecklistViewController` is its delegate.

The proper place to do that is in the `prepare(for:sender:)` method, also known as *prepare-for-segue*.

The `prepare(for:sender:)` method is invoked by UIKit when a segue from one screen to another is about to be performed. Recall that the segue is the arrow between two view controllers in the storyboard.

Using prepare-for-segue allows you to give data to the new view controller before it will be displayed. Usually you'll do that by setting its properties.

➤ Add this method to **ChecklistViewController.swift**:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  // 1
  if segue.identifier == "AddItem" {
    // 2
    let navigationController = segue.destination
                                                as! UINavigationController
    // 3
    let controller = navigationController.topViewController
                                                 as! AddItemViewController
    // 4
    controller.delegate = self
  }
}
```

This is what prepare-for-segue does, step-by-step:

1. Because there may be more than one segue per view controller, it's a good idea

to give each one a unique identifier and to check for that identifier first to make sure you're handling the correct segue. Swift's == comparison operator does not work on just numbers but also on strings and most other types of objects.

2. The new view controller can be found in `segue.destination`. The storyboard shows that the segue does not go directly to `AddItemViewController` but to the navigation controller that embeds it. So first you get ahold of this `UINavigationController` object.

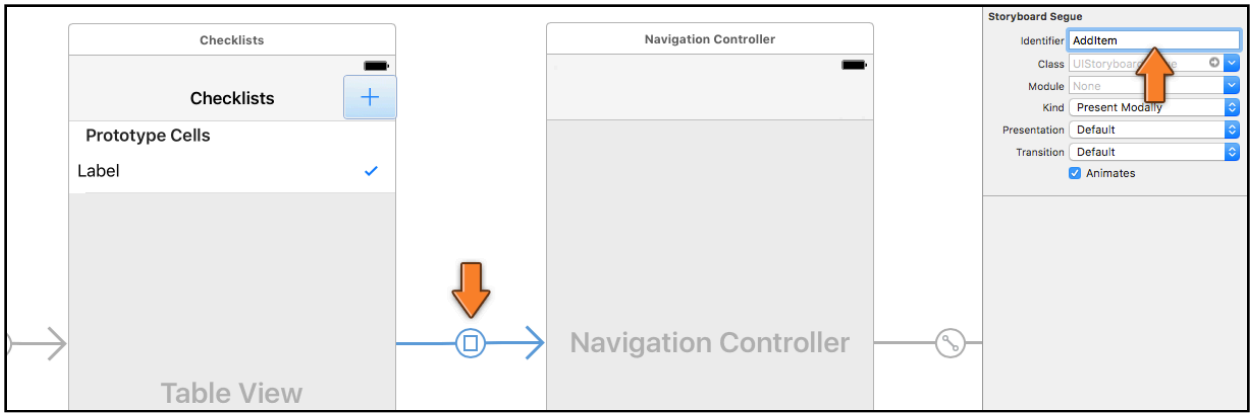
3. To find the `AddItemViewController`, you can look at the navigation controller's `topViewController` property. This property refers to the screen that is currently active inside the navigation controller.

4. Once you have a reference to the `AddItemViewController` object, you set its `delegate` property to `self` and the connection is complete. This tells the `AddItemViewController` that from now on, the object known as `self` is its delegate. But what is "self" here? Well, since you're editing **ChecklistViewController.swift**, `self` refers to the `ChecklistViewController`.

Excellent! `ChecklistViewController` is now the delegate of `AddItemViewController`. It took some work, but you're all set now.

➤ Open the storyboard and select the segue between the Checklist View Controller and the Navigation Controller on its right.

➤ In the **Attributes inspector**, type **AddItem** into the **Identifier** field:



Naming the segue between the Checklists scene and the navigation controller

➤ Run the app to see if it works. (Make sure the storyboard is saved before you press Run, or the app may crash.)

Pressing the + button will perform the segue to the Add Item screen with the Checklists screen set as its delegate.

When you press Cancel or Done, `AddItemViewController` sends a message to its delegate, `ChecklistViewController`. Currently the delegate simply closes the Add

Item screen, but now this works you can make it do more.

Let's add the new `ChecklistItem` to the data model and table view. Finally!

➤ Change the implementation of the "`didFinishAdding`" delegate method in **ChecklistViewController.swift** to the following:

```
func addItemViewController(_ controller: AddItemViewController,
                           didFinishAdding item: ChecklistItem) {
  let newRowIndex = items.count
  items.append(item)

  let indexPath = IndexPath(row: newRowIndex, section: 0)
  let indexPaths = [indexPath]
  tableView.insertRows(at: indexPaths, with: .automatic)

  dismiss(animated: true, completion: nil)
}
```

This is largely the same as what you did in `addItem()` before. In fact, I simply copied the contents of `addItem()` and pasted them into this delegate method. Compare the two methods and see for yourself.

The only difference is that you no longer create the `ChecklistItem` object here; that happens in the `AddItemViewController`. You merely have to insert this new object into the `items` array.

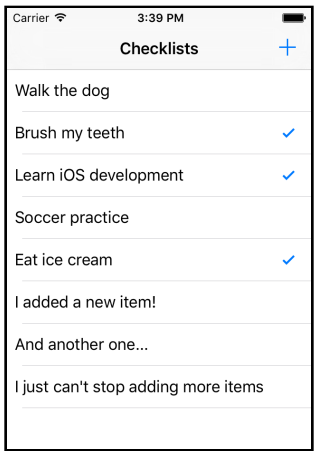
As before, you tell the table view you have a new row for it and then close the Add Items screen.

➤ Remove `addItem()` from **ChecklistViewController.swift** as you no longer need this method.

Just to make sure, open the storyboard and double-check that the + button is no longer connected to the `addItem` action. Bad things happen if buttons are connected to methods that no longer exist…

(You can see this in the Connections inspector for the + button, under Sent Actions. Nothing should be connected there. Only the modal segue under Triggered Segues should be present.)

➤ Run the app and you should be able to add your own items to the list!



You can finally add new items to the to-do list

You can find the project files for the app up to this point under **04 - Add Item Screen** in the tutorial's Source Code folder.

★ ★ ★

Weak

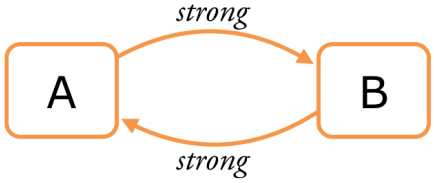
I still owe you an explanation about weak. Relationships between objects can be weak or strong. You use weak relationships to avoid so-called *ownership cycles*.

When object A has a strong reference to object B, and at the same time object B also has a strong reference back to A, then these two objects are involved in a dangerous kind of romance: an ownership cycle.

Normally, an object is destroyed – or *deallocated* – when there are no more strong references to it. But because A and B have strong references to each other, they're keeping each other alive.

The result is a potential *memory leak* where an object that ought to be destroyed isn't, and the memory for its data is never reclaimed. With enough such leaks, iOS will run out of available memory and your app will crash. I told you it was dangerous!

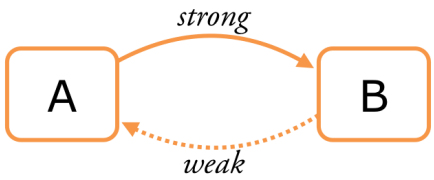
Due to the strong references between them, A owns B but also B owns A:



To avoid ownership cycles you can make one of these references weak.

In the case of a view controller and its delegate, screen A usually has a strong reference to screen B, but B only has a weak reference back to its delegate, A.

Because of the weak reference, B no longer owns A:



Now there is no ownership cycle.

Such cycles can occur in other situations too but they are most common with delegates. Therefore, delegates are always made `weak`.

(There is another relationship type, `unowned`, that is similar to `weak` and can be used for delegates too. The difference is that weak variables are allowed to become `nil` again. You may forget this right now.)

`@IBOutlets` are usually also declared with the `weak` keyword. This isn't done to avoid an ownership cycle, but to make it clear that the view controller isn't really the owner of the views from the outlets.

In the course of these tutorials you'll learn more about `weak`, `strong`, optionals, and the relationships between objects. These are important concepts in Swift, but they may take a while to make sense. Don't lose any sleep over it!

Editing existing checklist items

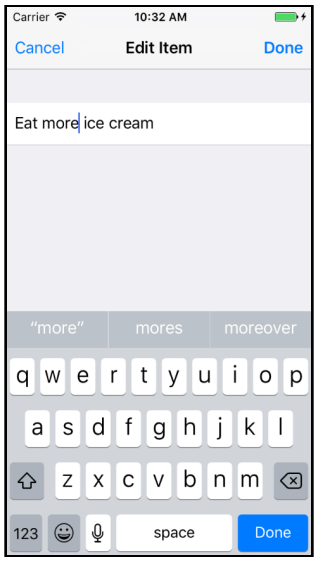
Adding new items to the list is a great step forward for the app, but there are usually three things an app needs to do with data:

1. adding new items (which you've tackled),
2. deleting items (you allow that with swipe-to-delete), and
3. editing existing items (uhh…).

The latter is useful for when you want to rename an item from your list. We all make typos.

You could make a completely new Edit Item screen but it would work mostly the same as the Add Item screen. The only difference is that it doesn't start out empty but with an existing to-do item.

So let's re-use the Add Item screen and make it capable of editing an existing `ChecklistItem` object.



Editing a to-do item

When the user presses Done you won't have to make a new `ChecklistItem` object, instead you will simply update the text in the existing `ChecklistItem`.

You'll also tell the delegate about these changes so that it can update the text label of the corresponding table view cell.

Exercise: Which changes would you need to make to the Add Item screen to enable it to edit existing items? ■

Answer:

1. The screen must be renamed to **Edit Item**.
2. You must be able to give it an existing `ChecklistItem` object.
3. You have to place the `ChecklistItem`'s text into the text field.
4. When the user presses Done, you should not add a new `ChecklistItem` object, but update the existing one.

There is a bit of a user interface problem, though… How will the user actually open the Edit Item screen? In many apps that is done by tapping on the item's row but in the Checklists app that already toggles the checkmark on or off.

To solve this problem, you'll have to revise the UI a little.

When a row is given two functions, the standard approach is to use a **detail disclosure button** for the secondary task:

The detail disclosure button

Tapping the row itself will still perform the row's main function, in this case toggling the checkmark. But tapping the disclosure button will open the Edit Item screen.

> **Note:** An alternative approach is taken by Apple's Reminders app. There the checkmark is on the left and tapping only this part of the row will toggle the checkmark. Tapping anywhere else in the row will bring up the Edit screen for that item.
>
> There are also apps that can toggle the whole screen into "Edit mode" and then let you change the text of an item inline. Which solution you choose depends on what works best for your data.

➤ Go to the table view cell in the storyboard and in the **Attributes inspector** set its **Accessory** to **Detail Disclosure**.

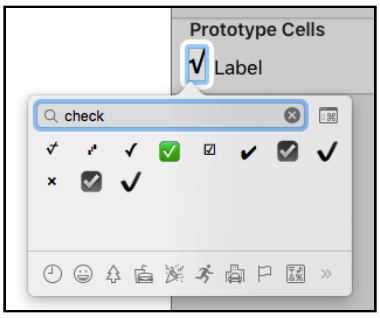
Instead of the checkmark you'll now see a chevron (>) and a blue info button on the right of the cell. This means you'll have to place the checkmark somewhere else.

➤ Drag a new **Label** into the cell and place it on the left of the text label. Give it the following attributes:

- Text: √ (you can type this with **Alt/Option+V**)
- Font: Helvetica Neue, Bold, size 22
- Tag: 1001

You've given this new label its own tag, so you can easily find it later.

If typing Option-V does not work for you, choose **Edit → Emoji & Symbols** from the Xcode menu bar. Use the search bar to search for "check" – or whatever takes your fancy. (Note that not all of these special symbols may actually work on your iPhone.)



The Emoji & Symbols palette

➤ Resize the text label so that it doesn't overlap the checkmark or the disclosure button. It should be about 215 points wide.

The design of the prototype cell now looks like this:

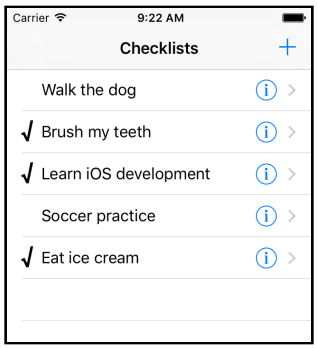
The new design of the prototype cell

➤ In **ChecklistViewController.swift**, change `configureCheckmark(for:with:)` to:

```
func configureCheckmark(for cell: UITableViewCell,
                        with item: ChecklistItem) {
  let label = cell.viewWithTag(1001) as! UILabel

  if item.checked {
    label.text = "√"
  } else {
    label.text = ""
  }
}
```

Instead of setting the cell's `accessoryType` property, this now changes the text in the new label.

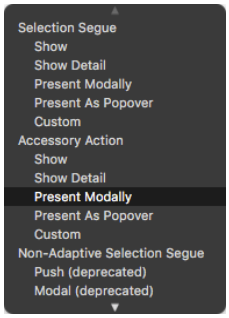
➤ Run the app and you'll see that the checkmark has moved to the left. There is also a blue detail disclosure button on the right. Tapping the row still toggles the checkmark, but tapping the blue button doesn't.



The checkmarks are now on the other side of the cell

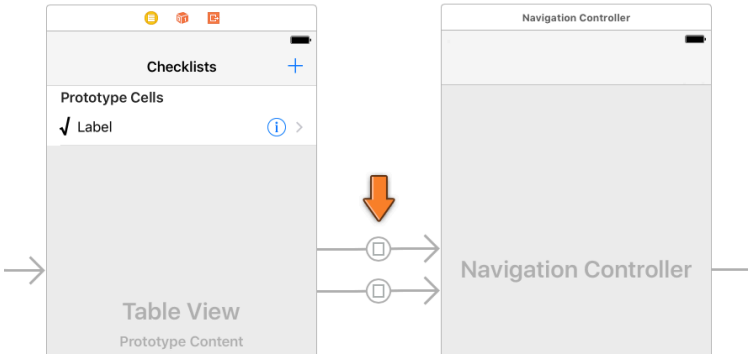
Next you're going to make the detail disclosure button open the Add/Edit Item screen. This is pretty simple because Interface Builder also allows you to make a segue for a disclosure button.

➤ Open the storyboard. Select the table view cell and **Ctrl-drag** to the Navigation Controller next door to make a segue. From the popup, choose **Present Modally** from the **Accessory Action** section (not from Selection Segue):



Making a modal segue from the detail disclosure button

There are now two segues going from the Checklists screen to the navigation controller. One is triggered by the + button, the other by the detail disclosure button from the prototype cell.



Two arrows for two segues

For the app to make a distinction between these two segues, they must have unique identifiers.

➤ Give this new segue the identifier **EditItem** (in the **Attributes inspector**).

If you run the app now, tapping the blue (i) button will also open the Add Item screen. But the Cancel and Done buttons won't work.

Exercise: Can you explain why not? ■

Answer: You haven't set the delegate yet. Remember that you set the delegate in `prepare(for:sender:)`, but only for when the + button is tapped to perform the "AddItem" segue. You haven't done the same for this new "EditItem" segue.

Before you fix the delegate business, you shall first make the Add/Edit Item screen capable of editing existing `ChecklistItem` objects.

➤ Add a new property for a `ChecklistItem` object below the other instance variables in **AddItemViewController.swift**:

```
  var itemToEdit: ChecklistItem?
```

This variable contains the existing `ChecklistItem` object that the user will be editing. But when adding a new to-do item, `itemToEdit` will be `nil`. That is how the view controller will make the distinction between adding and editing.

Because `itemToEdit` can be `nil`, it needs to be an optional. That explains the question mark.

➤ Add the `viewDidLoad()` method to **AddItemViewController.swift**:

```
  override func viewDidLoad() {
    super.viewDidLoad()

    if let item = itemToEdit {
      title = "Edit Item"
      textField.text = item.text
    }
  }
```

Recall that `viewDidLoad()` is called by UIKit when the view controller is loaded from the storyboard, but before it is shown on the screen. That gives you time to put the user interface in order.

In editing mode, when `itemToEdit` is not `nil`, you change the title in the navigation bar to "Edit Item". You do this by changing the `title` property.

Each view controller has a number of built-in properties and this is one of them. The navigation controller looks for the `title` property and automatically changes the text in the navigation bar.

You also set the text in the text field to the value from the item's `text` property.

if let

You cannot use optionals like you would regular variables. For example, if `viewDidLoad()` would have done the following,

```
    textField.text = itemToEdit.text
```

then Xcode would complain with the error message, "Value of optional type ChecklistItem? not unwrapped".

That's because `itemToEdit` is the optional version of `ChecklistItem`.

In order to use it you first need to *unwrap* the optional. You do that with the following special syntax:

```
if let temporaryConstant = optionalVariable {
  // temporaryConstant now contains the unwrapped value
  // of the optional variable
}
```

If the optional is not `nil`, then the code inside the if-statement is performed.

There are a few other ways to read the value of an optional, but using `if let` is the safest: if the optional has no value – i.e. it is `nil` – then the code inside the `if let` block is skipped over.

Do you find optionals weird and confusing? Take some comfort in the fact that everyone else does too. This feature of Swift isn't found in many other mainstream languages and most developers do a double take when they first encounter it.

Despite being a little odd, optionals will prevent mistakes with "null pointer dereferences" and help bulletproof your programs against crashes.

The `AddItemViewController` is now capable of recognizing when it needs to edit an item. If the `itemToEdit` property is given a `ChecklistItem` object, then the screen magically changes into the Edit Item screen.

But where do you fill up that `itemToEdit` property? In prepare-for-segue, of course! That's the ideal place for putting values into the properties of the new screen before it becomes visible.

➤ Change `prepare(for:sender:)` in **ChecklistViewController.swift** to the

following:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "AddItem" {
    . . .

  } else if segue.identifier == "EditItem" {
    let navigationController = segue.destination
                                                    as! UINavigationController
    let controller = navigationController.topViewController
                                                    as! AddItemViewController
    controller.delegate = self

    if let indexPath = tableView.indexPath(
                                        for: sender as! UITableViewCell) {
      controller.itemToEdit = items[indexPath.row]
    }
  }
}
```

As before, you get the navigation controller from the storyboard, and its embedded `AddItemViewController` using the `topViewController` property.

You also set the view controller's `delegate` property so you're notified when the user taps Cancel or Done. Nothing new there. This is the same as for the `"AddItem"` segue.

This is the interesting new bit:

```
if let indexPath = tableView.indexPath(for: sender as! UITableViewCell){
  controller.itemToEdit = items[indexPath.row]
}
```

You're in the `prepare(for:sender:)` method, which has a parameter named `sender`. This parameter contains a reference to the control that triggered the segue, in this case the table view cell whose disclosure button was tapped.

You use that `UITableViewCell` object to find the row number by looking up the corresponding index-path using `tableView.indexPath(for)`.

The return type of `UITableView`'s `indexPath(for)` method is `IndexPath?`, an optional, meaning it can possibly return `nil`. That's why you need to unwrap this optional value with `if let` before you can use it.

Once you have the row number you can obtain the `ChecklistItem` object to edit, and you assign this to `AddItemViewController`'s `itemToEdit` property.

Sending data between view controllers

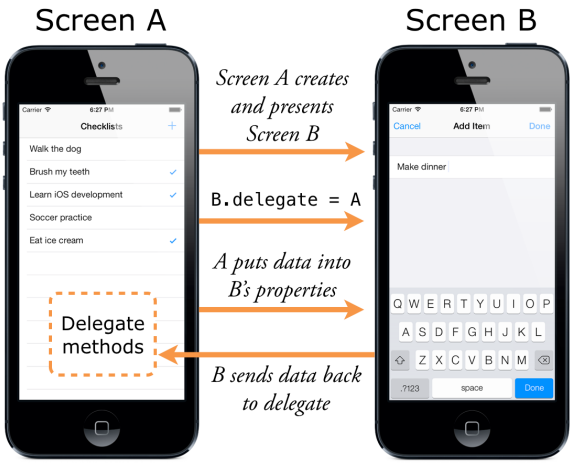
We've talked about screen B (the Add/Edit Item screen) passing data back to screen A (the Checklists screen) through the use of delegates.

But here you're passing a piece of data the other way around – from screen A to screen B – namely, the `ChecklistItem` to edit.

Data transfer between view controllers works two ways:

1. From A to B. When screen A opens screen B, A can give B the data it needs. You simply make a new instance variable in B's view controller. Screen A then puts an object into this property right before it makes screen B visible, usually in `prepare(for:sender:)`.
2. From B to A. To pass data back from B to A you use a delegate.

This illustration shows how screen A sends data to screen B by putting it into B's properties, and how screen B sends data back to the delegate:



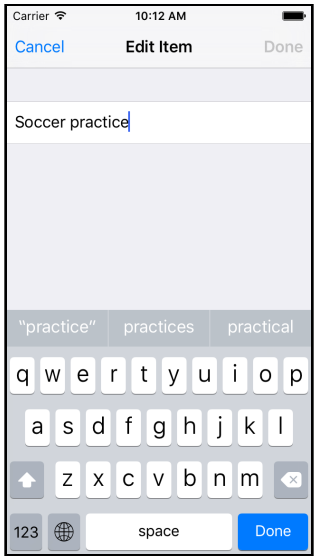
I hope the flow between view controllers is starting to make sense now. You're going to do this sort of thing a few more times in this lesson, just to make sure you get comfortable with it.

Making iOS apps is all about creating view controllers and sending messages between them, so you want this to become second nature.

★ ★ ★

➤ With these steps done, you can now run the app. A tap on the + button opens

the Add Item screen as before. But tap the accessory button on an existing row and the screen that opens is named Edit Item. It already contains the to-do item's text:



Editing an item

One small problem: the Done button in the navigation bar is initially disabled. This is because you originally set it to be disabled in the storyboard.

➤ Change `viewDidLoad()` in **AddItemViewController.swift** to fix this:

```
override func viewDidLoad() {
  super.viewDidLoad()

  if let item = itemToEdit {
    title = "Edit Item"
    textField.text = item.text
    doneBarButton.isEnabled = true    // add this line
  }
}
```

You can simply always enable the Done button; when editing an existing item you are guaranteed to pass in a text that is not empty.

The problems don't end here, though. Run the app, tap a row to edit it, and press Done. Instead of changing the text on the existing item, a brand new to-do item with the new text is added to the list.

You didn't write the code yet to update the data model, so the delegate always thinks it needs to add a new row.

To solve this you add a new method to the delegate protocol.

➤ Add the following line to the `protocol` section in **AddItemViewController.swift**:

```
func addItemViewController(_ controller: AddItemViewController,
                           didFinishEditing item: ChecklistItem)
```

The full protocol now looks like this:

```
protocol AddItemViewControllerDelegate: class {
func addItemViewControllerDidCancel(_ controller: AddItemViewController)
func addItemViewController(_ controller: AddItemViewController,
                           didFinishAdding item: ChecklistItem)
func addItemViewController(_ controller: AddItemViewController,
                           didFinishEditing item: ChecklistItem)
}
```

There is a method that is invoked when the user presses Cancel and two methods for when the user presses Done.

After adding a new item you call `didFinishAdding`, but when editing an existing item the new `didFinishEditing` method should now be called instead.

By using different methods the delegate (the `ChecklistViewController`) can make a distinction between those two situations.

➤ In **AddItemViewController.swift**, change the `done()` method to:

```
@IBAction func done() {
  if let item = itemToEdit {
    item.text = textField.text!
    delegate?.addItemViewController(self, didFinishEditing: item)

  } else {
    let item = ChecklistItem()
    item.text = textField.text!
    item.checked = false
    delegate?.addItemViewController(self, didFinishAdding: item)
  }
}
```

First this checks whether the `itemToEdit` property contains an object. You should recognize the `if let` syntax for unwrapping an optional.

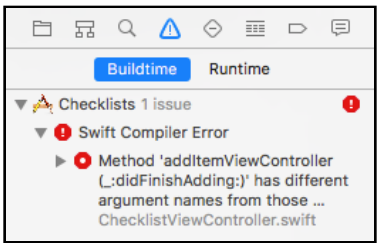
If the optional is not `nil`, you put the text from the text field into the existing `ChecklistItem` object and then call the new delegate method.

In the case that `itemToEdit` *is* `nil`, the user is adding a new item and you do the stuff you did before (inside the `else` block).

➤ Try to build the app. It won't work.

Xcode says "Build Failed" but there don't seem to be any error messages in **AddItemViewController.swift**. So what went wrong?

You can see all errors and warnings from Xcode in the **Issue navigator**:



Xcode warns about incomplete implementation

The error is apparently in `ChecklistViewController` because it does not implement a method from the protocol. That is not so strange because you just added the new `addItemViewController(didFinishEditing)` method to the delegate protocol. But you did not yet tell the view controller, who plays the role of the delegate, what to do with it.

> **Note:** The exact error message in my version of Xcode is "Method … has different argument names from those required by protocol …". That's a bit of a strange error message, wouldn't you say? It doesn't really describe what's wrong, just what Swift is confused about.
>
> As you're writing your own apps you'll probably run into other strange or even undecipherable Swift error messages. This should get better in time. The Swift compiler is quite new at the job and still needs to work on its bedside manner.

➤ Add the following to **ChecklistViewController.swift** and the compiler error will be history:

```
func addItemViewController(_ controller: AddItemViewController,
                           didFinishEditing item: ChecklistItem) {
  if let index = items.index(of: item) {
    let indexPath = IndexPath(row: index, section: 0)
    if let cell = tableView.cellForRow(at: indexPath) {
      configureText(for: cell, with: item)
    }
  }
  dismiss(animated: true, completion: nil)
}
```

The `ChecklistItem` object already has the new text – it was put there by `done()` – and the cell for it already exists in the table view. But you do need to update the label in its table view cell.

So in this new method you look for the cell that corresponds to the `ChecklistItem` object and, using the `configureText(for:with:)` method you wrote earlier, tell it to refresh its label.

The first statement is the most interesting:

```
  if let index = items.index(of: item) {
```

In order to make the `IndexPath` that you need to retrieve the cell, you first need to find the row number for this `ChecklistItem`. The row number is the same as the index of the `ChecklistItem` in the `items` array, and you can use the `index(of)` method to return that index.

Now, it won't happen here, but in theory it's possible that you use `index(of)` on an object that is not actually in the array. To account for the possibility, `index(of)` does not return a normal value, it returns an optional. If the object is not part of the array, the optional is `nil`.

That's why you need to use `if let` here to unwrap the return value of `index(of)`.

➤ Try to build the app. Whoops, I guess I spoke too soon. Xcode has found another reason to complain: "Cannot invoke index with an argument list of type blah blah blah". What does *that* mean?

This error happens because you can't use `index(of)` on just any object, only on objects that are "equatable". `index(of)` must be able to somehow compare the object that you're looking for to the objects in the array, to see if they are equal.

Your `ChecklistItem` object does not have any functionality for that yet. There are a few ways you can fix this, but we'll go for the easy one.

➤ In **ChecklistItem.swift**, change the class line to:

```
class ChecklistItem: NSObject {
```

If you've programmed in Objective-C before, then `NSObject` will look familiar.

Almost all objects in Objective-C programs are based on `NSObject`. It's the most basic building block provided by iOS, and it offers a bunch of useful functionality that standard Swift objects don't have.

You can write many Swift programs without having to resort to `NSObject`, but in times like these it comes in handy.

Building `ChecklistItem` on top of `NSObject` is enough to make it satisfy the "equatable" requirement. Later in the tutorial, when you learn about saving the checklist items, you would have had to make it an `NSObject` anyway, so this is a good solution for this app.

➤ Run the app again and verify that editing items works now. Excellent!

Refactoring the code

At this point you have an app that can add new items and edit existing items using the combined Add/Edit Item screen. Pretty sweet.

Given the recent changes, I don't think the name `AddItemViewController` is appropriate anymore as this screen is now used to both add and edit items.

I propose you rename it to `ItemDetailViewController`.

Now, I've got good news and I've got bad news. Which one do you want to hear first?

The good news is that Xcode has a special menu for refactoring source code, including a tool to rename items. You can find this menu under **Edit → Refactor**.

The bad news is that in Xcode 8.0 this tool does not yet work for Swift sources, only for Objective-C. So you can't actually use it. Boohoo!

For us poor slobs without a working Refactor menu there's only one option: manual labor – you'll have to make these changes by hand. Fortunately, Xcode has a very handy search & replace function.

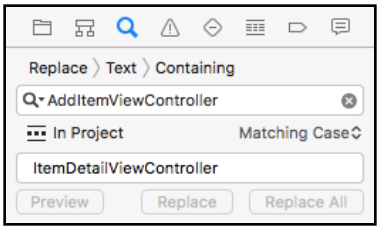
➤ Switch to the **Search navigator** (third tab in the navigator pane).

➤ Click on **Find** to change it to **Replace**.

➤ Change Ignoring Case to **Matching Case**.

➤ Type as the search text: **AddItemViewController**. Important: Make sure you spell it exactly like this!

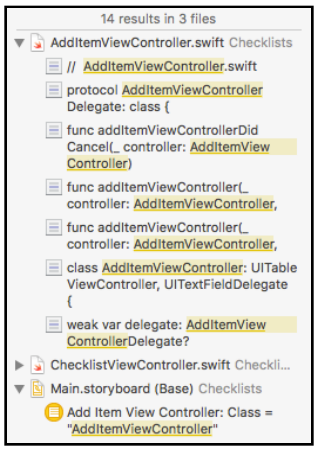
➤ Type in the replacement field: **ItemDetailViewController**.



The search & replace options

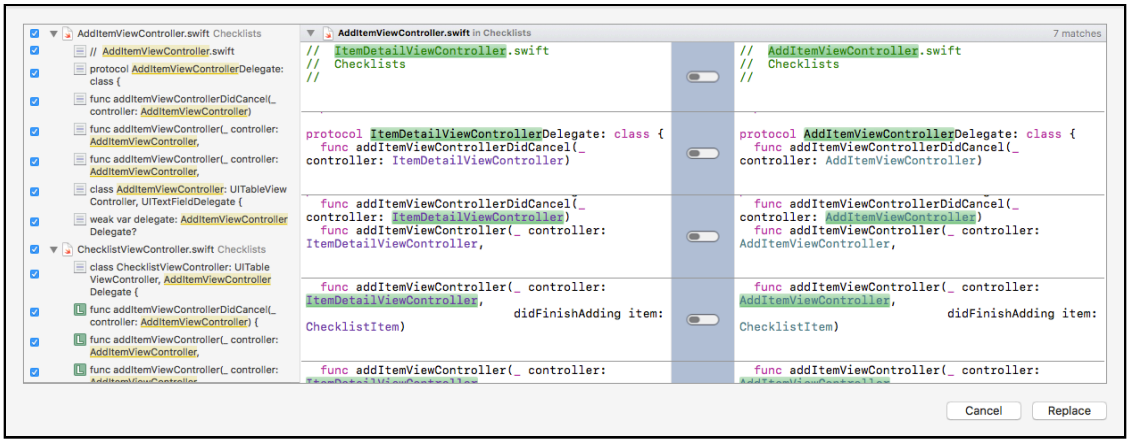
➤ Press **return** on your keyboard to start the search. This doesn't replace anything yet.

The search navigator shows the files containing matches for the search term. You should see the two Swift source files and Main.storyboard in this list.



The search results

➤ Click the **Preview** button. Xcode opens a screen with the files that are about to be changed and the individual changes inside each file:



The preview pane shows the changes that Xcode is proposing

Have a look through these files just to make sure Xcode isn't doing anything you'll regret later. It should only rename everything that says `AddItemViewController` to `ItemDetailViewController`, also inside your storyboard.

➤ Click **Replace** and pray. If Xcode asks for confirmation, click **Continue**.

Xcode did not automatically rename the Swift file, so let's do that now.

➤ In the project navigator, click **AddItemViewController.swift** to select it and then click again (slowly) to make the name become editable.

Type the new name: **ItemDetailViewController.swift**

You're not done yet with these refactorings. You still have to rename the methods from the delegate protocol.

➤ Switch to the **Search navigator** again

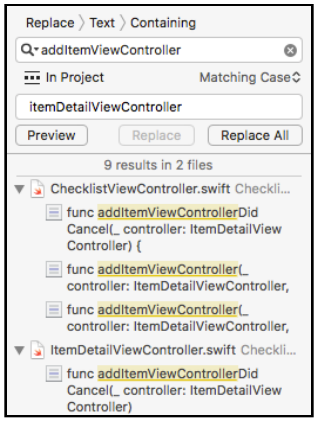
➤ Make sure you're in **Replace** mode (click Find at the top to switch to Replace)

➤ Make sure you're in **Matching Case** mode (if not, click Ignoring Case to switch)

➤ Type for the search text: **addItemViewController** with a lowercase "a"

➤ The as the replacement text: **itemDetailViewController**

➤ Press **return** to search through the entire project.



Using the Search navigator to find the methods to change

The search results should have only the method names from the delegate protocol, in both ChecklistViewController.swift and ItemDetailViewController.swift.

➤ If you're happy with the proposed changes, click **Replace All**.

I always repeat the search afterwards, ignoring case, to make sure I didn't skip anything by accident.

After these changes, the protocol in **ItemDetailViewController.swift** now has these methods:

```
protocol ItemDetailViewControllerDelegate: class {
  func itemDetailViewControllerDidCancel(
                                _ controller: ItemDetailViewController)

  func itemDetailViewController(_ controller: ItemDetailViewController,
                              didFinishAdding item: ChecklistItem)

  func itemDetailViewController(_ controller: ItemDetailViewController,
                              didFinishEditing item: ChecklistItem)
}
```

➤ Press **⌘+B** to compile the app. If you made all the changes without any mistakes, the app should build without errors.

> **Note:** Getting a "Build Failed" error? Then double-check your spelling.
>
> Swift is case-sensitive, so it considers "**i**temDetailViewController" and "**I**temDetailViewController" to be two completely different words.

If the app crashes for you at this point, double-check that in the storyboard the Custom Class of the view controller now says ItemDetailViewController (see the Identity inspector pane). Xcode sometimes skips this step and then you have to make this change manually.

Because you made quite a few changes all over the place, it's a good idea to do a *clean* just to make sure Xcode picks up all these changes and that there are no more warnings or compiler errors. You don't have to be paranoid about this, but it's good practice to clean house once in a while.

➤ From Xcode's menu bar choose **Product → Clean**. When the clean is done choose **Product → Build** (or simply press the Run button).

If there are no build issues, run the app again and test the various features just to make sure everything still works! (If the build succeeds but Xcode still shows red error icons in your source file, then close the project and open it again, or restart Xcode. Restarting Xcode is the solution that Always Works™.)

You can find the project files for the app up to this point under **05 - Edit Items** in the tutorial's Source Code folder.

Iterative development

If you think this approach to development we've taken in this tutorial is a little messy, then you're absolutely right.

You started out with one design but as you were developing you found out that things didn't work out so well in practice and that you had to refactor your approach a few times to find a way that works.

Well, this is how software development goes in practice.

You first build a small part of your app and everything looks and works fine. Then you add the next small part on top of that and suddenly everything breaks down. The proper thing to do is to go back and restructure your entire approach so that everything is hunky-dory again… Until the next change you need to make.

Software development is a constant process of refinement. In these tutorials I didn't want to just give you a perfect piece of code and explain how each part

works. That's not how software development happens in the real world.

Instead, you're working your way from zero to a full app, exactly the way a pro developer would, including the mistakes and dead ends.

Isn't it possible to create a design up-front – sometimes called a "software architecture design" – that deals with all of these situations, like a blueprint but for software?

I don't believe in such designs. Sure, it's always good to plan ahead. Before writing this tutorial, I made a few quick sketches of how I imagined the app would turn out. That was useful to envision the amount of work, but as usual some of my assumptions and guesses turned out to be wrong and the design stopped being useful about halfway in. And this is only a simple app!

That doesn't mean you shouldn't spend any time on planning and design, just not too much. ;-)

Simply start somewhere and keep going until you get stuck, then backtrack and improve on your approach. This is called *iterative development* and it's usually faster and provides better results than meticulous up-front planning.

Saving and loading the checklist items

Any new to-do items that you add to the list cease to exist when you terminate the app (by pressing the Stop button in Xcode, for example). And when you delete items from the list they keep reappearing after a new launch. That's not how a real app should behave, of course.

Thanks to the multitasking nature of iOS, an app stays in memory when you close it and go back to the home screen or switch to another app. The app goes into a suspended state where it does absolutely nothing but will still hang on to its data.

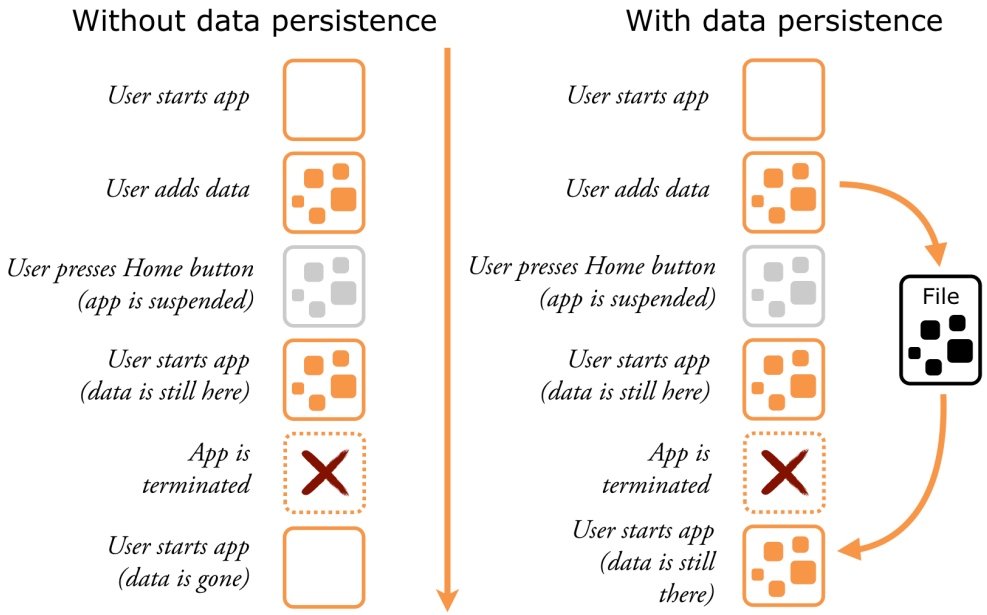
During normal usage, users will never truly terminate the app, just suspend it. However, the app can still be terminated when the iPhone runs out of available working memory, as iOS will terminate any suspended apps in order to free up memory when necessary. And if they really want to, users can kill apps by hand or reset their entire device.

Just keeping the list of items in memory is not good enough because there is no guarantee that the app will remain in memory forever, whether active or suspended.

Instead, you will need to **persist** this data in a file on the iPhone's long-term flash

storage. This is no different than saving a file from your word processor on your desktop computer except that iPhone apps should take care of this saving automatically.

The user shouldn't have to press a Save button just to make sure unsaved data is safely placed in long-term storage.



Apps need to persist data just in case the app is terminated

In this section you will:

- Determine where in the file system you can place the file that will remember the to-do list items.
- Save the to-do items to that file whenever the user changes something: adds a new item, toggles a checkmark, deletes an item, et cetera.
- Load the to-do items from that file when the app starts up again after it was terminated.

Let's get crackin'!

The documents directory

iOS apps live in a sheltered environment, also known as the **sandbox**. Each app has its own directory for storing files but cannot access the directories or files of any other apps.

This is a security measure, designed to prevent malicious software such as viruses from doing any damage. If an app can only change its own files, it cannot break any other part of the system.

Your apps can store files in the so-called "Documents" directory in the app's sandbox.

The contents of the Documents directory are backed up when the user syncs their device with iTunes or iCloud.

When you release a new version of your app and users install the update, the Documents folder is left untouched. Any data the app has saved into this folder stays there even if the app is updated.

In other words, the Documents folder is the perfect place for storing your user's data files.

Let's look at how this works.

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

```
func documentsDirectory() -> URL {
  let paths = FileManager.default.urls(for: .documentDirectory,
                                       in: .userDomainMask)
  return paths[0]
}

func dataFilePath() -> URL {
  return documentsDirectory().appendingPathComponent("Checklists.plist")
}
```

The `documentsDirectory()` method is something I've added for convenience. There is no standard method you can call to get the full path to the Documents folder, so I rolled my own.

The `dataFilePath()` method uses `documentsDirectory()` to construct the full path to the file that will store the checklist items. This file is named **Checklists.plist** and it lives inside the Documents directory.

Notice that both methods return a `URL` object. iOS uses URLs to refer to files on its filesystem. Where websites use `http://` or `https://` URLs, to find a file you use a `file://` URL.

> **Note:** Double check to make sure your code says `.documentDirectory` and not `.documentationDirectory`. Xcode's autocomplete can easily trip you up here!

➤ Still in **ChecklistViewController.swift**, add the following two `print` statements to the bottom of `init?(coder)`, below the call to `super.init()`:

```
required init?(coder aDecoder: NSCoder) {
  . . .
  super.init(coder: aDecoder)
```

```
    print("Documents folder is \(documentsDirectory())")
    print("Data file path is \(dataFilePath())")
  }
```

➤ Run the app. Xcode's debug area will now show you where your app's Documents directory is actually located.

If I run the app from the Simulator, on my system it says:

```
Documents folder is file:///Users/matthijs/Library/Developer/
CoreSimulator/Devices/66422991-21E3-4394-8DCE-0584865EA854/data/
Containers/Data/Application/96643C0B-2772-48D1-AD93-DBC2ACD4B779/
Documents/

Data file path is file:///Users/matthijs/Library/Developer/CoreSimulator/
Devices/66422991-21E3-4394-8DCE-0584865EA854/data/Containers/Data/
Application/96643C0B-2772-48D1-AD93-DBC2ACD4B779/Documents/
Checklists.plist
```

If you run it on your iPhone, the path will look somewhat different. Here's what mine says (this is on an iPod touch):

```
Documents folder is file:///var/mobile/Applications/
FDD50B54-9383-4DCC-9C19-C3DEBC1A96FE/Documents

Data file path is file:///var/mobile/Applications/
FDD50B54-9383-4DCC-9C19-C3DEBC1A96FE/Documents/Checklists.plist
```

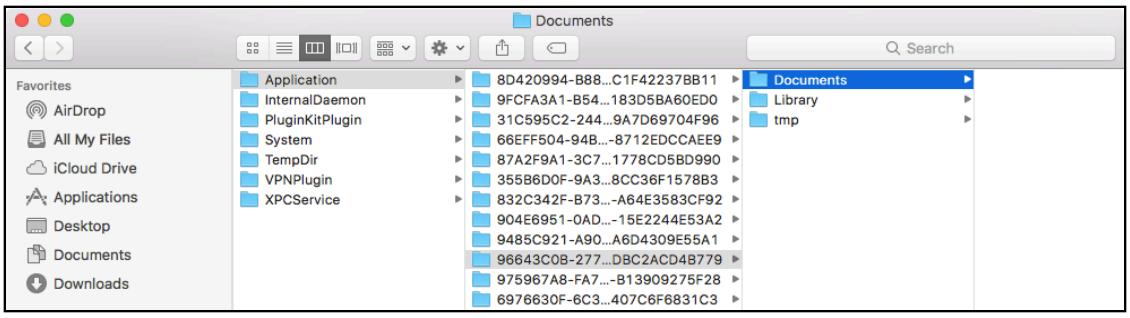
The name of the folder that contains the app's Documents folder is "96643C0B-2772-48D1-AD93-DBC2ACD4B779" (on the Simulator) and "FDD50B54-9383-4DCC-9C19-C3DEBC1A96FE" (on the device). There will be a quiz at the end of this section to see if you were able to memorize these folder names (I'm joking!).

The folder name is a random ID that Xcode picks when it installs the app on the Simulator or the device. Anything inside that folder is part of the app's sandbox.

For the rest of this section, run the app on the Simulator instead of a device. That makes it easier to look at the files you'll be writing into the Documents folder. Because the Simulator stores the app's files in a regular folder on your Mac, you can easily examine them from Finder.

➤ Open a new Finder window by clicking on the Desktop and typing ⌘**+N**. Then press ⌘**+Shift+G** and paste the full path to the Documents folder in the dialog. (Don't include the **file://** bit. The path starts with **/Users/yourname/...**)

The Finder window will go to that folder. Keep this window open so you can see that the Checklists.plist file is actually being created when you get to that part.



The app's directory structure in the Simulator

> **Tip:** If you want to navigate to the Simulator's app directories by hand, then you should know that the Library folder is hidden from your home directory. Hold down the Alt/Option key and click on Finder's Go menu. This will reveal the Library folder.

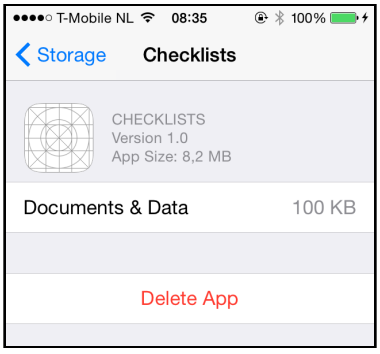
You can see several things inside the app's directory:

- The Documents directory where the app will put its data files. Currently the Documents folder is still empty.
- The Library directory has cache files and preferences files. The contents of this directory are managed by the operating system.
- The tmp directory is for temporary files. Sometimes apps need to create files for temporary usage. You don't want these to clutter up your Documents folder, so tmp is a good place to put them. iOS will clear out this folder from time to time.

It is also possible to look inside the Documents directory of apps on your device.

➤ On your iPhone or iPod, go to **Settings → General → Usage**. Under **Storage & iCloud Usage** tap **Manage Storage** and then the name of an app.

You'll now see the contents of its Documents folder:



Viewing the Documents folder on the device

Saving the checklist items

In this section you are going to write code that saves the list of to-do items to the Checklists.plist file when the user adds a new item or edits an existing item. Once you are able to save the items you'll add the code that is required to load this list again when the app starts up.

So what is a **.plist** file?

You've already seen a file named Info.plist in the Bull's Eye lesson. All apps have one, including the Checklists app (see the project navigator). Info.plist contains several configuration options that give iOS additional information about the app, such as what name to display under the app's icon on the home screen.

"plist" stands for Property List and is an XML file format that stores structured data, usually in the form of a list of settings and their values. Property List files are very common in iOS. They are suitable for many types of data storage, and best of all they are simple to use. What's not to like!

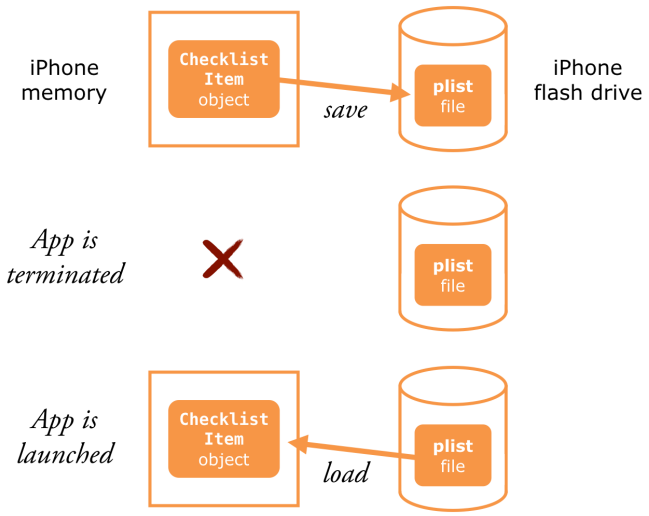
To save the checklist items you'll use the `NSCoder` system, which lets objects store their data in a structured file format.

You actually don't have to care much about that format. In this case it happens to be a .plist file but you're not directly going to mess with that file. All you care about is that the data gets stored in some kind of file in the app's Documents folder, but you'll leave the technical details for `NSCoder` to deal with.

You have already used `NSCoder` behind the scenes because that's exactly how storyboards work. When you add a view controller to a storyboard, Xcode uses the `NSCoder` system to write this object to a file (encoding). Then when your application starts up, it uses `NSCoder` again to read the objects from the storyboard file (decoding).

The process of converting objects to files and back again is also known as **serialization**. It's a big topic in software engineering.

I like to think of this whole process as freezing objects. You take a living object and freeze it so that it is suspended in time. You store that frozen object into a file on the iPhone's flash drive where it will spend some time in cryostasis. Later you can read that file into memory and defrost the object to bring it back to life again.



The process of freezing (saving) and unfreezing (loading) objects

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

```
func saveChecklistItems() {
  let data = NSMutableData()
  let archiver = NSKeyedArchiver(forWritingWith: data)
  archiver.encode(items, forKey: "ChecklistItems")
  archiver.finishEncoding()
  data.write(to: dataFilePath(), atomically: true)
}
```

This method takes the contents of the `items` array and in two steps converts it to a block of binary data and then writes this data to a file:

1. `NSKeyedArchiver`, which is a form of `NSCoder` that creates plist files, encodes the array and all the `ChecklistItems` in it into some sort of binary data format that can be written to a file.
2. That data is placed in an `NSMutableData object`, which will write itself to the file specified by `dataFilePath()`.

It's not really important that you understand how `NSKeyedArchiver` works internally. The format that it stores the data in isn't of great significance. All you care about is that it allows you to put your objects into a file and read them back later.

You have to call this new `saveChecklistItems()` method whenever the list of items is modified.

Exercise: Where in the source code would you call this method? ■

Answer: Look at where the `items` array is being changed. This happens inside the `ItemDetailViewControllerDelegate` methods. That's where the party's at!

➤ Add a call to `saveChecklistItems()` to the end of these methods inside **ChecklistViewController.swift**:

```
func itemDetailViewController(_ controller: ItemDetailViewController,
                              didFinishAdding item: ChecklistItem) {
  . . .
  saveChecklistItems()
}
```

```
func itemDetailViewController(_ controller: ItemDetailViewController,
                              didFinishEditing item: ChecklistItem) {
  . . .
  saveChecklistItems()
}
```

➤ Let's not forget the swipe-to-delete function:

```
override func tableView(_ tableView: UITableView,
                        commit editingStyle: UITableViewCellEditingStyle,
                        forRowAt indexPath: IndexPath) {
  . . .
  saveChecklistItems()
}
```

➤ And toggling the checkmark on a row on or off:

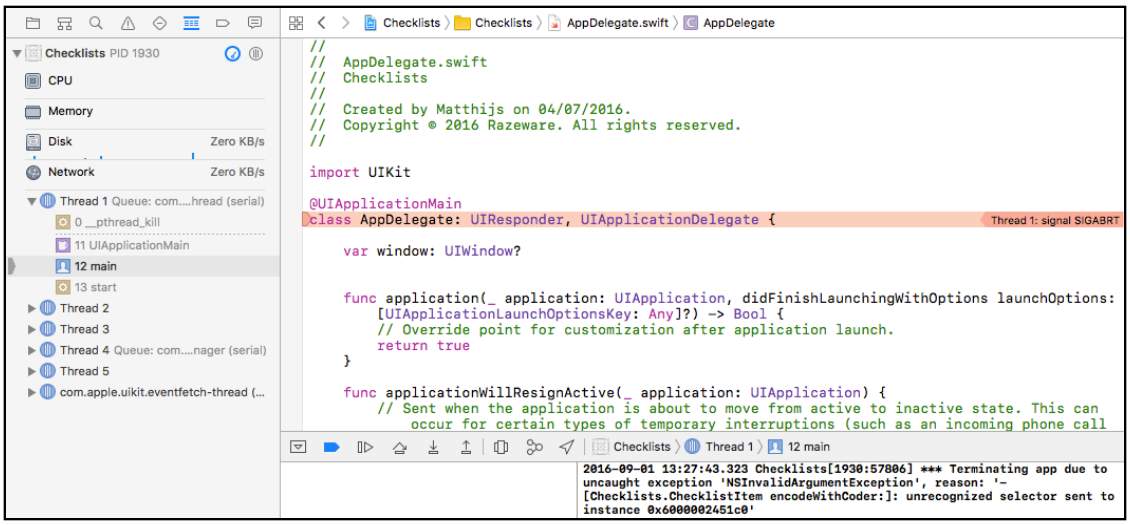
```
override func tableView(_ tableView: UITableView,
                         didSelectRowAt indexPath: IndexPath) {
  . . .
  saveChecklistItems()
}
```

Just calling `NSKeyedArchiver` on the `items` array is not enough. If you were to run the app now and do something that results in a save, such as tapping a row to flip the checkmark, the app crashes with the following error (try it out):

```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '-[Checklists.ChecklistItem
encodeWithCoder:]: unrecognized selector sent to instance 0x7f8d6af3aac0
```

A *selector* is a term Objective-C uses for the name of a method, so this warning means the app tried to call a method named `encodeWithCoder()` that doesn't actually exist anywhere. (Swift doesn't really use the term selector, but because the iOS frameworks are written in Objective-C you'll still see it being used in the documentation and in error messages.)

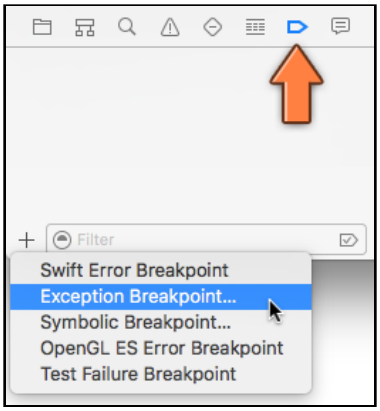
The Xcode window has switched to the *debugger* and points out which line caused the crash, more or less:



Xcode isn't being very helpful here

The debugger points at the AppDelegate.swift source file as the cause for the crash, but that's a little misleading. If this happened to you too, then you need to enable the Exception Breakpoint.

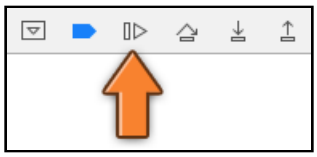
➤ Switch to the **Breakpoint navigator** and click the + button at the bottom:



Adding the Exception Breakpoint

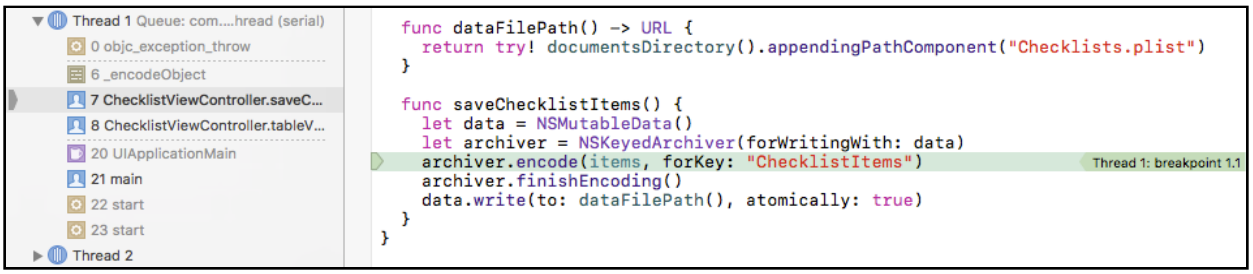
Now try running the app again.

Note: If the app now appears to "crash" right away, then somewhere deep inside UIKit an exception got triggered. This has nothing to do with your code. The exception breakpoint catches *any* exception, even if it is not fatal or it happens in the system frameworks. In that case, press the following button a few times until the app properly appears:



This is the "Continue program execution" button and you can find it near the bottom of the Xcode window.

➤ Tap a row to toggle the checkmark. This time Xcode points at the correct line, inside `saveChecklistItems()`:



The app crashes on the code you just added

This line is the culprit:

```
archiver.encode(items, forKey: "ChecklistItems")
```

Apparently the app crashes when trying to encode the `items` array, or rather the things inside the array.

The "unrecognized selector" crash message means you forgot to implement a certain method. In this case, the missing method appears to be `encode(with)` on the `ChecklistItem` object – that's what the crash message says.

Here is what happened: You asked `NSKeyedArchiver` to encode the array of items, so it not only has to encode the array itself but also each `ChecklistItem` object inside that array.

`NSKeyedArchiver` knows how to encode an `Array` object but it doesn't know anything about `ChecklistItem`. So you have to help it out a bit.

➤ Add `NSCoding` to the `class` line in **ChecklistItem.swift**:

```
class ChecklistItem: NSObject, NSCoding {
```

You're telling the compiler that `ChecklistItem` will conform to a new protocol, `NSCoding`.

The names get confusing: If you want to use the `NSCoder` system on an object, the object needs to implement the `NSCoding` protocol.

You've seen that a protocol is just a list of method names. Making an object conform to the protocol means the object should add implementations for the methods from that protocol.

The methods from the `NSCoding` protocol are:

- `func encode(with aCoder: NSCoder)`
- `init?(coder aDecoder: NSCoder)`

Only two methods, that can't be too bad! The first one is a regular method and it is used for saving – or encoding – the objects.

The other is a special init method. Recall that init methods are used during the creation of new objects. This one is for creating objects by loading – or decoding – them from a plist file.

➤ Add the following to **ChecklistItem.swift**:

```
func encode(with aCoder: NSCoder) {
  aCoder.encode(text, forKey: "Text")
  aCoder.encode(checked, forKey: "Checked")
}
```

This is the missing method from the unrecognized selector error.

When `NSKeyedArchiver` tries to encode the `ChecklistItem` object it will send the checklist item an `encode(with)` message.

Here you simply say: a `ChecklistItem` should save an object named "Text" that contains the value of the instance variable `text`, and an object named "Checked" that contains the value of the variable `checked`.

Just these two lines are enough to make the coder system work, at least for saving the to-do items.

Before you can build and run the app, you need to add some more code. Swift requires that you always implement all the required methods from a protocol and `NSCoding` has two methods, one for saving and one for loading.

➤ Add the second method to **ChecklistItem.swift**:

```
required init?(coder aDecoder: NSCoder) {
  super.init()
}
```

You're not going to use this right away, but it's needed to make the app compile without errors.

> **Note:** This should look familiar. You've already used `init?(coder)` before, to initialize the `ChecklistViewController`. That's because storyboards also use the

`NSCoding` system to load objects into the app.

Now, init methods are special in Swift. Because you just added `init?(coder)` you also need to add an `init()` method that takes no parameters. Without this, the app won't build. You'll learn more about why soon.

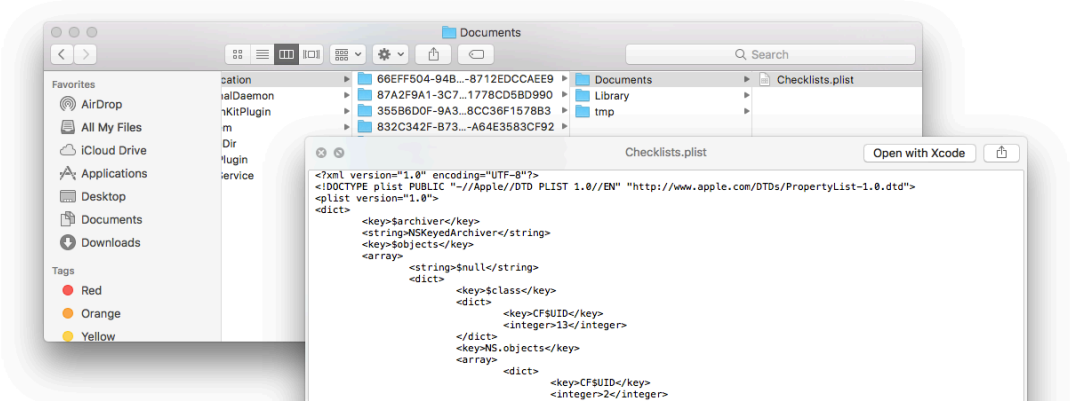
➤ Also add this method to **ChecklistItem.swift**:

```
override init() {
  super.init()
}
```

It doesn't do anything useful, but it keeps the compiler happy.

➤ Run the app again and tap a row to toggle a checkmark. The app didn't crash? Good!

➤ Go to the Finder window that has the app's Documents directory open:



The Documents directory now contains a Checklists.plist file

There is now a **Checklists.plist** file in the Documents folder, which contains the items from the list.

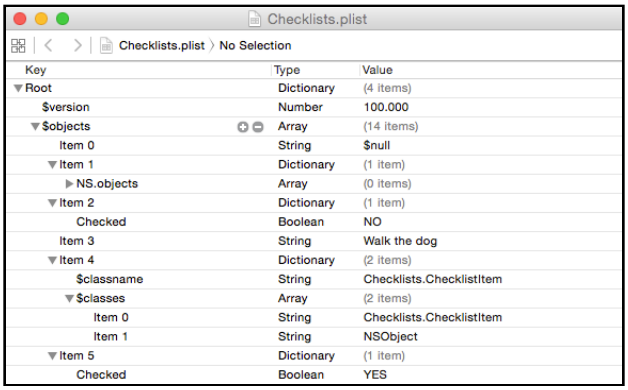
You can look inside this file if you want, but the contents won't make much sense. Even though it is XML, this file wasn't intended to be read by humans, only by the `NSKeyedArchiver` system.

If you're having trouble viewing the XML it may be because the plist file isn't stored as text but as a binary format. Some text editors support this file format and can read it as if it were text (TextWrangler is a good one and is a free download on the Mac App Store).

You can also use Finder's Quick Look feature to view the file. Simply select the file in Finder and press the space bar.

Naturally, you can also open the plist file with Xcode.

➤ Right-click the Checklists.plist file and choose **Open With → Xcode**.



Checklist.plist in Xcode

It still won't make much sense but it's fun to look at anyway.

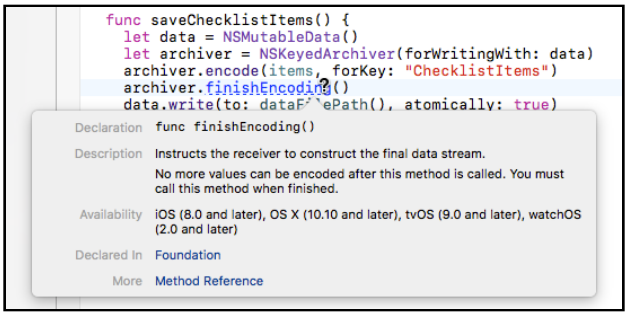
Expand some of the rows and you'll see this file was made by `NSKeyedArchiver` and that the names of the `ChecklistItems` are also in there. But exactly how all these data items fit together, I have no idea.

★ ★ ★

"NS" objects

Objects whose name start the "NS" prefix means are provided by the Foundation framework. NS stands for NextStep, the operating system from the 1990's that later became Mac OS X and also forms the basis of iOS.

If you are curious about exactly how objects such as `NSKeyedArchiver` and `NSCoding` work, you can Alt/Option-click any item in your source code to bring up a popup with a brief description:



I use this all the time to remind myself of how to use framework objects and their methods.

It's good to have a general idea of what objects are available in the frameworks, but no one can remember all the specifics. So get into the habit of looking up the documentation for any new objects and methods that you encounter. It'll make you learn the iOS frameworks much quicker!

Loading the file

Saving is all well and good but pretty useless by itself, so let's also implement the loading of the Checklists.plist file. It's very straightforward – you're basically going to do the same thing you just did but in reverse.

You've already added an empty `init?(coder)` to **ChecklistItem.swift**. This is the method for unfreezing the objects from the file.

➤ Make the following changes to `init?(coder)`:

```
required init?(coder aDecoder: NSCoder) {
  text = aDecoder.decodeObject(forKey: "Text") as! String
  checked = aDecoder.decodeBool(forKey: "Checked")
  super.init()
}
```

Inside `init?(coder)` you do the opposite from `encode(with)`. You take objects from the `NSCoder`'s decoder object and put their values inside your own properties. That's all it takes!

What you stored earlier under the "Text" key now goes back into the `text` instance variable. Likewise for `checked` and the boolean "Checked" value. Pay close attention: for `text` you use `decodeObject()` but for `checked` you need to use `decodeBool()`.

Initializers

Methods named "init" are special in Swift. They are only used when you're creating new objects, to make those new objects ready for use.

Think of it as having bought new clothes. The clothes are in your possession (the memory for the object is allocated) but they're still in the bag. You need to go change and put the new clothes on (initialization) before you're ready to go out and

party.

When you write the following to create a new object,

```
let item = ChecklistItem()
```

Swift first allocates a chunk of memory big enough to hold the new object and then calls `ChecklistItem`'s `init()` method with no parameters.

Loading the Checklists.plist file will be done by an `NSKeyedUnarchiver` object (you'll add this code in a minute). That unarchiver does the following behind the scenes to create the `ChecklistItem` objects:

```
let item = ChecklistItem(coder: someDecoderObject)
```

This also allocates memory for the new `ChecklistItem` but it calls `init?(coder)` instead of the regular `init()`.

It is pretty common for objects to have more than one init method. Which one is used depends on the circumstances.

You use `init()` for creating `ChecklistItem` objects when the user presses the + button, and you use `init?(coder)` to restore `ChecklistItems` that were saved to disk.

The implementations of these init methods, whether they're just called `init()` or `init?(coder)` or something else, always follow the same series of steps. When you write your own init methods, you need to stick to those steps as well.

This is the standard way to write an init method:

```
init() {
  // Put values into your instance variables and constants.

  super.init()

  // Other initialization code, such as calling methods, goes here.
}
```

Note that unlike other methods, `init` does not have the `func` keyword.

Sometimes you'll see it written as `override init` or `required init?`. That is necessary when you're adding the init method to an object that is a subclass of some other object. Much more about that later.

The question mark is for when `init?` can potentially fail and return a `nil` value instead of a real object. You can imagine that decoding an object can fail if not enough information is present in the plist file.

Inside the init method, you first need to make sure that all your instance variables and constants have a value. Recall that in Swift all variables must always have a value, except for optionals.

When you declare an instance variable you can give it an initial value, like so:

```
var checked = false
```

It's also possible to write just the variable name and its type, but not give the variable a value yet:

```
var checked: Bool
```

In that case, you have to give this variable a value inside your init method:

```
init() {
  checked = false
  super.init()
}
```

You must use either one of these approaches; if you don't give the variable a value at all, Swift considers this an error. The only exception is optionals, they do not need to have a value (in which case they are `nil`).

Once you've given all your instance variables and constants values, you call `super.init()` to initialize this object's superclass. If you haven't done any object-oriented programming at all, you may not know what a *superclass* is. That's fine; we'll completely ignore this topic until the next tutorial.

Just remember that sometimes objects need to send messages to something called `super` and if you forget to do this, bad things are likely to happen.

After calling `super.init()`, you can do additional initialization, such as calling the object's own methods. You're not allowed to do that before the call to `super.init()` because Swift has no guarantee that your object's variables all have proper values until then.

You don't always need to provide an init method. If your init method doesn't need to do anything – if there are no instance variables to fill in – then you can leave it out completely and the compiler will provide one for you.

When you first made `ChecklistItem`, it didn't have an `init()` method either. But now that you've added `init?(coder)` you also have to provide an `init()` that doesn't take any parameters.

Swift's rules for initializers can be a bit complicated, but fortunately the compiler will remind you when you forget to provide an init method.

The implementation of `ChecklistItem` is now complete. It can bring back to life the

objects that were serialized (frozen) into the plist file. But you still have to write the code that will actually load this plist. That happens in `ChecklistViewController`.

A table view controller, like many objects, has more than one init method. There is:

- `init?(coder)` for view controllers that are automatically loaded from a storyboard
- `init(nibName, bundle)` for view controllers that you manually want to load from a nib (a nib is like a storyboard but only contains a single view controller)
- `init(style)` for table view controllers that you want to create without using a storyboard or nib

This view controller comes from a storyboard, so you'll put the plist loading code into its `init?(coder)`. Yup, that's actually the same kind of method you've just implemented in `ChecklistItem`.

The `UITableViewController` object gets loaded and unfrozen from the storyboard file using the same `NSCoder` system that you used for your own files. If it's good enough for storyboards then it's certainly good enough for us!

➤ In **ChecklistViewController.swift**, replace `init?(coder)` with:

```
required init?(coder aDecoder: NSCoder) {
  items = [ChecklistItem]()
  super.init(coder: aDecoder)
  loadChecklistItems()
}
```

This follows the pattern for init methods:

1. First you make sure the instance variable `items` has a proper value (a new array).
2. Then you call super's version of `init()`. This time you call `super.init(coder)` to ensure the rest of the view controller is properly unfrozen from the storyboard.
3. Finally, you can call other methods. Here you call a new method to do the real work of loading the plist file.

> **Note:** Did you notice that `init?(coder)` has parameters with different external and internal labels? The label `coder` is part of the method name, but inside the method this parameter is called `aDecoder`.
>
> When you call `super.init`, you use the label `coder` to refer to the parameter of super's init method, and the object from `aDecoder` as that parameter's value.

➤ Also add the `loadChecklistItems()` method:

```
func loadChecklistItems() {
  // 1
```

```
    let path = dataFilePath()
    // 2
    if let data = try? Data(contentsOf: path) {
      // 3
      let unarchiver = NSKeyedUnarchiver(forReadingWith: data)
      items = unarchiver.decodeObject(forKey: "ChecklistItems")
                                                    as! [ChecklistItem]
      unarchiver.finishDecoding()
    }
  }
```

Let's go through this step-by-step:

1. First you put the results of `dataFilePath()` in a temporary constant named `path`.
2. Try to load the contents of Checklists.plist into a new `Data` object. The `try?` command attempts to create the `Data` object, but returns `nil` if it fails. That's why you put it in an `if let` statement.

   Why would it fail? If there is no Checklists.plist then there are obviously no `ChecklistItem` objects to load. This is what happens when the app is started up for the very first time. In that case, you'll skip the rest of this method.
3. When the app does find a Checklists.plist file, you'll load the entire array and its contents from the file. This is essentially the reverse of `saveChecklistItems()`.

   You create an `NSKeyedUnarchiver` object (note: this is an *un*archiver) and ask it to decode that data into the items array. This populates the array with exact copies of the `ChecklistItem` objects that were frozen into this file.

> **Note:** Double-check that both methods `loadChecklistItems()` and `saveChecklistItems()` use the same key name `"ChecklistItems"` for encoding and decoding the array. If you make a typo here, then the app won't work as expected.
>
> Normally Xcode is very good at pointing out typos but it's not smart enough to realize that the key name in the load and save methods must be the same. That's up to you, the human.

➤ Run the app and make some changes to the to-do items. Press Stop to terminate the app. Start it again and notice that your changes are still there.

➤ Stop the app again. Go to the Finder window with the Documents folder and remove the Checklists.plist file. Run the app once more. You should now have an empty screen.

➤ Add an item and notice that the Checklists.plist file re-appears.

Awesome! You've written an app that not only lets you add and edit data, but that also persists the data between sessions. These techniques form the basis of many,

many apps.

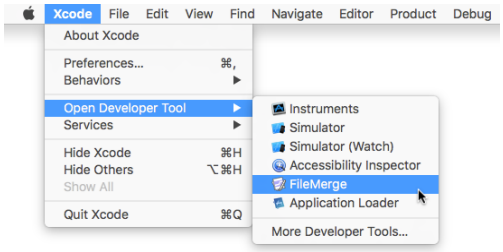
Being able to use a navigation controller, show modal edit screens, and pass data around through delegates are essential iOS development skills.

You can find the project files for the app up to this point under **06 - Saving and Loading** in the tutorial's Source Code folder.

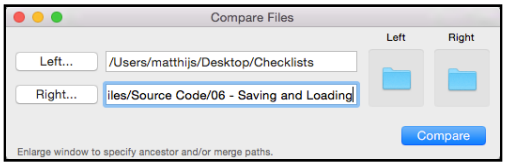
★ ★ ★

Using FileMerge to compare files

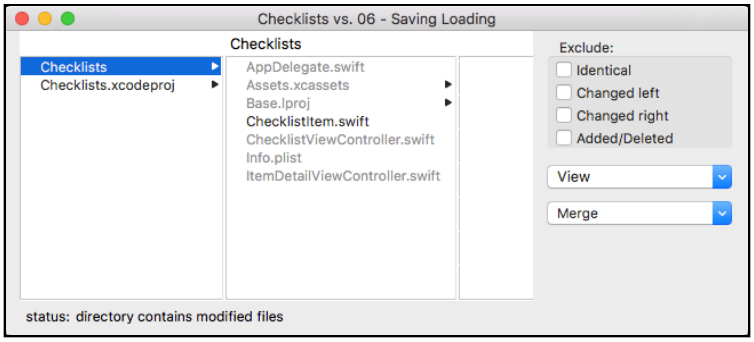
You can compare your own work with my version of the app using the FileMerge tool. Open this tool from the Xcode menu bar, under **Xcode → Open Developer Tool → FileMerge**:



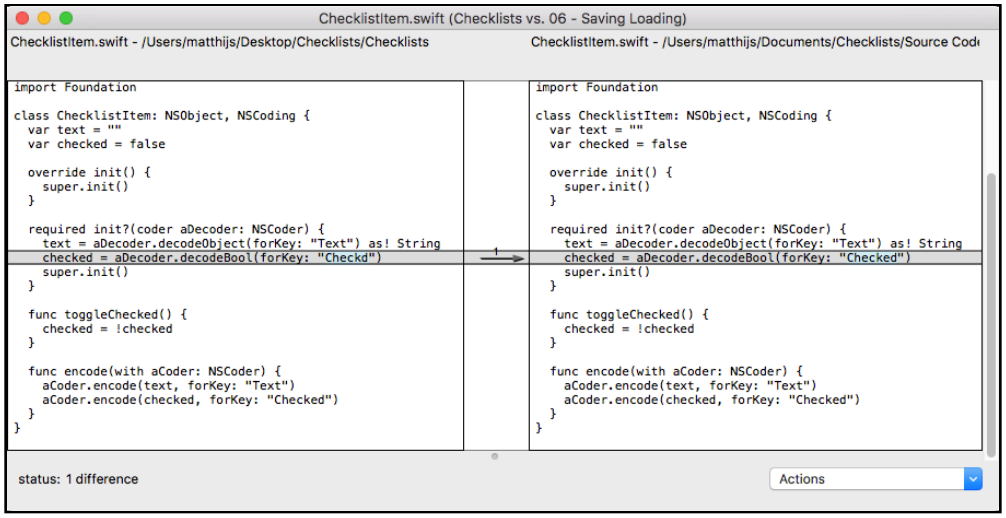
You give FileMerge two files or two folders to compare:



After working hard for a few seconds or so, FileMerge tells you what is different:



Double-click on a filename from the list to view the differences between the two files:



FileMerge is a wonderful tool for spotting the differences between two files or even entire folders. I use it all the time!

If something from the tutorials doesn't work as it should, then do a "diff" – that's what you're supposed to call it – between your own files and the ones from the Source Code folder to see if you can find any anomalies.

★ ★ ★

This is a good time to take a break, put your feet up, and daydream about all the cool apps you'll soon be writing.

It's also smart to go back and repeat those parts you're still a bit fuzzy about. Don't rush through these tutorials – there are no prizes for finishing first. Rather than going fast, take your time to truly understand what you've been doing.

As always, feel free to change the app and experiment. Breaking things is allowed – even encouraged – here at iOS Apprentice Academy!

Just to make sure you truly get everything you've done so far, next up you'll expand the app with new features that more or less repeat what you just did.

But I'll also throw in a few twists to keep it interesting…

Multiple checklists

The app is named Checklist**s** for a reason: it allows you to keep more than one list of to-do items. So far the app has only supported a single list but now you'll give it the capability to handle multiple checklists.

The steps for this section are:

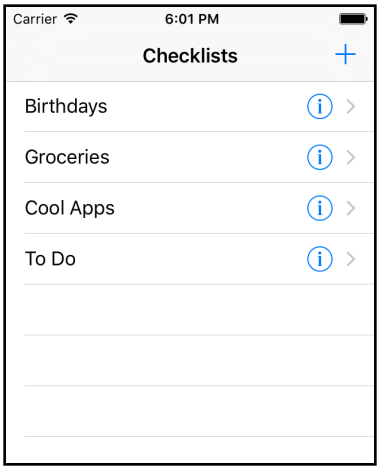
- Add a new screen that shows all the checklists.
- Create a screen that lets users add/edit checklists.
- Show the to-do items that belong to a particular checklist when you tap the name of that list.
- Save all the checklists to a file and load them in again.

Two new screens means two new view controllers:

1. `AllListsViewController` shows all the user's lists, and
2. `ListDetailViewController` allows adding a new list and editing the name and icon of an existing list.

You will first add the `AllListsViewController`. This becomes the new main screen of the app.

When you're done this is what it will look like:



The new main screen of the app

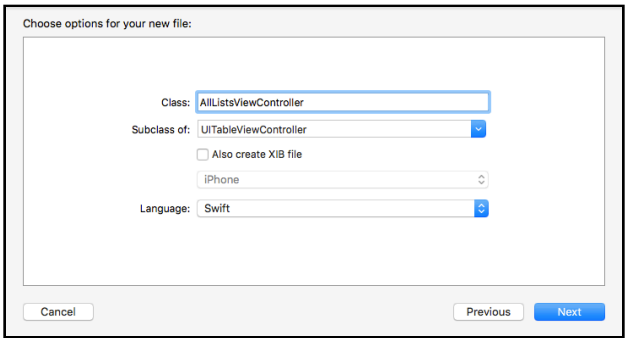
This screen is very similar to what you created before. It's a table view controller that shows a list of `Checklist` objects (not `ChecklistItem` objects).

From now on, I will refer to this screen as the "All Lists" screen and to the screen that shows the to-do items from a single checklist as the "Checklist" screen.

➤ Right-click the Checklists group in the project navigator and choose **New File**. Choose the **Cocoa Touch Class** template (under iOS, Source).

In the next step, choose the following options:

- Class: **AllListsViewController**
- Subclass of: **UITableViewController**
- Also create XIB file: Uncheck this
- Language: **Swift**



Choosing the options for the new view controller

Note: Make sure the "Subclass of" field is set to UI**Table**ViewController, not just "UIViewController". Also be careful that Xcode didn't rename what you typed into Class to "AllListsTableViewController" with the extra word "Table". It can be sneaky like that…

➤ Press **Next** and then **Create** to finish.

The Xcode template for table view controller objects puts a lot of stuff in this new file that you don't need. The template assumes you'll fill in this placeholder code before you run the app again. So let's clean that up first.

You'll also put some fake data in the table view just to get it up and running. As you know by now, I always like to take as small a step as possible and then run the app to see if it's working. Once everything works, you can move forward and put in the real data.

➤ In **AllListsViewController.swift**, remove the `numberOfSections(in)` method. Without it, there will always be a single section in the table view.

➤ Change the `tableView(numberOfRowsInSection)` method to:

```
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
  return 3
}
```

➤ Implement the `tableView(cellForRowAt)` method to put some text into the cells, just so there is something to see.

Note that the template already contains a commented-out version of this method. You can uncomment it by removing the `/*` and `*/` surrounding the method, and make your changes there.

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

  let cell = makeCell(for: tableView)
  cell.textLabel!.text = "List \(indexPath.row)"
  return cell
}
```

In `ChecklistViewController` the table view used prototype cells that you designed in Interface Builder. Just for the fun of it, in `AllListsViewController` you are taking a different approach where you'll create the cells in code instead.

➤ That requires you to add the following helper method:

```
func makeCell(for tableView: UITableView) -> UITableViewCell {
  let cellIdentifier = "Cell"
  if let cell =
         tableView.dequeueReusableCell(withIdentifier: cellIdentifier) {
    return cell
  } else {
    return UITableViewCell(style: .default,
                           reuseIdentifier: cellIdentifier)
  }
}
```

Later on I'll explain in more detail how this works, but for now recognize that you're using `dequeueReusableCell(withIdentifier)` here too. If it returns `nil`, there is no cell that can be recycled and you construct a new one with `UITableViewCell(style, reuseIdentifier)`.

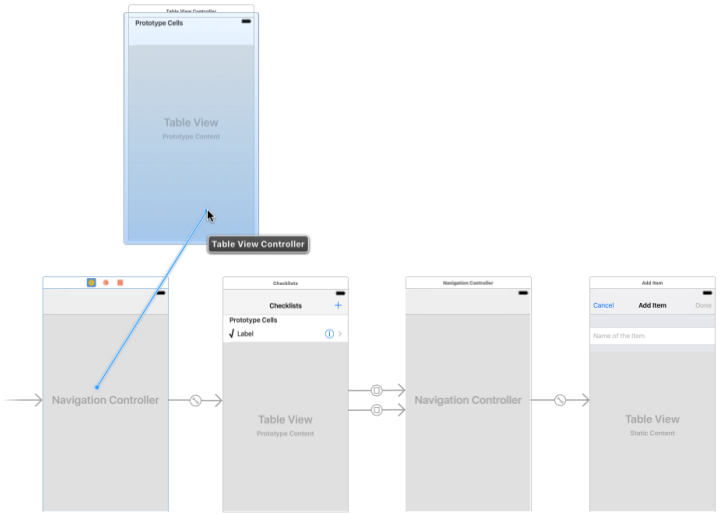
The reason you put this logic into a separate method is that it keeps the code in `tableView(cellForRowAt)` simple and clean. I find it more readable that way.

➤ Remove all the commented-out cruft from **AllListsViewController.swift**. Xcode puts it there to be helpful, but it also makes a mess of things.

The final step is to add this new view controller to the storyboard.

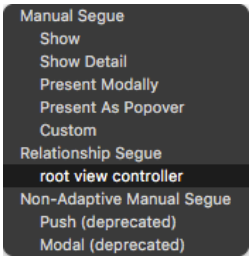
➤ Open the storyboard and drag a new **Table View Controller** onto the canvas. Put it somewhere near the first navigation controller.

➤ **Ctrl-drag** from the very first navigation controller to this new table view controller:



Ctrl-drag from the navigation controller to the new table view controller

From the popup menu choose **Relationship Segue - root view controller**:



Relationships are also segues

This will break the connection that existed between the navigation controller and the `ChecklistViewController` so that "Checklists" is no longer the app's main screen.

➤ Select the new table view controller and set its **Class** in the **Identity inspector** to **AllListsViewController**.

➤ Double-click the new view controller's navigation bar and change its title to **Checklists**.

This makes Xcode rename the view controller in the outline pane from All Lists View Controller to just Checklists, which is a bit confusing because there's a Checklists view controller already. You'll fix that in a minute.

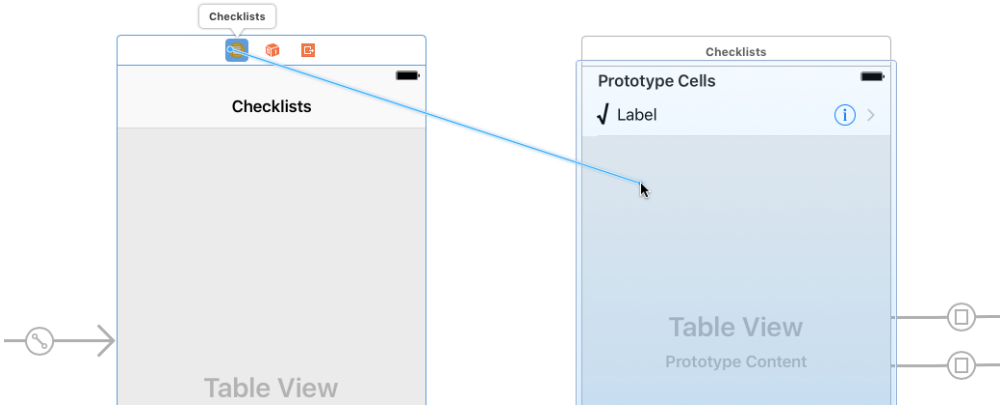
You may want to reorganize your storyboard at this point to make everything look neat again. The new table view controller goes in between the other scenes.

As I mentioned, you're not going to use prototype cells for this table view. It would be perfectly fine if you did, and as an exercise you could rewrite the code to use prototype cells later, but I want to show you another way of making table view

cells.

➤ Delete the empty prototype cell from the All Lists View Controller. (Simply select the Table View Cell and press **delete** on your keyboard.)

➤ **Ctrl-drag** from the yellow circle icon at the top of All Lists View Controller into the Checklist View Controller and create a **Show** segue.



Ctrl-dragging from the All Lists scene to the Checklist scene

This adds a "push" transition from the All Lists screen to the Checklist screen. It also puts the navigation bar back on the Checklist scene (the one on the right).

➤ Double-click the navigation bar to change its title to **(Name of the Checklist)**. This is just placeholder text. It helps tell the view controllers apart in the outline pane.

> **Note:** The outline pane doesn't show the name of the view controller object but the text from the navigation item. Very confusing, Xcode!
>
> When I refer to the All Lists View Controller, it's the plural "Checklists Scene" in the outline pane.
>
> The Checklist View Controller that shows a single list of to-do items is now found under "(Name of the Checklist) Scene".

Note that the new segue isn't attached to any button or table view cell.

There is nothing on the All Lists screen that you can tap or otherwise interact with in order to trigger this segue. That means you have to perform it programmatically.

➤ Click on the new segue to select it, go to the **Attributes inspector** and give it the identifier **ShowChecklist**.

The segue **Kind** should be **Show (e.g. Push)** because you're pushing the Checklist View Controller onto the navigation stack when performing this segue.

➤ In **AllListsViewController.swift**, add the `tableView(didSelectRowAt)` method:

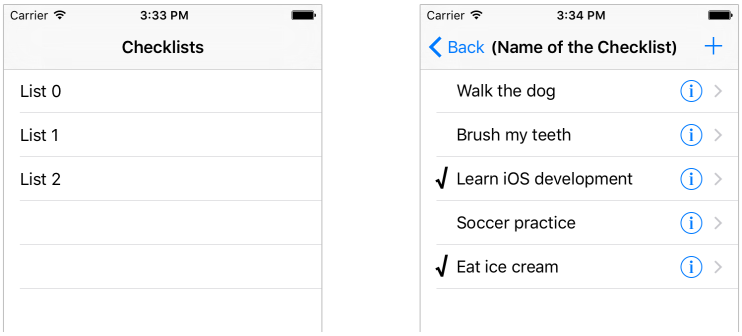
```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  performSegue(withIdentifier: "ShowChecklist", sender: nil)
}
```

Recall that this table view delegate method is invoked when you tap a row.

Previously, a tap on a row would automatically perform the segue because you had hooked up the segue to the prototype cell. However, the table view for this screen isn't using prototype cells and therefore you have to perform the segue manually.

That's simple enough: just call `performSegue(withIdentifier, sender)` with the name of the segue and things will start moving.

➤ Run the app. It now looks like this:



The first version of the All Lists screen (left). Tapping a row opens the Checklist screen (right).

Tap a row and the familiar `ChecklistViewController` slides into the screen.

You can tap the "Back" button in the top-left to go back to the main list. Now you're truly using the power of the navigation controller!

Putting lists into the All Lists screen

You're going to duplicate most of the functionality from the Checklist View Controller for this new All Lists screen.

There will be a + button at the top that lets users add new checklists, they can do swipe-to-delete, and they can tap the disclosure button to edit the name of the checklist.

Of course, you'll also save the array of `Checklist` objects to the Checklists.plist file.

Because you've already seen how this works, we'll go through the steps a bit quicker this time.

You begin by creating a data model object that represents a checklist.

➤ Add a new file to the project based on the **Cocoa Touch Class** template. Name it **Checklist** and make it a subclass of **NSObject**.

This adds the file Checklist.swift to the project.

Just like `ChecklistItem`, you're building `Checklist` on top of `NSObject`. This is a requirement for using the `NSCoder` system to load and save these objects.

➤ Give **Checklist.swift** a `name` property:

```
import UIKit

class Checklist: NSObject {
  var name = ""
}
```

Next, you'll give `AllListsViewController` an array that will store these new `Checklist` objects.

➤ Add a new instance variable to **AllListsViewController.swift**:

```
var lists: [Checklist]
```

This is an array that will hold the `Checklist` objects.

> **Note:** You can also write the above as follows:
>
> `var lists: Array<Checklist>`
>
> The version with the square brackets is what's known as *syntactic sugar* for the complete notation, which is `Array<`*type of the objects to put in the array*`>`.
>
> You will see both forms used in Swift programs and they do exactly the same thing. Because arrays are used a lot, the designers of Swift included the handy shorthand with the square brackets.

As a first step you will fill this new array with test data, which you'll do from the `init?(coder)` method. Remember that UIKit automatically invokes this method as it loads the view controller from the storyboard.

In **AllListsViewController.swift** you could add the following `init?(coder)` method (don't actually add it just yet, just read along with the description):

```
required init?(coder aDecoder: NSCoder) {
  // 1
  lists = [Checklist]()

  // 2
  super.init(coder: aDecoder)

  // 3
  var list = Checklist()
```

```
    list.name = "Birthdays"
    lists.append(list)

    // 4
    list = Checklist()
    list.name = "Groceries"
    lists.append(list)

    list = Checklist()
    list.name = "Cool Apps"
    lists.append(list)

    list = Checklist()
    list.name = "To Do"
    lists.append(list)
  }
```

You've seen something very much like it a while ago when you added the fake test data to `ChecklistViewController`. Here is what it does step-by-step:

1. Give the `lists` variable a value. You can also write this as `lists = Array<Checklist>()` – that does the exact same thing. I just like the square brackets better.
2. Call `super`'s version of `init?(coder)`. Without this, the view controller won't be properly loaded from the storyboard. But don't worry too much about forgetting to call `super`; if you don't, Xcode gives an error message.
3. Create a new `Checklist` object, give it a name, and add it to the array.
4. Here you create three more `Checklist` objects. Because you declared the local variable `list` as `var` instead of `let`, you can re-use it.

Notice how this is performing the same two steps for every new `Checklist` object you're creating?

```
  list = Checklist()
  list.name = "Name of the checklist"
```

It seems likely that every `Checklist` you'll ever make will also have a name. You can make this a requirement by writing your own init method that takes the name as a parameter. Then you can simply write:

```
  list = Checklist(name: "Name of the checklist")
```

➤ Go to **Checklist.swift** and add the new init method:

```
  init(name: String) {
    self.name = name
    super.init()
  }
```

This initializer takes one parameter, `name`, and places it into the instance variable,

which is also called `name`.

Because both the parameter and the instance variable are called `name`, you use `self.name` to refer to the instance variable.

If you tried to do this,

```
init(name: String) {
  name = name
  super.init()
}
```

then Swift wouldn't understand that the first `name` referred to the instance variable.

To disambiguate, you use `self`. Recall that `self` refers to the object that you're in, so `self.name` means the `name` variable of the current `Checklist` object.

➤ Go back to **AllListsViewController.swift** and add `init?(coder)`, for real this time:

```
required init?(coder aDecoder: NSCoder) {
  lists = [Checklist]()

  super.init(coder: aDecoder)

  var list = Checklist(name: "Birthdays")
  lists.append(list)

  list = Checklist(name: "Groceries")
  lists.append(list)

  list = Checklist(name: "Cool Apps")
  lists.append(list)

  list = Checklist(name: "To Do")
  lists.append(list)
}
```

That's a bit shorter than what I showed you before, and it guarantees that new `Checklist` objects will now always have their `name` property filled in.

Note that you don't write:

```
var list = Checklist.init(name: "Birthdays")
```

Even though the method is named `init`, it's not a regular method. Initializers are only used to construct new objects and you write that as:

```
var object = ObjectName(parameter1: value1, parameter2: value2, . . .)
```

Depending on the parameters that you specified, Swift will locate the corresponding `init` method and call that.

Clear? Great! Let's continue building the All Lists screen.

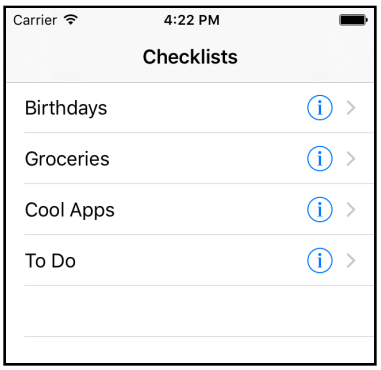
➤ Change the `tableView(numberOfRowsInSection)` method to return the number of objects in the new array:

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

➤ Finally, change `tableView(cellForRowAt)` to fill in the cells for the rows:

```
override func tableView(_ tableView: UITableView,
                 cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let cell = makeCell(for: tableView)

  let checklist = lists[indexPath.row]
  cell.textLabel!.text = checklist.name
  cell.accessoryType = .detailDisclosureButton

  return cell
}
```

➤ Run the app. It looks like this:



The table view shows Checklist objects

It has a table view with cells representing `Checklist` objects. The rest of the screen doesn't do much yet, but it's a start.

★ ★ ★

The many ways to make table view cells

Creating a new table view cell in `AllListsViewController` is a little more involved than what happens in `ChecklistViewController`. There you just did the following to obtain a new table view cell:

```
let cell = tableView.dequeueReusableCell(
```

```
                            withIdentifier: "ChecklistItem", for: indexPath)
```

But here you have a whole chunk of code to accomplish the same:

```
  let cellIdentifier = "Cell"
  if let cell =
         tableView.dequeueReusableCell(withIdentifier: cellIdentifier) {
    return cell
  } else {
    return UITableViewCell(style: .default,
                           reuseIdentifier: cellIdentifier)
  }
```

The call to `dequeueReusableCell(withIdentifier)` is still there, except that previously the storyboard had a prototype cell with that identifier and now it doesn't.

If the table view cannot find a cell to re-use (and it won't until it has enough cells to fill the entire visible area), this method will return `nil` and you have to create your own cell by hand. That's what happens in the `else` section.

There are actually two versions of `dequeueReusableCell(…)`, one with an extra `for` parameter that takes an `IndexPath`, and one without. Here you're using the one without. The difference is that `dequeueReusableCell(withIdentifier, for)` only works with prototype cells. If you tried to use it here, it would crash the app.

There are four ways that you can make table view cells:

1. Using prototype cells. This is the simplest and quickest way. You did this in `ChecklistViewController`.
2. Using static cells. You did this for the Add/Edit Item screen. Static cells are limited to screens where you know in advance which cells you'll have. The big advantage with static cells is that you don't need to provide any of the data source methods (`cellForRowAt` and so on).
3. Using a *nib* file. A nib (also known as a XIB) is like a mini storyboard that only contains a single customized `UITableViewCell` object. This is very similar to using prototype cells, except that you can do it outside of a storyboard.
4. By hand, what you did above. This is how you were supposed to do it in the early days of iOS. Chances are you'll run across code examples that do it this way, especially from older articles and books. It's a bit more work but also offers you the most flexibility.

When you create a cell by hand you specify a certain **cell style**, which gives you a cell with a preconfigured layout that already has labels and an image view.

For the All Lists View Controller you're using the "Default" style. Later in this tutorial you'll switch it to "Subtitle", which gives the cell a second, smaller label below the main label.

Using standard cell styles means you don't have to design your own cell layout. For many apps these standard layouts are sufficient so that saves you some work.

Prototype cells and static cells can also use these standard cell styles. The default style for a prototype or static cell is "Custom", which requires you to use your own labels, but you can change that to one of the built-in styles with Interface Builder.

And finally, a gentle warning: Sometimes I see code that creates a new cell for every row rather than trying to reuse cells. Don't do that! Always ask the table view first whether it has a cell available that can be recycled, using one of the `dequeueReusableCell(…)` methods.

Creating a new cell for each row will cause your app to slow down, as object creation is slower than simply re-using an existing object. Creating all these new objects also takes up more memory, which is a precious commodity on mobile devices. For the best performance, reuse those cells!

Viewing the checklists

Right now, the data model consists of the `lists` array from `AllListsViewController` that contains a handful of `Checklist` objects. There is also a separate `items` array in `ChecklistViewController` with `ChecklistItem` objects.

You may have noticed that when you tap the name of a list, the Checklist screen slides into view but it currently always shows the same to-do items, regardless of which list you tapped on.

Each checklist should really have its own to-do items. You'll work on that later in this tutorial, as this requires a significant change to the data model.

As a start, let's set the title of the screen to reflect the chosen checklist.

➤ Add a new instance variable to **ChecklistViewController.swift**:

```
var checklist: Checklist!
```

I'll explain why the exclamation mark is necessary in a moment.

➤ Change the `viewDidLoad()` method in **ChecklistViewController.swift** to:

```
override func viewDidLoad() {
  super.viewDidLoad()
  title = checklist.name
}
```

This changes the title of the screen, which is shown in the navigation bar, to the

name of the `Checklist` object.

You'll give this `Checklist` object to the `ChecklistViewController` when the segue is performed.

➤ In **AllListsViewController.swift**, update `tableView(didSelectRowAt)` to the following:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  let checklist = lists[indexPath.row]
  performSegue(withIdentifier: "ShowChecklist", sender: checklist)
}
```

As before, you use `performSegue()` to start the segue. This method has a `sender` parameter that you previously set to `nil`. Now you'll use it to send along the `Checklist` object from the row that the user tapped on.

You can put anything you want into `sender`. If the segue is performed by the storyboard (rather than manually like you do here) then `sender` will refer to the control that triggered it, for example the `UIBarButtonItem` object for the Add button or the `UITableViewCell` for a row in the table.

But because you start this particular segue by hand, you can put into `sender` whatever is most convenient.

Putting the `Checklist` object into the `sender` parameter doesn't give this object to the `ChecklistViewController` yet. That happens in "prepare-for-segue", which you still need to write for this view controller.

➤ Add the `prepare(for:sender:)` method to **AllListsViewController.swift**:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "ShowChecklist" {
    let controller = segue.destination as! ChecklistViewController
    controller.checklist = sender as! Checklist
  }
}
```

You've seen this method before. `prepare(for:sender:)` is called right before the segue happens. Here you get a chance to set the properties of the new view controller before it will become visible.

> **Note:** The segue's `destination` is the `ChecklistViewController`, not a `UINavigationController`. That is different from before.
>
> The segue to the Add/Edit Item screen was to a modally presented view controller that was embedded inside a navigation controller.
>
> This time the "push" segue is directly to the Checklist view controller.
>
> Look in the storyboard and you'll see there is no navigation controller between the All Lists screen and the Checklist screen. The segue goes directly from one

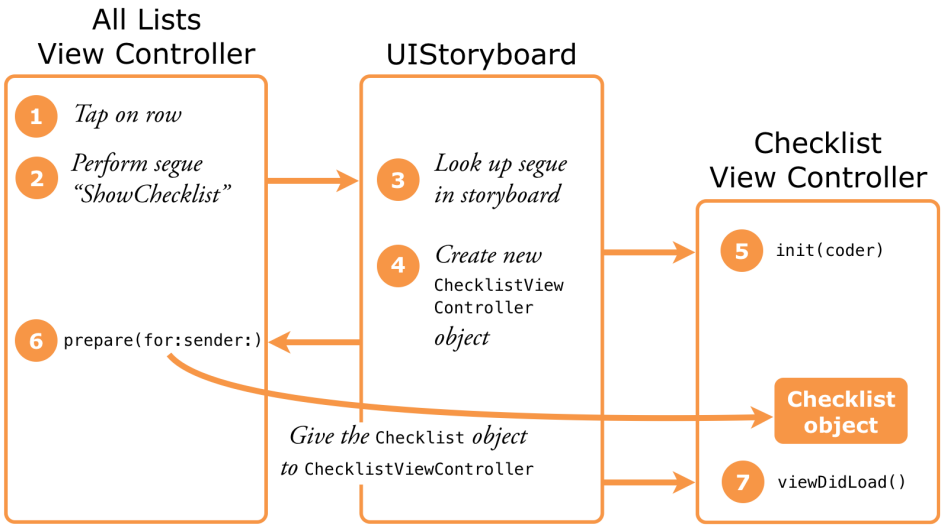
to the other.

Inside `prepare(for:sender:)`, you need to give the `ChecklistViewController` the `Checklist` object from the row that the user tapped. That's why you put that object in the `sender` parameter earlier.

(You could have temporarily stored the `Checklist` object in an instance variable instead but passing it along in the `sender` parameter is much easier.)

All of this happens a short time after `ChecklistViewController` is instantiated but just before `ChecklistViewController`'s view is loaded. That means its `viewDidLoad()` method is called after `prepare(for:sender:)`.

At this point, the view controller's `checklist` property is filled in with the `Checklist` object from `sender`, and `viewDidLoad()` can set the title of the screen accordingly.



The steps involved in performing a segue

This sequence of events is why the `checklist` property is declared as `Checklist!` with an exclamation point. That allows its value to be temporarily `nil` until `viewDidLoad()` happens.

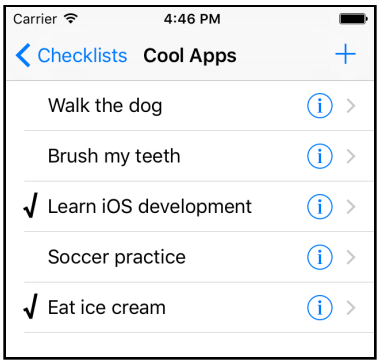
`nil` is normally not an allowed value for variables in Swift but by using the `!` you override that.

Does this sound an awful lot like optionals? The exclamation point turns `checklist` into a special kind of optional. It's very similar to optionals with a question mark, but you don't have to write `if let` to unwrap it.

Such *implicitly unwrapped* optionals should be used sparingly and with care, as they

do not have any of the anti-crash protection that normal optionals do.

➤ Run the app and notice that when you tap the row for a checklist, the next screen properly takes over the title.



The name of the chosen checklist now appears in the navigation bar

Note that giving the `Checklist` object to the `ChecklistViewController` does not make a copy of it.

You only pass the view controller a *reference* to that object – any changes the user makes to that `Checklist` object are also seen by `AllListsViewController`.

Both view controllers have access to the exact same `Checklist` object. You'll use that to your advantage later in order to add new `ChecklistItems` to the `Checklist`.

★ ★ ★

Type Casts

In `prepare(for:sender:)` you do this:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  . . .
  controller.checklist = sender as! Checklist
  . . .
}
```

What is that "`as! Checklist`" thing?

If you've been paying attention – of course you have! – then you've seen this "as something" used quite a few times now. This is known as a *type cast*.

A type cast tells Swift to interpret a value as having a different data type.

(It's the opposite of what happens to certain actors in the movies. For them typecasting results in always playing the same character; in Swift, a type cast

actually changes the character of an object.)

Here, `sender` has type `Any?`, meaning that it can be any sort of object: a `UIBarButtonItem`, a `UITableViewCell`, or in this case a `Checklist`. Thanks to the question mark it can even be `nil`.

But the `controller.checklist` property always expects a proper `Checklist` object – it wouldn't know what to do with a `UITableViewCell`… Hence, Swift demands that you only put `Checklist` objects into the `checklist` property.

By writing "`sender as! Checklist`", you tell Swift that it can safely treat `sender` as a `Checklist` object.

Another example of a typecast is:

```
let controller = segue.destination as! ChecklistViewController
```

The segue's `destination` property refers to the view controller on the receiving end of the segue. But obviously the engineers at Apple could not predict beforehand that we would call it `ChecklistViewController`.

So you have to cast it from its generic type (`UIViewController`) to the specific type used in this app (`ChecklistViewController`) before you can access any of its properties.

One final example, from `loadChecklistItems()`:

```
items = unarchiver.decodeObjectForKey("ChecklistItems")
                                                as! [ChecklistItem]
```

The `NSKeyedUnarchiver` object decodes the object frozen under the key `"ChecklistItems"` into an array, but you still need to tell Swift that this really is an array containing `ChecklistItem` objects.

Without this type cast, Swift considers it an `Any` object, which is incompatible with the data type of the `items` array.

Note that there is also `as?` with a question mark. This is for casting optionals, or when the type cast is allowed to fail. You'll see some examples of that later.

Don't worry if any of this goes over your head right now. You'll see plenty more examples of type casting in action.

The main reason you need all these type casts is interoperability with the iOS frameworks that are written in Objective-C. Swift is less forgiving about types than Objective-C and requires you to be much more explicit (about types; it's not encouraging you to swear more).

Adding and editing checklists

Let's quickly add the Add Checklist / Edit Checklist screen. This is going to be yet another `UITableViewController`, with static cells, and you'll present it modally from the `AllListsViewController`.

If the previous sentence made perfect sense to you, then you're getting the hang of this!

➤ Add a new file to the project, **ListDetailViewController.swift**. You can either use the **Cocoa Touch Class** template or the **Swift File** template for this.

➤ Replace the contents of **ListDetailViewController.swift** with:

```
import UIKit

protocol ListDetailViewControllerDelegate: class {
  func listDetailViewControllerDidCancel(
                                  _ controller: ListDetailViewController)

  func listDetailViewController(_ controller: ListDetailViewController,
                                  didFinishAdding checklist: Checklist)

  func listDetailViewController(_ controller: ListDetailViewController,
                                  didFinishEditing checklist: Checklist)
}

class ListDetailViewController: UITableViewController,
                                                  UITextFieldDelegate {
  @IBOutlet weak var textField: UITextField!
  @IBOutlet weak var doneBarButton: UIBarButtonItem!

  weak var delegate: ListDetailViewControllerDelegate?

  var checklistToEdit: Checklist?
}
```

I simply took the contents of ItemDetailViewController.swift and changed the names. Also, instead of a property for a `ChecklistItem` you're now dealing with a `Checklist`.

➤ Add the `viewDidLoad()` method:

```
override func viewDidLoad() {
  super.viewDidLoad()

  if let checklist = checklistToEdit {
    title = "Edit Checklist"
    textField.text = checklist.name
```

```
      doneBarButton.isEnabled = true
    }
  }
```

This changes the title of the screen if the user is editing an existing checklist, and it puts the checklist's name into the text field already.

➤ Also add the `viewWillAppear()` method to pop up the keyboard:

```
  override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    textField.becomeFirstResponder()
  }
```

➤ Add the action methods for the Cancel and Done buttons:

```
  @IBAction func cancel() {
    delegate?.listDetailViewControllerDidCancel(self)
  }

  @IBAction func done() {
    if let checklist = checklistToEdit {
      checklist.name = textField.text!
      delegate?.listDetailViewController(self,
                                         didFinishEditing: checklist)
    } else {
      let checklist = Checklist(name: textField.text!)
      delegate?.listDetailViewController(self,
                                         didFinishAdding: checklist)
    }
  }
```

This should look familiar as well. It's essentially the same as what the Add/Edit Item screen does.

To create the new `Checklist` object in `done()`, you use its `init(name)` method and pass the contents of `textField.text` into the `name` parameter.

You cannot write this the way you did for `ChecklistItems` – this won't work:

```
  let checklist = Checklist()
  checklist.name = textField.text!
```

Because `Checklist` does not have an `init()` method that takes no parameters, writing `Checklist()` results in a compiler error. It only has an `init(name)` method, and you must always use that initializer to create new `Checklist` objects.

➤ Also make sure the user cannot select the table cell with the text field:

```
  override func tableView(_ tableView: UITableView,
                          willSelectRowAt indexPath: IndexPath) -> IndexPath? {
    return nil
  }
```

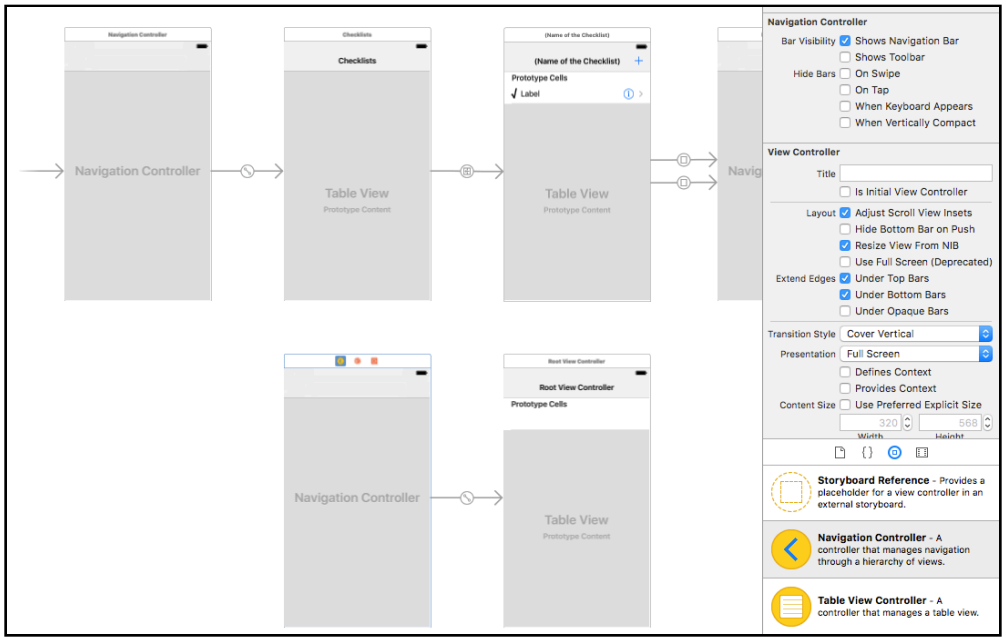
➤ And finally, add the text field delegate method that enables or disables the Done button depending on whether the text field is empty or not.

```
func textField(_ textField: UITextField,
               shouldChangeCharactersIn range: NSRange,
               replacementString string: String) -> Bool {

  let oldText = textField.text! as NSString
  let newText = oldText.replacingCharacters(in: range, with: string)
                                            as NSString

  doneBarButton.isEnabled = (newText.length > 0)
  return true
}
```

Again, this is the same as what you did in `ItemDetailViewController`.

Let's make the user interface for this new view controller in Interface Builder.

➤ Open the storyboard. Drag a new **Navigation Controller** from the Object Library into the canvas and move it below the other view controllers.



Dragging a new navigation controller into the canvas

Interface Builder already assumes that you want to embed a table view controller inside the navigation controller, so that saves you some work.

➤ Select the new Table View Controller (the one named "Root View Controller") and go to the **Identity inspector**. Change its class to **ListDetailViewController**.

➤ Change the navigation bar title from "Root View Controller" to **Add Checklist**.

(If double-clicking the navigation bar doesn't work, select the Root View Controller

navigation item in the outline pane and use the Attributes inspector.)

➤ Add **Cancel** and **Done** bar button items and hook them up to the action methods in the view controller. Also connect the Done button to the **doneBarButton** outlet and uncheck its **Enabled** option.

Remember, you can Ctrl-drag from a button to the view controller to connect it to an action method. To connect an outlet, do it the other way around: Ctrl-drag from the view controller to the button.

Tip: My Xcode acted a bit buggy and wouldn't let me drop the bar buttons on the navigation bar. If this happens to you too, drop them on the navigation item – now called Add Checklist – in the outline pane. You can also Ctrl-drag in the outline pane to make the connections to the actions and the outlet.

➤ Change the table view to **Static Cells**, style **Grouped**. You only need one cell, so remove the bottom two.

➤ Drop a new **Text Field** into the cell. Here are the configuration options:

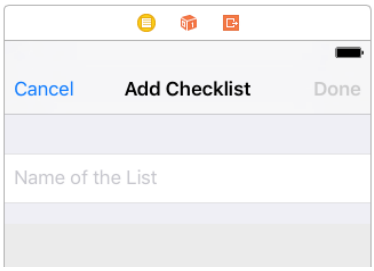
- Border Style: none
- Font size: 17
- Placeholder text: **Name of the List**
- Adjust to Fit: disabled
- Capitalization: Sentences
- Return Key: Done
- Auto-enable Return key: check

➤ Ctrl-drag from the view controller to the Text Field and connect it to the **textField** outlet.

➤ Then Ctrl-drag the other way around, from the Text Field back to the view controller, and choose **delegate** under **Outlets**. Now the view controller is the delegate for the text field.

➤ Connect the text field's **Did End on Exit** event to the **done** action on the view controller.

This completes the steps for converting this view controller to the Add / Edit Checklist screen:



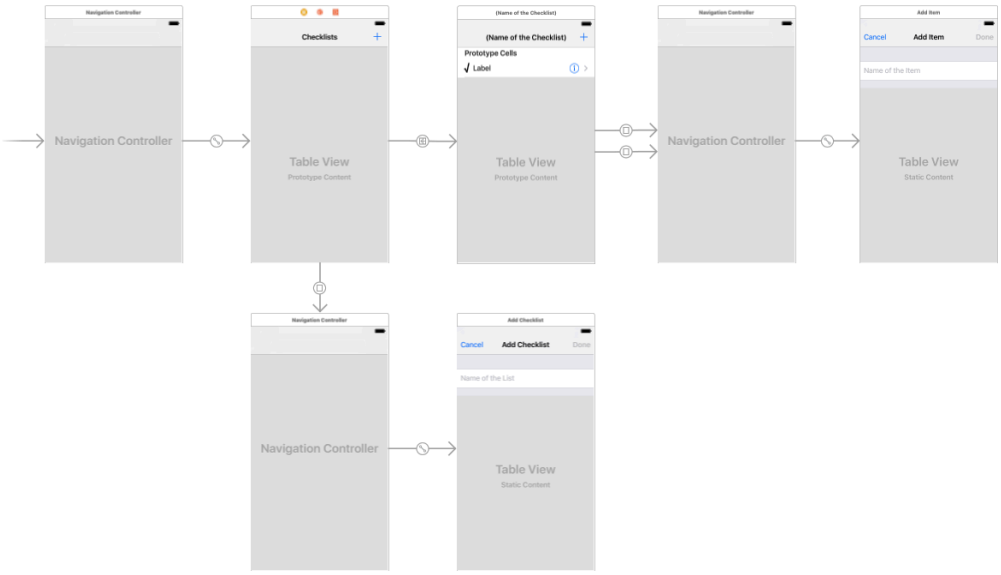
The finished design of the ListDetailViewController

➤ Go to the **All Lists View Controller** (the one titled "Checklists") and drag a **Bar Button Item** into its navigation bar. Change it into an **Add** button.

➤ **Ctrl-drag** from this new bar button to the navigation controller below to add a new **Present Modally** segue.

➤ Click on the new segue and name it **AddChecklist**.

Your storyboard should now look like this:



The full storyboard: 3 navigation controllers, 4 table view controllers

Almost there. You still have to make the `AllListsViewController` the delegate for the `ListDetailViewController` and then you're done. Again, it's very similar to what you did before.

➤ Declare the All Lists view controller to conform to the delegate protocol by `adding ListDetailViewControllerDelegate` to its class line.

You do this in **AllListsViewController.swift**:

```
class AllListsViewController: UITableViewController,
                                    ListDetailViewControllerDelegate {
```

(This goes all on one line)

➤ Also in **AllListsViewController.swift**, first extend `prepare(for:sender:)` to:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "ShowChecklist" {
    . . .
  } else if segue.identifier == "AddChecklist" {
    let navigationController = segue.destination
                                                  as! UINavigationController
    let controller = navigationController.topViewController
                                                as! ListDetailViewController
    controller.delegate = self
    controller.checklistToEdit = nil
  }
}
```

The first `if` doesn't change. You've added a second `if` for the new "AddChecklist" segue that you just defined in the storyboard.

As before, you look for the view controller inside the navigation controller (which is the `ListDetailViewController`) and set its `delegate` property to `self`.

➤ At the bottom of the **AllListsViewController.swift**, implement the following delegate methods.

```
func listDetailViewControllerDidCancel(
                                _ controller: ListDetailViewController) {
  dismiss(animated: true, completion: nil)
}

func listDetailViewController(_ controller: ListDetailViewController,
                              didFinishAdding checklist: Checklist) {
  let newRowIndex = lists.count
  lists.append(checklist)

  let indexPath = IndexPath(row: newRowIndex, section: 0)
  let indexPaths = [indexPath]
  tableView.insertRows(at: indexPaths, with: .automatic)

  dismiss(animated: true, completion: nil)
}

func listDetailViewController(_ controller: ListDetailViewController,
                              didFinishEditing checklist: Checklist) {
  if let index = lists.index(of: checklist) {
    let indexPath = IndexPath(row: index, section: 0)
    if let cell = tableView.cellForRow(at: indexPath) {
      cell.textLabel!.text = checklist.name
    }
  }
  dismiss(animated: true, completion: nil)
}
```

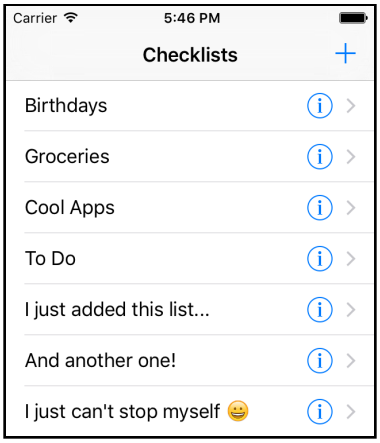
These methods are called when the user presses Cancel or Done inside the new Add/Edit Checklist screen.

None of this code should surprise you. It's exactly what you did before but now for the `ListDetailViewController` and `Checklist` objects.

➤ Also add the table view data source method that allows the user to delete checklists:

```
override func tableView(_ tableView: UITableView,
                        commit editingStyle: UITableViewCellEditingStyle,
                        forRowAt indexPath: IndexPath) {
  lists.remove(at: indexPath.row)

  let indexPaths = [indexPath]
  tableView.deleteRows(at: indexPaths, with: .automatic)
}
```

➤ Run the app. Now you can add new checklists and delete them again:



Adding new lists

Note: If the app crashes, then go back and make sure you made all the connections properly in Interface Builder. It's really easy to miss just one tiny thing, but even the tiniest of mistakes can bring the app down in flames…

You can't edit the names of existing lists yet. That requires one last addition to the code.

To bring up the Edit Checklist screen, the user taps the blue accessory button. In the `ChecklistViewController` that triggered a segue. You could use a segue here too, but I want to show you another way.

This time you're not going to use a segue at all, but load the new view controller by hand from the storyboard. Just because you can.

➤ Add the following `tableView(accessoryButtonTappedForRowWith)` method to **AllListsViewController.swift**. This method comes from the table view delegate protocol and the name is hopefully obvious enough to guess what it does.

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

  let navigationController = storyboard!.instantiateViewController(
                      withIdentifier: "ListDetailNavigationController")
                          as! UINavigationController

  let controller = navigationController.topViewController
                                          as! ListDetailViewController
  controller.delegate = self

  let checklist = lists[indexPath.row]
  controller.checklistToEdit = checklist

  present(navigationController, animated: true, completion: nil)
}
```

Inside this method you create the view controller object for the Add/Edit Checklist screen and show it ("present" it) on the screen. This is roughly equivalent to what a segue would do behind the scenes. The view controller is embedded in a storyboard and you have to ask the storyboard object to load it.

Where did you get that storyboard object? As it happens, each view controller has a `storyboard` property that refers to the storyboard the view controller was loaded from. You can use that property to do all kinds of things with the storyboard, such as instantiating other view controllers.

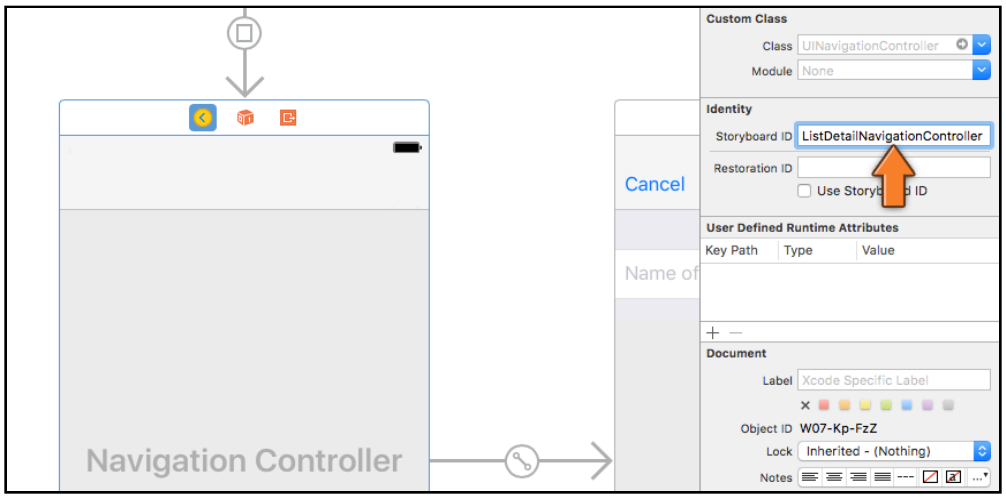
The `storyboard` property is optional because view controllers are not always loaded from a storyboard. But this one is, which is why you can use `!` to *force unwrap* the optional. It's like using `if let`, but because you can safely assume storyboard will not be `nil` in this app you don't have to unwrap it inside an if-statement.

The call to `instantiateViewController(withIdentifier)` takes an identifier string, `"ListDetailNavigationController"`. That is how you ask the storyboard to create the new view controller. In your case, this will be the navigation controller that contains the `ListDetailViewController`.

You could instantiate the `ListDetailViewController` directly, but it was designed to work inside the navigation controller. Instantiating it by itself wouldn't make much sense – it would no longer have a title bar or Cancel and Done buttons.

You still have to set this identifier on the navigation controller; otherwise the storyboard cannot find it.

➤ Open the storyboard and select the **navigation controller** that points to List Detail View Controller. Go to the **Identity inspector** and into the field **Storyboard ID** type **ListDetailNavigationController**:



Setting an identifier on the navigation controller

➤ That should do the trick. Run the app and tap some detail disclosure buttons.

(If the app crashes, make sure the storyboard is saved before you press Run.)

Exercise: Set the **ListDetailNavigationController** identifier on the List Detail View Controller instead of the navigation controller and see what happens when you run the app. Can you explain this? If you can, kudos! ■

You can find the project files for the app up to this point under **07 - Lists** in the tutorial's Source Code folder.

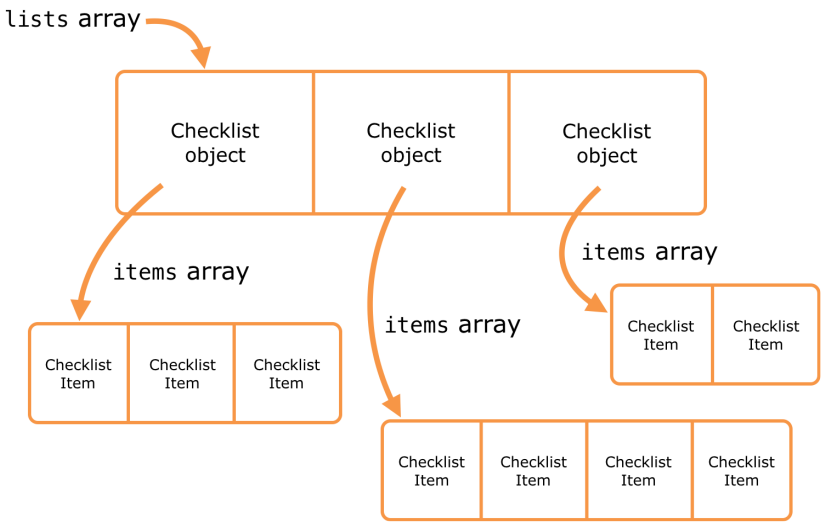
> **Are you still with me?**
>
> If at this point your eyes are glazing over and you feel like giving up: don't.
>
> Learning new things is hard and programming doubly so. Set the tutorial aside, sleep on it, and come back in a few days.
>
> Chances are that in the mean time you'll have an a-ha! moment where the thing that didn't make any sense suddenly becomes clear as day.
>
> If you have specific questions, join us on the forums. I usually check in a few times a day to help people out and so do many members of our community. Don't be embarrassed to ask for help! forums.raywenderlich.com

Putting to-do items into the checklists

Everything you've done in the previous section is all well and good, but checklists don't actually contain any to-do items yet.

So far, the list of to-do items and the actual checklists have been separate from each other.

Let's change the data model to look like this:



Each Checklist object has an array of ChecklistItem objects

There will still be the `lists` array that contains all the `Checklist` objects, but each of these `Checklists` will have its own array of `ChecklistItem` objects.

➤ Add a new property to **Checklist.swift**:

```
class Checklist: NSObject {
  var name = ""
  var items = [ChecklistItem]()     // add this line
  . . .
```

This creates a new, empty, array that can hold `ChecklistItem` objects and assigns it to the `items` instance variable.

This is slightly different from what you did before in ChecklistViewController.swift. There you declared the array and initialized it in two different steps:

```
var items: [ChecklistItem]

required init?(coder aDecoder: NSCoder) {
  items = [ChecklistItem]()
  . . .
}
```

But it's just as easy to do it in a single line, which keeps everything nice and compact.

If you're a stickler for completeness, you can also write it as follows:

```
var items: [ChecklistItem] = [ChecklistItem]()
```

I personally don't like this way of declaring variables because it violates the "DRY" principle – Don't Repeat Yourself. Fortunately, thanks to Swift's type inference, you can save yourself some keystrokes.

Another way you see it sometimes written is:

```
var items: [ChecklistItem] = []
```

The notation `[]` means: make an empty array of the inferred type.

Regardless of the way you choose to write it, the `Checklist` object now contains an array of `ChecklistItem` objects. Initially, that array is empty.

Earlier you fixed `prepare(for:sender:)` in AllListsViewController.swift so that tapping a row makes the app segue into the `ChecklistViewController`, passing along the `Checklist` object that belongs to that row.

Currently `ChecklistViewController` still gets the `ChecklistItem` objects from its own private `items` array. You will change that so it reads from the `items` array inside the `Checklist` object instead.

➤ Remove the `items` instance variable from **ChecklistViewController.swift.**

➤ Then make the following changes in this source file. Anywhere it says `items` you change it to say `checklist.items` instead.

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

```
override func tableView(_ tableView: UITableView,
                  cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  . . .
  let item = checklist.items[indexPath.row]
  . . .
}
```

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  . . .
  let item = checklist.items[indexPath.row]
  . . .
}
```

```
override func tableView(_ tableView: UITableView,
                        commit editingStyle: UITableViewCellEditingStyle,
                        forRowAt indexPath: IndexPath) {

  checklist.items.remove(at: indexPath.row)
  . . .
}
```

```
func itemDetailViewController(_ controller: ItemDetailViewController,
                              didFinishAdding item: ChecklistItem) {
  let newRowIndex = checklist.items.count
  checklist.items.append(item)
  . . .
}
```

```
func itemDetailViewController(_ controller: ItemDetailViewController,
                             didFinishEditing item: ChecklistItem) {

  if let index = checklist.items.index(of:item) {
  . . .
}
```

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    . . .
    controller.itemToEdit = checklist.items[indexPath.row]
    . . .
}
```

➤ Delete the following methods from **ChecklistViewController.swift**. (Tip: You may want to set aside the code from these methods in a temporary file somewhere; shortly you'll be using them again in a slightly modified form.)

- `func documentsDirectory()`
- `func dataFilePath()`
- `func saveChecklistItems()`
- `func loadChecklistItems()`

You recently added these methods to load and save the checklist items from a file. That is no longer the responsibility of this view controller. It is better for the app's design if you make the `Checklist` object do that.

Loading and saving data model objects really belongs in the data model itself, rather than in a controller.

But before you get to that, let's first test whether these changes were successful. Xcode is complaining about 5 or so errors because you still call the method `saveChecklistItems()` at several places in the code. You should remove those lines, as you will soon be saving the items from a different place.

➤ Remove the lines that call `saveChecklistItems()`.

➤ Also delete `init?(coder)` from **ChecklistViewController.swift**.

➤ Press **⌘+B** to make sure the app builds without errors.

Fake it 'til you make it

Let's add some fake data into the various `Checklist` objects so that you can test whether this new design actually works.

In `AllListsViewController`'s `init?(coder)` method you already put fake `Checklist` objects into the `lists` array. It's time to add something new to this method.

➤ Add the following to the bottom of **AllListsViewController.swift**'s `init?(coder)`:

```
for list in lists {
  let item = ChecklistItem()
  item.text = "Item for \(list.name)"
  list.items.append(item)
}
```

This introduces something you haven't seen before in these tutorials: the `for in` statement. Like `if`, this is a special language construct.

Programming language constructs

For the sake of review, let's go over the programming language stuff you've already seen. Most modern programming languages offer at least the following basic building blocks:

- The ability to remember values by storing things into variables. Some variables are simple, such as `Int` and `Bool`. Others can store objects (`ChecklistItem`, `UIButton`) or even collections of objects (`Array`).
- The ability to read values from variables and use them for basic arithmetic (multiply, add) and comparisons (greater than, not equals, etc).
- The ability to make decisions. You've already seen the `if` statement, but there is also a `switch` statement that is shorthand for `if` with many `else if`s.
- The ability to group functionality into units such as methods and functions. You can call those methods and receive back a result value that you can then use in further computations.
- The ability to bundle functionality (methods) and data (variables) together into objects.
- The ability to repeat a set of statements more than once. This is what the `for in` statement does. There are other ways to perform repetitions as well: `while` and `repeat`. Endlessly repeating things is what computers are good at.

Everything else is built on top of these building blocks. You've seen most of these already, but repetitions (or **loops** in programmer slang) are new.

If you grok the concepts from this list, you're well on your way to becoming a

software developer. And if not, well, just hang in there!

Let's go through that `for` loop line-by-line:

```
for list in lists {
  . . .
}
```

This means the following: for every `Checklist` object in the `lists` array, perform the statements that are in between the curly braces.

The first time through the loop, the temporary `list` variable will hold a reference to the Birthdays checklist, as that is the first `Checklist` object that you created and added to the `lists` array.

Inside the loop you do:

```
let item = ChecklistItem()
item.text = "Item for \(list.name)"
list.items.append(item)
```

This should be familiar. You first create a new `ChecklistItem` object. Then you set its `text` property to "Item for Birthdays" because the `\(…)` placeholder gets replaced with the name of the `Checklist` object, `list.name`, which is "Birthdays".

Finally, you add this new `ChecklistItem` to the Birthdays `Checklist` object, or rather, to its `items` array.

That concludes the first pass through this loop. Now the `for in` statement will look at the `lists` array again and sees that there are three more `Checklist` objects in that array. So it puts the next one, Groceries, into the `list` variable and the process repeats.

This time the text is "Item for Groceries", which is put into its own `ChecklistItem` object that goes into the `items` array of the Groceries `Checklist` object.

After that, the loop adds a new `ChecklistItem` with the text "Item for Cool Apps" to the Cool Apps checklist, and "Item for To Do" to the To Do checklist.

Then there are no more objects left to look at in the `lists` array and the loop ends.

Using loops will often save you a lot of time. You could have written this code as follows:

```
var item = ChecklistItem()
item.text = "Item for Birthdays"
```

```
lists[0].items.append(item)

item = ChecklistItem()
item.text = "Item for Groceries"
lists[1].items.append(item)

item = ChecklistItem()
item.text = "Item for Cool Apps"
lists[2].items.append(item)

item = ChecklistItem()
item.text = "Item for To Do"
lists[3].items.append(item)
```

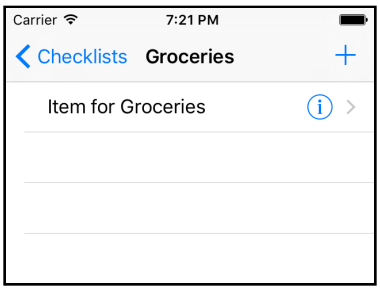
That's very repetitive, which is a good sign it's better to use a loop. Imagine if you had 100 `Checklist` objects… would you be willing to copy-paste that code a hundred times? I'd rather use a loop.

Most of the time you won't even know in advance how many objects you'll have, so it's impossible to write it all out by hand. By using a loop you don't need to worry about that. The loop will work just as well for three items as for three hundred.

As you can imagine, loops and arrays work quite well together.

➤ Run the app. You'll see that each checklist now has its own set of items.

Play with it for a minute, remove items, add items, and verify that each list indeed is completely separate from the others.



Each Checklist now has its own items

Let's put the load/save code back in. This time you'll make `AllListsViewController` do the loading and saving.

➤ Add the following to **AllListsViewController.swift** (you may want to copy this from that temporary file, but be sure to make the highlighted changes):

```
func documentsDirectory() -> URL {
  let paths = FileManager.default.urls(for: .documentDirectory,
                                       in: .userDomainMask)
  return paths[0]
}
```

```
func dataFilePath() -> URL {
  return documentsDirectory().appendingPathComponent("Checklists.plist")
}

// this method is now called saveChecklists()
func saveChecklists() {
  let data = NSMutableData()
  let archiver = NSKeyedArchiver(forWritingWith: data)

  // this line is different from before
  archiver.encode(lists, forKey: "Checklists")

  archiver.finishEncoding()
  data.write(to: dataFilePath(), atomically: true)
}

// this method is now called loadChecklists()
func loadChecklists() {
  let path = dataFilePath()
  if let data = try? Data(contentsOf: path) {
   let unarchiver = NSKeyedUnarchiver(forReadingWith: data)

   // this line is different from before
   lists = unarchiver.decodeObject(forKey: "Checklists") as! [Checklist]

   unarchiver.finishDecoding()
  }
}
```

This is mostly identical to what you had before in `ChecklistViewController`, except that you load and save the `lists` array instead of the `items` array. Note that the key under which the data gets stored is now `"Checklists"` instead of `"ChecklistItems"`. Also, the names of the methods changed slightly.

➤ Change `init?(coder)` to:

```
required init?(coder aDecoder: NSCoder) {
  lists = [Checklist]()
  super.init(coder: aDecoder)
  loadChecklists()
}
```

This gets rid of the test data you put there earlier and makes the `loadChecklists()` method do all the work.

You also have to make the `Checklist` object compliant with `NSCoding`.

➤ Add the `NSCoding` protocol in **Checklist.swift**:

```
class Checklist: NSObject, NSCoding {
```

Recall that the `NSCoding` protocol requires that you add two methods, `init?(coder)` and `encode(with)`.

➤ Add those methods to **Checklist.swift**:

```
required init?(coder aDecoder: NSCoder) {
  name = aDecoder.decodeObject(forKey: "Name") as! String
  items = aDecoder.decodeObject(forKey: "Items") as! [ChecklistItem]
  super.init()
}

func encode(with aCoder: NSCoder) {
  aCoder.encode(name, forKey: "Name")
  aCoder.encode(items, forKey: "Items")
}
```

This loads and saves the `Checklist`'s `name` and `items` properties.

➤ **Important:** Before you run the app, remove the old **Checklists.plist** file from the Simulator's Documents folder.

If you don't, the app might crash because the internal format of the file no longer corresponds to the data you're loading and saving.

Weird crashes

When I first wrote this tutorial, I didn't think to remove the Checklists.plist file before running the app. That was a mistake, but the app appeared to work fine… until I added a new checklist. At that point the app aborted with a strange error message from `UITableView` that made no sense at all.

I started to wonder whether I tested the code properly. But then I thought of the old file, removed it and ran the app again. It worked perfectly. Just to make sure it was the fault of that file, I put a copy of the old file back and ran the app again. Sure enough, when I tried to add a new checklist it crashed.

The explanation for this kind of error is that somehow the code managed to load the old file, even though its format no longer corresponded to the new data model. This put the table view into a bad state. Any subsequent operations on the table view caused app to crash.

You'll run into this type of bug every so often, where the crash isn't directly caused by what you're doing but by something that went wrong earlier on. These kinds of bugs can be tricky to solve, because you can't fix them until you find the true cause.

There is a section devoted to debugging techniques in tutorial 4 because it's inevitable that you'll introduce bugs in your code. Knowing how to find and eradicate them is an essential skill that any programmer should master – if only to save you a lot of time and aggravation!

➤ Run the app and add a checklist and a few to-do items.

➤ Exit the app (with the Stop button) and run it again. You'll see that the list is empty again. All your to-do items are gone.

You can add all the checklists and items you want, but nothing gets saved anymore.

What's going on here?

Doing saves differently

Previously, you saved the data whenever the user changed something: adding a new item, deleting an item, and toggling a checkmark all caused Checklists.plist to be re-saved. That used to happen in `ChecklistViewController`.

However, you just moved the saving logic into `AllListsViewController`. How do you make sure changes to the to-do items get saved now? The `AllListsViewController` doesn't know when a checkmark is toggled on or off.

You could give `ChecklistViewController` a reference to the `AllListsViewController` and have it call its `saveChecklists()` method whenever the user changes something, but that introduces a so-called *child-parent dependency* and you've been trying hard to avoid those (ownership cycles, remember?).

Parents and their children

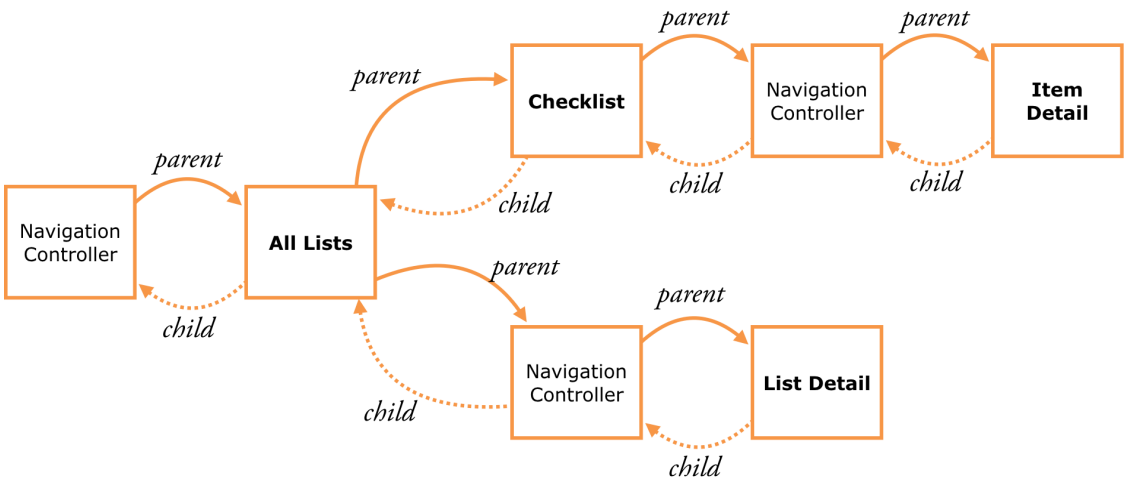
The terms *parent* and *child* are common in software development.

A parent is an object higher up in some hierarchy; a child is an object lower in the hierarchy.

In this case, the "hierarchy" represents the navigation flow between the different screens of the app.

The All Lists screen is the parent of the Checklist screen, because All Lists was "born" first. It creates a new `ChecklistViewController` "baby" every time the user performs the segue.

Likewise, All Lists is also the parent of the List Detail screen. The Item Detail screen, however, is the child of the Checklist view controller.



Generally speaking, it's OK if the parent knows everything about its children, but not the other way around (just like in real life, every parent has horrible secrets they don't want their kids to know about… or so I've been told).

As a result, you don't want parent objects to be dependent on their child objects, but the other way around is fine. So the `ChecklistViewController` asking the `AllListsViewController` to do things is a big no-no.

★ ★ ★

You may think: ah, I could use a delegate for this. True – and if you thought that indeed I'm very proud – but instead we'll rethink our saving strategy.

Is it really necessary to save changes all the time? While the app is running, the data model sits in working memory and is always up-to-date.

The only time you have to load anything from the file (the long-term storage memory) is when the app first starts up, but never afterwards. From then on you always make the changes to the objects in the working memory.

But when changes are made, the file becomes out-of-date. That is why you save those changes – to keep the file in sync with what is in memory.

The reason you save to a file is so that you can restore the data model in working memory after the app gets terminated. But until that happens, the data in the short-term working memory will do just fine.

You just need to make sure that you save the data to the file just before the app gets terminated. In other words, the only time you save is when you actually need to keep the data safe.

Not only is this more efficient, especially if you have a lot of data, it also is simpler to program. You no longer need to worry about saving every time the user makes a change to the data, only right before the app terminates.

There are three situations in which an app can terminate:

1. While the user is running the app. This doesn't happen very often anymore, but earlier versions of iOS did not support multitasking apps. Receiving an incoming phone call, for example, would kill the currently running app. As of iOS 4, the app will simply be suspended in the background when that happens.

   There are still situations where iOS may forcefully terminate a running app, for example if the app becomes unresponsive or runs out of memory.

2. When the app is suspended in the background. Most of the time iOS keeps these apps around for a long time. Their data is frozen in memory and no computations are taking place. (When you resume a suspended app, it literally continues from where it left off.)

   Sometimes the OS needs to make room for an app that requires a lot of working memory – often a game – and then it simply kills the suspended apps and wipes them from memory. The apps are not notified of this.

3. The app crashes. There are ways to detect crashes but handling them can be very tricky. Trying to deal with the crash may actually make things worse. The best way to avoid crashes is to make no programming mistakes! :-)

Fortunately for us, iOS will inform the app about significant changes such as, "you are about to be terminated", and, "you are about to be suspended".

You can listen for these events and save your data at that point. That will ensure the on-file representation of the data model is always up-to-date when the app does terminate.

The ideal place for handling these notifications is inside the **application delegate**. You haven't spent much time with this object before, but every app has one. As its name implies, it is the delegate object for notifications that concern the app as a whole.

This is where you receive the "app will terminate" and "app will be suspended" notifications.

In fact, if you look inside **AppDelegate.swift**, you'll see the methods:

```
func applicationDidEnterBackground(_ application: UIApplication)
```

and:

```
func applicationWillTerminate(_ application: UIApplication)
```

There are a few others, but these are the ones you need. (The Xcode template put

helpful comments inside these methods, so you know what to do with them.)

Now the trick is, how do you call `AllListsViewController`'s `saveChecklists()` method from these delegate methods? The app delegate does not know anything about `AllListsViewController` yet.

You have to use some trickery to find the All Lists View Controller from within the app delegate.

➤ Add this new method to **AppDelegate.swift**:

```
func saveData() {
  let navigationController = window!.rootViewController
                                                  as! UINavigationController
  let controller = navigationController.viewControllers[0]
                                                  as! AllListsViewController
  controller.saveChecklists()
}
```

The `saveData()` method looks at the `window` property to find the `UIWindow` object that contains the storyboard.

`UIWindow` is the top-level container for all your app's views. There is only one `UIWindow` object in your app (unlike desktop apps, which usually have multiple windows).

Exercise: Can you explain why you wrote `window!` with an exclamation point? ■

★ ★ ★

Unwrapping optionals

At the top of AppDelegate.swift you can see that `window` is declared as an optional:

```
var window: UIWindow?
```

To *unwrap* an optional you normally use the `if let` syntax:

```
if let w = window {
  // if window is not nil, w is the real UIWindow object
  let navigationController = w.rootViewController
}
```

As a shorthand you can use *optional chaining*:

```
let navigationController = window?.rootViewController
```

If window is `nil`, then the app won't even bother to look at the rest of the statement and `navigationController` will also be `nil`.

For apps that use a storyboard (and most of them do), you're guaranteed that `window` is never `nil`, even though it is an optional. UIKit promises that it will put a valid reference to the app's `UIWindow` object inside the `window` variable when the app starts up.

So why is it an optional? There is a brief moment between when the app is launched and the storyboard is loaded where the `window` property does not have a valid value yet. And if a variable can be `nil` – no matter how briefly – then Swift requires it to be an optional.

If you're *sure* an optional will not be `nil` when you're going to use it, you can *force unwrap* it by adding an exclamation point:

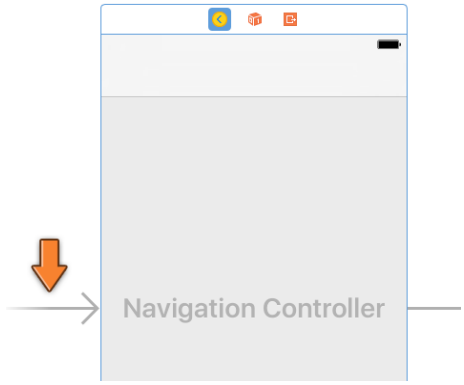
```
let navigationController = window!.rootViewController
```

That's exactly what you're doing in the `saveData()` method. Force unwrapping is the simplest way to deal with optionals, but it comes with a danger: if you're wrong and the optional *is* `nil`, the app will crash. Use with caution!

(You've actually used force unwrapping already when you read the text from the `UITextField` objects in the Item Detail and List Detail view controllers. The `UITextField text` property is an optional `String` but it will never be `nil`, which is why you can read it with `textField.text!` – the exclamation point converts the optional `String` value to a regular `String`.)

Normally you don't need to do anything with your `UIWindow`, but in cases such as this you have to ask it for its `rootViewController`. The "root" or "initial" view controller is the very first scene from the storyboard, the navigation controller all the way over on the left.

You can see this in Interface Builder because this navigation controller has the big arrow pointing at it. This is the one:



The left-most navigation controller is the window's root view controller

(The Attributes inspector for this navigation controller also has the **Is Initial View Controller** box checked, that's the same thing. In the outline pane it is called the Storyboard Entry Point.)

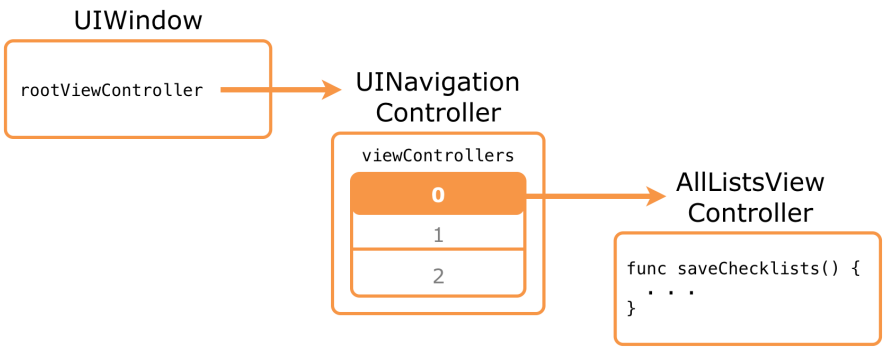
Once you have the navigation controller, you can find the `AllListsViewController`. After all, that's the view controller that is embedded in the navigation controller.

Unfortunately, the `UINavigationController` does not have a "rootViewController" property of its own, so you have to look into its `viewControllers` array to find it:

```
let controller = navigationController.viewControllers[0]
                                                as! AllListsViewController
```

As usual, a type cast is necessary because the `viewControllers` array does not know anything about the types of your own view controllers. Once you have a reference to `AllListsViewController` you can call its `saveChecklists()`method.

It's a bit of work to dig through the window and navigation controller to find the view controller you need, but that's life as an iOS developer.



From the root view controller to the AllListsViewController

> **Note:** By the way, the `UINavigationController` does have a `topViewController` property but you cannot use it here: the "top" view controller is the screen that is currently displaying, which may be the `ChecklistViewController` if the user is looking at to-do items. You don't want to send the `saveChecklists()` message to that screen – it has no method to handle that message and the app will crash!

➤ Change the `applicationDidEnterBackground()` and `applicationWillTerminate()` methods to call `saveData()`:

```
func applicationDidEnterBackground(_ application: UIApplication) {
  saveData()
}

func applicationWillTerminate(_ application: UIApplication) {
  saveData()
}
```

➤ Run the app, add some checklists, add items to those lists, and set some checkmarks.

➤ Press **Shift+⌘+H** or pick **Hardware → Home** from the Simulator's menu bar to make the app go to the background. This simulates what happens when a user taps the home button on their iPhone.

Look inside the app's Documents folder using Finder. There is a new Checklists.plist file here.

➤ Press Stop in Xcode to terminate the app. Run the app again and your data should still be there. Awesome!

> **Xcode's Stop button**
>
> Important note: When you press Xcode's Stop button, the application delegate will *not* receive the `applicationWillTerminate()` notification. Xcode kills the app without mercy.
>
> Therefore, to test the saving behavior, always simulate a tap on the home button to make the app go into the background before you press Stop. If you don't press **Shift+⌘+H** first, you'll lose your data. *Caveat developer.*

Improving the data model

The above code works but you can still do a little better. You have made data model objects for `Checklist` and `ChecklistItem` but the code for loading and saving the Checklists.plist file currently lives in `AllListsViewController`. To follow good programming practice, we should put that in the data model as well.

I prefer to create a top-level `DataModel` object for many of my apps. For this app,

`DataModel` will contain the array of `Checklist` objects. You can move the code for loading and saving into this new `DataModel` object.

➤ Add a new file to the project using the **Swift File** template. Save it as **DataModel.swift** (you don't need to make this a subclass of anything).

➤ Change **DataModel.swift** to the following:

```
import Foundation

class DataModel {
  var lists = [Checklist]()
}
```

This defines the new `DataModel` object and gives it a `lists` property.

Unlike `Checklist` and `ChecklistItem`, `DataModel` does not need to be built on top of `NSObject`. It also does not need to conform to the `NSCoding` protocol.

`DataModel` will be taking over the responsibilities for loading and saving the to-do lists from `AllListsViewController`.

➤ Cut the following methods out of **AllListsViewController.swift** and paste them into **DataModel.swift**:

- `func documentsDirectory()`
- `func dataFilePath()`
- `func saveChecklists()`
- `func loadChecklists()`

➤ Add an `init()` method to **DataModel.swift**:

```
init() {
  loadChecklists()
}
```

This makes sure that, as soon as the `DataModel` object is created, it will attempt to load Checklists.plist.

The declaration of `lists` already includes an initial value, so you don't need to do anything with it inside `init()`.

Also, you don't have to call `super.init()` because `DataModel` does not have a superclass (it is not built on `NSObject`).

Switch to **AllListsViewController.swift** and make the following changes:

➤ Remove the `lists` instance variable.

➤ Remove the `init?(coder)` method.

➤ Add a new instance variable:

```
var dataModel: DataModel!
```

The `!` is necessary because `dataModel` will temporarily be `nil` when the app starts up. It doesn't have to be a true optional – with `?` – because once `dataModel` is given a value, it will never become `nil` again.

Xcode still finds a number of errors in **AllListsViewController.swift**. You can no longer reference the `lists` variable directly, because it no longer exists. Instead, you'll have to ask the `DataModel` for its `lists` property.

➤ Everywhere the code for `AllListsViewController` says `lists`, replace this with `dataModel.lists`. You need to do this in the following methods:

- `tableView(numberOfRowsInSection)`
- `tableView(cellForRowAt)`
- `tableView(didSelectRowAt)`
- `tableView(commit, forRowAt)`
- `tableView(accessoryButtonTappedForRowWith)`
- `listDetailViewController(didFinishAdding)`
- `listDetailViewController(didFinishEditing)`

Phew, that's a big list! Fortunately, the change is very simple.

To recap, you created a new `DataModel` object that owns the array of `Checklist` objects and knows how to load and save the checklists and their items.

Instead of its own array, the `AllListsViewController` now uses this `DataModel` object, which it accesses through the `dataModel` property.

But where does this `DataModel` object get created? There is no place in the code that currently does `dataModel = DataModel()`.

The best place for this is in the app delegate. You can consider the app delegate to be the top-level object in your app. Therefore it makes sense to make it the "owner" of the data model.

The app delegate then gives this `DataModel` object to any view controllers that need to use it.

➤ In **AppDelegate.swift**, add a new property:

```
let dataModel = DataModel()
```

This creates the `DataModel` object and puts it in a constant named `dataModel`.

Even though `AllListsViewController` also has an instance variable named `dataModel`, these two things are totally separate from each other. Here you're only putting the `DataModel` object into `AppDelegate`'s `dataModel` property.

➤ Simplify the `saveData()` method to just this:

```
func saveData() {
  dataModel.saveChecklists()
}
```

If you run the app now, it will crash at once because `AllListsViewController`'s own reference to `DataModel` is still `nil`. I told you those `nil`s were no-gooders!

The best place to share the `DataModel` instance with `AllListsViewController` is in the `application(didFinishLaunchingWithOptions)` method, which gets called as soon as the app starts up.

➤ Change that method to:

```
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions:
                                 [UIApplicationLaunchOptionsKey: Any]?)
                 -> Bool {

  let navigationController = window!.rootViewController
                                                 as! UINavigationController
  let controller = navigationController.viewControllers[0]
                                                 as! AllListsViewController
  controller.dataModel = dataModel

  return true
}
```

This finds the `AllListsViewController` by looking in the storyboard (as before) and then sets its `dataModel` property. Now the All Lists screen can access the array of `Checklist` objects again.

➤ Do a clean build (**Product → Clean**) and run the app. Verify that everything still works. Great!

You can find the project files for the app up to this point under **08 - Improved Data Model** in the tutorial's Source Code folder.

★ ★ ★

I'm still confused about var and let!

If `var` makes a variable and `let` makes a constant, then why were you able to do this in AppDelegate.swift:

```
let dataModel = DataModel()
```

You'd think that when something is constant it cannot change, right?

Then how come the app lets you add new `Checklist` objects to `DataModel`? Obviously the `DataModel` object *can* be changed…

Here's the trick: Swift makes a distinction between so-called **value types** and **reference types**, and `let` works a bit differently for both.

An example of a value type is `Int`. Once you create a constant of type `Int` you can never change it afterwards:

```
let i = 100
i = 200       // not allowed
i += 1        // not allowed

var j = 100
j = 200       // allowed
j += 1        // allowed
```

The same goes for other value types such as `Float`, `String`, and even `Array`. They are called value types because the variable or constant directly stores their value.

When you assign the contents of one variable to another, the value is copied into the new variable:

```
var s = "hello"
var u = s          // u has its own copy of "hello"
s += " there"      // s and u are now different
```

But objects that you define with the keyword `class` (such as `DataModel`) are reference types. The variable or constant does not contain the actual object, only a reference to the object.

```
var d = DataModel()
var e = d                  // e refers to the same object as d
d.lists.remove(at: 0)      // this also changes e
```

You can also write this using `let` and it would do the exact same thing:

```
let d = DataModel()
let e = d                  // e refers to the same object as d
d.lists.remove(at: 0)      // this also changes e
```

So what is the difference between `var` and `let` for reference types?

When you use `let` it is not the object that is constant but the *reference* to the object. That means you cannot do this:

```
let d = DataModel()
d = someOtherDataModel   // error: cannot change the reference
```

The constant `d` can never point to another object, but the object itself can still change.

It's OK if you have trouble wrapping your head around this. The distinction between

value types and reference types is an important idea in software development, but also takes a while to understand.

My suggestion is that you use `let` whenever you can and change to `var` when the compiler complains. Note that optionals always need to be `var`, because being an optional implies that it can change its value at some point.

Using UserDefaults to remember stuff

You now have an app that lets you create lists and add to-do items to those lists. All of this data is saved to long-term storage so that even if the app gets terminated, nothing is lost.

There are some user interface improvements you can make, though.

Imagine the user is on the Birthdays checklist and switches to another app. The Checklists app is now suspended. It is possible that at some point the app gets terminated and is removed from memory.

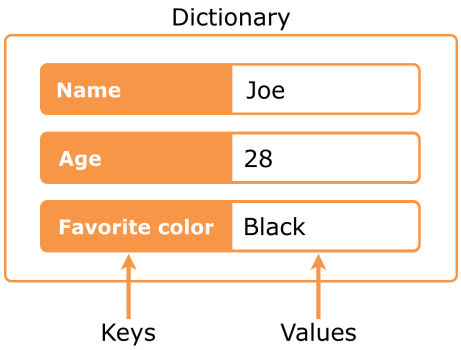
When the user reopens the app some time later it no longer is on Birthdays but on the main screen. Because it was terminated the app didn't simply resume where it left off, but got launched anew.

You might be able to get away with this, as apps don't get terminated often (unless your users play a lot of games that eat up memory) but little things like this matter in iOS apps.

Fortunately, it's fairly easy to remember whether the user has opened a checklist and to switch to it when the app starts up.

You could store this information in the Checklists.plist file, but especially for simple settings such as this there is the `UserDefaults` object.

`UserDefaults` works like a dictionary, which is a collection object for storing key-value pairs. You've already seen the array collection, which stores an ordered list of objects. The dictionary is another very common collection that looks like this:



A dictionary is a collection of key-value pairs

Dictionaries in Swift are handled by the `Dictionary` object (who would've guessed).

You can put objects into the dictionary under a reference key and then retrieve it later using that key. This is, in fact, how Info.plist works.

The Info.plist file is read into a dictionary and then iOS uses the various keys (on the left hand) to obtain the values (on the right hand). Keys are usually strings but values can be any type of object.

To be fair, `UserDefaults` isn't a true dictionary, but it certainly acts like one.

When you insert new values into `UserDefaults`, they are saved somewhere in your app's sandbox so these values persist even after the app terminates.

You don't want to store huge amounts of data inside `UserDefaults`, but it's ideal for small things like settings – and for remembering what screen the app was on when it closed.

This is what you are going to do:

1. On the segue from the main screen (`AllListsViewController`) to the checklist screen (`ChecklistViewController`), you write the row index of the selected list into `UserDefaults`. This is how you'll remember which checklist was active.

   You could have saved the name of the checklist instead of the row index, but what would happen if two checklists have the same name? Unlikely, but not impossible. Using the row index guarantees that you'll always select the proper one.

2. When the user presses the back button to return to the main screen, you have to remove this value from `UserDefaults` again. It is common to set a value such as this to -1 to mean "no value".

   Why -1? You start counting rows at 0, so you can't use 0. Positive numbers are also out of the question, unless you use a huge number such as 1000000 as it's very unlikely the user will make that many checklists. -1 is not a valid row index – and because it's a negative value it looks weird, making it easy to spot during

debugging.

(If you're wondering why you're not using an optional for this – good question! – the answer is that `UserDefaults` cannot handle optionals. Sad face.)

3. If the app starts up and the value from `UserDefaults` isn't -1, the user was previously viewing the contents of a checklist and you have to manually perform a segue to the `ChecklistViewController` for the corresponding row.

Phew, it's more work to explain this in English than writing the actual code. ;-)

Let's start with the segue from the main screen. Recall that this segue is triggered from code rather than from the storyboard.

➤ In **AllListsViewController.swift**, change `tableView(didSelectRowAt)` to the following:

```
override func tableView(_tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  // add this line:
  UserDefaults.standard.set(indexPath.row, forKey: "ChecklistIndex")

  let checklist = dataModel.lists[indexPath.row]
  performSegue(withIdentifier: "ShowChecklist", sender: checklist)
}
```

In addition to what this method did before, you now store the index of the selected row into `UserDefaults` under the key "ChecklistIndex".

To recognize whether the user presses the back button on the navigation bar, you have to become a delegate of the navigation controller. Being the delegate means that the navigation controller tells you when it pushes or pops view controllers on the navigation stack.

The logical place for this delegate is the `AllListsViewController`.

➤ Add the delegate protocol to the `class` line in **AllListsViewController.swift**:

```
class AllListsViewController: UITableViewController,
      ListDetailViewControllerDelegate, UINavigationControllerDelegate {
```

As you can see, a view controller can be a delegate for many other objects at once.

`AllListsViewController` is now the delegate for both the `ListDetailViewController` and the `UINavigationController`, but also implicitly for the `UITableView` (because it is a table view controller).

➤ Add the delegate method to the bottom of **AllListsViewController.swift**:

```
func navigationController(
                      _ navigationController: UINavigationController,
                      willShow viewController: UIViewController,
                      animated: Bool) {
```

```
    // Was the back button tapped?
    if viewController === self {
      UserDefaults.standard.set(-1, forKey: "ChecklistIndex")
    }
  }
```

This method is called whenever the navigation controller will slide to a new screen.

If the back button was pressed, the new view controller is `AllListsViewController` itself and you set the "ChecklistIndex" value in `UserDefaults` to `-1`, meaning that no checklist is currently selected.

Equal or identical

To determine whether the `AllListsViewController` is the newly activated view controller, you wrote:

```
  if viewController === self {
```

Yep, it's not a typo, that's three equals signs in a row.

Previously to compare objects you used only two equals signs:

```
  if segue.identifier == "AddItem" {
```

You may be wondering what the difference is between these two operators. It's subtle but important question about identity. (Who said programmers couldn't be philosophical?)

If you use ==, you're checking whether two variables have the same value.

With === you're checking whether two variables refer to the exact same object.

Imagine two people who are both called Joe. They're different people who just happen to have the same name.

If we'd compare them using `joe1 === joe2` then the result would be false, as they're not the same person.

But `joe1.name == joe2.name` would be true.

On the other hand, if I'm telling you an amusing (or embarrassing!) story about Joe and this story seems awfully familiar to you, then maybe we happen to know this same Joe.

In that case, `joe1 === joe2` would be true as well.

By the way, the above code would have worked just fine if you had written,

```
if viewController == self
```

with just two equals signs. For objects such as view controllers, equality is tested by comparing the references, just like === would do. But technically speaking, === is more correct here than ==.

The only thing that remains is to check at startup which checklist you need to show and then perform the segue manually. You'll do that in `viewDidAppear()`.

➤ Add the `viewDidAppear()` method to **AllListsViewController.swift**:

```
override func viewDidAppear(_ animated: Bool) {
  super.viewDidAppear(animated)

  navigationController?.delegate = self

  let index = UserDefaults.standard.integer(forKey: "ChecklistIndex")
  if index != -1 {
    let checklist = dataModel.lists[index]
    performSegue(withIdentifier: "ShowChecklist", sender: checklist)
  }
}
```

UIKit automatically calls this method after the view controller has become visible.

First, the view controller makes itself the delegate for the navigation controller.

Every view controller has a built-in `navigationController` property. To access it you use the notation `navigationController?.delegate` because it is optional.

(You could also have written `navigationController!` instead of `?`. The difference between the two is that `!` will crash the app if this view controller would ever be shown outside of a `UINavigationController`, while `?` won't crash but simply ignore the rest of that line. For our app, this does not matter.)

Then it checks `UserDefaults` to see whether it has to perform the segue.

If the value of the "ChecklistIndex" setting is -1, then the user was on the app's main screen before the app was terminated and we don't have to do anything.

However, if the value of the "ChecklistIndex" setting is *not* -1, then the user was previously viewing a checklist and the app should segue to that screen. As before, you place the relevant `Checklist` object into the `sender` parameter of `performSegue(withIdentifier, sender)`.

The != operator means: not equal. It is the opposite of the == operator. If you're mathematically-inclined, with some imagination != looks like ≠. (Some languages use <> for not equal but that won't work in Swift.)

> **Note:** It may not be immediately obvious what's going on here.
>
> `viewDidAppear()` isn't just called when the app starts up but also every time the navigation controller slides the main screen back into view.
>
> Checking whether to restore the checklist screen needs to happen only once when the app starts, so why did you put this logic in `viewDidAppear()` if it gets called more than once?
>
> Here's the reason:
>
> The very first time `AllListsViewController`'s screen becomes visible you don't want the `navigationController(willShow…)` delegate method to be called yet, as that would always overwrite the old value of "ChecklistIndex" with -1, before you've had a chance to restore the old screen.
>
> By waiting to register `AllListsViewController` as the navigation controller delegate until it is visible, you avoid this problem. `viewDidAppear()` is the ideal place for that, so it makes sense to do it from that method.
>
> However, as mentioned, `viewDidAppear()` also gets called after the user presses the back button to return to the All Lists screen. That shouldn't have any unwanted side effects, such as triggering the segue again.
>
> Naturally, the navigation controller calls `navigationController(willShow…)` when the back button is pressed, but this happens before `viewDidAppear()`. The delegate method always sets the value of "ChecklistIndex" back to -1, and as a result `viewDidAppear()` does not trigger a segue again.
>
> And so it all works out… The logic that you added to `viewDidAppear()` only does its job once during app startup. There are other ways to solve this particular issue but this approach is simple, so I like it.
>
> Is all of this going way over your head? Don't fret about it. To get a better idea of what is going on, sprinkle `print()` statements around the various methods to see in which order they get called. Change things around to see what the effect is. Jumping into the code and playing with it is the quickest way to learn!

Double-check that all the lines with `UserDefaults` use the same key name, "ChecklistIndex". If one of them is misspelled, `UserDefaults` is reading from and writing to different items.

➤ Run the app and go to a checklist screen. Exit to the home screen (Shift+⌘+H in the Simulator), followed by Stop to quit the app.

Tip: You need to exit to the home screen first because `UserDefaults` may not immediately save its settings to disk and therefore you may lose your changes if

you kill the app from within Xcode.

> **Note:** Does the app crash for you at this point? That happens if you didn't add any lists or to-do items yet. That's the exact problem we're solving in the next section. You can either comment out the code in `viewDidAppear()`, add some to-do items, and enable the code again to try it. Or simply move on to the next section.

➤ Run the app again and you'll notice that Xcode immediately switches to the screen where you were last at. Cool, huh!

Defensive programming

➤ Now do the following: Stop the app and reset the Simulator using the menu item **Simulator → Reset Contents and Settings**.

(Just holding down the app icon until it starts to wiggle and then deleting it is not enough; you need to reset the entire Simulator.)

Then run the app again from within Xcode and watch it crash:

```
fatal error: Index out of range
```

The app crashes in `viewDidAppear()` on the line:

```
let checklist = dataModel.lists[index]
```

What's going on here? Apparently the value of `index` is not -1, because the code entered the if-statement.

As it turns out `index` is 0, even though there should be nothing in `UserDefaults` yet because this is a fresh install of the app. The app didn't write anything in the "ChecklistIndex" key yet.

Here's the thing: `UserDefaults`'s `integer(forKey)` method returns 0 if it cannot find the value for the key you specify, but in this app 0 is a valid row index.

At this point the app doesn't have any checklists yet, so index 0 does not exist in the `lists` array. That is why the app crashes.

What you would like instead, is that `UserDefaults` returns -1 also if nothing is set yet for "ChecklistIndex", because to this app -1 means: show the main screen instead of a specific checklist.

Fortunately, `UserDefaults` will let you set default values for the default values. Yep, you read that correctly. Let's do that in the `DataModel` object.

➤ Add the following method inside **DataModel.swift**:

```
func registerDefaults() {
  let dictionary: [String: Any] = [ "ChecklistIndex": -1 ]

  UserDefaults.standard.register(defaults: dictionary)
}
```

This creates a new `Dictionary` instance and adds the value -1 for the key "ChecklistIndex".

The square bracket notation is not only used to make arrays but also dictionaries. The difference is that for a dictionary it always looks like,

```
[ key1: value1, key2: value2, . . . ]
```

while an array is just:

```
[ value1, value2, value3, . . . ]
```

`UserDefaults` will use the values from this dictionary if you ask it for a key but it cannot find anything under that key.

➤ Change **DataModel.swift**'s `init()` to call this new method:

```
init() {
  loadChecklists()
  registerDefaults()
}
```

➤ Run the app again and now it should no longer crash.

Why did you do this in `DataModel`? Well, I don't really like to sprinkle all of these calls to `UserDefaults` throughout the code.

In fact, let's move all of the `UserDefaults` stuff into `DataModel`.

➤ Add the following to **DataModel.swift**:

```
var indexOfSelectedChecklist: Int {
  get {
    return UserDefaults.standard.integer(forKey: "ChecklistIndex")
  }
  set {
    UserDefaults.standard.set(newValue, forKey: "ChecklistIndex")
  }
}
```

This does something you haven't seen before. It appears to declare a new instance variable `indexOfSelectedChecklist` of type `Int`, but what are these `get { }` and `set { }` blocks?

This is an example of a *computed property*.

There isn't any storage allocated for this property (so it's not really a variable).

Instead, when the app tries to read the value of `indexOfSelectedChecklist`, the code in the `get` block is performed. And when the app tries to put a new value into `indexOfSelectedChecklist`, the `set` block is performed.

From now on you can simply use `indexOfSelectedChecklist` and it will automatically update `UserDefaults`. How cool is that?

You're doing this so the rest of the code won't have to worry about `UserDefaults` anymore. The other objects just have to use the `indexOfSelectedChecklist` property on `DataModel`.

Hiding implementation details is an important object-oriented programming principle, and this is one way to do it.

If you decide later that you want to store these settings somewhere else, for example in a database or in iCloud, then you only have to change this in one place, in `DataModel`. The rest of the code will be oblivious to these changes and that's a good thing.

➤ Update the code in **AllListsViewController.swift** to use this new computed property:

```
override func viewDidAppear(_ animated: Bool) {
  super.viewDidAppear(animated)

  navigationController?.delegate = self

  let index = dataModel.indexOfSelectedChecklist        // change this
  if index != -1 {
    let checklist = dataModel.lists[index]
    performSegue(withIdentifier: "ShowChecklist", sender: checklist)
  }
}
```

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  // change this line
  dataModel.indexOfSelectedChecklist = indexPath.row

  let checklist = dataModel.lists[indexPath.row]
  performSegue(withIdentifier: "ShowChecklist", sender: checklist)
}
```

```
func navigationController(
                    _ navigationController: UINavigationController,
                    willShow viewController: UIViewController,
                    animated: Bool) {
  if viewController === self {
    dataModel.indexOfSelectedChecklist = -1             // change this
  }
}
```

The intent of the code is now much clearer. `AllListsViewController` no longer has to worry about the "how" – storing values in `UserDefaults` – and can simply focus

on the "what" – changing the index of the selected checklist.

➤ Run the app again and make sure everything still works.

It's pretty nice that the app now remembers what screen you were on, but this new feature has also introduced a subtle bug in the app. Here's how to reproduce it:

➤ Start the app and add a new checklist. Also add a new to-do item to this list. Now kill the app from within Xcode.

Because you did not exit to the home screen first, the new checklist and its item were not saved to Checklists.plist.

However, there is a (small) chance that `UserDefaults` did save its changes to disk and now thinks this new list is selected. That's a problem because that list doesn't exist anymore (it never made it into Checklists.plist).

`UserDefaults` will save its changes at indeterminate times so it could have saved before you terminated the app.

➤ Run the app again and – if you're lucky? – it will crash with:

```
fatal error: Index out of range
```

If you can't get this error to happen, add the following line to the `set` block of `indexOfSelectedChecklist` and try again. This forces `UserDefaults` to save its changes every time `indexOfSelectedChecklist` changes:

```
  set {
    UserDefaults.standard.set(newValue, forKey: "ChecklistIndex")
    UserDefaults.standard.synchronize()
  }
```

The reason for the crash is that `UserDefaults` and the contents of Checklists.plist are out-of-sync. `UserDefaults` thinks the app needs to select a checklist that doesn't actually exist. Every time you run the app it will now crash. Yikes!

This situation shouldn't really happen during regular usage because you used the Xcode Stop button to kill the app before it had a chance to save the plist file.

Under normal circumstances the user would press the home button. As the app goes into the background it properly saves both Checklists.plist and `UserDefaults` and everything is in sync again.

However, the OS can always decide to terminate the app and then this same situation could occur.

Even though there's only a small chance that this can go wrong in practice, you should really protect the app against it. These are the kinds of bug reports you don't want to receive because often you have no idea what the user did to make it happen.

This is where the practice of *defensive programming* becomes important. Your code should always check for such boundary cases and be able to gracefully handle them even if they are unlikely to occur.

In our case, you can easily fix `AllListsViewController`'s `viewDidAppear()` method to deal with this situation.

➤ Change the if-statement in `viewDidAppear()` to:

```
if index >= 0 && index < dataModel.lists.count {
```

Instead of just checking for `index != -1`, you now do a more precise check to determine whether `index` is valid. It should be between 0 and the number of checklists in the data model. If not, then you simply don't segue.

This prevents `dataModel.lists[index]` from asking for an object at an `index` that doesn't exist.

You haven't seen the `&&` operator before. This symbol means "logical and". It is used as follows:

```
if something && somethingElse {
  // do stuff
}
```

This reads: if something is true **and** something else is also true, then do stuff.

In `viewDidAppear()` you only perform the segue when `index` is 0 or greater and also less than the number of checklists, which means it's only valid if it lies in between those two values.

With this defensive check in place, you're guaranteed that the app will not try to segue to a checklist that doesn't exist, even if the data is out-of-sync.

Note: Even though the app remembers what checklist the user was on, it won't bother to remember whether the user had the Add/Edit Checklist or Add/Edit Item screen open.

These kinds of modal screens are supposed to be temporary. You open them to make a few changes and then close them again. If the app goes to the background and is terminated, then it's no big deal if the modal screen disappears.

At least that is true for this app. If you have an app that allows the user to make many complicated edits in a modal screen, you may want to persist those changes when the app closes so the user won't lose all his work in case the app is killed.

In this tutorial you used `UserDefaults` to remember which screen was open, but iOS actually has a dedicated API for this kind of thing, State Preservation and Restoration. You can read more about this on raywenderlich.com.

The first-run experience

Let's use `UserDefaults` for something else. It would be nice if the first time you ran the app it created a default checklist for you, simply named "List", and switched over to that list. This enables you to start adding to-do items right away.

That's how the standard Notes app works too: you can start typing a note right after launching the app for the very first time, but you can also go one level back in the navigation hierarchy to see a list of all notes.

To pull this off, you need to keep track in `UserDefaults` whether this is the first time the user runs the app. If it is, you create a new `Checklist` object.

You can perform all of this logic inside `DataModel`.

It's a good idea to add a new default setting to the `registerDefaults()` method. The key for this value is "FirstTime".

➤ Change the `registerDefaults()` method in **DataModel.swift** (don't miss the comma after the first line of the dictionary):

```
func registerDefaults() {
  let dictionary: [String: Any] = [ "ChecklistIndex": -1,
                                    "FirstTime": true ]

  UserDefaults.standard.register(defaults: dictionary)
}
```

The "FirstTime" setting can be a boolean value because it's either true (this is the first time) or false (this is any other than the first time).

The value of "FirstTime" needs to be true if this is the first launch of the app after a fresh install.

➤ Still in **DataModel.swift**, add a new `handleFirstTime()` method:

```
func handleFirstTime() {
  let userDefaults = UserDefaults.standard
  let firstTime = userDefaults.bool(forKey: "FirstTime")

  if firstTime {
    let checklist = Checklist(name: "List")
    lists.append(checklist)

    indexOfSelectedChecklist = 0
    userDefaults.set(false, forKey: "FirstTime")
    userDefaults.synchronize()
  }
}
```

Here you check `UserDefaults` for the value of the "FirstTime" key. If the value for "FirstTime" is true, then this is the first time the app is being run. In that case, you create a new `Checklist` object and add it to the array.

You also set `indexOfSelectedChecklist` to 0, which is the index of this newly added `Checklist` object, to make sure the app will automatically segue to the new list in `AllListsViewController`'s `viewDidAppear()`.

Finally, you set the value of "FirstTime" to false, so this code won't be executed again the next time the app starts up.

➤ Call this new method from `DataModel`'s `init()`:

```
init() {
  loadChecklists()
  registerDefaults()
  handleFirstTime()
}
```

➤ Reset the Simulator to remove the app and its associated data, and run the app again from Xcode.

Because it's the first time you run the app (at least from the app's perspective) after a fresh install, it will automatically create a new checklist named List and switch to it.

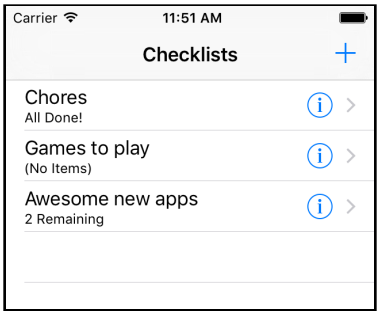
You can find the project for the app up to this point under **09 - UserDefaults** in the tutorial's Source Code folder.

Improving the user experience

There are a few small features I'd like to add, just to polish the app a little more. After all, you're building a real app here – if you want to make top-notch apps, you have to pay attention to those details.

Showing the number of to-do items remaining

In the main screen, for each checklist the app will show the number of to-do items that do not have checkmarks yet:



Each checklist shows how many items are still left to-do

First, you need a way to count these items.

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

```
func countUncheckedItems() -> Int {
  var count = 0
  for item in items where !item.checked {
    count += 1
  }
  return count
}
```

With this method you can ask any `Checklist` object how many of its `ChecklistItem` objects do not yet have their checkmark set. The method returns this count as an `Int` value.

You use a `for in` to loop through the `ChecklistItem` objects from the `items` array. If an `item` object has its `checked` property set to false, you increment the local variable `count` by 1.

Remember that the `!` operator negates the result. So if `item.checked` is true, then `!item.checked` will make it false. You should read it as "where not item.checked".

> **Note:** If the `!` symbol is written in front of something then it is the logical **not** operator, as you see here. When the `!` is written behind something, it's related to optionals. This is another example of a symbol that has more than one meaning in Swift. The correct interpretation depends on the context where it is being used.

When the loop is over and you've looked at all the objects, you return the total value of the count to the caller.

Exercise: What would happen if you used `let` instead of `var` to make the `count` variable? ■

Answer: When `count` is a constant, Swift won't let you change its value, so the line that does `+= 1` gives an error message.

By the way, you could also have written the loop as follows:

```
  for item in items {
    if !item.checked {
      count += 1
    }
  }
```

This uses the more familiar `if` statement instead. Personally, I like the brevity of the `for in where` loop, but using an `if` is just as correct.

➤ Go to **AllListsViewController.swift** and in `makeCell(for)` change `style: .default` to `style: .subtitle`.

The rest of the code stays the same, except you now use `.subtitle` for the cell style instead of `.default`. The "subtitle" cell style adds a second, smaller label below the main label. You can use the cell's `detailTextLabel` property to access this subtitle label.

➤ That happens in `tableView(cellForRowAt)`. Add the following line just before `return cell`:

```
cell.detailTextLabel!.text =
                            "\(checklist.countUncheckedItems()) Remaining"
```

You call the `countUncheckedItems()` method on the `Checklist` object and put the count into a new string that you place into the `detailTextLabel`.

As usual, you use `\(…)` to do the string interpolation. Notice that you can even call methods inside interpolated strings. Sweet!

To put text into the cell's labels, you wrote:

```
cell.textLabel!.text = someString
cell.detailTextLabel!.text = anotherString
```

The `!` is necessary because `textLabel` and `detailTextLabel` are optionals.

The `textLabel` property is only present on table view cells that use one of the built-in cell styles; it is `nil` on custom cell designs. Likewise, not all of the cell styles have a detail label and `detailTextLabel` will be `nil` in those cases.

Here you're using the "Subtitle" cell style, which is guaranteed to have both labels. Because these optionals will never be `nil` for a "Subtitle" cell, you can use `!` to *force unwrap* them. This turns the optional into an actual object that you can use.

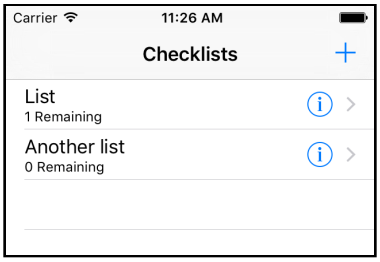
Be careful with this, though… using `!` on an optional that *is* `nil` will crash your app immediately.

You could also have written it as:

```
if let label = cell.textLabel {
  label.text = someString
}
if let label = cell.detailTextLabel {
  label.text = anotherString
}
```

That is safer – no chance of crashing here – but also a bit more cumbersome. Writing `!` is just more convenient in this case.

➤ Run the app. For each checklist it will now show how many items still remain to be done.



The cells now have a subtitle label

One problem: The to-do count never changes. If you toggle a checkmark on or off, or add new items, the "to do" count remains the same. That's because you create these table view cells once and never update their labels. (Try it out!)

Exercise: Think of all the situations that will cause this "still to do" count to change. ■

Answer:

- The user toggles a checkmark on an item. When the checkmark is set, the count goes down. When the checkmark gets removed, the count goes up again.
- The user adds a new item. New items don't have their checkmark set, so adding a new item should increment the count.
- The user deletes an item. The count should go down but only if that item had no checkmark.

These changes all happen in the `ChecklistViewController` but the "still to do" label is shown in the `AllListsViewController`.

So how do you let the All Lists View Controller know about this?

If you thought, "That's easy, let's use a delegate!", then you're starting to get the hang of this. You could make a new `ChecklistViewControllerDelegate` protocol that sends messages when the following things happen:

- the user toggles a checkmark on an item
- the user adds a new item
- the user deletes an item

But what would the delegate – which would be `AllListsViewController` – do in response? It would simply set a new text on the cell's `detailTextLabel` in all cases.

The delegate approach sounds good, only you're going to cheat and not use a delegate at all. There is a simpler solution and a smart programmer always picks the simplest way to solve a problem.

➤ Go to **AllListsViewController.swift** and add the `viewWillAppear()` method to do the following:

```
override func viewWillAppear(_ animated: Bool) {
  super.viewWillAppear(animated)
  tableView.reloadData()
}
```

Don't confuse this method with `viewDidAppear()`. The difference is in the verb: *will* versus *did*. `viewWillAppear()` is called before `viewDidAppear()`, when the view is about to become visible but the animation hasn't started yet. `viewDidAppear()` is called after the view is visible on the screen and the animation has completed. There may be half a second or so difference between them as the animation takes place.

The iOS API often does this: there is a "will" method that is invoked before something happens and a "did" method that is invoked after that something happened. Sometimes you need to do things before, sometimes after, and having two methods gives you the ability to choose whichever situation works best for you.

> **API (ay-pee-eye)** stands for Application Programming Interface. When people say "the iOS API" they mean all the frameworks, objects, protocols and functions that are provided by iOS that you as a programmer can use to write apps.
>
> The iOS API consists of everything from UIKit, Foundation, Core Graphics, and so on. Likewise, when people talk about "the Facebook API" or "the Google API", they mean the services that these companies provide that allow you to write apps for those platforms.

Here, `viewWillAppear()` tells the table view to reload its entire contents. That will cause `tableView(cellForRowAt)` to be called again for every visible row.

When you tap the back button on the `ChecklistViewController`'s navigation bar, the `AllListsViewController` screen will slide back into view. Just before that happens, `viewWillAppear()` is called. Thanks to the call to `tableView.reloadData()` the app will update all of the table cells, including the `detailTextLabels`.

Reloading all of the cells may seem like overkill but in this situation you can easily get away with it. It's unlikely the All Lists screen will contain many rows (say, less than 100) and only about 14 visible cells, so reloading them is quite fast. And it saves you some work of having to make yet another delegate.

Sometimes a delegate is the best solution; sometimes you can simply reload the entire table.

➤ Run the app and test that it works!

Exercise. Change the label to read "All Done!" when there are no more to-do items left to check. ■

Answer: Change the relevant code in `tableView(cellForRowAt)` to:

```
let count = checklist.countUncheckedItems()
if count == 0 {
  cell.detailTextLabel!.text = "All Done!"
} else {
  cell.detailTextLabel!.text = "\(count) Remaining"
}
```

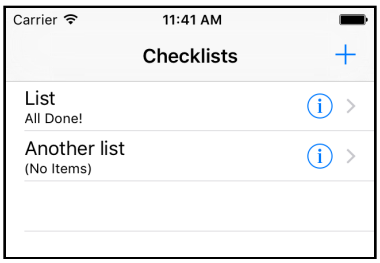
You put the count into a local constant because you refer to it twice. Calculating the count once and storing it into a temporary constant is more optimal than doing the same calculation two times.

Exercise: Now update the label to say "No Items" when the list is empty. ■

Answer:

```
let count = checklist.countUncheckedItems()
if checklist.items.count == 0 {
  cell.detailTextLabel!.text = "(No Items)"
} else if count == 0 {
  cell.detailTextLabel!.text = "All Done!"
} else {
  cell.detailTextLabel!.text = "\(count) Remaining"
}
```

Just looking at the result of `countUncheckedItems()` is not enough. If this returns 0, you don't know whether that means all items are checked off or if the list has no items at all. You also need to look at the total number of items in the checklist, with `checklist.items.count`.



The text in the detail label changes depending on how many items are checked off

Little details like these matter – they make your app more fun to use. Ask yourself, what would make you feel better about having done your chores, the rather bland

message "0 Remaining" or the joyous exclamation "All Done!"

A short diversion into Functional Programming

Swift is primarily an object-oriented language, but there is another style of writing code that has become quite popular in recent years: *functional programming*.

The term "functional" means that programs can be expressed purely in terms of mathematical functions that transform data.

Unlike the methods and functions in Swift, these mathematical functions are not allowed to have "side effects". For any given inputs, a function should always produce the same output. Methods are much less strict.

Even though Swift is not a purely functional language, it does let you use certain functional programming techniques in your apps. They can really make your code a lot shorter.

For example, let's look at `countUncheckedItems()` again:

```
func countUncheckedItems() -> Int {
  var count = 0
  for item in items where !item.checked {
    count += 1
  }
  return count
}
```

That's quite a bit of code for something that's fairly simple. You can actually write this in a single line of code:

```
func countUncheckedItems() -> Int {
  return items.reduce(0) { cnt, item in cnt + (item.checked ? 0 : 1) }
}
```

`reduce()` is a method that looks at each item and performs the code in the `{ }` block. Initially, the `cnt` variable contains the value 0, but after each item it is incremented by either 0 or 1, depending on whether the item was checked.

When `reduce()` is done, its return value is the total count of unchecked items.

You don't have to remember any of this for now, but it's pretty cool to see that Swift allows you to express this kind of algorithm very succinctly.

Sorting the lists

Another thing you often need to do with lists is sort them in some particular order.

Let's sort the list of checklists by name. Currently when you add a new checklist it is always appended to the end of the table, regardless of alphabetical order.

Before we figure out how to sort an array, let's think about when you need to perform this sort:

- When a new checklist is added
- When a checklist is renamed

There is no need to re-sort when a checklist is deleted because that doesn't have any impact on the order of the other objects.

Currently you handle these two situations in `AllListsViewController`'s implementation of "`didFinishAdding`" and "`didFinishEditing`".

➤ Change these methods to the following:

```
func listDetailViewController(_ controller: ListDetailViewController,
                              didFinishAdding checklist: Checklist) {
  dataModel.lists.append(checklist)
  dataModel.sortChecklists()
  tableView.reloadData()
  dismiss(animated: true, completion: nil)
}

func listDetailViewController(_ controller: ListDetailViewController,
                              didFinishEditing checklist: Checklist) {
  dataModel.sortChecklists()
  tableView.reloadData()
  dismiss(animated: true, completion: nil)
}
```

You were able to remove a whole bunch of stuff from both methods because you now always do `reloadData()` on the table view.

It is no longer necessary to insert the new row manually, or to update the cell's `textLabel`. Instead you simply call `tableView.reloadData()` to refresh the entire table's contents.

Again, you can get away with this because the table will only hold a handful of rows. If this table had hundreds of rows, a more advanced approach might be necessary. (You could figure out where the new or renamed `Checklist` object should be inserted and just update that row.)

The `sortChecklists()` method on `DataModel` is new and you still need to add it. But before that, we need to have a short discussion about how sorting works.

When you sort a list of items, the app will compare the items one-by-one to figure out what the proper order is. But what does it mean to compare two `Checklist` objects?

In our app we obviously want to sort them by name, but we need some way to tell the app that's what we mean.

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

```
func sortChecklists() {
  lists.sort(by: { checklist1, checklist2 in
    return checklist1.name.localizedStandardCompare(checklist2.name) ==
                                                        .orderedAscending })
}
```

Here you tell the `lists` array that the `Checklists` it contains should be sorted using some formula.

That formula is provided in the shape of a *closure*. You can tell by the `{ }` brackets around the sorting code; they are what makes it into a closure:

```
lists.sort(by: { /* the sorting code goes here */ })
```

You've briefly seen closures with the alert box in the Bull's Eye tutorial. They wrap a piece of source code into an anonymous, inline method.

The purpose of the closure is to determine whether one `Checklist` object comes before another, based on our rules for sorting.

The sort algorithm will repeatedly ask one `Checklist` object from the list how it compares to the other `Checklist` objects using the formula from the closure, and then shuffles them around until the array is sorted.

This allows `sort()` to sort the contents of the array in any order you desire. If you wanted to sort on other criteria all you'd have to do is change the logic inside the closure.

The actual sorting formula is this:

```
checklist1.name.localizedStandardCompare(checklist2.name) ==
                                                    .orderedAscending
```

To compare these two `Checklist` objects, you're only looking at their names.

The `localizedStandardCompare()` method compares the two name strings while ignoring lowercase vs. uppercase (so "a" and "A" are considered equal) and taking into consideration the rules of the current *locale*.

A locale is an object that knows about country and language-specific rules. Sorting

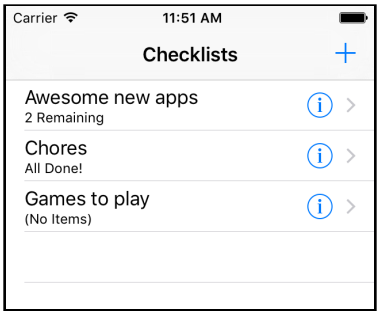
in German may be different than sorting in English, for example.

That's all you have to do to sort the array: call `sort()` and give it a closure with the logic that compares two `Checklist` objects.

➤ Just to make sure the existing lists are also sorted in the right order, you should also call `sortChecklists()` when the plist file is loaded:

```
func loadChecklists() {
  let path = dataFilePath()
  if let data = try? Data(contentsOf: path) {
    . . .
    sortChecklists()
  }
}
```

➤ Run the app and add some new checklists. Change their names and notice that the list is always sorted alphabetically.

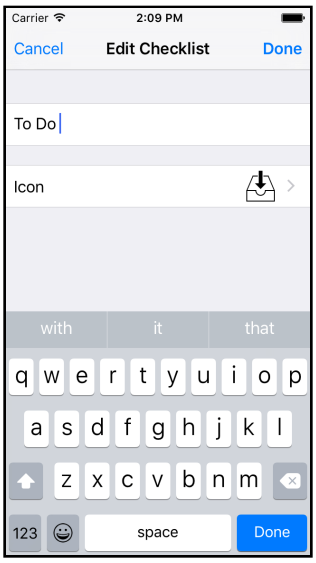


New checklists are always sorted alphabetically

Adding icons to the checklists

Because true iOS developers can't get enough of view controllers and delegates, let's add a new property to the `Checklist` object that lets you choose an icon. We're really going to cement these principles in your mind.

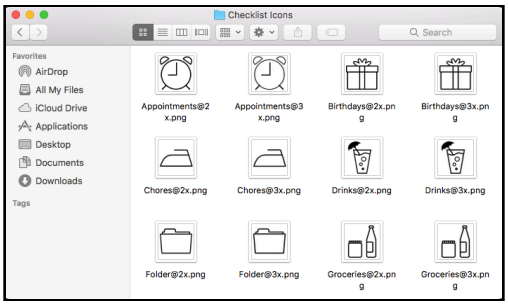
When you're done, the Add/Edit Checklist screen will look like this:



You can assign an icon to a checklist

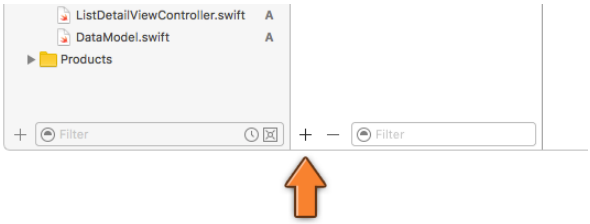
You are going to add a row to the Add/Edit Checklist screen that opens a new screen for picking an icon. This icon picker is a new view controller. You won't show it modally this time but push it on the navigation stack so it slides into the screen.

The Resources folder for this tutorial contains a folder named **Checklist Icons** with a selection of PNG images that depict different categories.



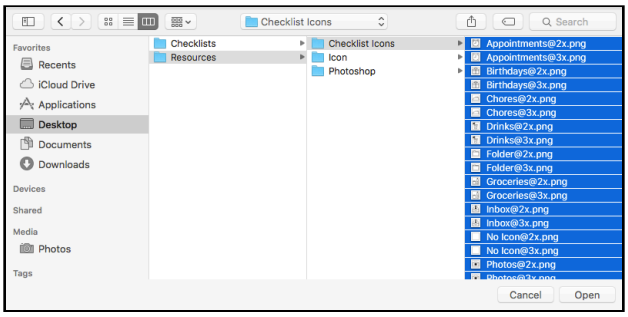
The various checklist icon images

➤ Add the images from this folder to the asset catalog. Select **Assets.xcassets** in the project navigator, click the **+** button at the bottom and choose **Import...**



Importing new images into the asset catalog

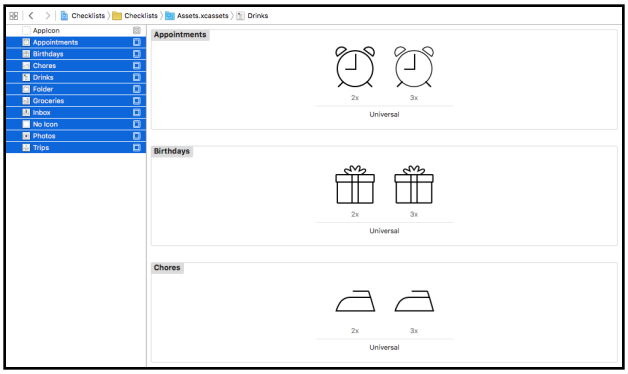
Navigate to the **Checklist Icons** folder and select all the files inside:



Selecting the image files to import

Note: Make sure to select the actual image files, not just the folder.

Click **Open** to import the images. The asset catalog should now look like this:



The asset catalog after importing the checklist icons

Each image comes with a 2x version for Retina devices and a 3x version for the iPhone 6s Plus and 7 Plus with its incredible Retina HD screen.

As I pointed out in the previous tutorial, you don't need low-resolution 1x graphics anymore. All iPhones, iPads, and iPod touch devices that can run iOS 10 have Retina 2x or 3x screens.

➤ Add the following property to **Checklist.swift**:

```
var iconName: String
```

The `iconName` variable holds the filename of the icon image.

➤ Extend `init?(coder)` and `encode(with)` to respectively load and save this icon name in the Checklists.plist file:

```
required init?(coder aDecoder: NSCoder) {
  name = aDecoder.decodeObject(forKey: "Name") as! String
  items = aDecoder.decodeObject(forKey: "Items") as! [ChecklistItem]
```

```
    iconName = aDecoder.decodeObject(forKey: "IconName") as! String
    super.init()
  }

  func encode(with aCoder: NSCoder) {
    aCoder.encode(name, forKey: "Name")
    aCoder.encode(items, forKey: "Items")
    aCoder.encode(iconName, forKey: "IconName")
  }
```

Just in case you feel like extending this app with new features of your own, remember that this is something you need to do for every new property that you add to `Checklist`. Otherwise it won't get saved to the plist file.

Xcode now complains about the `init(name)` method. Apparently it doesn't like that "Property self.iconName is not initialized at super.init call".

That means `iconName` doesn't have a value yet if the `Checklist` object is initialized with `init(name)` instead of `init?(coder)`. And as you know by now, all variables that are not optionals must always have a value.

➤ Update `init(name)` to the following:

```
  init(name: String) {
    self.name = name
    iconName = "Appointments"
    super.init()
  }
```

This will give all new checklists the "Appointments" icon.

At this point you just want to see that you can make an icon – any icon – show up in the table view. When that works you can worry about letting the user pick their own icons.

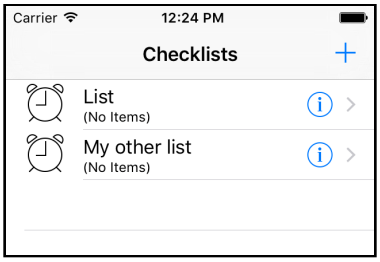
➤ Change `tableView(cellForRowAt)` in **AllListsViewController.swift** to put the icon into the table view cell:

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

    cell.imageView!.image = UIImage(named: checklist.iconName)
    return cell
  }
```

Cells using the standard `.subtitle` cell style come with a built-in `UIImageView` on the left. You can simply give it the image and it will automatically appear. Easy peasy.

➤ Before running the app, **remove the Checklists.plist file** or reset the Simulator, because by adding the "IconName" field in `init?(coder)` and `encode(with)` you've modified the file format again. You don't want any weird crashes…

➤ Run the app and now each checklist should have an alarm clock icon in front of its name.



The checklists have an icon

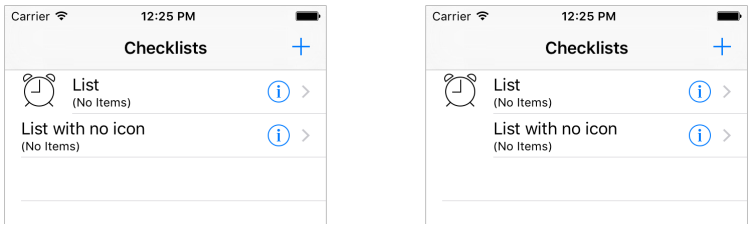
Satisfied that this works, you can now change `Checklist`'s `init(name)` to give each `Checklist` object an icon named "No Icon" by default.

➤ In **Checklist.swift**, in `init(name)` change the line that sets `iconName` to:

```
iconName = "No Icon"
```

The "No Icon" image is a fully transparent PNG image with the same dimensions as the other icons. Using a transparent image is necessary to make all the checklists line up properly, even if they have no icon.

If you were to set `iconName` to an empty string instead, the image view in the table view cell would remain empty and the text would align with the left margin of the screen. That looks bad when other cells do have icons:



Using an empty image to properly align the text labels (right)

Let's create the icon picker screen.

➤ Add a new Swift file to the project. Name it **IconPickerViewController**.

➤ Replace the contents of **IconPickerViewController.swift** with:

```
import UIKit

protocol IconPickerViewControllerDelegate: class {
  func iconPicker(_ picker: IconPickerViewController,
                  didPick iconName: String)
}
```

```
class IconPickerViewController: UITableViewController {
  weak var delegate: IconPickerViewControllerDelegate?
}
```

This defines the `IconPickerViewController` object, which is a table view controller, and a delegate protocol that it uses to communicate with other objects in the app.

➤ Add a constant (inside the `class` brackets) to hold the array of icons:

```
let icons = [
    "No Icon",
    "Appointments",
    "Birthdays",
    "Chores",
    "Drinks",
    "Folder",
    "Groceries",
    "Inbox",
    "Photos",
    "Trips" ]
```

This is an array that contains a list of icon names. These strings are both the text you will show on the screen and the name of the PNG file inside the asset catalog.

The `icons` array is the data model for this table view. Note that it is a non-mutable array (it is defined with `let` and arrays are "value" types), because the user cannot add or delete icons.

This new view controller is a `UITableViewController`, so you have to implement the data source methods for the table view.

➤ Add the following methods to the source file:

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

This simply returns the number of icons in the array.

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

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

  let iconName = icons[indexPath.row]
  cell.textLabel!.text = iconName
  cell.imageView!.image = UIImage(named: iconName)

  return cell
}
```

Here you obtain a table view cell and give it text and an image. You will design this

cell in the storyboard momentarily.

It will be a prototype cell with the cell style "Default" (or "Basic" as it is called in Interface Builder). Cells with this style already contain a text label and an image view, which is very convenient.

➤ Open the storyboard. Drag a new **Table View Controller** from the Object Library and place it next to the List Detail View Controller (the one that says "Add Checklist").

➤ In the **Identity inspector**, change the class of this new table view controller to **IconPickerViewController**.

➤ Select the prototype cell and set its **Style** to **Basic** and its (re-use) **Identifier** to **IconCell**.

That takes care of the design for the icon picker. Now you need to have some place to call it from. To do this, you will add a new row to the Add/Edit Checklist screen.

➤ Go to the **List Detail View Controller** and add a new section to the table view. You can do this by changing the **Sections** field in the **Attributes inspector** for the table view from 1 to 2. This will duplicate the existing section.

➤ Delete the Text Field from the new cell; you don't need it.

➤ Add a **Label** to this cell and name it **Icon**.

➤ Set the cell's **Accessory** to **Disclosure Indicator**. That adds a gray chevron.

➤ Add an **Image View** to the right of the cell. Make it 36 × 36 points big. (Tip: use the Size inspector for this.)

> **Note:** If you're on macOS Sierra, Xcode may show the Image View as a white rectangle, making it very hard to see. I consider this a bug in Xcode (on OS X El Capitan the Image View shows up as a blue rectangle).

➤ Use the **Assistant Editor** to add an outlet property for this image view to **ListDetailViewController.swift** and name it **iconImageView**.

Now that you've finished the designs for both screens, you can connect them with a segue.

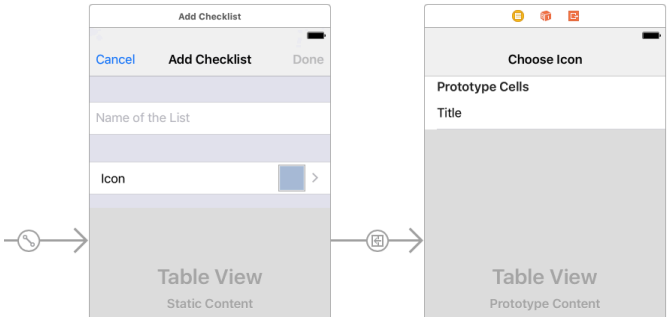
➤ **Ctrl-drag** from the "Icon" table view cell to the Icon Picker View Controller and add a segue of type **Selection Segue – Show**. (Make sure you're dragging from the Table View Cell, not its Content View or any of the other subviews.)

➤ Give the segue the identifier **PickIcon**.

➤ Thanks to the segue, the new view controller has been given a navigation bar. Double-click that navigation bar and change its title to **Choose Icon**.

> **Note:** If Xcode won't let you change the navigation title, you may first need to drag a Navigation Item from the Object Library into the view controller. Xcode is supposed to add this Navigation Item for you, but for some reason (bug?) it doesn't.

This part of the storyboard should now look like this:



The Icon Picker view controller in the storyboard

➤ In **ListDetailViewController.swift**, change the "`willSelectRowAt`" table view delegate method to:

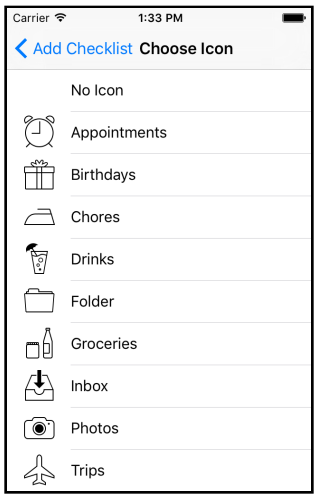
```
override func tableView(_ tableView: UITableView,
                        willSelectRowAt indexPath: IndexPath) -> IndexPath? {
  if indexPath.section == 1 {
    return indexPath
  } else {
    return nil
  }
}
```

This is necessary otherwise you cannot tap the "Icon" cell to trigger the segue.

Previously this method always returned `nil`, which meant tapping on rows was not possible. Now, however, you want to allow the user to tap the Icon cell, so this method should return the index-path for that cell.

Because the Icon cell is the only row in the second section, you only have to check `indexPath.section`. There is no need to check the row number too. Users still can't select the cell with the text field (from section 0).

➤ Run the app and verify that there is now an Icon row in the Add/Edit Checklist screen. Tapping it will open the Choose Icon screen and show a list of icons.



The icon picker screen

You can press the back button to go back but selecting an icon doesn't do anything yet. It just colors the row gray but doesn't put the icon into the checklist.

To make this work, you have to hook up the icon picker to the Add/Edit Checklist screen through its own delegate protocol.

➤ First, add an instance variable in **ListDetailViewController.swift**:

```
var iconName = "Folder"
```

You use this variable to keep track of the chosen icon name.

Even though the `Checklist` object now has an `iconName` property, you cannot keep track of the chosen icon in the `Checklist` object for the simple reason that you may not always have a `Checklist` object, i.e. when the user is adding a new checklist.

So you'll store the icon name in a temporary variable and copy that into the `Checklist`'s `iconName` property at the right time.

You should initialize the `iconName` variable with something reasonable. Let's go with the folder icon. This is only necessary for new `Checklists`, which get the Folder icon by default.

➤ Update `viewDidLoad()` to the following:

```
override func viewDidLoad() {
  super.viewDidLoad()

  if let checklist = checklistToEdit {
    title = "Edit Checklist"
    textField.text = checklist.name
    doneBarButton.isEnabled = true
    iconName = checklist.iconName                    // add this
  }
```

```
    iconImageView.image = UIImage(named: iconName)      // add this
  }
```

This has two new lines: If the `checklistToEdit` optional is not `nil`, then you copy the `Checklist` object's icon name into the `iconName` instance variable. You also load the icon's image file into a new `UIImage` object and set it on the `iconImageView` so it shows up in the Icon row.

Earlier you created a push segue named "PickIcon". You still need to implement `prepare(for:sender:)` in order to tell the `IconPickerViewController` that this screen is now its delegate.

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

```
  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "PickIcon" {
      let controller = segue.destination as! IconPickerViewController
      controller.delegate = self
    }
  }
```

This should have no big surprises for you.

Of course, Xcode has found something to complain about: it does not like that you wrote "`controller.delegate = self`", because (and I quote),

Cannot assign a value of type 'ListDetailViewController' to a value of type 'IconPickerViewControllerDelegate?'

Exercise: What did we forget? ■

Answer: You haven't made the view controller conform to the delegate protocol yet, so Swift won't let `ListDetailViewController` become the delegate of the icon picker!

➤ Add the name of that protocol to the `class` line:

```
  class ListDetailViewController: UITableViewController,
                  UITextFieldDelegate, IconPickerViewControllerDelegate {
```

➤ And add the implementation of the method from that delegate protocol somewhere inside the `ListDetailViewController` class:

```
  func iconPicker(_ picker: IconPickerViewController,
                  didPick iconName: String) {
    self.iconName = iconName
    iconImageView.image = UIImage(named: iconName)
    let _ = navigationController?.popViewController(animated: true)
  }
```

This puts the name of the chosen icon into the `iconName` variable to remember it, and also updates the image view with the new image.

You don't call `dismiss()` here but `popViewController(animated)` because the Icon Picker is on the navigation stack. When creating the segue you used the segue style "show" instead of "present modally", which pushes the new view controller on the navigation stack. To return you need to "pop" it off again. (`dismiss()` is for modal screens only, not for push screens.)

Recall that `navigationController` is an optional property of the view controller, so you need to use `?` (or `!`) to access the actual `UINavigationController` object. By writing `let _ =` you tell Xcode you don't care about the return value from `popViewController()` – without this `let` Xcode gives a warning. The _ symbol is called the *wildcard* and you can use it instead of a variable name.

> **Note:** You've seen `self` used to refer to the object itself. Here you've written:
>
> `self.iconName = iconName`
>
> The reason is that `iconName` can refer to two different things here: 1) the parameter from the delegate method, and 2) the instance variable.
>
> To remove the ambiguity, you prefix the instance variable with "`self.`", so it's clear to the compiler which of the two `iconName`s you intended to use.

➤ Change the `done()` action so that it puts the chosen icon name into the `Checklist` object when the user closes the screen:

```
@IBAction func done() {
  if let checklist = checklistToEdit {
    checklist.name = textField.text!
    checklist.iconName = iconName                          // add this
    delegate?.listDetailViewController(self,
                                       didFinishEditing: checklist)
  } else {
    let checklist = Checklist(name: textField.text!)
    checklist.iconName = iconName                          // add this
    delegate?.listDetailViewController(self,
                                       didFinishAdding: checklist)
  }
}
```

Finally, you must change `IconPickerViewController` to actually call the delegate method when a row is tapped.

➤ Add the following method to the bottom of **IconPickerViewController.swift**:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  if let delegate = delegate {
    let iconName = icons[indexPath.row]
    delegate.iconPicker(self, didPick: iconName)
  }
}
```

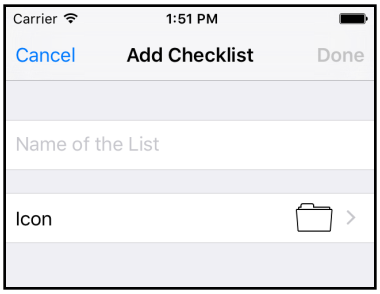
And that's it. You can now set icons on the `Checklist` objects.

To recap, you:

- added a new view controller object,
- designed its user interface in the storyboard editor, and
- hooked it up to the Add/Edit Checklist screen using a segue and a delegate.

Those are the basic steps you need to take with any new screen that you add.

➤ Run the app to try it out.



You can now give each list its own icon

Achievement unlocked: users can pick icons!

There's still a small improvement to make to the code. In `done()`, you currently do this:

```
let checklist = Checklist(name: textField.text!)
checklist.iconName = iconName
```

Setting the icon name can be considered part of the initialization of `Checklist`, so it would be nice if you could write:

```
let checklist = Checklist(name: textField.text!, iconName: iconName)
```

➤ In **ListDetailViewController.swift**'s `done()` method, replace the code that creates the new `Checklist` object with the above.

To make this work, you have to add a new `init` method to **Checklist.swift** that takes two parameters: `name` and `iconName`.

➤ Add the new init method to **Checklist.swift**:

```
init(name: String, iconName: String) {
  self.name = name
  self.iconName = iconName
  super.init()
}
```

`Checklist` now has three init methods:

- `init(name)` for when you just have a name
- `init(name, iconName)` for when you also have an icon name
- `init?(coder)` for loading the objects from the plist file

Note that at this point `init(name)` and `init(name, iconName)` do almost the same things. Both initializers assign values to `self.name` and `iconName`, and call `super.init()`.

> **Note:** The only difference is that `init(name)` does not have to use the notation "`self.iconName`" because there `iconName` can only mean one thing.

You can improve on this by making `init(name)` call `init(name, iconName)` with `"No Icon"` as the value for the `iconName` parameter.

➤ Replace `init(name)` with:

```
convenience init(name: String) {
  self.init(name: name, iconName: "No Icon")
}
```

Instead of `super.init()` it now calls `self.init(name, iconName)`.

Because it farms out its work to another init method, `init(name)` is now known as a *convenience initializer*.

It does the same thing as `init(name, iconName)` but saves you from having to type `iconName: "No Icon"` whenever you want to use it.

`init(name, iconName)` has become the so-called *designated initializer* for `Checklist`. It is the primary way to create new `Checklist` objects, while `init(name)` exists only for the convenience of lazy developers… such as you and me. :-)

➤ Build the app to verify it still works.

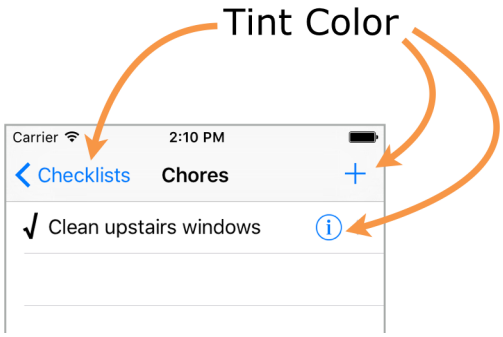
Exercise: Give `ChecklistItem` an `init(text)` method that is used instead of the parameter-less `init()`. Or how about an `init(text, checked)` method? ■

Making the app look good

You're going to keep it simple in this tutorial as far as fancying up the graphics goes. The standard look of navigation controllers and table views is perfectly adequate, although a little bland. In the next tutorials you'll see how you can customize the look of these UI elements.

Even though this app uses the stock visuals, there is a simple trick to give the app its own personality: changing the **tint color**.

The tint color is what UIKit uses to indicate that things can be interacted with, such as buttons. The default tint color is a medium blue.

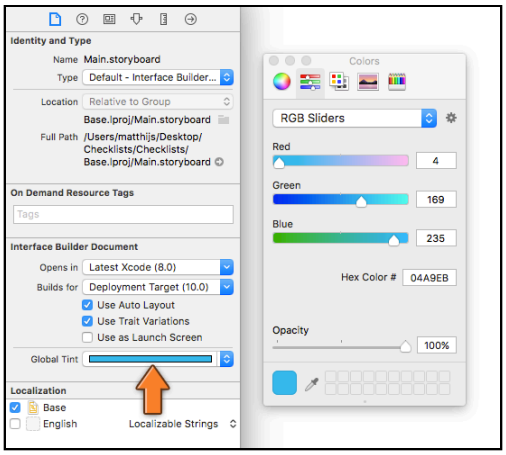


The buttons all use the same tint color

Changing the tint color is pretty easy.

➤ Open the storyboard and go to the **File inspector** (the first tab).

➤ Click **Global Tint** to open the color picker and choose Red: 4, Green: 169, Blue: 235. That makes the tint color a lighter shade of blue.



Changing the Global Tint color for the storyboard

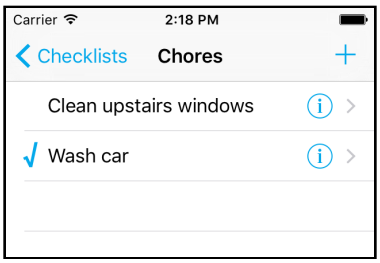
Tip: If the color picker only shows a black & white bar, then click the box that says Gray Scale Slider and change it to RGB Sliders.

It would also look nice if the checkmark wasn't black but used the tint color too.

➤ To make that happen, add the following line to `configureCheckmark(for:with:)` in **ChecklistViewController.swift**:

```
  label.textColor = view.tintColor
```

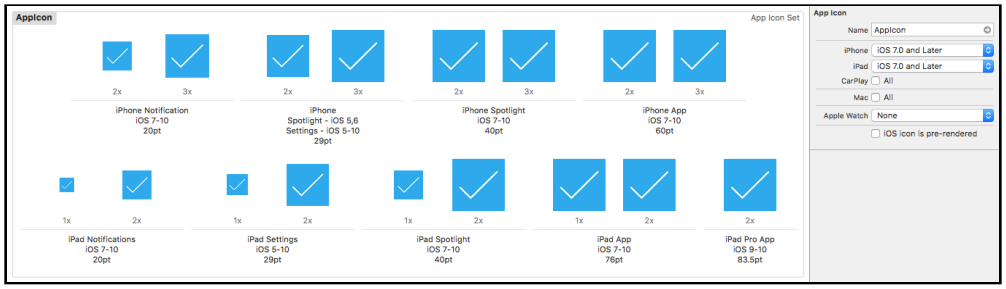
➤ Run the app. It already looks a lot more interesting:



The tint color makes the app less plain looking

No app is complete without an icon. The Resources folder for this tutorial contains a folder named **Icon** with the app icon image in various sizes. Notice that it uses the same blue as the tint color.

➤ Add these icons to the asset catalog (**Assets.xcassets**). Recall that icons go into the **AppIcon** section. Simply drag them from the Finder into the slots.



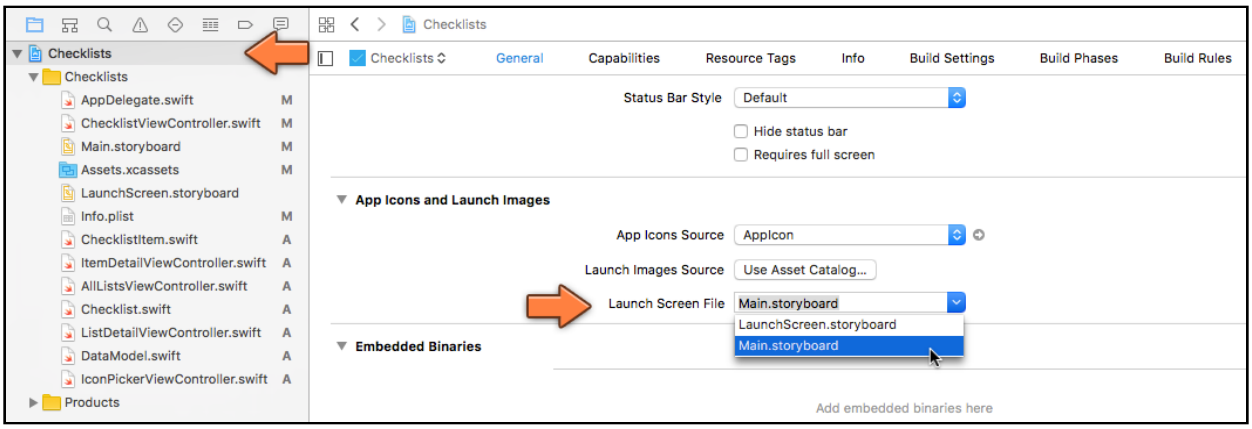
The app icons in the asset catalog

Apps should also have a launch image or launch file. Showing a static picture of the app's UI will give the illusion the app is loading faster than it really is. It's all smoke and mirrors.

The Xcode template includes the file **LaunchScreen.storyboard** that is used as the launch file. With some effort you could make this look like the initial screen of the app, but there's an easier solution.

➤ Open the **Project Settings** screen. In the **General** tab, scroll down to the **App Icons and Launch Images** section.

➤ In the **Launch Screen File** box, press the arrow and select **Main.storyboard**.



Changing the launch screen file

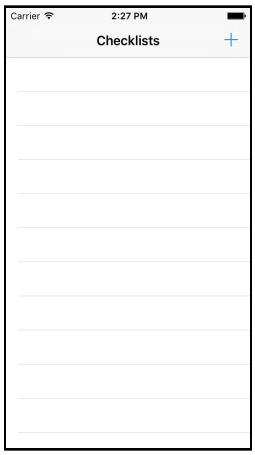
This tells the app you'll be using the design from the storyboard as the launch file.

Upon startup, the app finds the initial view controller and converts it into a static launch image. For this app that is the All Lists View Controller inside its navigation controller.

➤ Delete **LaunchScreen.storyboard** from the project.

➤ From the **Product** menu choose **Clean**. It's also a good idea to delete the app from the Simulator just so it no longer has any copies of the old launch file lying around (hold down on the icon until it starts to wiggle, just like on a real iPhone).

➤ Run the app. Just before the real UI appears you should briefly see the following launch screen:



The empty launch screen

The launch screen simply has a navigation bar and an empty table view. This gives the illusion the app's UI has already been loaded but that the data hasn't been filled in yet.

Using a proper launch screen makes the app look more professional – and faster!

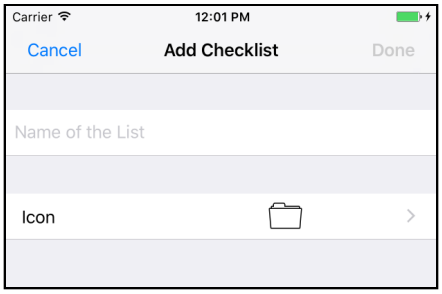
For many apps you can simply use the main storyboard as the launch file, making it a no-brainer to add. Besides, you need a launch file to support the larger screens of the iPhone 6s, 7, and Plus models.

Supporting all iPhone models

The app should run without major problems on all current iPhone models, from the smallest (iPhone SE) to the largest (iPhone 7 Plus). Table view controllers are very flexible and will automatically resize to fit the screen, no matter how large or small. Give it a try in the different Simulators!

Well, I said no *major* problems. But there are still a few tweaks you can make here and there.

So far I've been showing you screenshots of the iPhone SE simulator, and I also designed my screens in Interface Builder using the dimensions of the iPhone SE. But this is what happens when running the app on a larger simulator such as the iPhone 7 Plus:



The icon is in the wrong place

The icon is no longer nicely aligned on the right. Also try typing some text: it gets cut off because the text field is too small. Why does this happen?

When you design the user interface for your app in Interface Builder, it doesn't automatically fit all possible iPhone models, only the one you're designing for. You need to help Interface Builder out and tell it how to adjust your UI for different screen sizes. That's where Auto Layout comes in.

What you want to happen is that the image view stays glued to the right edge of the screen, always at the same distance from the disclosure indicator. When the view controller grows or shrinks to fit the iPhone screen, the image view should move along with it.

The solution is to add Auto Layout constraints to the image view that tell the app what the relationship is between the image view and the edges of the screen.

➤ Select the **Icon Image View**. Bring up the **Pin menu** using the icon at the

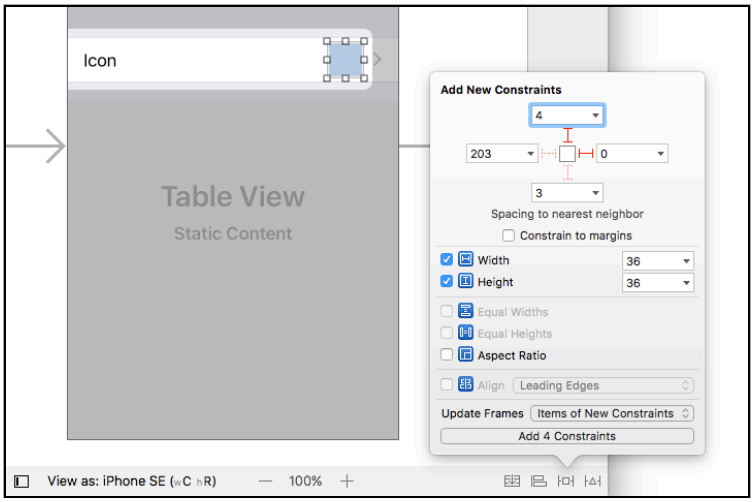
bottom of the canvas.

➤ First, uncheck **Constrain to margins**.

➤ Activate the bars at the top and the right so they turn red.

➤ Put checkmarks in front of **Width** and **Height**.

➤ For Update Frames choose **Items of New Constraints**.



Adding constraints to the Image View

➤ Finally, click **Add 4 Constraints** to finish.

The image view should now look like this:

The Image View with the constraints

Make sure the bars representing the constraints are blue. If they are orange or red you may have forgotten something in the Pin menu. (Either try again or use the **Editor → Resolve Auto Layout Issues → Update Frames** menu item.)

The most important constraint is the one on the right. This tells UIKit that the right-hand side of the image view should always stick to the right-hand edge of the table view cell's content view.

In other words, no matter how wide or narrow the screen is, the image view will always have the same location relative to the disclosure indicator.

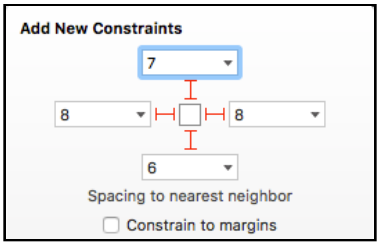
The other three constraints – top, width, and height – were necessary only because all views must always have enough constraints to determine their position and size.

If you don't specify any constraints of your own, Interface Builder will come up with reasonable default constraints. But as soon as you add just one custom constraint, you'll have to add the others too.

➤ To verify that your changes do the right thing you don't necessarily need to run the app in the simulator. Use the **View as:** panel at the bottom to switch between the different iPhone models right inside Interface Builder. If your constraints are correct, then the icon should always be in the right place.

While you're at it, you might just as well fix the text field so that it stretches to the entire width of the screen.

➤ Select the **Text Field** and in the **Pin menu** activate the four bars so they all become red:

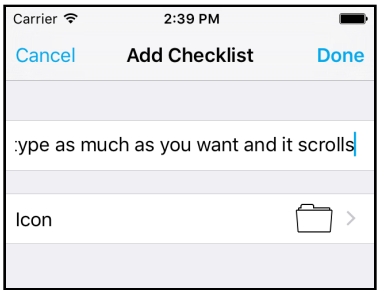


Pinning the text field

These options will make the text field stick to the sides of the table view cell. (The numbers here don't really matter, so it's fine if your numbers are slightly different. The important thing is that there are four red bars to make the four constraints.)

➤ Also do this for the text field on the Add/Edit Item screen.

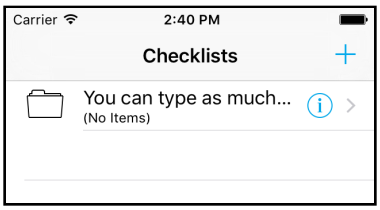
Now you can type all the way to the edge and then the text will start scrolling:



Type to your heart's content

Let's say you enter a very long text. What happens to that text when it gets shown in the other table view?

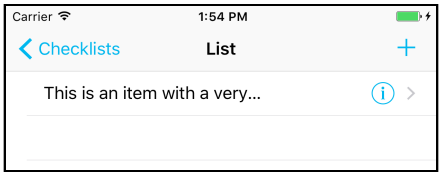
There is no problem on the All Lists screen:



Built-in cell styles automatically resize

This table view uses the built-in "Subtitle" cell style, which automatically resizes to fit the width of the screen. It also truncates the text with … when it becomes too large.

For the to-do items table, however, the picture doesn't look so rosy. The text gets cut off too soon on larger devices:



The text gets cut off

Because this is a custom prototype cell design, you'll have to add some constraints to prevent this from happening.

➤ In the storyboard, go to the Checklist screen and select the label inside the prototype cell.

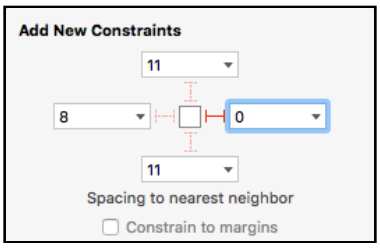
➤ First use **Editor → Size to Fit Content** to give the label its ideal size. That makes it a lot smaller, but that's OK. Without doing this first you may run into issues on the next steps. (Don't worry if doing this also moves the label.)

You want to pin the label to the right edge of the content view so it sticks to the disclosure button. Let's make that constraint first.

➤ Open the **Pin menu** and uncheck **Constrain to margins**.

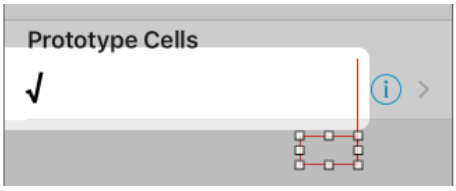
➤ Activate the red bar on the **right**. Give it the value 0 so there is no spacing between the label and the disclosure button.

➤ As always, set Update Frames to **Items of New Constraints**. Click **Add 1 Constraint** to add the new constraint.



Pinning the label to the right

Whoops, that messes up the label:

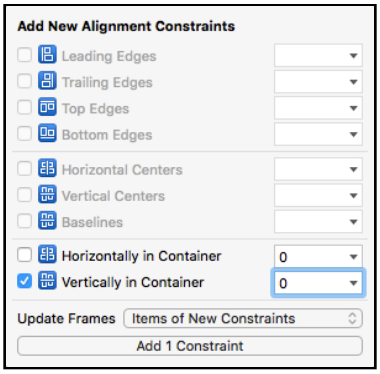


The label doesn't have enough constraints yet

Remember that you always need to specify enough constraints to determine the position and size of a view. Here you only added a constraint for the right edge of the label, which is not enough.

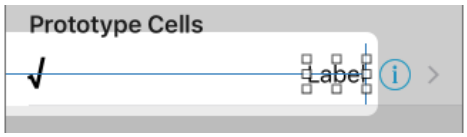
No panic! This sort of thing is common while you're adding constraints. To fix it you simply have to add the missing ones.

➤ With the label still selected, open the **Align menu** (next to Pin). Check **Vertically in Container**. Update Frames should be **Items of New Constraints**.



Centering the label vertically

Now everything turns blue again. The label has a valid position, both X and Y.



All blue bars but still in the wrong place

> **Note:** Even though you didn't specify any constraints for the label's size, the bars are all blue. How come they are not still orange?
>
> Without size constraints the label uses its contents – the text and the font – to calculate how big needs to be. This is called the *intrinsic content size*.
>
> UI components with an intrinsic size, such as `UILabel`, don't need to have Width or Height constraints, but this is only valid if you've used Size to Fit Content to reset the label to its intrinsic size first.

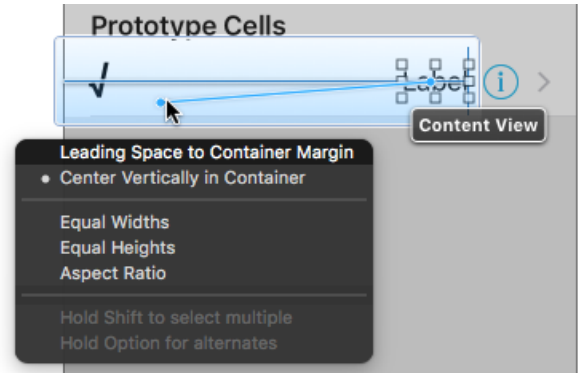
Unfortunately, the label is now right aligned. That's not what you wanted… the label should be on the left and just as wide as the cell's content view.

The easiest way to make this happen is to add a new constraint on the left to glue the label to the left edge of the screen as well.

You can't use the Pin menu to make this constraint because that would connect the label to the checkmark, which is not what you want (the size of the checkmark label changes depending on whether the check is set or not). Instead, you'll use another technique to make the new constraint.

➤ Select the label again. **Ctrl-drag** from the label to anywhere within the cell. When you let go, a popup appears. The options inside this popup depend on the direction you dragged in, so what you see may be slightly different from the illustration.

To make the constraint, select **Leading Space to Container Margin** from the popup.



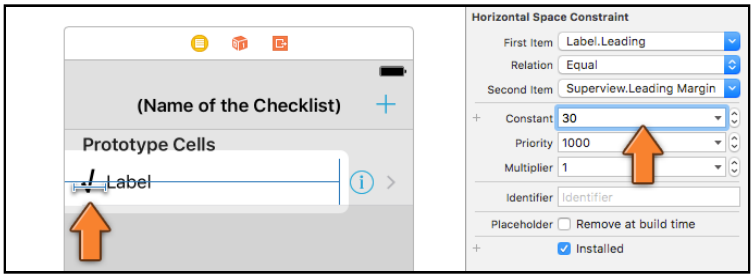
Ctrl-drag to make a constraint

This adds a very big blue bar for the new constraint, but doesn't actually move the label yet.

The new leading space constraint

➤ Select the blue bar. In the **Size inspector**, change **Constant** to 30.

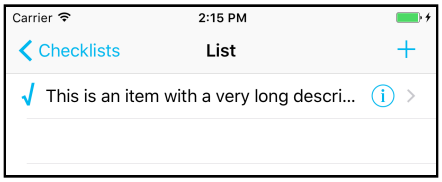
That's better:



Making the leading space constraint smaller

The label is now pinned to both edges of the table view cell's content view, so it will get stretched to however wide the table view cell is.

➤ Run the app and the label should properly truncate:



The label uses as much room as it can

You can find the project for the app up to this point under **10 - UI Improvements** in the tutorial's Source Code folder.

Extra feature: local notifications

I hope you're still with me! We have discussed in great detail view controllers, navigation controllers, storyboards, segues, table views and cells, and the data model.

These are all essential topics to master if you want to build iOS apps because almost every app uses these building blocks.

In this section you're going to expand the app to add a new feature: **local notifications**, using the brand new User Notifications framework that is introduced with iOS 10.

A local notification allows the app to schedule a reminder to the user that will be displayed even when the app is not running.

You will add a "due date" field to the `ChecklistItem` object and then remind the user about this deadline with a local notification.

If this sounds like fun, then keep reading. :-)

The steps for this section are as follows:

- Try out a local notification just to see how it works.
- Allow the user to pick a due date for to-do items.
- Create a date picker control.
- Schedule local notifications for the to-do items, and update them when the user changes the due date.

Before you wonder about how to integrate this in the app, let's just schedule a local notification and see what happens.

By the way, local notifications are different from *push* notifications (also known as *remote* notifications). Push notifications allow your app to receive messages about external events, such as your favorite team winning the World Series.

Local notifications are more similar to an alarm clock: you set a specific time and then it "beeps".

An app is only allowed to show local notifications after it has asked the user for permission. If the user denies permission, then any local notifications for your app simply won't appear. You only need to ask for permission once, so let's do that first.

➤ Open **AppDelegate.swift** and add an new import to the top of the file:

```
import UserNotifications
```

This tells Xcode that we're going to use the User Notifications framework.

➤ Add the following to the method `application(didFinishLaunchingWithOptions)`, just before the `return true` line:

```
let center = UNUserNotificationCenter.current()
center.requestAuthorization(options: [.alert, .sound]) {
  granted, error in
  if granted {
    print("We have permission")
  } else {
    print("Permission denied")
```

```
    }
}
```

Recall that `application(didFinishLaunchingWithOptions)` is called when the app starts up. It is the *entry point* for the app, the first place in the code where you can do something after the app launches.

Because you're just playing with these local notifications now, this is a good place to ask for permission.

You tell iOS that the app wishes to send notifications of type "alert" with a sound effect. Later you'll put this code into a more appropriate place.

Things that start with a dot

Throughout the app you've seen things like `.none`, `.checkmark`, and `.subtitle` – and now `.alert` and `.sound`. These are *enumeration* symbols.

An enumeration, or enum for short, is a data type that consists of a list of possible symbols and their values.

For example, the `UNAuthorizationOptions` enum contains the symbols:

```
.badge
.sound
.alert
.carPlay
```

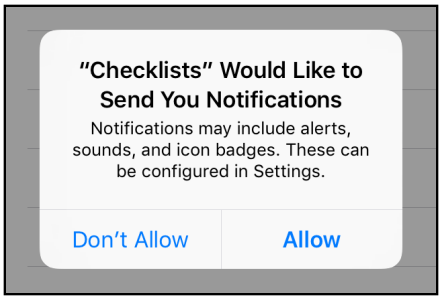
You can combine these names in an array to define what sort of notifications the app will show to the user. Here you've chosen the combination of an alert and a sound effect by writing `[.alert, .sound]`.

It's easy to spot when an enum is being used because of the dot in front of the symbol name. This is actually shorthand notation; you could also have written it like this:

```
center.requestAuthorization(options:
    [UNAuthorizationOptions.alert, UNAuthorizationOptions.sound]) { . . .
```

Fortunately, Swift is smart enough to realize that `.alert` and `.sound` are from the enum `UNAuthorizationOptions`, so you can save yourself some keystrokes.

➤ Run the app. You should immediately get a popup asking for permission:



The permission dialog

Tap **Allow**. The next time you run the app you won't be asked again; iOS remembers what you chose.

(If you tapped Don't Allow – naughty! – then you can always reset the Simulator to get the permissions dialog again. You can also change the notification options in the Settings app.)

➤ Stop the app and add the following code to `didFinishLaunchingWithOptions`:

```
let content = UNMutableNotificationContent()
content.title = "Hello!"
content.body = "I am a local notification"
content.sound = UNNotificationSound.default()

let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10,
                                                repeats: false)
let request = UNNotificationRequest(identifier: "MyNotification",
                                    content: content, trigger: trigger)
center.add(request)
```

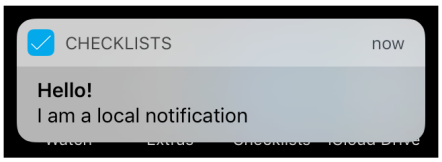
This creates a new local notification. Because you wrote `timeInterval: 10`, it will fire exactly 10 seconds after the app has started.

The `UNMutableNotificationContent` describes what the local notification will say. Here you set a text so that an alert message will be shown when the notification fires. You also set a sound.

Finally, you add the notification to the `UNUserNotificationCenter`. This object is responsible for keeping track of all the local notifications and making them appear when their time is up.

➤ Run the app. Immediately after it has started, exit to the home screen (use the **Hardware → Home** menu item on the Simulator).

Wait 10 seconds… I know, it seems like an eternity! After an agonizing 10 seconds a message should pop up:



The local notification message

➤ Tap the notification to go back to the app.

And that's a local notification. Pretty cool, huh?

Why did I want you to exit to the home screen? iOS will only show an alert with the notification message if the app is not currently active.

➤ Stop the app and run it again. This time don't press Home and just wait.

Well, don't wait too long – nothing will happen. The local notification does get fired, but it is not shown to the user. To handle this situation, we must listen somehow to interesting events that concern these notifications. How? Through a delegate, of course!

➤ Add the following to `AppDelegate`'s `class` declaration:

```
class AppDelegate: UIResponder, UIApplicationDelegate,
                                    UNUserNotificationCenterDelegate {
```

This makes `AppDelegate` the delegate for the `UNUserNotificationCenter`.

➤ Also add the following method to **AppDelegate.swift**:

```
func userNotificationCenter(_ center: UNUserNotificationCenter,
                        willPresent notification: UNNotification,
                        withCompletionHandler completionHandler:
                @escaping (UNNotificationPresentationOptions) -> Void) {
  print("Received local notification \(notification)")
}
```

This method will be invoked when the local notification is posted and the app is still running. You won't do anything here except log a message to the debug pane.

When your app is active and in the foreground, it is supposed to handle any fired notifications in its own manner. Depending on the type of app it may make sense to react to the notification, for example to show a message to the user or to refresh the screen.

➤ Finally, tell the `UNUserNotificationCenter` that `AppDelegate` is now its delegate. You do this in `application(didFinishLaunchingWithOptions)`:

```
center.delegate = self
```

➤ Run the app again and just wait (don't press Home).

After 10 seconds you should see a message in the debug area. It displays something like this:

```
Received local notification <UNNotification: 0x7ff54af135e0; date:
2016-07-11 14:21:27 +0000, request: <UNNotificationRequest: . . .
identifier: MyNotification, content: <UNNotificationContent: . . .
title: Hello!, subtitle: (null), body: I am a local notification,
. . .
```

All right, now you know that it works, you should remove the test code from **AppDelegate.swift** because you don't really want to schedule a new notification every time the user starts the app.

➤ Remove the the local notification code from `didFinishLaunchingWithOptions`, but keep these lines:

```
let center = UNUserNotificationCenter.current()
center.delegate = self
```

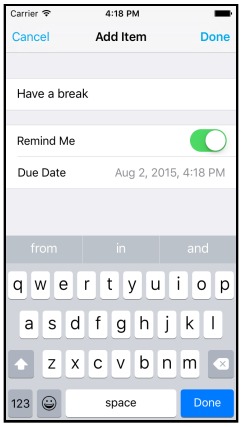
You can also keep the `userNotificationCenter(willPresent…)` method, as it will come in handy when debugging the local notifications.

Extending the data model

Let's think about how the app will handle these notifications. Each `ChecklistItem` will get a due date field (a `Date` object, which specifies a certain date and time) and a `Bool` that says whether the user wants to be reminded of this item or not.

Users might not want to be reminded of everything, so you shouldn't schedule local notifications for those items. Such a `Bool` variable is often called a *flag*. Let's name it `shouldRemind`.

You will add settings for these two new fields to the Add/Edit Item screen and make it look like this:



The Add/Edit Item screen now has Remind Me and Due Date fields

The due date field will require some sort of date picker control. iOS comes with a cool date picker view that you'll add into the table view.

First, let's figure out how and when to schedule the notifications. I can think of the following situations:

- When the user adds a new `ChecklistItem` object that has the `shouldRemind` flag set, you must schedule a new notification.
- When the user changes the due date on an existing `ChecklistItem`, the old notification should be cancelled (if there is one) and a new one scheduled in its place (if `shouldRemind` is still set).
- When the user toggles the `shouldRemind` flag from on to off, the existing notification should be cancelled. The other way around, from off to on, should schedule a new notification.
- When the user deletes a `ChecklistItem`, its notification should be cancelled if it had one.
- When the user deletes an entire `Checklist`, all the notifications for those items should be cancelled.

This makes it obvious that you don't need just a way to schedule new notifications but also a way to cancel them.

You should probably also check that you don't create notifications for to-do items whose due dates are in the past. I'm sure iOS is smart enough to ignore those notifications, but let's be good citizens anyway.

We need some way to associate `ChecklistItem` objects with their local notifications. This requires some changes to our data model.

When you schedule a local notification you create a `UNNotificationRequest` object. It is tempting to put the `UNNotificationRequest` object as an instance variable in `ChecklistItem`, so you always know what it is. However, this is not the correct approach.

Instead, you'll use an *identifier*. When you create a local notification you need to give it an identifier, which is just a `String`. It doesn't really matter what is in this string, as long as it is unique for each notification.

To cancel a notification at a later point you don't use the `UNNotificationRequest` object but the identifier you gave it. The right thing to do is to store this identifier in the `ChecklistItem` object.

Even though the identifier for the local notification is a `String`, you'll give give each `ChecklistItem` an identifier that is simply a number. You'll also save this item ID in the Checklists.plist file. When it's time to schedule or cancel a local notification, you'll turn that number into a string. Then you can easily find the notification when you have the `ChecklistItem` object, or the `ChecklistItem` object when you have the

notification object.

Assigning numeric IDs to objects is a common approach when creating data models – it is very similar to giving records in a relational database a numeric primary key, if you're familiar with that sort of thing.

➤ Make these changes to **ChecklistItem.swift**:

```
var dueDate = Date()
var shouldRemind = false
var itemID: Int
```

Note that you called it `itemID` and not simply "id". The reason is that `id` is a special keyword in Objective-C, and this could cause trouble if you ever wanted to mix your Swift code with Objective-C code.

The `dueDate` and `shouldRemind` variables have initial values, but `itemID` does not. That's why you had to specify the type for `itemID` – it's an `Int` – but not for the other two variables.

Swift is smart enough to infer that `dueDate` cannot be anything but a `Date`, and that `shouldRemind` should be a `Bool`.

You have to extend `init?(coder)` and `encode(with)` in order to be able to load and save these new properties along with the `ChecklistItem` objects.

➤ Add these lines to `init?(coder)` in **ChecklistItem.swift**:

```
dueDate = aDecoder.decodeObject(forKey: "DueDate") as! Date
shouldRemind = aDecoder.decodeBool(forKey: "ShouldRemind")
itemID = aDecoder.decodeInteger(forKey: "ItemID")
```

➤ And add the following lines to `encode(with)`:

```
aCoder.encode(dueDate, forKey: "DueDate")
aCoder.encode(shouldRemind, forKey: "ShouldRemind")
aCoder.encode(itemID, forKey: "ItemID")
```

For `dueDate` you call `decodeObject(forKey)`, but for `shouldRemind` it is `decodeBool()`, and for `itemID` it is `decodeInteger()`. Why do you need different methods to encode and decode these things?

This is necessary because the `NSCoder` system is written in Objective-C and that language makes a distinction between *primitive types* and objects.

In Objective-C, `Int`, `Float`, and `Bool` are primitive types. Everything else, such as `String` and `Date`, is an object. That is different from Swift, which basically treats everything as an object. But because you're talking to an Objective-C framework here, you need to play by the rules of Objective-C.

Great, that takes care of saving and loading existing objects.

Xcode has spotted one remaining error: `init()` still needs to give `itemID` a value. That makes sense: you also have to assign an ID to new `ChecklistItem` objects, which happens in `init()`.

➤ Make the following changes to `init()`:

```
override init() {
  itemID = DataModel.nextChecklistItemID()
  super.init()
}
```

This asks the `DataModel` object for a new item ID whenever the app creates a new `ChecklistItem` object.

Now let's add this new `nextChecklistItemID()` method to `DataModel`. As you can guess from its name this method will return a new, unique ID every time you call it.

➤ Hop on over to **DataModel.swift** and add this new method:

```
class func nextChecklistItemID() -> Int {
  let userDefaults = UserDefaults.standard
  let itemID = userDefaults.integer(forKey: "ChecklistItemID")
  userDefaults.set(itemID + 1, forKey: "ChecklistItemID")
  userDefaults.synchronize()
  return itemID
}
```

You're using your old friend `UserDefaults` again.

This method gets the current "ChecklistItemID" value from `UserDefaults`, adds 1 to it, and writes it back to `UserDefaults`. It returns the previous value to the caller.

The method also does `userDefaults.synchronize()` to force `UserDefaults` to write these changes to disk immediately, so they won't get lost if you kill the app from Xcode before it had a chance to save.

This is important because you never want two or more `ChecklistItems` to get the same ID.

➤ Add a default value for "ChecklistItemID" to the `registerDefaults()` method (note the added comma after "FirstTime"):

```
func registerDefaults() {
  let dictionary: [String: Any] = [ "ChecklistIndex": -1,
                                    "FirstTime": true,
                                    "ChecklistItemID": 0 ]
  . . .
}
```

The first time `nextChecklistItemID()` is called it will return the ID 0. The second time it is called it will return the ID 1, the third time it will return the ID 2, and so on. The number is incremented by one each time. You can call this method a few billion times before you run out of unique IDs.

Class methods vs. instance methods

If you are wondering why you wrote,

```
class func nextChecklistItemID()
```

and not just:

```
func nextChecklistItemID()
```

then I'm glad you're paying attention. :-)

Adding the `class` keyword means that you can call this method without having a reference to the `DataModel` object.

Remember, you did,

```
itemID = DataModel.nextChecklistItemID()
```

instead of:

```
itemID = dataModel.nextChecklistItemID()
```

This is because `ChecklistItem` objects do not have a `dataModel` property with a reference to the `DataModel` object. You could certainly give them such a reference, but I decided that using a *class method* was easier.

The declaration of a class method begins with `class func`. This kind of method applies to the class as a whole.

So far you've been using *instance methods*. They just have the word `func` (without `class`) and work only on a specific instance of that class.

We haven't discussed the difference between classes and instances before, and you'll get into that in more detail in the next tutorial. For now, just remember that a method starting with `class func` allows you to call methods on an object even when you don't have a reference to that object.

I had to make a trade-off: is it worth giving each `ChecklistItem` object a reference to the `DataModel` object, or can I get away with a simple class method? To keep things simple, I chose the latter. It's very well possible that, if you were to develop this app further, it would make more sense to give `ChecklistItem` a `dataModel` property instead.

★ ★ ★

For a quick test to see if assigning these IDs works, you can put them inside the text that is shown in the `ChecklistItem` cell label. This is just a temporary thing for testing purposes, as users couldn't care less about the internal identifier of these objects.

➤ In **ChecklistViewController.swift**, change the `configureText(for:with:)` method to:

```
func configureText(for cell: UITableViewCell,
                   with item: ChecklistItem) {
  let label = cell.viewWithTag(1000) as! UILabel

  //label.text = item.text
  label.text = "\(item.itemID): \(item.text)"
}
```

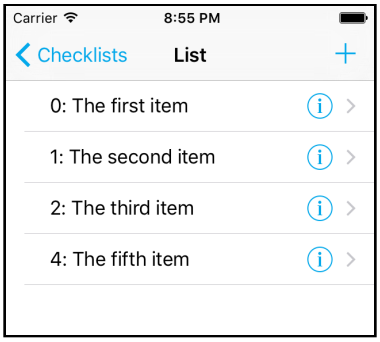
I have commented out the original line because you want to put that back later. The new one uses `\( … )` to add the to-do item's `itemID` property into the text.

➤ Before you run the app, make sure to **reset the Simulator first** or throw away Checklists.plist from the app's Documents directory.

You have changed the format of the Checklists.plist file again and reading an incompatible file may cause crashes.

➤ Run the app and add some checklist items. Each new item should get a unique identifier. Exit to the home screen (to make sure everything is saved properly) and stop the app.

Run the app again and add some new items; the IDs for these new items should start counting at where they left off.



The items with their IDs. Note that the item with ID 3 was deleted in this example.

OK, that takes care of the IDs. Now lets add the "due date" and "should remind"

fields to the Add/Edit Item screen.

(Keep `configureText(for:with:)` the way it is for the time being; that will come in handy with testing the notifications.)

➤ Add the following outlets to **ItemDetailViewController.swift**:

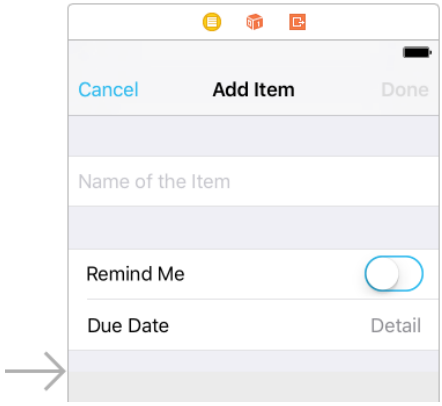
```
@IBOutlet weak var shouldRemindSwitch: UISwitch!
@IBOutlet weak var dueDateLabel: UILabel!
```

➤ Open the storyboard and select the Table View in the Item Detail View Controller (the one that says "Add Item").

➤ Add a new section to the table. The easiest way to do this is to increment the **Sections** field in the **Attributes inspector**. This duplicates the existing section and cell.

➤ Remove the Text Field from the new cell. Drag a new **Table View Cell** from the Object Library and drop it below this one, so that the second section has two rows.

You will now design the new cells to look as follows:



The new design of the Add/Edit Item screen

➤ Add a **Label** to the first cell and give it the text **Remind Me**. Set the font to **System**, size **17**.

➤ Also drag a **Switch** control into the cell. Hook it up to the `shouldRemindSwitch` outlet on the view controller. In the Attributes inspector, set its Value to **Off** so it is no longer green.

➤ Pin the Switch to the top and right edges of the table view cell. This makes sure the control will be visible regardless of the width of the device's screen.

➤ The third cell has two labels: Due Date on the left and the label that will hold the actual chosen date on the right. You don't have to add these labels yourself: simply set the **Style** of the cell to **Right Detail** and rename Title to **Due Date**.

➤ The label on the right should be hooked up to the `dueDateLabel` outlet. (You may need to click it a few times before it is selected and you can make the connection.)

You may need to move the Remind Me label and the switch around a bit to align them nicely with the labels from the "due date" cell. Tip: select the "Due Date" and "Detail" labels and look in the Size inspector what their margins are (should be 15 points from the edges).

Let's write the code for this.

➤ Add a new `dueDate` instance variable to **ItemDetailViewController.swift**:

```
var dueDate = Date()
```

For a new `ChecklistItem` item, the due date is right now, i.e. `Date()`. That sounds reasonable but by the time the user has filled in the rest of the fields and pressed Done, that due date will be in the past.

But you do have to suggest something here. An alternative default value could be this time tomorrow, or ten minutes from now, but in most cases the user will have to pick their own due date anyway.

➤ Change `viewDidLoad()` to the following:

```
override func viewDidLoad() {
  super.viewDidLoad()

  if let item = itemToEdit {
    title = "Edit Item"
    textField.text = item.text
    doneBarButton.isEnabled = true
    shouldRemindSwitch.isOn = item.shouldRemind       // add this
    dueDate = item.dueDate                            // add this
  }

  updateDueDateLabel()                                // add this
}
```

If there already is an existing `ChecklistItem` object, you set the switch control to on or off, depending on the value of the object's `shouldRemind` property. If the user is adding a new item, the switch is initially off (you did that in the storyboard).

You also get the due date from the `ChecklistItem`.

➤ The `updateDueDateLabel()` method is new. Add it to the file:

```
func updateDueDateLabel() {
  let formatter = DateFormatter()
  formatter.dateStyle = .medium
  formatter.timeStyle = .short
  dueDateLabel.text = formatter.string(from: dueDate)
}
```

To convert the `Date` value to text, you use the `DateFormatter` object.

The way it works is very straightforward: you give it a style for the date component and a separate style for the time component, and then ask it to format the `Date` object.

You can play with different styles here but space in the label is limited so you can't fit in the full month name, for example.

The cool thing about `DateFormatter` is that it takes the current locale into consideration so the time will look good to the user no matter where she is on the globe.

➤ The last thing to change in this file is the `done()` action. Change it to:

```
@IBAction func done() {
  if let item = itemToEdit {
    item.text = textField.text!

    item.shouldRemind = shouldRemindSwitch.isOn           // add this
    item.dueDate = dueDate                                // add this

    delegate?.itemDetailViewController(self, didFinishEditing: item)
  } else {
    let item = ChecklistItem()
    item.text = textField.text!
    item.checked = false

    item.shouldRemind = shouldRemindSwitch.isOn           // add this
    item.dueDate = dueDate                                // add this

    delegate?.itemDetailViewController(self, didFinishAdding: item)
  }
}
```

Here you put the value of the switch control and the `dueDate` instance variable back into the `ChecklistItem` object when the user presses the Done button.

➤ Run the app and change the position of the switch control. The app will remember this setting when you terminate it (but be sure to exit to the home screen first).

The due date row doesn't really do anything yet, however. In order to make that work, you first have to create a date picker.

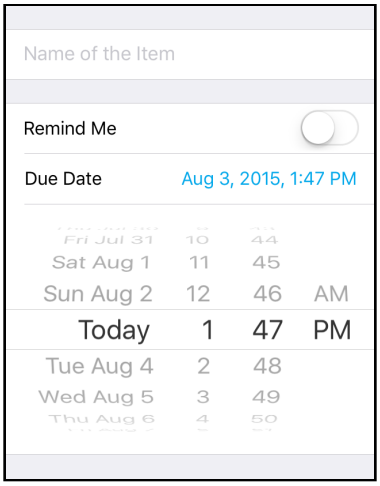
Note: Maybe you're wondering why you're using an instance variable for the `dueDate` but not for `shouldRemind`.

You don't need one for `shouldRemind` because it's easy to get the state of the switch control: you just look at its `isOn` property, which is either `true` or `false`.

However, it is hard to read the chosen date back out of the `dueDateLabel` because the label stores text (a `String`), not a `Date`. So it's easier to keep track of the chosen date separately in a `Date` instance variable.

The date picker

The date picker is not a new view controller. Tapping the Due Date row will insert a new `UIDatePicker` component directly into the table view, just like what happens in the built-in Calendar app.



The date picker in the Add Item screen

➤ Add a new instance variable to **ItemDetailViewController.swift**, to keep track of whether the date picker is currently visible:

```
var datePickerVisible = false
```

➤ And add the `showDatePicker()` method:

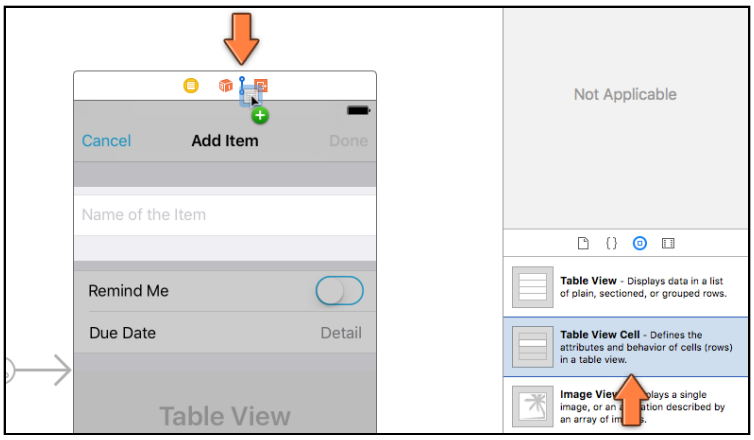
```
func showDatePicker() {
  datePickerVisible = true

  let indexPathDatePicker = IndexPath(row: 2, section: 1)
  tableView.insertRows(at: [indexPathDatePicker], with: .fade)
}
```

This sets the new instance variable to `true`, and tells the table view to insert a new row below the Due Date cell. This new row will contain the `UIDatePicker`.

The question is: where does the cell for this new date picker row come from? You can't put it into the table view as a static cell already because then it would always be visible. You only want to show it after the user taps the Due Date row.

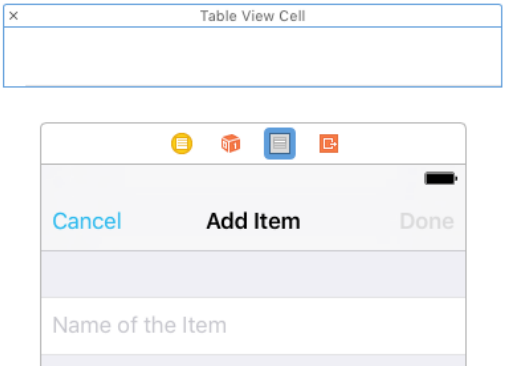
Xcode has a cool new feature that lets you add additional views to a scene that are not immediately visible. That's a great solution to this problem!

➤ Open the storyboard and go to the **Add Item** scene. From the Object Library, pick up a new **Table View Cell**. Don't drag it into the view controller itself but into the scene dock at the top:



Dragging a table view cell into the scene dock

After dragging, the storyboard should look like this:



The new table view cell sits in its own area

The new Table View Cell object belongs to the scene but it is not (yet) part of the scene's table view.

The cell is a bit too small to fit a date picker, so first you'll make it bigger.

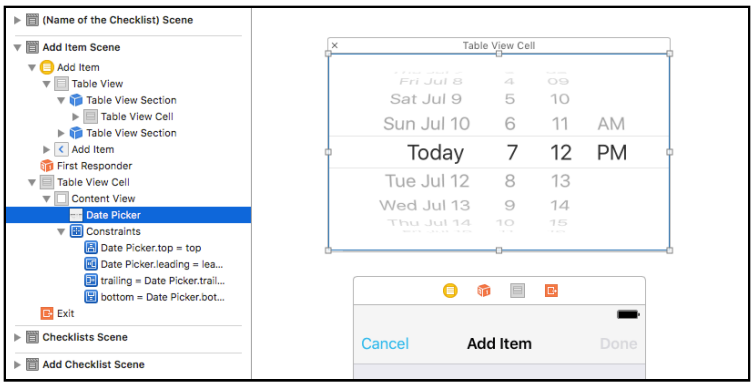
➤ Select the Table View Cell and in the **Size inspector** set the **Height** to 217. The date picker is 216 points tall, plus one point for the separator line at the bottom of the cell.

➤ In the **Attributes inspector**, set **Selection** to **None** so this cell won't turn gray when you tap on it.

➤ From the Object Library, drag a **Date Picker** into the cell. It should fit exactly.

➤ Use the **Pin menu** to glue the Date Picker to the four sides of the cell. Turn off **Constrain to margins** and then select the four bars to make them red (they all should be 0).

When you're done, the new cell looks like this:



The finished date picker cell

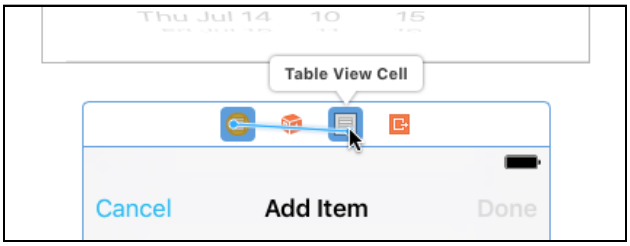
So how do you get this cell into the table view? First, make two new outlets and connect them to the cell and the date picker, respectively. That way you can refer to these views from code.

➤ Add these lines to **ItemDetailViewController.swift**:

```
@IBOutlet weak var datePickerCell: UITableViewCell!
@IBOutlet weak var datePicker: UIDatePicker!
```

Back in the storyboard, take a look at that scene dock again. Besides an icon for the table view cell you just added it also has a round yellow icon. This represents the view controller.

➤ To connect the outlet, simply Ctrl-drag from that yellow icon to the gray icon for the Table View Cell, and select the **datePickerCell** outlet:



Ctrl-drag between the icons in the scene dock

➤ To connect the date picker, Ctrl-drag from the yellow icon to the big Date Picker above it and select the **datePicker** outlet.

Great! Now that you have outlets for the cell and the date picker inside it, you can write the code to add them to the table view.

Normally you would implement the `tableView(cellForRowAt)` method, but remember that this screen uses a table view with static cells. Such a table view does not have a data source and therefore does not use "`cellForRowAt`".

If you look in **ItemDetailViewController.swift** you won't find that method

anywhere. However, with a bit of trickery you can override the data source for a static table view and provide your own methods.

➤ Add the `tableView(cellForRowAt)` method to **ItemDetailViewController.swift**:

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

  if indexPath.section == 1 && indexPath.row == 2 {
    return datePickerCell
  } else {
    return super.tableView(tableView, cellForRowAt: indexPath)
  }
}
```

Danger: You shouldn't really mess around too much with this method when it's being used by a static table view, because it may interfere with the inner workings of those static cells. But if you're careful you can get away with it.

The `if`-statement checks whether "`cellForRowAt`" is being called with the index-path for the date picker row. If so, it returns the new `datePickerCell` that you just designed. This is safe to do because the table view from the storyboard doesn't know anything about row 2 in section 1, so you're not interfering with an existing static cell.

For any index-paths that are not the date picker cell, this method will call through to `super` (which is `UITableViewController`). This is the trick that makes sure the other static cells still work.

➤ You also need to override `tableView(numberOfRowsInSection)`:

```
override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
  if section == 1 && datePickerVisible {
    return 3
  } else {
    return super.tableView(tableView, numberOfRowsInSection: section)
  }
}
```

If the date picker is visible, then section 1 has three rows. If the date picker isn't visible, you can simply pass through to the original data source.

➤ Likewise, you also need to provide the `tableView(heightForRowAt)` method:

```
override func tableView(_ tableView: UITableView,
                        heightForRowAt indexPath: IndexPath) -> CGFloat {
  if indexPath.section == 1 && indexPath.row == 2 {
    return 217
  } else {
    return super.tableView(tableView, heightForRowAt: indexPath)
  }
}
```

So far the cells in your table views all had the same height (44 points), but this is not a hard requirement. By providing the "`heightForRowAt`" method you can give each cell its own height.

The `UIDatePicker` component is 216 points tall, plus 1 point for the separator line, making for a total row height of 217 points.

The date picker is only made visible after the user taps the Due Date cell, which happens in `tableView(didSelectRowAt)`.

➤ Add that method:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  tableView.deselectRow(at: indexPath, animated: true)
  textField.resignFirstResponder()

  if indexPath.section == 1 && indexPath.row == 1 {
    showDatePicker()
  }
}
```

This calls `showDatePicker()` when the index-path indicates that the Due Date row was tapped. It also hides the on-screen keyboard if that was visible.

At this point you have most of the pieces in place, but the Due Date row isn't actually tap-able yet. That's because **ItemDetailViewController.swift** already has a "`willSelectRowAt`" method that always returns `nil`, causing taps on all rows to be ignored.

➤ Change `tableView(willSelectRowAt)` to:

```
override func tableView(_ tableView: UITableView,
                        willSelectRowAt indexPath: IndexPath) -> IndexPath? {
  if indexPath.section == 1 && indexPath.row == 1 {
    return indexPath
  } else {
    return nil
  }
}
```

Now the Due Date row responds to taps, but the other rows don't.

➤ Run the app to try it out. Add a new checklist item and tap the Due Date row.

Whoops. The app crashes. After some investigating I found that when you override the data source for a static table view cell, you also need to provide the delegate method `tableView(indentationLevelForRowAt)`.

That's not a method you'd typically use, but because you're messing with the data source for a static table view you do need to override it. I told you this was a little tricky.

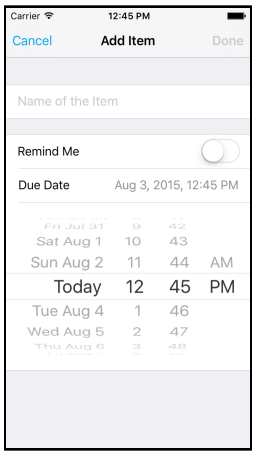
➤ Add the `tableView(indentationLevelForRowAt)` method:

```
override func tableView(_ tableView: UITableView,
                  indentationLevelForRowAt indexPath: IndexPath) -> Int {
  var newIndexPath = indexPath
  if indexPath.section == 1 && indexPath.row == 2 {
    newIndexPath = IndexPath(row: 0, section: indexPath.section)
  }
  return super.tableView(tableView,
                         indentationLevelForRowAt: newIndexPath)
}
```

The reason the app crashed on this method was that the standard data source doesn't know anything about the cell at row 2 in section 1 (the one with the date picker), because that cell isn't part of the table view's design in the storyboard.

So after inserting the new date picker cell the data source gets confused and it crashes the app. To fix this, you have to trick the data source into believing there really are three rows in that section when the date picker is visible.

➤ Run the app again. This time the date picker cell shows up where it should:



The date picker appears in a new cell

Interacting with the date picker should change the date in the Due Date row but currently this has no effect (try it out: spin the wheels).

You have to listen to the date picker's "Value Changed" event. That event gets sent whenever the wheels settle on a new value. For that, you need to add a new action method.

➤ Add the `dateChanged()` method to **ItemDetailViewController.swift**:

```
@IBAction func dateChanged(_ datePicker: UIDatePicker) {
  dueDate = datePicker.date
  updateDueDateLabel()
}
```

This is pretty simple. It updates the `dueDate` instance variable with the new date and then updates the text on the Due Date label.

➤ In the storyboard, Ctrl-drag from the Date Picker to the view controller and select the **dateChanged:** action method. Now everything is properly hooked up.

(You can verify that the action method is indeed connected to the date picker's Value Changed event by looking at the Connections inspector.)

➤ Run the app to try it out. When you turn the wheels on the date picker, the text in the Due Date row updates too. Cool.

However, when you edit an existing to-do item, the date picker does not show the date from that item. It always starts on the current date and time.

➤ Add the following line to the bottom of `showDatePicker()`:

```
  datePicker.setDate(dueDate, animated: false)
```

This gives the proper date to the `UIDatePicker` component.

➤ Verify that it works: click on the (i) button from an existing to-do item, preferably one you made a while ago, and confirm that the date picker shows the same date and time as the Due Date label. Excellent!

Speaking of the label, it would be nice if this becomes highlighted when the date picker is active. You can use the tint color for this (that's also what the Calendar app does).

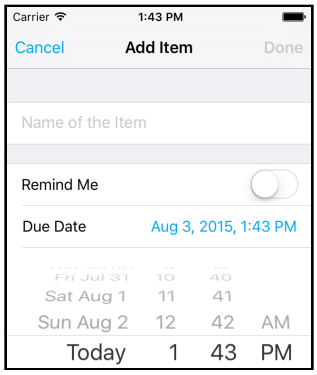
➤ Change `showDatePicker()` one last time:

```
  func showDatePicker() {
    datePickerVisible = true

    let indexPathDateRow = IndexPath(row: 1, section: 1)
    let indexPathDatePicker = IndexPath(row: 2, section: 1)

    if let dateCell = tableView.cellForRow(at: indexPathDateRow) {
      dateCell.detailTextLabel!.textColor =
                                    dateCell.detailTextLabel!.tintColor
    }

    tableView.beginUpdates()
    tableView.insertRows(at: [indexPathDatePicker], with: .fade)
    tableView.reloadRows(at: [indexPathDateRow], with: .none)
    tableView.endUpdates()

    datePicker.setDate(dueDate, animated: false)
  }
```

This sets the `textColor` of the `detailTextLabel` to the tint color. It also tells the table view to reload the Due Date row. Without that, the separator lines between the cells don't update properly.

Because you're doing two operations on the table view at the same time – inserting a new row and reloading another – you need to put this in between calls to `beginUpdates()` and `endUpdates()`, so that the table view can animate everything at the same time.

➤ Run the app. The date now appears in blue:



The date label appears in the tint color while the date picker is visible

When the user taps the Due Date row again, the date picker should disappear. If you try that right now the app will crash – what did you expect! – which obviously won't win it many favorable reviews.

➤ Add the new `hideDatePicker()` method:

```
func hideDatePicker() {
  if datePickerVisible {
    datePickerVisible = false

    let indexPathDateRow = IndexPath(row: 1, section: 1)
    let indexPathDatePicker = IndexPath(row: 2, section: 1)

    if let cell = tableView.cellForRow(at: indexPathDateRow) {
      cell.detailTextLabel!.textColor = UIColor(white: 0, alpha: 0.5)
    }

    tableView.beginUpdates()
    tableView.reloadRows(at: [indexPathDateRow], with: .none)
    tableView.deleteRows(at: [indexPathDatePicker], with: .fade)
    tableView.endUpdates()
  }
}
```

This does the opposite of `showDatePicker()`. It deletes the date picker cell from the table view and restores the color of the date label to medium gray.

➤ Change `tableView(didSelectRowAt)` to toggle between the visible and hidden states:

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

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

    if indexPath.section == 1 && indexPath.row == 1 {
      if !datePickerVisible {
        showDatePicker()
      } else {
        hideDatePicker()
      }
    }
  }
```

There is another situation where it's a good idea to hide the date picker: when the user taps inside the text field.

It won't look very nice if the keyboard partially overlaps the date picker, so you might as well hide it. The view controller is already the delegate for the text field, making this easy.

➤ Add the `textFieldDidBeginEditing()` method:

```
  func textFieldDidBeginEditing(_ textField: UITextField) {
    hideDatePicker()
  }
```

And with that you have a cool inline date picker!

➤ Run the app and verify that hiding the date picker works, also when you activate the text field.

Scheduling the local notifications

One of the principles of object-oriented programming is that objects can do as much as possible themselves. Therefore, it makes sense that the `ChecklistItem` object can schedule its own notifications.

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

```
  func scheduleNotification() {
    if shouldRemind && dueDate > Date() {
      print("We should schedule a notification!")
    }
  }
```

This compares the due date on the item with the current date. You can always get the current time by making a new `Date` object with `Date()`.

The statement `dueDate > Date()` compares the two dates and returns `true` if `dueDate` is in the future and `false` if it is in the past.

If the due date is in the past, the `print()` will not be performed.

Note the use of the `&&` "and" operator. You only print the text when the Remind Me

switch is set to "on" *and* the due date is in the future.

You will call this method when the user presses the Done button after adding or editing a to-do item.

➤ In the `done()` action in **ItemDetailViewController.swift**, add the following line just before the call to `didFinishEditing` and also before `didFinishaAdding:

```
item.scheduleNotification()
```

➤ Run the app and try it out. Add a new item, set the switch to ON but don't change the due date. Press Done.

There should be no message in the debug area because the due date has already passed (it is several seconds in the past by the time you press Done).

➤ Add another item, set the switch to ON, and choose a due date in the future.

When you press Done now, there should be a `print` in the debug area ("We should schedule a notification!").

Now that you've verified the method is called in the proper place, let's actually schedule a new local notification object. First consider the case of a new to-do item being added.

➤ In **ChecklistItem.swift**, change `scheduleNotification()` to:

```
func scheduleNotification() {
  if shouldRemind && dueDate > Date() {
    // 1
    let content = UNMutableNotificationContent()
    content.title = "Reminder:"
    content.body = text
    content.sound = UNNotificationSound.default()

    // 2
    let calendar = Calendar(identifier: .gregorian)
    let components = calendar.dateComponents(
                          [.month, .day, .hour, .minute], from: dueDate)
    // 3
    let trigger = UNCalendarNotificationTrigger(
                                dateMatching: components, repeats: false)
    // 4
    let request = UNNotificationRequest(
            identifier: "\(itemID)", content: content, trigger: trigger)
    // 5
    let center = UNUserNotificationCenter.current()
    center.add(request)

    print("Scheduled notification \(request) for itemID \(itemID)")
  }
}
```

You've seen this code before when you tried out local notifications for the first time,

but there are a few differences.

1. Put the item's `text` into the notification message.
2. Extract the month, day, hour, and minute from the `dueDate`. We don't care about the year or the number of seconds – the notification doesn't need to be scheduled with millisecond precision, on the minute is precise enough.
3. To test the local notifications you used a `UNTimeIntervalNotificationTrigger`, which scheduled the notification to appear after a number of seconds. Here, you're using a `UNCalendarNotificationTrigger`, which shows the notification at the specified date.
4. Create the `UNNotificationRequest` object. Important here is that we convert the item's numeric ID into a `String` and use it to identify the notification. That is how you'll be able to find this notification later in case you need to cancel it.
5. Add the new notification to the `UNUserNotificationCenter`.

Xcode is not so impressed with this new code and gives a bunch of error messages.

What is wrong here? `UNUserNotificationCenter` and the other objects are provided by the User Notifications framework – you can tell by the "UN" in their names.

However, `ChecklistItem` hasn't used any code from that framework until now. The only frameworks object it has used, `NSObject` and `NSCoder`, came from another framework, Foundation.

➤ To tell `ChecklistItem` about the User Notifications framework, you need to add the following line to the top of the file, below the other `import`:

```
import UserNotifications
```

Now the errors disappear like snow in the sun.

There's another small problem, though. If you've reset the Simulator recently (and you probably have when you made the most recent data model changes) then the app no longer has permission to send local notifications.

➤ Try it out. Run the app, add a new checklist item, set the due date a minute into the future, and press Done. You should not see a notification.

You can't assume the app has permission anymore. When you were just messing around at this beginning of this section, you placed the permission request code in the `AppDelegate` and ran it immediately upon launch. That's not recommended.

Don't you just hate those apps that prompt you for ten different things before you've even had a chance to properly look at them? Let's be a bit more user friendly with our own app!

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

```
@IBAction func shouldRemindToggled(_ switchControl: UISwitch) {
  textField.resignFirstResponder()

  if switchControl.isOn {
    let center = UNUserNotificationCenter.current()
    center.requestAuthorization(options: [.alert, .sound]) {
      granted, error in /* do nothing */
    }
  }
}
```

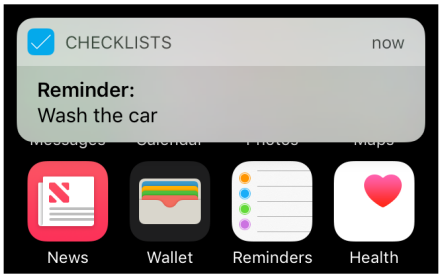
When the switch is toggled to ON, this prompts the user for permission to send local notifications. Once the user has given permission, the app won't put up a prompt again.

➤ Also add an `import UserNotifications` or the above method won't compile.

➤ Open the storyboard and connect the **shouldRemindToggled:** action to the switch control.

➤ Test it out. Run the app, add a new checklist item, set the due date a minute into the future, press Done and exit to the home screen.

Wait one minute (patience…) and the notification should appear. Pretty cool!



The local notification when the app is in the background

That takes care of the case where you're adding a new notification. There are two situations left: 1) the user edits an existing item, and 2) the user deletes an item. Let's do editing first.

When the user edits an item, the following situations can occur:

- Remind Me was switched off and is now switched on. You have to schedule a new notification.
- Remind Me was switched on and is now switched off. You have to cancel the existing notification.
- Remind Me stays switched on but the due date changes. You have to cancel the existing notification and schedule a new one.
- Remind Me stays switched on but the due date doesn't change. You don't have to

do anything.

- Remind Me stays switched off. Here you also don't have to do anything.

Of course, in all those situations you'll only schedule the notification if the due date is in the future.

Phew, that's quite a list. It's always a good idea to take stock of all possible scenarios before you start programming because this gives you a clear picture of everything you need to tackle.

It may seem like you need to write a lot of logic here to deal with all these situations, but actually it turns out to be quite simple.

First you'll look if there is an existing notification for this to-do item. If there is, you simply cancel it. Then you determine whether the item should have a notification and if so, you schedule a new one.

That should take care of all the above situations, even if sometimes you simply could have left the existing notification alone. The algorithm is crude, but effective.

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

```
func removeNotification() {
  let center = UNUserNotificationCenter.current()
  center.removePendingNotificationRequests(
                                        withIdentifiers: ["\(itemID)"])
}
```

This removes the local notification for this `ChecklistItem`, if it exists. Note that `removePendingNotificationRequests()` requires an array of identifiers, so we first put our `itemID` into a string with `\(…)` and then into an array using `[]`.

➤ Call this new method from to the top of `scheduleNotification()`:

```
func scheduleNotification() {
  removeNotification()
  . . .
}
```

Let's try it out.

➤ Run the app and add a to-do item with a due date two minutes into the future. A new notification will be scheduled. Go to the home screen and wait until it shows up.

➤ Edit the item and change the due date to three minutes into the future. The old notification will be removed and a new one scheduled for the new time.

➤ Add a new to-do item with a due date two minutes into the future. Edit the to-do item but now set the switch to OFF. The old notification will be removed and no new notification will be scheduled.

➤ Edit again and put the time a few minutes into the future but don't change anything else; no new notification will be scheduled because the switch is still off.

These tests should also work if you terminate the app in between.

There is one last case to handle: deletion of the `ChecklistItem`object. This can happen in two ways:

1. the user can delete an individual item using swipe-to-delete,
2. the user can delete an entire checklist in which case all its `ChecklistItem` objects are also deleted.

An object is notified when it is about to be deleted using the `deinit` message. You can simply implement this method, look if there is a scheduled notification for this item and then cancel it.

➤ Add the following to the bottom of **ChecklistItem.swift**:

```
deinit {
  removeNotification()
}
```

That's all you have to do. The special `deinit` method will be invoked when you delete an individual `ChecklistItem` but also when you delete a whole `Checklist` – because all its `ChecklistItems` will be destroyed as well, as the array they are in is deallocated.

➤ Run the app and try it out. First schedule some notifications a minute or so into the future and then remove that to-do item or its entire checklist. Wait until the due date comes and you shouldn't get a notification.

Once you're convinced everything works, you can remove the `print()` statements. They are only temporary for debugging purposes. You probably don't want to leave them in the final app. The `print()` statements won't hurt any, but the end user can't see those messages anyway.

➤ Also remove the item ID from the label in the `ChecklistViewController` – that was only used for debugging.

You can find the final project files for the Checklists app under **11 - Local Notifications** in the tutorial's Source Code folder.

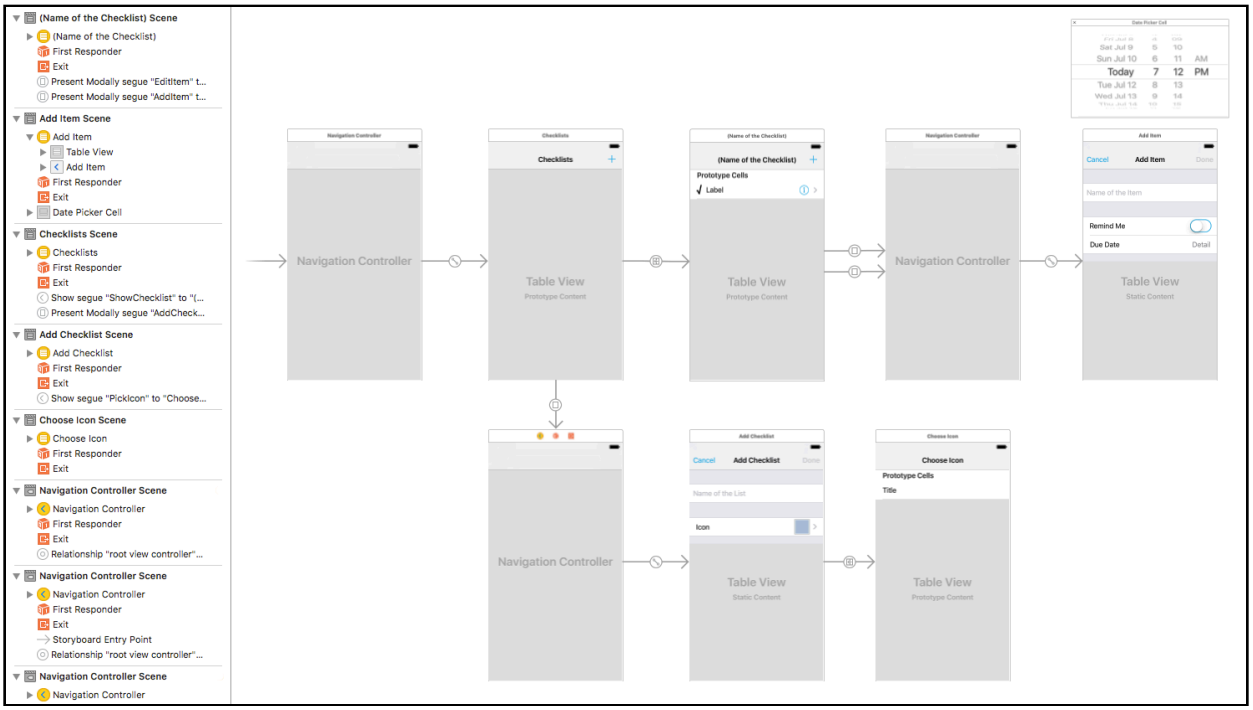
That's a wrap!

Things should be starting to make sense by now. I've thrown you into the deep end by writing an entire app from scratch, and we've touched on a number of advanced topics already, but hopefully you were able to follow along quite well with what we've been doing. Kudos for sticking with it until the end!

It's OK if you're still a bit fuzzy on the details. Sleep on it for a bit and keep tinkering with the code. Programming requires its own way of thinking and you won't learn that overnight. Don't be afraid to do this tutorial again from the start – it will make more sense the second time around!

This lesson focused mainly on UIKit and its most important controls and patterns. In the next lesson we'll take a few steps back to talk more about the Swift language itself and of course you'll build another cool app.

Here is the final storyboard for **Checklists**:



The final storyboard

I had trouble fitting that on my screen!

Take a well-deserved break, and when you're ready continue on to the next tutorial, where you'll make an app that knows its place! :-)

Haven't had enough yet? Here are some challenges to sink your teeth into:

Exercise: Put the due date in a label on the table view cells under the text of the to-do item. ■

Exercise: Sort the to-do items list based on the due date. This is similar to what you did with the list of `Checklists` except that now you're sorting `ChecklistItem` objects and you'll be comparing `Date` objects instead of strings. ■