Fully updated for iOS 7!

iOS Apprentice

SECOND EDITION

Tutorial 2: Checklists

By Matthijs Hollemans

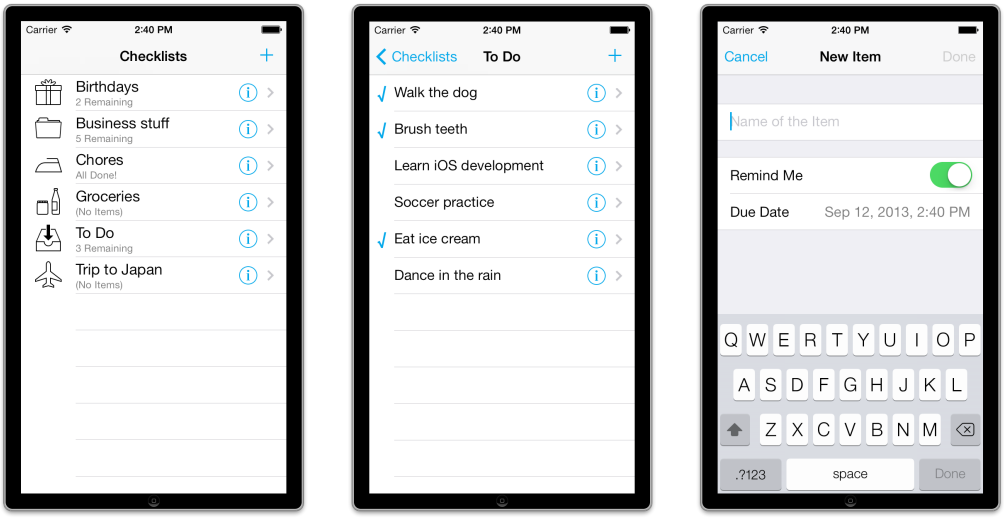
The iOS Apprentice:
Checklists

By Matthijs Hollemans

Table of Contents

To-do list apps are one of the most popular types of app on the App Store, second only to fart apps. Apple even included their own Reminders app as of iOS 5 (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 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.

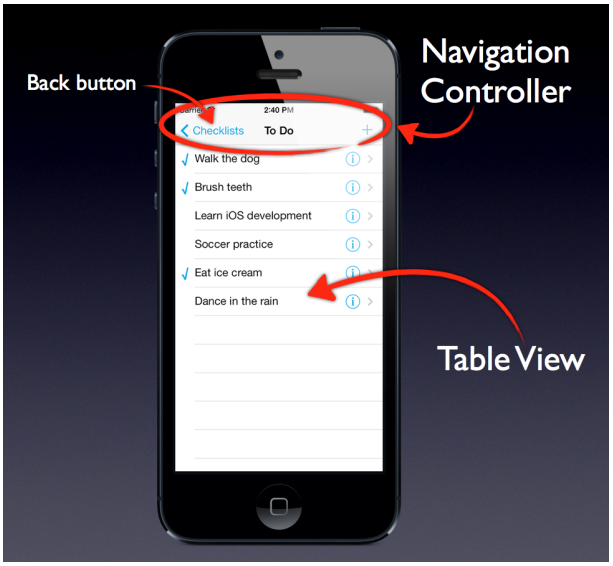
As far as to-do list apps go, Checklists is very basic, but don't let that fool you. Even a simple app such as this already has five different screens and a lot of complexity behind the scenes.

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**.

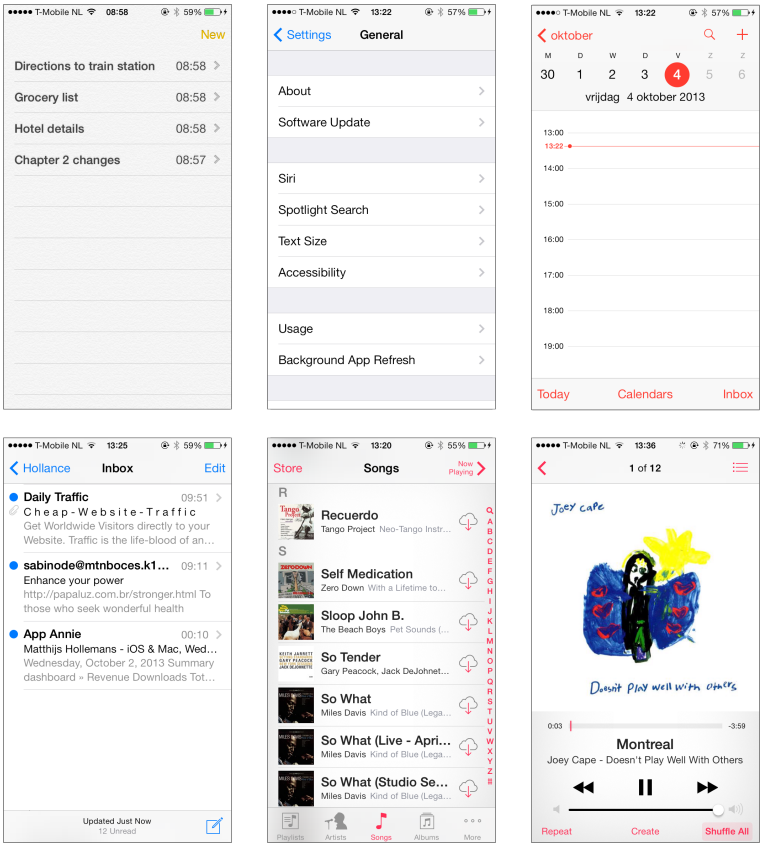
- A table view shows a list of things. All three of the screens above use a table view. In fact, all of this app's screens are made with table views. This component is extremely versatile and the most important one to master in iOS development.
- The navigation controller allows you to build a hierarchy of screens that lead from one to another. It adds a navigation bar at the top of the screen with a title and optional buttons. In this app, tapping on the name of a checklist – "To Do", for example – slides in the screen that contains the to-do items from that list. A tap on the "back button" in the upper-left corner takes you back to the previous screen with a swift animation. That is the navigation controller at work; you have no doubt seen it before in other apps.

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.

If you take a look at the apps that come with your iPhone – Calendar, Notes, Contacts, Mail, Settings – you'll notice that even though they look slightly different, all these apps still work in very much the same way. That's because they all use table views and navigation controllers. (The Music app also has a tab bar at the bottom, something we'll talk about in the next tutorial.)



These are all table views inside navigation controllers: Notes, Settings, Calendar, Mail, Music

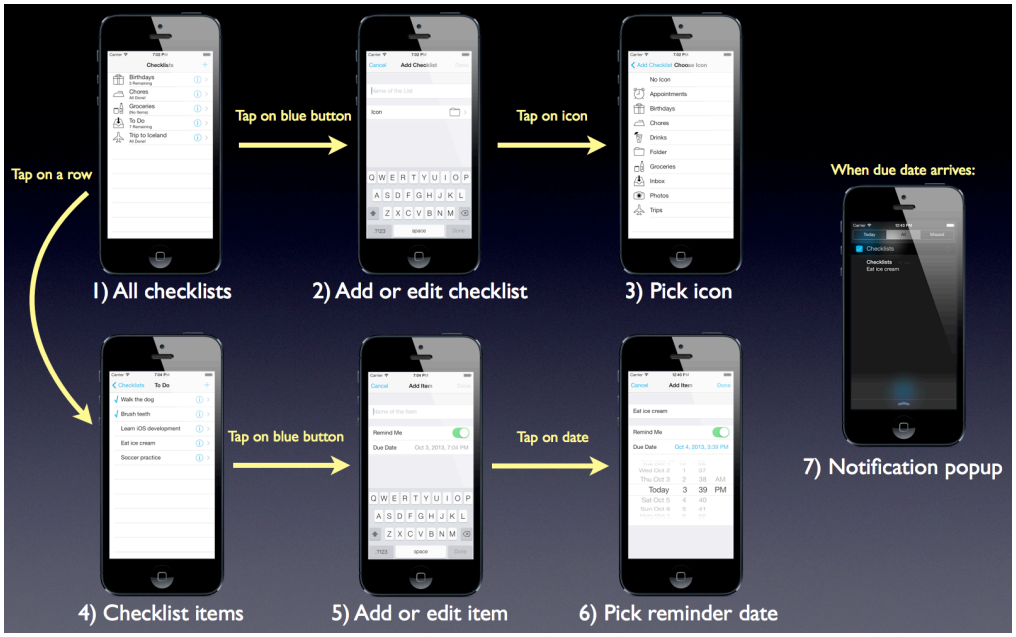
If you want to learn how to program iOS apps, you need to master these two components as they make an appearance in almost every app. That's exactly what you'll focus on in this tutorial. You'll also learn how to pass data from one screen to another, a very important topic that often puzzles beginners.

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 will be writing. Change stuff and see what it does, even if it breaks the app. Playing with the code is the quickest way to learn!

The Checklists 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 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). An 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).

The app uses local notifications to automatically notify the user of checklist items that have their "remind me" option set, even if the app isn't running (6). Pretty cool.

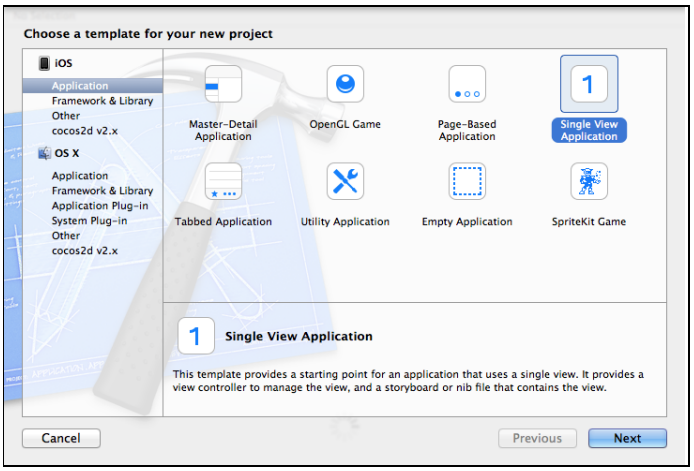
Playing with table views

Seeing as table views are so important, you will start out by examining how table views work. Because I always like to 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 on a row in the table to toggle a checkmark on and off

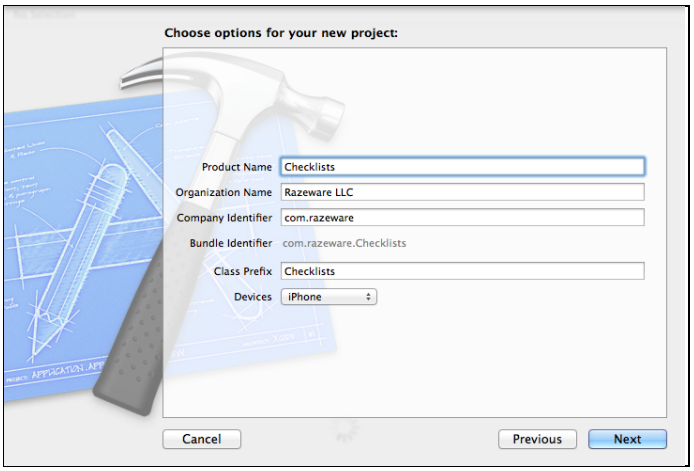
Once you have these basics up and running, you'll keep adding new functionality to it over the course of this tutorial until you end up with the full-blown app.

› 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:

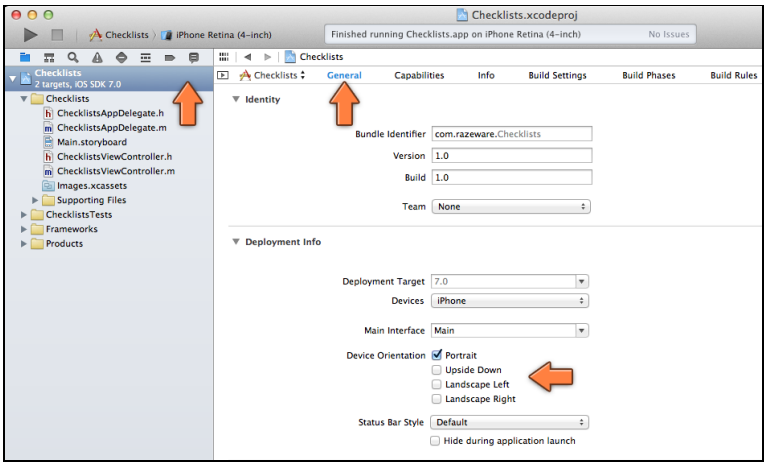
- Product Name: **Checklists**
- Organization Name: Your name or the name of your company
- Company Identifier: Use your own identifier here, using reverse domain name notation
- Class Prefix: **Checklists**
- Devices: iPhone

› 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 make sure the General tab is selected. Under **Deployment Info**, **Device Orientation**, de-select the Landscape Left and Landscape Right buttons so 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, then 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. If they were to answer with the phone upside down, the microphone is at the wrong end. (iPad apps, on the other hand, are supposed to support all four orientations including upside-down.)

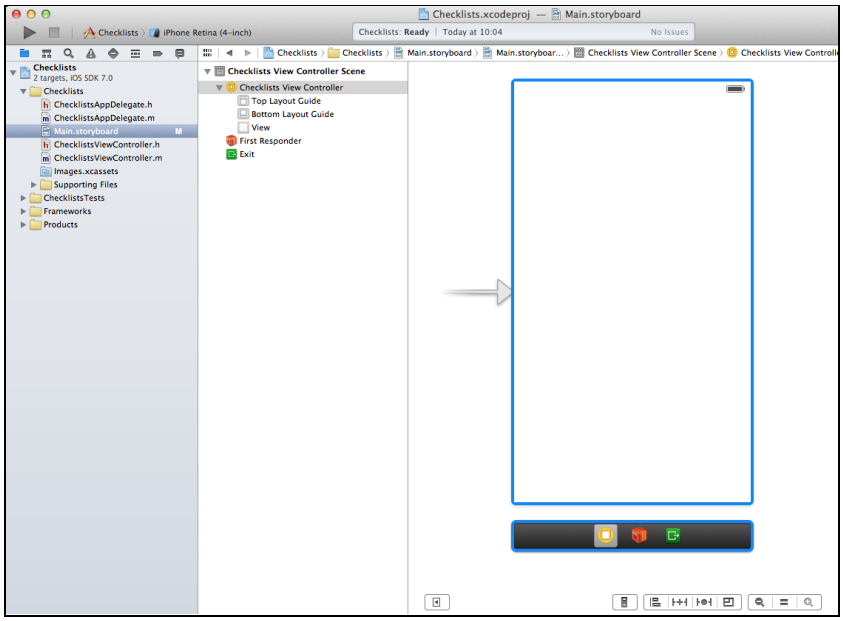
Storyboards

For this app you will use **storyboarding**, a technique introduced with iOS 5 that makes apps easier and quicker to write. Before iOS 5 you had to make a separate file for each of your app's screens but with storyboarding the designs for all your view controllers are combined into a single Storyboard file. You already used a storyboard in Bull's Eye but in this tutorial you will unlock their full power.

As you can see in the project navigator on the left side of the screen, Xcode automatically made the **ChecklistsViewController.h** and **.m** files for you, which

contain the view controller source code for the main screen. Recall that a view controller represents one screen of your app.

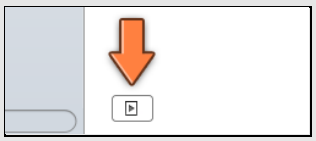
› Click on **Main.storyboard** to open Interface Builder.



The storyboard editor with the app's only scene

In storyboard terminology, each view controller is named a **scene**.

› Remove the **Checklists View Controller Scene** from the storyboard so that the canvas is empty (the outline pane to the left should say "No Scenes").

> **Note:** 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.
>
> 

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 `ChecklistsViewController`'s type to a table view controller, you first have to edit its .h file.

› Click on **ChecklistsViewController.h** to open it in the source code editor and change the `@interface` line from this:

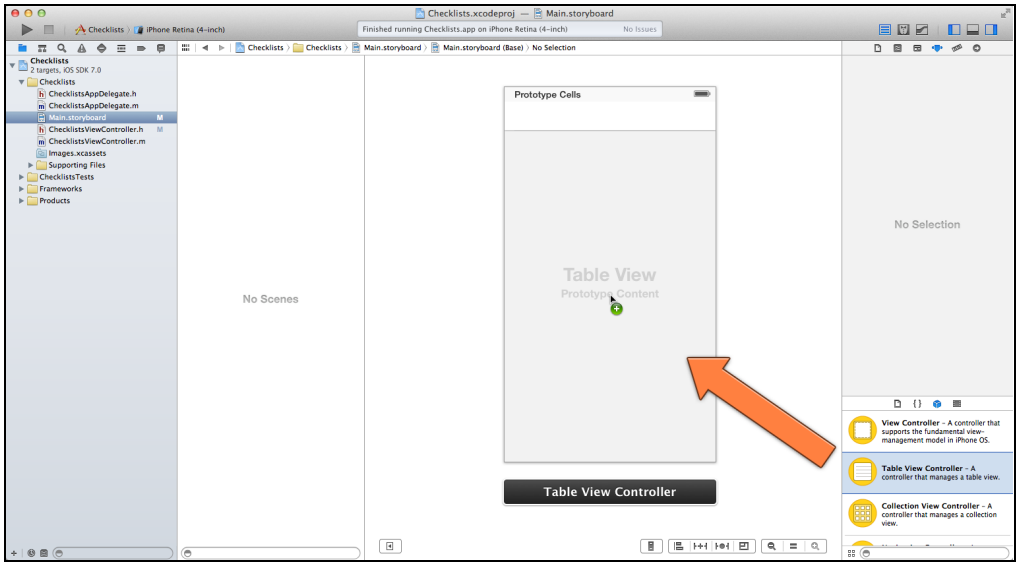
```
@interface ChecklistsViewController : UIViewController
```

to this:

```
@interface ChecklistsViewController : UITableViewController
```

With this change you tell the Objective-C compiler that the view controller is now a `UITableViewController` object instead of a regular `UIViewController`.

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

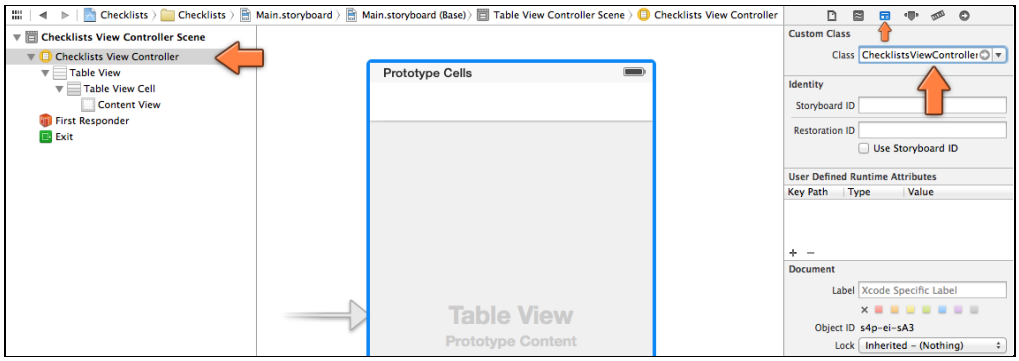


Dragging a Table View Controller into the storyboard

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 **ChecklistsViewController** (or choose it using the blue arrow).

The name of the scene in the Scene List on the left changes to "Checklists View Controller Scene". (When you do this, make sure the actual Table View Controller is selected, not the Table View inside it. There should be a fat blue border around the scene.)



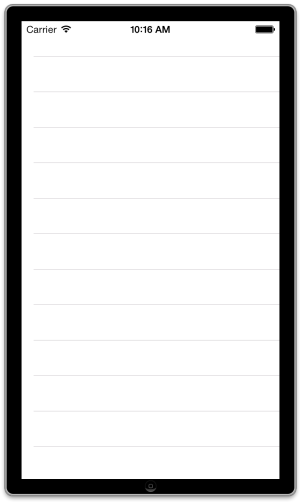
Changing the Custom Class of the Table View Controller

You have now changed the `ChecklistsViewController` from a regular view controller 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.

› Run the app on the **iPhone Retina (4-inch)** simulator.

Instead of a plain white screen you'll now 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.

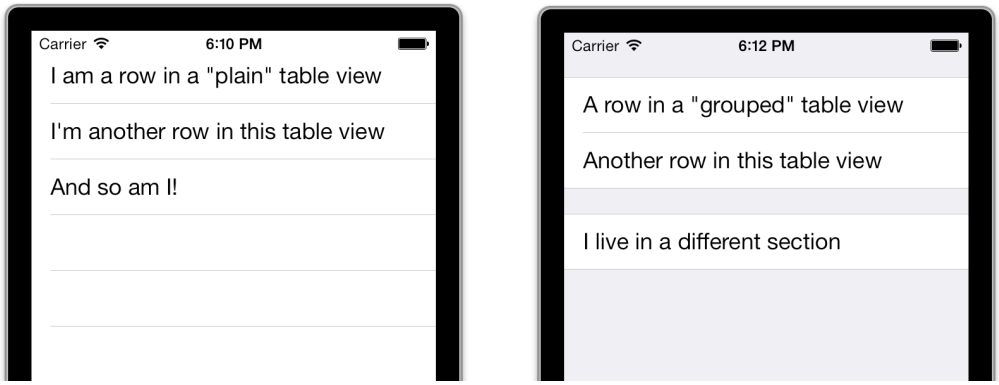


The app now uses a table view controller

The anatomy of a table view

First, let's talk a bit more about table views. A `UITableView` object displays a list of things. I'm not sure why it's named a table because a table is commonly thought of as a spreadsheet-type thing 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.

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 combined into boxes (the groups) on a light gray background.

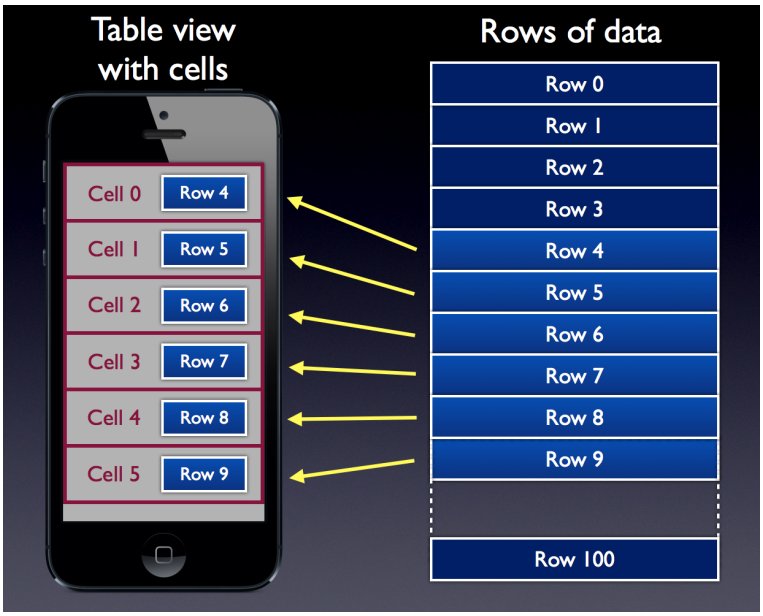


A plain-style table (left) and a grouped table (right)

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...

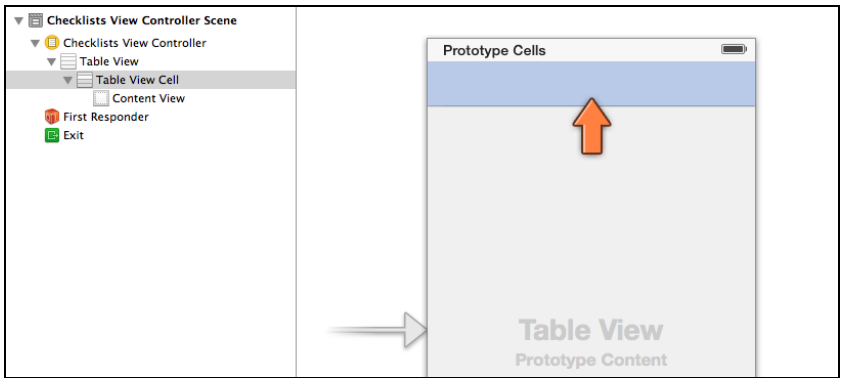
Tables display their data in **cells**. A cell is related to a row but it's not exactly the same. A cell is a view that shows a row of data that happens to be visible at that moment. If your table can show 10 rows at a time on the screen, then it only has 10 cells, even though there may be hundreds of 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

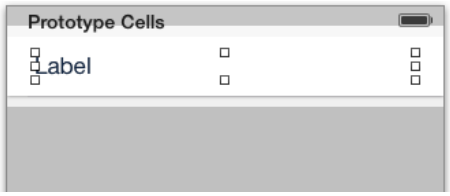
Until iOS 5 you had to put in quite a bit of effort to create cells for your tables but these days Xcode has a very handy new feature named **prototype cells** that lets you design your cells visually in Interface Builder.

› Open the storyboard and click the empty cell to select it. It will become blue:



Selecting the prototype cell

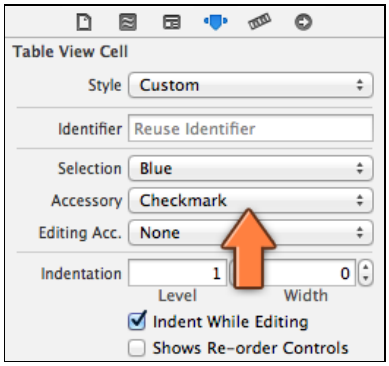
› Drag a **Label** from the Object Library into this cell. Give the label some placeholder text: **Checklist Item**. Make sure the label spans the entire width of the cell (but leave a small margin on the sides).



Adding the label to the prototype cell

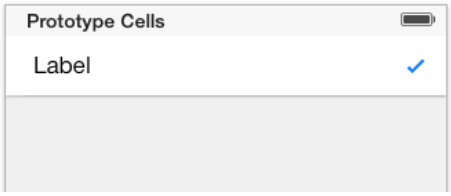
Besides the label you will also add a checkmark to the cell's design. The checkmark is provided by something called the **accessory**, which is a built-in subview that appears on the right side of the cell. You can choose from a few standard accessory controls or provide your own.

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



Changing the accessory to get a checkmark

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.

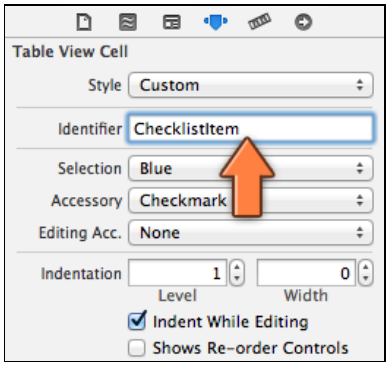
You also need to set a **reuse identifier** on the cell. This is an internal name that the table view uses to find free cells to reuse when rows scroll off the screen and new rows must become visible. The table needs to assign cells to those new rows and recycling existing cells is more efficient than creating new cells. This technique is what makes your 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.



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 **ChecklistsViewController.m** and add the following methods right before the `@end` line at the bottom of the file:

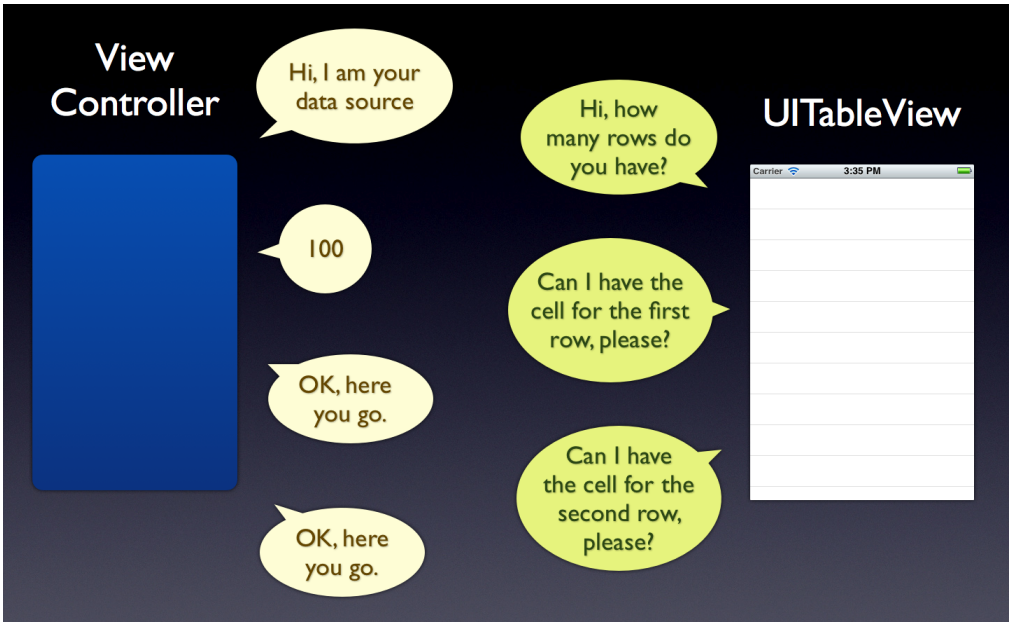
```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
  return 1;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
            dequeueReusableCellWithIdentifier:@"ChecklistItem"];

  return cell;
}
```

These two 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, the table view sends a `numberOfRowsInSection` message when it wants to know how many rows there are. And when the table view needs to display a particular row it sends the `cellForRowAtIndexPath` message to ask the data source for a cell.

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

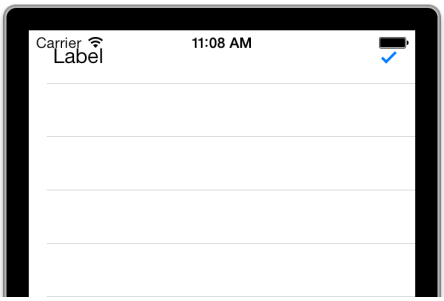


The dating ritual of a data source and a table view

Your implementation of `numberOfRowsInSection` returns the value 1. This tells the table view that you just have one row of data.

The table view then calls the `cellForRowAtIndexPath` method to obtain a cell for that row. Inside `cellForRowAtIndexPath` you simply grab a copy of the prototype cell and give that back to the table view. This is where you would normally put the row data into the cell, but the app doesn't have any row data yet.

▸ If you haven't already, run the app and you'll see that there is now a single cell in the table:



The table now has one row

Notice how the iPhone's status bar partially overlaps the table view. This is new in iOS 7. On previous versions of iOS the status bar had its own separate area, but now it is simply drawn on top of the view controller. Later in this chapter you will fix this small cosmetic problem by placing a navigation bar on top of the table view.

Methods with multiple parameters

Most of the methods you have used so far took only one parameter or did not have any parameters at all, but these table view data source methods take two:

```
- (NSInteger)tableView:(UITableView *)tableView  // parameter 1
       numberOfRowsInSection:(NSInteger)section  // parameter 2
{
    . . .
}

- (UITableViewCell *)tableView:(UITableView *)tableView   // 1
         cellForRowAtIndexPath:(NSIndexPath *)indexPath   // 2

{
    . . .
}
```

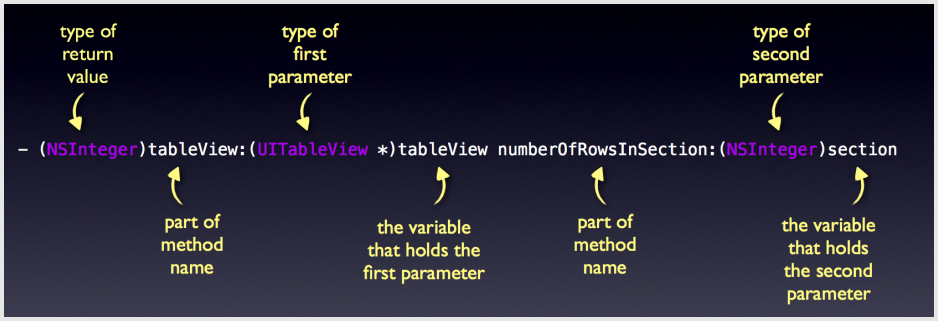
The first parameter of both methods is the `UITableView` object in question (the table view on whose behalf these methods are invoked) and the second parameter is either the section number in the case of `numberOfRowsInSection`, or something called the "indexPath" in the case of `cellForRowAtIndexPath`.

In other programming languages a method with multiple parameters typically looks like this:

```
void numberOfRowsInSection(UITableView *tableView,
                                          NSInteger section)
{
    . . .
}
```

But that's not the way we do it in Objective-C. It may look a little weird if you're coming from another language, but once you get used to it you'll find that this notation is actually quite readable.

To summarize, these are the various parts that make up a method declaration:



The name of this method is officially `tableView:numberOfRowsInSection:` (including the colons). If you pronounce that out loud, it actually makes sense. It asks for the number of rows in a particular section in a particular table view.

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

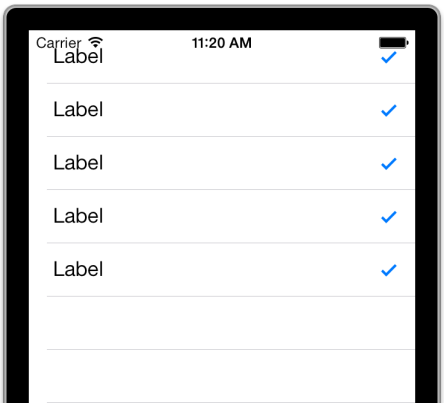
That shouldn't have been too hard:

```
- (NSInteger)tableView:(UITableView *)tableView
       numberOfRowsInSection:(NSInteger)section
{
  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 `numberOfRowsInSection` return the number 5, you tell the table view that there will be five rows. The table view then calls `cellForRowAtIndexPath` five times, once for each row.

Because `cellForRowAtIndexPath` currently just returns a copy of the prototype cell, your table view now shows five identical rows:



The table now has five identical rows

There are several ways that you can create cells in `cellForRowAtIndexPath` but by far the easiest approach is what you've done here: you add a prototype cell to the table view in the storyboard and then call `[tableView dequeueReusableCellWithIdentifier]` with that cell's reuse identifier. It sounds scary but this simply 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.

Index paths

You've seen that the table view asks the data source for a cell using the `cellForRowAtIndexPath` method. So what is an **index-path**?

`NSIndexPath` is simply an object that points to a specific row in the table. It is a combination of a row number and a section number, that's all. 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.

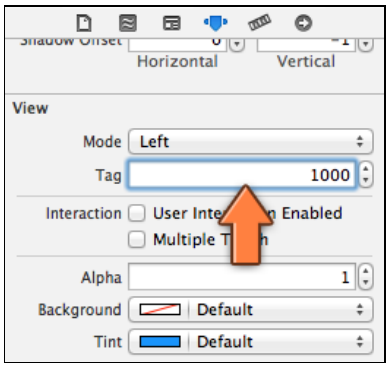
It is also possible for tables to group rows into sections. In an address book app you might sort contacts by last name. All contacts whose last name starts with "A" are grouped into their own section, all contacts whose last name starts with "B" are in another section, and so on. To find out to which section a row belongs, you would 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 `NSIndexPath` for now.

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. Therefore, for the first row in the first section, `indexPath.row` is 0 and `indexPath.section` is also 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.

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, or any type of view really, 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.

Note: Double check to make sure you set the tag on the *label*. It's a common mistake to set it on the table view cell itself instead of the label and then the results won't be what you expect!

➤ In **ChecklistsViewController.m**, change `cellForRowAtIndexPath` to:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
           dequeueReusableCellWithIdentifier:@"ChecklistItem"];

  UILabel *label = (UILabel *)[cell viewWithTag:1000];

  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, which gets a copy of the prototype cell (either a new one or a recycled one) and puts it into the `cell` local variable:

```
  UITableViewCell *cell = [tableView
            dequeueReusableCellWithIdentifier:@"ChecklistItem"];
```

The first new line in this method is:

```
  UILabel *label = (UILabel *)[cell viewWithTag:1000];
```

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 that `UILabel` object. Using tags is a handy trick to get a reference to a control without having to make a `@property` for it.

Exercise: Why can't you simply add an `IBOutlet` property 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… □

Answer: There will likely be more than one cell in the table (at least enough of them to cover all the visible rows) 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 property would 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. (That doesn't mean you cannot use properties with table view cells at all. In the MyLocations tutorial I'll show you how to use properties for the controls in your table view cells.)

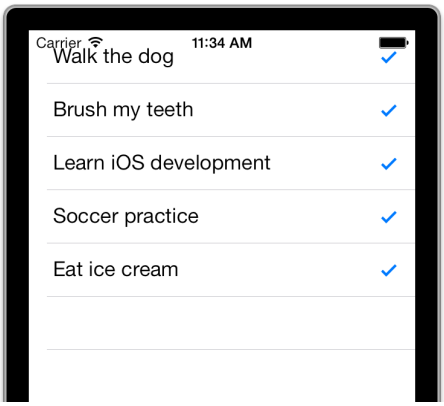
Back to the code. The next bit shouldn't give you too much trouble:

```
  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 – remember that you start counting at index 0 – gets the text "Walk the dog", the cell for the second row gets the text "Brush my teeth", and so on.

▸ Run the app and see that it has five rows, each with their own text:



The rows in the table now have their own text

That is how you write the `cellForRowAtIndexPath` 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 `NSIndexPath`.

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

▸ Change the code to the following (the highlighted bits indicate the changes):

```
- (NSInteger)tableView:(UITableView *)tableView
     numberOfRowsInSection:(NSInteger)section
{
  return 100;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
       cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"ChecklistItem"];

  UILabel *label = (UILabel *)[cell viewWithTag:1000];
```

```
  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";
  }

  return cell;
}
```

It is mostly the same as before, except that `numberOfRowsInSection` returns 100 and `cellForRowAtIndexPath` uses a slightly different method to determine which text to display where:

```
if (indexPath.row % 5 == 0) {
} else if (indexPath.row % 5 == 1) {
} else if (indexPath.row % 5 == 2) {
} else if (indexPath.row % 5 == 3) {
} else if (indexPath.row % 5 == 4) {
}
```

This uses the **modulo operator**, represented by the `%` sign, to determine what row you're on.

If you're unsure of the modulo operator: it returns the remainder of a division. You may have seen this in maths class. For example 13 % 4 = 1, because 13 divided by 4 is 3.25, which is 3 with a leftover of 0.25. The modulus of 13 and 4 is therefore 1 because what remains is one-fourth. 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...

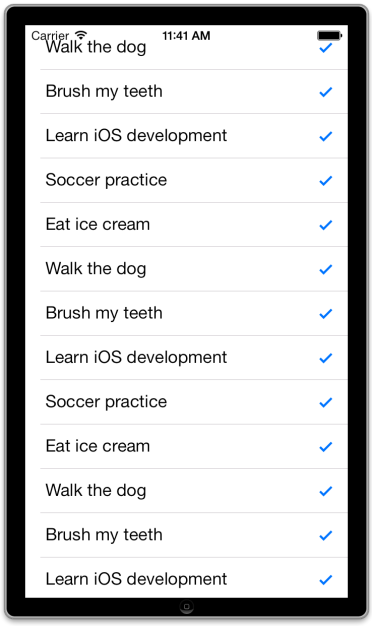
I think you get the picture, every five rows these lines repeat. Rather than typing in all the possibilities all the way up to a hundred, you let the computer calculate this for you (that is what they are good at):

```
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
                                                   repeats here
Seventh row:   6 % 5 = 1  (same as second row)
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
                                                   repeats again
Twelfth row:  11 % 5 = 1  (same as second row)
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.

➤ Run the app and you should see this:



The table now has 100 rows

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

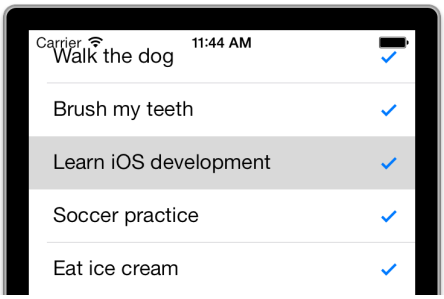
Answer: There are 100 rows but only 14 fit on the screen at a time. If you count the number of visible rows in the screenshot above you'll get up to 13 but it's possible to scroll the table in such a way that the top cell is still visible and a new cell is pulled in from below. So that makes at least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

If you scroll really fast, then I guess it is possible that the table view needs to make a few more temporary cells, but I'm not sure about that. Is this important to know? Not really. You should let the table view take care of juggling the cells behind the

scenes. All you have to do is give the table view a cell when it asks for it and fill it up with the data from the corresponding row.

Tapping on the rows

When you tap a row, notice that it colors gray. But when you let go of the row, it 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 that 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 delegation pattern

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.

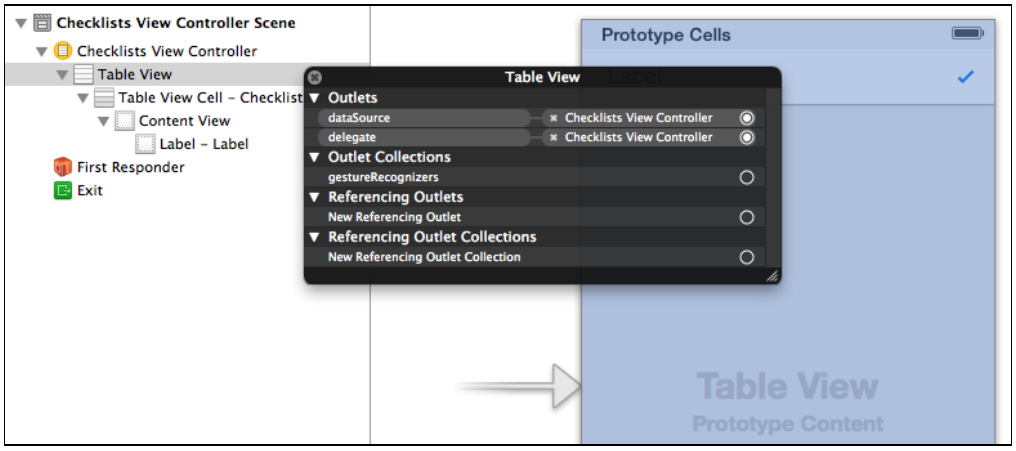
The table view doesn't really care who its data source is or what kind of data your app deals with, just that it can send the `cellForRowAtIndexPath` message and that it will receive a cell in return. This keeps the table view component simple and moves the responsibility for handling the data to where 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 but 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. (Sometimes it's not entirely clear to which of these a particular piece of functionality belongs so you may have to check the documentation for both.)

▸ Open the storyboard and **Ctrl-click** on the table view. 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 regular view controllers but then you'll have to connect the data source and delegate manually.)



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

▸ Add the following method to **ChecklistsViewController.m**, just before `@end`:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Run the app and tap a row; you'll see that the row briefly turns gray and then becomes de-selected again.

▸ Let's make `didSelectRowAtIndexPath` toggle the checkmark, so change it to:

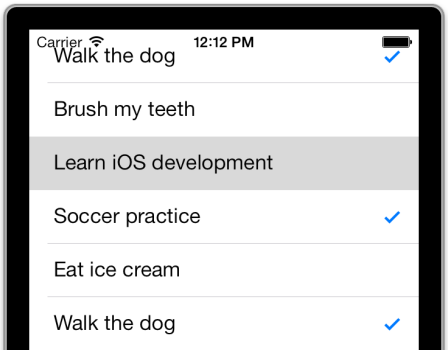
```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
```

```
  UITableViewCell *cell = [tableView
                               cellForRowAtIndexPath:indexPath];

  if (cell.accessoryType == UITableViewCellAccessoryNone) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

First, you get the `UITableViewCell` object in question. You simply ask the table view: what is the cell at this `indexPath` you've given me? (Note that you call `cellForRowAtIndexPath` directly on the table view, not on the view controller. They are different methods with the same name in different objects.)

Then 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.

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



You can now tap on a row to toggle the checkmark

Note: If the checkmark only appears after you select *another* row, then make sure the method name is `tableView:didSelectRowAtIndexPath:` and not `didDeselect`! Xcode's autocompletion may trick you here into picking the wrong method name.

Sweet. However, you may have noticed there is a problem with the app. Here's how to reproduce it:

▸ Tap a row to remove the checkmark. Scroll that row off the screen and now scroll back (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 whether you should 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.

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.

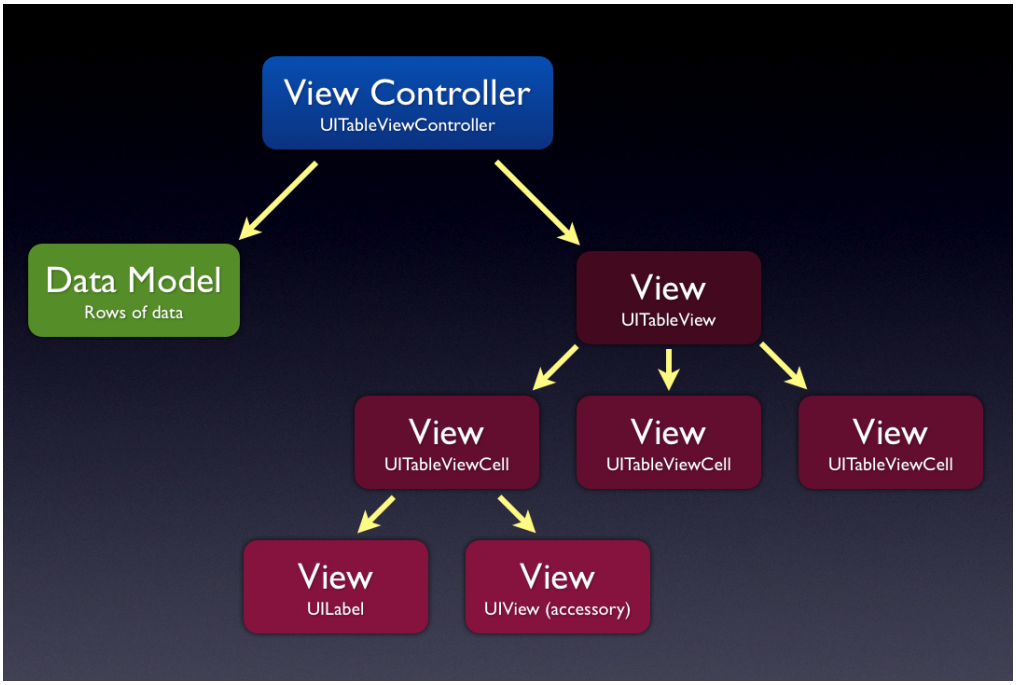
Model-View-Controller

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 roughly means that all 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 are 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, and so on. In a game the views are 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 view 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 view controller is in charge.

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 `self.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 a storyboard (or a nib file).

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 has its own views and is handled by its own view controller. 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 mail compose controller that lets you write email, the image picker controller for photos, and the tweet sheet for sending Twitter messages.

Views vs. view controllers

Note 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.

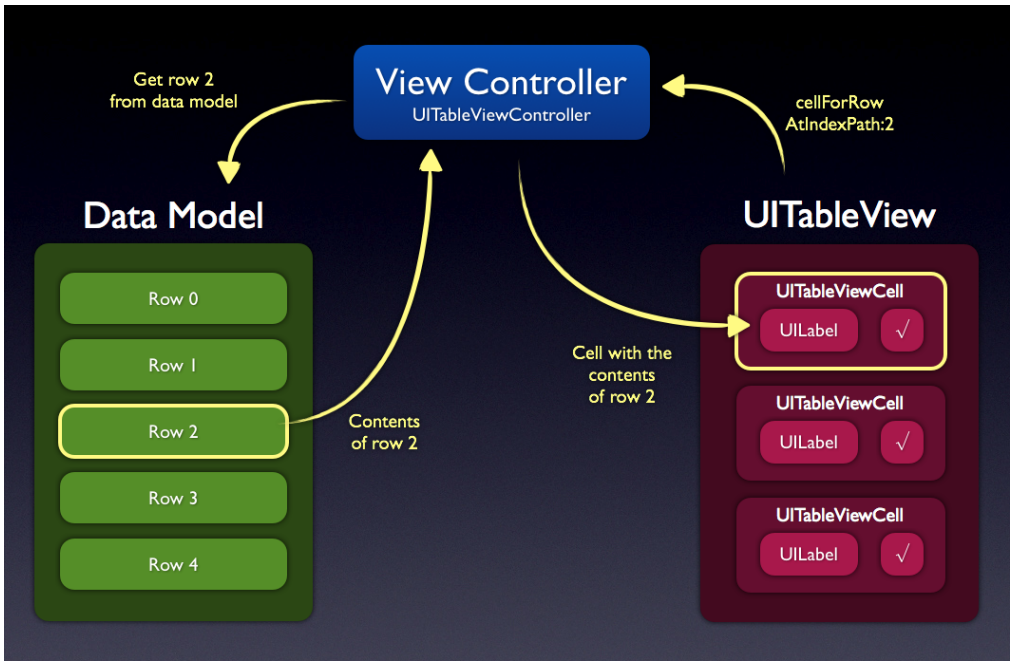
A lot of beginners give their view controllers names such as `FirstView` and `MainView`. Don't do that, it is very confusing! If something is a view controller, call it "ViewController" and not "View".

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. The view controller doesn't just control a view, it also controls your model. It is the bridge that sits between the two.

Creating the data model

So far you've put a bunch of fake data into the table view. As you saw, you cannot just use the cells to remember the data as cells get re-used all the time and their old contents get overwritten. The cell is part of the view and is just used to display the data, but that data actually comes from somewhere else: the data model.

The rows are the data, the cells are the views. The table view controller is the thing that ties them together as it implements 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. Note: this is what you *shouldn't* do. It will work but it isn't very smart. Even though this is the wrong 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 bad so you'll be able to appreciate the proper solution better.

➤ In **ChecklistsViewController.m**, add the following instance variables after the `@implementation` line:

```
@implementation ChecklistsViewController
{
  NSString *_row0text;
  NSString *_row1text;
  NSString *_row2text;
  NSString *_row3text;
  NSString *_row4text;
}
```

They are instance variables so their names begin with an underscore.

➤ Change the `viewDidLoad` method into the following:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  _row0text = @"Walk the dog";
  _row1text = @"Brush teeth";
  _row2text = @"Learn iOS development";
  _row3text = @"Soccer practice";
  _row4text = @"Eat ice cream";
}
```

➤ Change the data source methods into:

```
- (NSInteger)tableView:(UITableView *)tableView
      numberOfRowsInSection:(NSInteger)section
{
  return 5;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
          dequeueReusableCellWithIdentifier:@"ChecklistItem"];
```

```
  UILabel *label = (UILabel *)[cell viewWithTag:1000];

  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. Those five instance variables are your data model.

In `cellForRowAtIndexPath` 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.

➤ Add the following instance variables:

```
@implementation ChecklistsViewController
{
  NSString *_row0text;
  NSString *_row1text;
  NSString *_row2text;
  NSString *_row3text;
  NSString *_row4text;

  BOOL _row0checked;
  BOOL _row1checked;
  BOOL _row2checked;
  BOOL _row3checked;
  BOOL _row4checked;
}
```

`BOOL` is a datatype just like `int` and `NSString`, except that it can hold only two possible values: `YES` and `NO`. In other languages these are commonly called "true" and "false" but Objective-C uses the simpler terms `YES` and `NO` (in all capitals).

`BOOL` is short for "boolean", after Englishman George Boole who long ago invented a type 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 (`YES`) or not (`NO`). The names of boolean variables often start with the verb "is" or "has", as in `isHungry` or `hasIceCream`.

In your case, the instance variable `_row0checked` is `YES` if the first row has its checkmark set and `NO` 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.

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.

➤ Replace `didSelectRowAtIndexPath` with the following:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                                  cellForRowAtIndexPath:indexPath];

  BOOL isChecked = NO;

  if (indexPath.row == 0) {
    isChecked = _row0checked;
    _row0checked = !_row0checked;
  } else if (indexPath.row == 1) {
    isChecked = _row1checked;
    _row1checked = !_row1checked;
  } else if (indexPath.row == 2) {
    isChecked = _row2checked;
    _row2checked = !_row2checked;
  } else if (indexPath.row == 3) {
    isChecked = _row3checked;
    _row3checked = !_row3checked;
  } else if (indexPath.row == 4) {
    isChecked = _row4checked;
    _row4checked = !_row4checked;
  }

  if (isChecked) {
    cell.accessoryType = UITableViewCellAccessoryNone;
```

```
    } else {
      cell.accessoryType = UITableViewCellAccessoryCheckmark;
    }

    [tableView deselectRowAtIndexPath:indexPath animated:YES];
  }
```

You examine `indexPath.row` to find the row in question, and then look up the proper "row checked" instance variable. For the first row that is `_row0checked`, for the second row it is `_row1checked`, and so on. You store its value into the temporary (local) variable `isChecked`, which you use at the bottom of the method to set or remove the checkmark on the cell.

You do the following to flip the 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 `YES`, then `!` makes it `NO`. Conversely, `!NO` is `YES`. Think of `!` as "not": not yes is no and not no is yes. Yes?

➤ Run the app and observe... that it 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: if you don't set a value in a `BOOL` variable then its default value is `NO`. So `_row0checked` and the others think that there is no checkmark on the row, but the table draws one anyway 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 `YES` to begin with, or you could remove the checkmark from the prototype cell in the storyboard. Neither is a foolproof solution because 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 forgot to set the checkmark properly in `cellForRowAtIndexPath`.

When you are asked for a new cell, you always should configure all of its properties. The call to `dequeueReusableCellWithIdentifier` could return a cell that was previously used for a row with a checkmark, so if the new row doesn't have a checkmark you have to remove it from the cell at this point (and vice versa). Let's fix that.

➤ Add the following method above `cellForRowAtIndexPath`:

```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell
                      atIndexPath:(NSIndexPath *)indexPath
{
  BOOL isChecked = NO;
  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 = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
}
```

This new method looks at the cell for a certain row (specified by `indexPath`) and makes the checkmark visible if the corresponding "row checked" variable is `YES`, or hides it if `NO`.

You'll call this method in `cellForRowAtIndexPath`, just before you return the cell.

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

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  . . .

  [self configureCheckmarkForCell:cell atIndexPath: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 guarantees that each cell always has the value that corresponds to its row.

Why did you make `configureCheckmarkForCell` a method of its own? Well, you can use it to simplify `didSelectRowAtIndexPath`. That method handles taps on the row and toggles the "row checked" variable and then updates the cell. You can simplify things by letting `configureCheckmarkForCell` do some of the work.

➤ Replace the `tableView:didSelectRowAtIndexPath:` method with the following:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                                cellForRowAtIndexPath: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;
  }

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

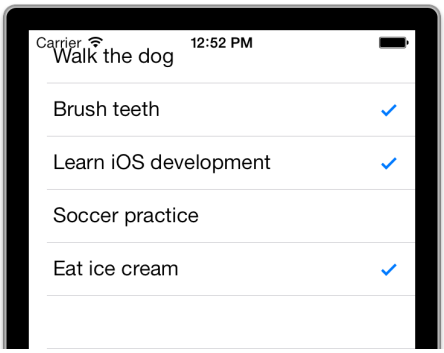
This method no longer sets or clears the checkmark from the cell, but only toggles the
"checked" state in the data model and then calls `configureCheckmarkForCell` to update the view.

➤ Run the app again and it should still work.

➤ Add the following to `viewDidLoad` and run the app again:

```
- (void)viewDidLoad
{
  . . .

  _row1checked = YES;
  _row2checked = YES;
  _row4checked = YES;
}
```

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



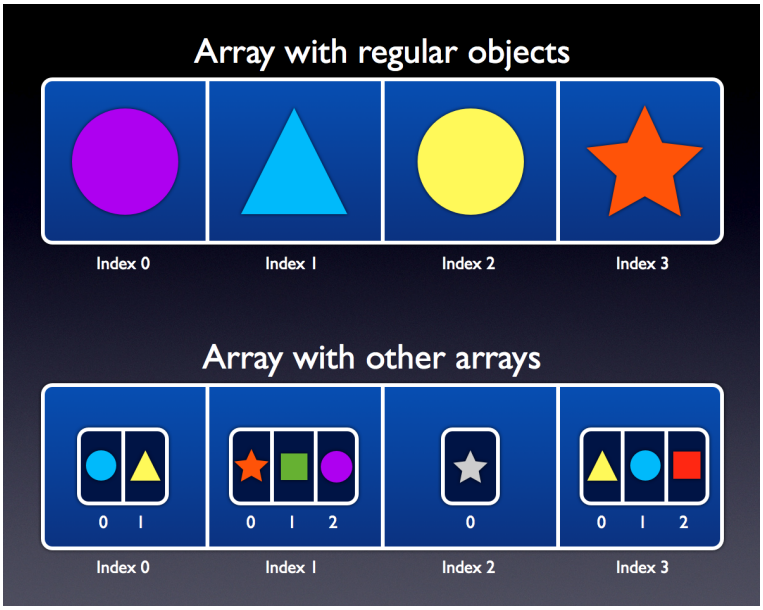
The data model and the table view cells are now always in-sync

This approach works, but you'll have to agree that the code is becoming unwieldy very quickly. 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" instance 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. Of course, the array itself is also an object (named `NSArray`) that you can put into a variable. And because arrays are objects, arrays can contain other arrays.



Arrays are ordered lists that can contain objects, including other arrays

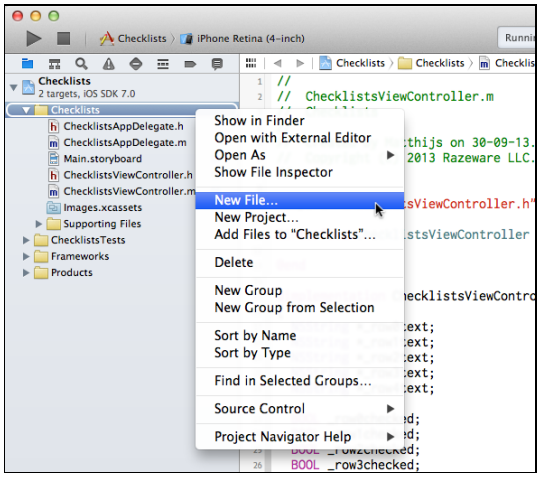
The objects inside an array are indexed by numbers, starting at 0 as usual. To ask the array for the first object, you do `[array objectAtIndex:0]` or the shorthand `array[0]`. The array is *ordered*, which means that the order of the objects it contains matters. The object at index 0 always comes before the object at index 1.

`NSArray` is a so-called **collection** object. There are several other collection objects, such as `NSDictionary` and `NSSet`, and they all organize their objects in a different fashion. (A dictionary contains key-value pairs, just like a real dictionary contains a list of words and a description for each of those words. A set is like an array except that the order of the objects doesn't matter. You'll use these other collections 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 that you put your data model's rows into an array.

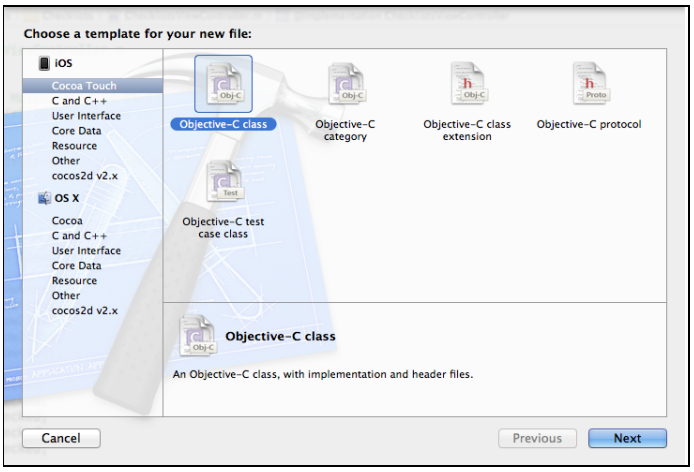
Arrays store objects, but your rows currently consist of two separate pieces of data: the text and the checked state. It would be easier if you made a single object for each row, because then the row number from the table simply becomes the index in the array. Let's combine the text and checkmark state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



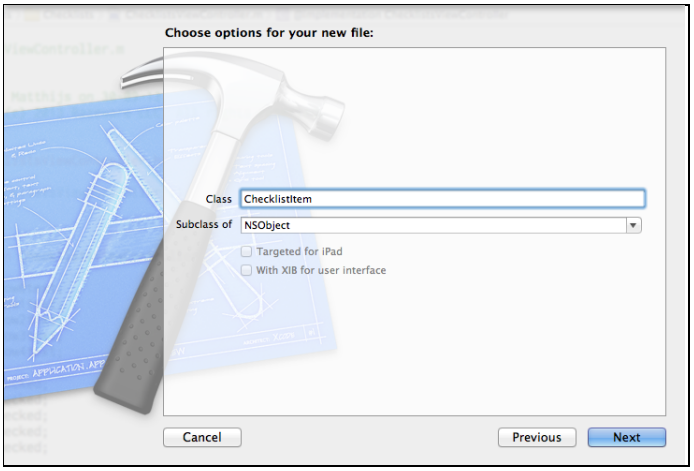
Adding a new file to the project

Under the **Cocoa Touch** section choose **Objective-C class**:



Choosing the Objective-C class template

The next screen gives you some options to fill out:

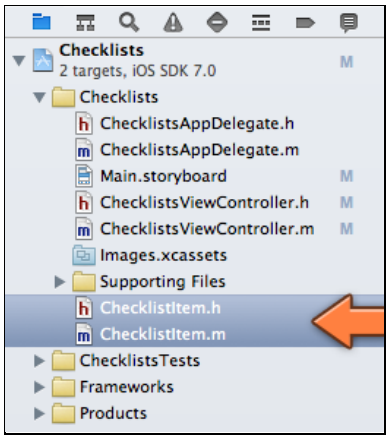


Options for the new file

Enter the following:

- Class: **ChecklistItem**
- Subclass of: **NSObject**

The screen has a few more options, but you will not use them here. Press **Next** to create the new files, **ChecklistItem.h** and **ChecklistItem.m**.



The new files are added to the project navigator

The files themselves look like the following (without the comments at the top). This is **ChecklistItem.h**:

```
#import <Foundation/Foundation.h>

@interface ChecklistItem : NSObject

@end
```

And this is **ChecklistItem.m**:

```
#import "ChecklistItem.h"

@implementation ChecklistItem

@end
```

What you see here is the absolute minimum amount of stuff you need in order to make a new object.

➤ Add the following to **ChecklistItem.h**, before the `@end` line:

```
@property (nonatomic, copy) NSString *text;
@property (nonatomic, assign) BOOL checked;
```

These are the two data items that you're adding to the object, in the form of properties. 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.

Why are you adding these data items as properties and not as instance variables? Instance variables are really supposed to be used on the insides of objects only;

they should not be visible to other objects. In this case you do want the text and checked values to be visible, so that the view controller can use them. These data items are part of the `ChecklistItem` object's so-called *public interface* – it is no coincidence that they are placed in the `@interface` section.

Unlike the properties you've used in the previous tutorial, these two do not have the `IBOutlet` symbol as they are not outlets. You only declare something as an outlet when you want to be able to make connections to it from Interface Builder. That is not the case for these properties as they are part of the data model, not the user interface of the app.

That's all for now. You don't need to make any changes to **ChecklistItem.m**. The `ChecklistItem` object currently only serves to combine the text and the checked flag into one object.

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

First, you need to tell the view controller about the `ChecklistItem` object or it won't be able to use it. To do so, you add an `#import` statement to the top of the file.

▸ Add the following to **ChecklistsViewController.m** below the other import:

```
#import "ChecklistItem.h"
```

▸ Remove the old `NSString` and `BOOL` instance variables and replace them with `ChecklistItem` objects:

```
@implementation ChecklistsViewController
{
  ChecklistItem *_row0item;
  ChecklistItem *_row1item;
  ChecklistItem *_row2item;
  ChecklistItem *_row3item;
  ChecklistItem *_row4item;
}
```

Because you just removed the `_rowXtext` and `_rowXchecked` instance variables but some methods in the view controller still refer to these variables, Xcode detects several errors. Before you can run the app again you need to fix these errors, so let's do that now.

Previously you filled in the "row text" and "row checked" variables in `viewDidLoad`. You'll do the same for the `ChecklistItem` objects.

▸ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
```

```
    [super viewDidLoad];

    _row0item = [[ChecklistItem alloc] init];
    _row0item.text = @"Walk the dog";
    _row0item.checked = NO;

    _row1item = [[ChecklistItem alloc] init];
    _row1item.text = @"Brush my teeth";
    _row1item.checked = YES;

    _row2item = [[ChecklistItem alloc] init];
    _row2item.text = @"Learn iOS development";
    _row2item.checked = YES;

    _row3item = [[ChecklistItem alloc] init];
    _row3item.text = @"Soccer practice";
    _row3item.checked = NO;

    _row4item = [[ChecklistItem alloc] init];
    _row4item.text = @"Eat ice cream";
    _row4item.checked = YES;
}
```

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

Before you can set the properties, you first create a new `ChecklistItem` object:

```
_row0item = [[ChecklistItem alloc] init];
```

You've seen something similar in the first tutorial when you created the `UIAlertView`. There you did:

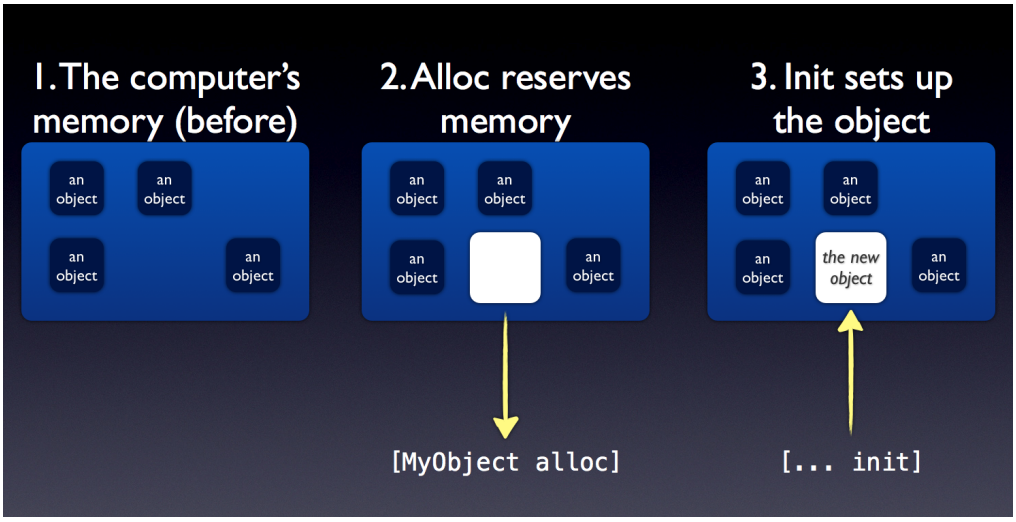
```
alertView = [[UIAlertView alloc] initWithTitle:...];
```

That is how you create objects in Objective-C, you first call `alloc` to reserve memory for this new object, followed by some form of `init` to *initialize* this object. Initialization means that you put the object in a usable state, usually by giving your instance variables and properties meaningful values.

The `ChecklistItem` object has a default initialization method that is simply named `init`, but not all objects are as concise. The full name of the method for `UIAlertView` is `initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:`. That's quite a mouthful! It's possible for an object to have more than one init method, but

you'll always only call one of them (it doesn't make sense to initialize an object twice).

You'll be seeing this pattern a lot, almost every time you make a new object. First `alloc`, then `init`. This gives you an *instance* of the object, a new copy of the object in memory. In case you haven't learned enough fancy words today, the whole process of allocation followed by initialization is also called *instantiation*.



Allocation and initialization

So in `viewDidLoad`, after creating the `ChecklistItem` object, you put values into its `text` and `checked` properties. This is repeated for the four other rows. Each row gets its own `ChecklistItem` object that you store it its own instance variable.

➤ You also need to change the other methods to use the new instance variables:

```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell
                      atIndexPath:(NSIndexPath *)indexPath
{
  BOOL isChecked = NO;

  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;
  }
```

```
  if (isChecked) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
            dequeueReusableCellWithIdentifier:@"ChecklistItem"];

  UILabel *label = (UILabel *)[cell viewWithTag:1000];

  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;
  }

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  return cell;
}

- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                              cellForRowAtIndexPath:indexPath];

  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;
  }

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Instead of using the separate `_row0text` and `_row0checked` variables, you now use `_row0item.text` and `_row0item.checked`. Likewise for the other rows.

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

The code is still unwieldy because you still need to keep around a `ChecklistItem` instance variable for each row. Time to put that array into action!

Mutable and non-mutable

There are actually two types of arrays: the *mutable* array (`NSMutableArray`) and the *non-mutable* or *immutable* array (`NSArray`). Mutable means: can be changed. An `NSArray`, which is non-mutable, cannot be changed once it is created. You cannot add new objects to it or remove objects from it, only access the objects that are already inside the array.

You see this in other places in the iOS SDK as well. `NSString` is also immutable. Once you've made an `NSString` object, you cannot change its text. You can only create a new string object that is derived from this one.

For example, `[string lowercase]` will create a new string with all the characters converted to lowercase. The original string object is still there, unmodified. If you need to create a string that you can change afterwards, you should use the `NSMutableString` object instead.

Note that even if you have a non-mutable array, you can still modify the objects that it contains. It is the array itself that cannot change – you cannot take objects out of it or put new objects into it – but once you have obtained a reference to one of its objects using `array[ objectAtIndex:]` or `array[index]` you can do with that object what you want.

Think of an immutable array as being stuck in a traffic jam. The cars in front of you and behind you don't change and no one is going anywhere, but you can certainly step out of your car and spray paint it pink.

You need to use a mutable array because this app will let the user add new items to the list and remove items as well.

➤ In **ChecklistsViewController.m**, throw away all the instance variables and replace them with a single `NSMutableArray` variable named `_items`:

```
@implementation ChecklistsViewController
{
  NSMutableArray *_items;
}
```

➤ Change the `viewDidLoad` method to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  _items = [[NSMutableArray alloc] initWithCapacity:20];

  ChecklistItem *item;

  item = [[ChecklistItem alloc] init];
  item.text = @"Walk the dog";
  item.checked = NO;
  [_items addObject:item];

  item = [[ChecklistItem alloc] init];
  item.text = @"Brush my teeth";
  item.checked = YES;
  [_items addObject:item];

  item = [[ChecklistItem alloc] init];
  item.text = @"Learn iOS development";
  item.checked = YES;
  [_items addObject:item];

  item = [[ChecklistItem alloc] init];
  item.text = @"Soccer practice";
  item.checked = NO;
  [_items addObject:item];

  item = [[ChecklistItem alloc] init];
  item.text = @"Eat ice cream";
  item.checked = YES;
  [_items addObject:item];
}
```

This is not so different from before, except that you first make the array object:

```
_items = [[NSMutableArray alloc] initWithCapacity:20];
```

Again, notice the `[[alloc] init...]` pattern to create and initialize the object. It is important to realize 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 call `alloc` and `init` to create the object and put it into that variable.

When you did this,

```
@implementation ChecklistsViewController
{
  NSMutableArray *_items;
}
```

you just said: I have a variable named `_items` that can contain an `NSMutableArray` object. But until you instantiate an actual `NSMutableArray` object and put that into `_items`, the variable is empty. Its value is "nil" in programmer-speak, although some programmers like to call this "null". You can still send messages to a `nil` variable, but they won't arrive anywhere so that's quite pointless.

That's why in `viewDidLoad`, you first make the actual `NSMutableArray` object and stuff it into `_items`. Now you can use this array object through the `_items` variable.

`NSMutableArray` has an init method named `initWithCapacity:` that reserves space for a certain number of items (20 in this case). That doesn't mean the array can only store 20 items and no more! It's just a hint that you give to the array. You expect about 20 items, but if you add more than that the array will grow to make room.

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

```
item = [[ChecklistItem alloc] init];
item.text = @"Walk the dog";
item.checked = NO;
 [_items addObject:item];
```

At the end of `viewDidLoad`, 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.

▸ Change these methods to:

```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell
                      atIndexPath:(NSIndexPath *)indexPath
{
  ChecklistItem *item = _items[indexPath.row];
```

```
  if (item.checked) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
           dequeueReusableCellWithIdentifier:@"ChecklistItem"];

  ChecklistItem *item = _items[indexPath.row];

  UILabel *label = (UILabel *)[cell viewWithTag:1000];
  label.text = item.text;

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  return cell;
}

- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                               cellForRowAtIndexPath:indexPath];

  ChecklistItem *item = _items[indexPath.row];
  item.checked = !item.checked;

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

In each method, you do:

```
ChecklistItem *item = _items[indexPath.row];
```

This asks the array for the `ChecklistItem` object at the index that corresponds with 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).

> **Note:** To get an item from an array you can also write,
>
> `ChecklistItem *item = [_items objectAtIndex:indexPath.row];`
>
> For many years the `objectAtIndex:` method was the only way to read objects from arrays in Objective-C, but recently the notation `array[index]` was added to the language to make this a bit more convenient. If you've programmed in other languages using arrays or lists, that notation should look very familiar.

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:

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
  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 `viewDidLoad` for this to work. □

Cleaning up the code

There are a few more things I want to do to clean up this code.

▸ Make these changes:

```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell
                  withChecklistItem:(ChecklistItem *)item
{
  if (item.checked) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
}
```

```
- (void)configureTextForCell:(UITableViewCell *)cell
           withChecklistItem:(ChecklistItem *)item
{
  UILabel *label = (UILabel *)[cell viewWithTag:1000];
  label.text = item.text;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
            dequeueReusableCellWithIdentifier:@"ChecklistItem"];

  ChecklistItem *item = _items[indexPath.row];

  [self configureTextForCell:cell withChecklistItem:item];
  [self configureCheckmarkForCell:cell withChecklistItem:item];

  return cell;
}

- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                                cellForRowAtIndexPath:indexPath];

  ChecklistItem *item = _items[indexPath.row];
  [item toggleChecked];

  [self configureCheckmarkForCell:cell withChecklistItem:item];

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Exercise: Spot the differences. Can you see what was changed? Can you reason why?

Answer: The `configureCheckmarkForCell:atIndexPath:` method was renamed to `configureCheckmarkForCell:withChecklistItem:`. If you think this is a long method name, you're in for a surprise. A lot of the object names and method names in the iOS SDK are huge, but at least that should give you a good idea of what they mean. Fortunately, Xcode has an auto-completion feature, so you only have to type the first few characters and it will automatically fill out the rest. Otherwise you'd be doing a lot of typing!

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

```
  ChecklistItem *item = _items[indexPath.row];
```

But in both `cellForRowAtIndexPath` and `didSelectRowAtIndexPath` you already do that as well. So it makes more sense to pass that `ChecklistItem` object directly to the "configure" method instead of making it do the same work twice. Anything that simplifies the code is good.

You also added a `configureTextForCell:withChecklistItem:` method. That sets the item's text on the cell's label. Previously you did that in `cellForRowAtIndexPath` but it's clearer to put that in its own method.

Finally, `didSelectRowAtIndexPath` 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 method to `ChecklistItem` otherwise the code won't run.

➤ Add the following to **ChecklistItem.h**, before `@end`:

```
- (void)toggleChecked;
```

➤ Add the implementation of this method to **ChecklistItem.m**:

```
- (void)toggleChecked
{
  self.checked = !self.checked;
}
```

As you can see, the method does exactly what `didSelectRowAtIndexPath` 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.

➤ Run the app, and well, it still should work exactly the same as before. :-)

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%

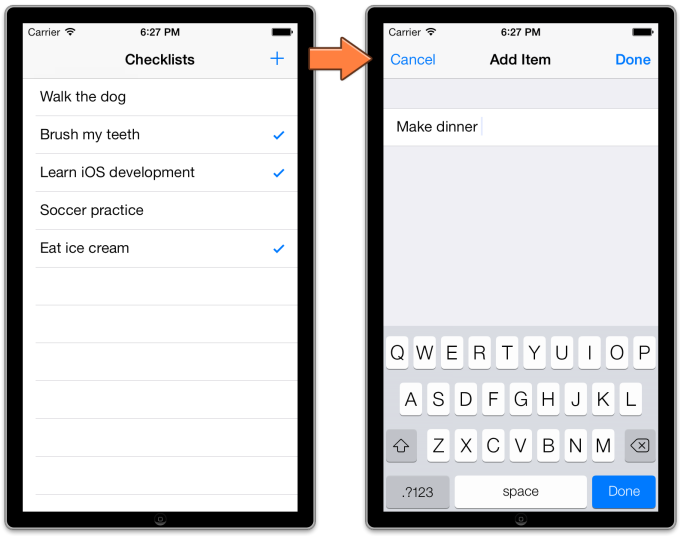
perfect right from the start. So you write code until it becomes messy and then you clean it up. Then after a little while it becomes a big mess again and you clean it up again. The process for cleaning up code is called *refactoring* and it's a cycle that never ends.

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 then 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 +) that opens 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:

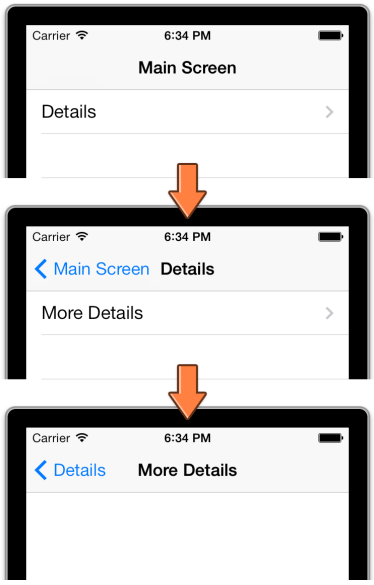
- Add a navigation controller
- Put the Add button into the navigation bar
- Add a fake item to the list when you press the Add button
- Delete items with swipe-to-delete
- Open the Add Item screen that lets the user type the text for the item

As always, we take it in small steps. After you've put the Add button on the screen, you 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 actually 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:

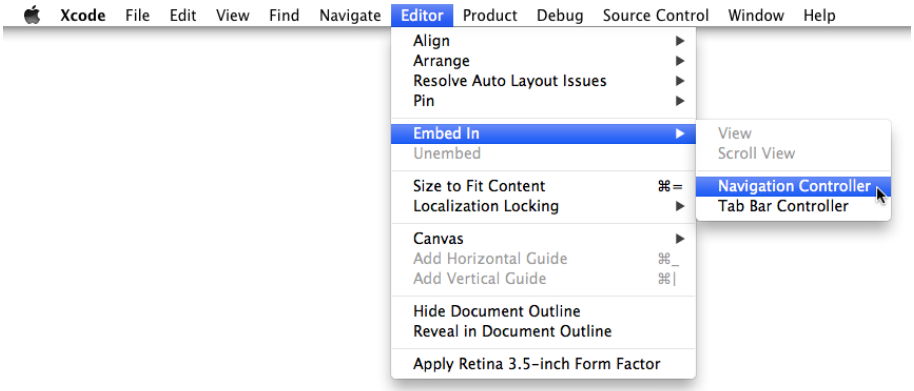


A navigation controller in action

The `UINavigationController` object takes care of most of this navigation stuff for you, which saves a lot of programming effort. You get a title in the middle of the screen 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.

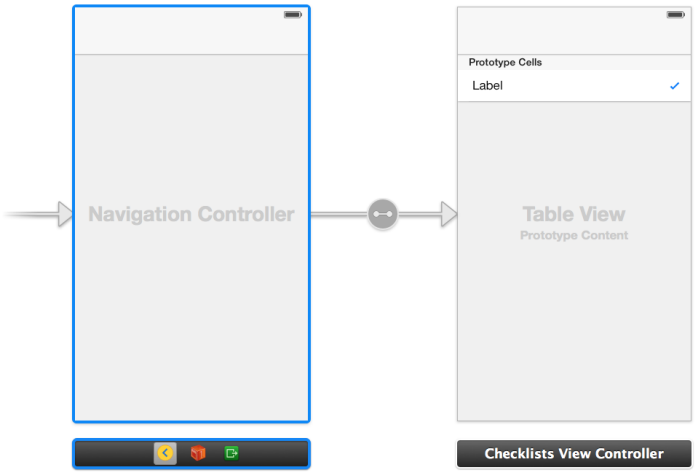
Adding a navigation controller is really easy.

› Open **Main.storyboard** and select the Checklists View Controller scene. From the menu bar at the top of the screen, choose **Editor** → **Embed In** → **Navigation Controller**.



Putting the view controller inside a navigation controller

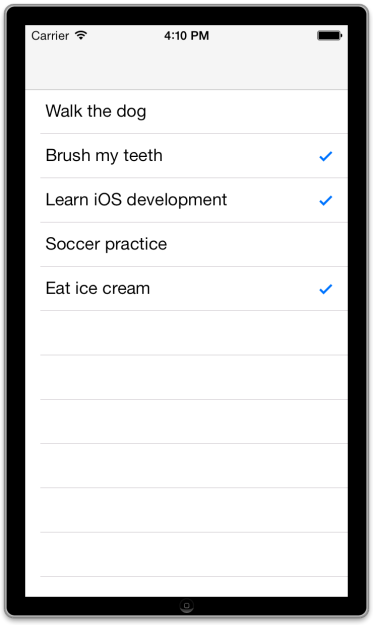
That's it. Interface Builder has now added a new Navigation Controller scene and made a relationship between it and your view controller.



The navigation controller is now linked with your view controller

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

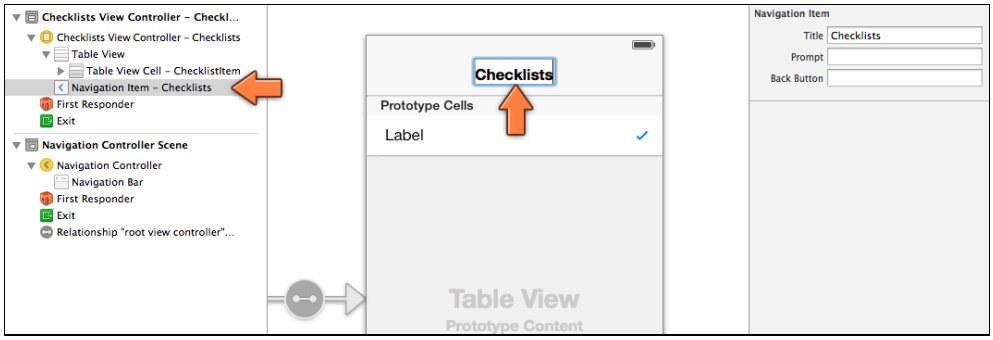
› Run the app and try it out.



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.

› Go back to the storyboard and double-click on the navigation bar inside the Checklists View Controller to make the title editable. Name it **Checklists**.

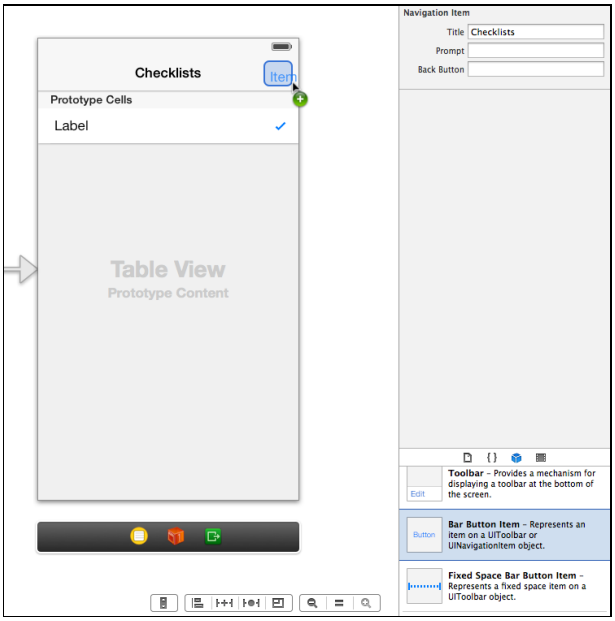


Changing the title in the navigation bar

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 will 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 is 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.

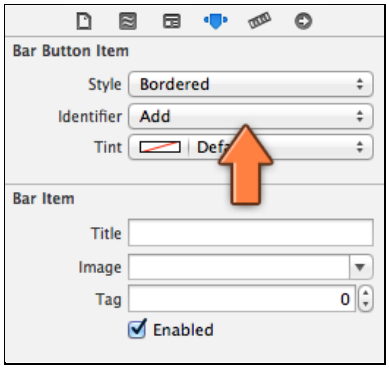
› Go to the Object Library and look for **Bar Button Item**. Drag it into the right-side slot of the navigation bar. Be sure to use the navigation bar on the Checklists View Controller, not the one from the navigation controller!



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.

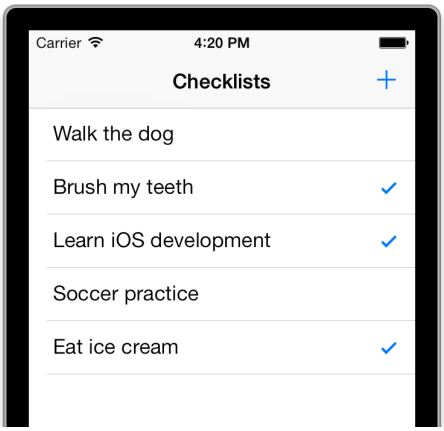
› In the **Attributes inspector** for the bar button item, choose **Identifier**: **Add**.



Bar Button Item attributes

If you look through the Identifier list you 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

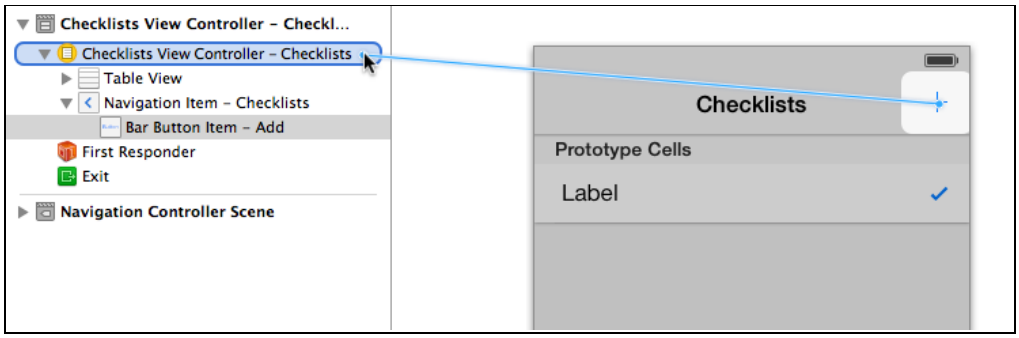
Of course, pressing the button doesn't actually do anything because you haven't hooked it up to an action yet. In a little while you will create the Add Item screen and show this screen when you tap the button. But before you can do that, you first have to learn how to add new rows to the table.

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

▸ Add a declaration for a new action method to **ChecklistsViewController.h**:

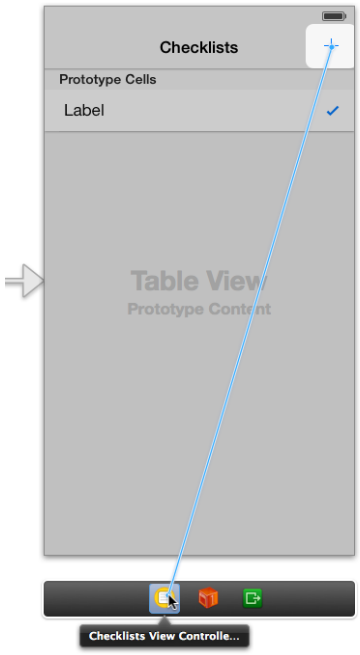
```
@interface ChecklistsViewController : UITableViewController

- (IBAction)addItem;

@end
```

▸ Open the storyboard and hook up the Add button to this action. **Ctrl-drag** from the + button to the Checklists View Controller item in the sidebar:



Ctrl-drag from Add button to View Controller

Or, even simpler, Ctrl-drag from the Add button to the view controller item in the dock area below the scene:



Ctrl-drag from Add button to 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 is a good spot).

➤ After dragging, pick **addItem** from the list (under **Sent Actions**). Now the connection is made and a tap on the + button will send the `addItem` message to the view controller.

Let's give `addItem` something to do.

➤ Add the body of this new method to the bottom of **ChecklistsViewController.m**, just before `@end`:

```
- (IBAction)addItem
{
  NSInteger newRowIndex = [_items count];

  ChecklistItem *item = [[ChecklistItem alloc] init];
  item.text = @"I am a new row";
  item.checked = NO;
  [_items addObject:item];

  NSIndexPath *indexPath = [NSIndexPath
                       indexPathForRow:newRowIndex inSection:0];
  NSArray *indexPaths = @[indexPath];
  [self.tableView insertRowsAtIndexPaths:indexPaths
              withRowAnimation:UITableViewRowAnimationAutomatic];
}
```

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

Let's take it section by section:

```
NSInteger newRowIndex = [_items count];
```

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. You add the new row to the end of the array, so the index for that new row will be 5.

In other words, when you're adding a row to the end of a table 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 variable `newRowIndex`.

The following few lines should look familiar:

```
ChecklistItem *item = [[ChecklistItem alloc] init];
item.text = @"I am a new row";
item.checked = NO;
[_items addObject:item];
```

You have seen this code before in `viewDidLoad`. 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 `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.

Here it gets tricky:

```
NSIndexPath *indexPath = [NSIndexPath
                        indexPathForRow:newRowIndex inSection:0];
```

Just adding the new `ChecklistItem` object to the data model 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. As you know by now, table views use index-paths to identify rows, so first you make an `NSIndexPath` 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:

```
NSArray *indexPaths = @[indexPath];
```

You will use the table view method `insertRowsAtIndexPaths` to tell it 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 `NSIndexPath` object, you need to give it an array

of index-paths. Not very convenient, but that's the way it is. Fortunately it is easy to create an array that contains a single index-path object using `@[indexPath]`. The notation `@[]` creates a new `NSArray` object that contains the objects between the brackets.

Finally, you tell the table view to insert this new row with a nice animation:

```
[self.tableView insertRowsAtIndexPaths:indexPaths
                      withRowAnimation:UITableViewRowAnimationAutomatic];
```

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.

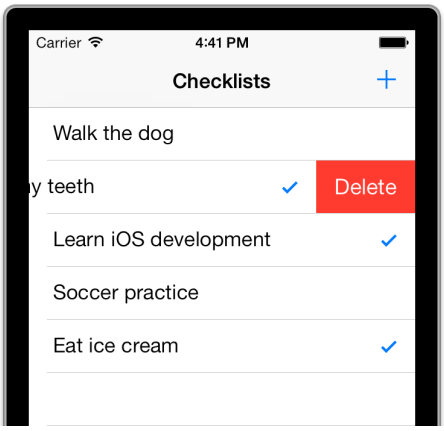
➤ Try it out. You can now add many new rows to the table. You can also tap these new rows to turn their checkmarks on and off again. When you scroll the table up and down, the checkmarks stay with the proper rows.

Note that the rows are always added to both the table and your data model. When you send the `insertRowsAtIndexPaths` message to the table, 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. You then tap the Delete button to confirm the removal, or anywhere else to cancel.



Swipe-to-delete in action

Swipe-to-delete is very easy to implement.

➤ Add the following method to the bottom of **ChecklistsViewController.m**, before `@end`:

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
  [_items removeObjectAtIndex:indexPath.row];

  NSArray *indexPaths = @[indexPath];
  [tableView deleteRowsAtIndexPaths:indexPaths
            withRowAnimation:UITableViewRowAnimationAutomatic];
}
```

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 remove the item from the data model,

```
[_items removeObjectAtIndex:indexPath.row];
```

and delete the corresponding row from the table view:

```
NSArray *indexPaths = @[indexPath];
 [tableView deleteRowsAtIndexPaths:indexPaths
            withRowAnimation:UITableViewRowAnimationAutomatic];
```

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

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

By the way, when you do `[items removeObjectAtIndex]`, that not only takes the `ChecklistItem` at that index out of the array but it also permanently destroys that `ChecklistItem` object.

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, *deallocated*.

What does it mean for an object to be destroyed? Each object occupies a small section of the computer's memory. When you call `alloc` to create an object, a

chunk of memory is reserved to hold the object's values. 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 is not valid anymore and you can no longer use it.

On versions of iOS before 5.0 you had to take care of this memory management by hand and if you made a mistake it was possible to keep using an object that already had been deleted. This so-called zombie object is no longer valid but you're still trying to access the memory that used to be reserved for it. Sometimes it even works – which is what makes these kinds of bugs so insidious – but eventually your app will crash. As of iOS 5 and its ARC technology (Automated Reference Counting) it's a lot harder to use such undead objects, but not impossible.

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 `ChecklistItem`s.

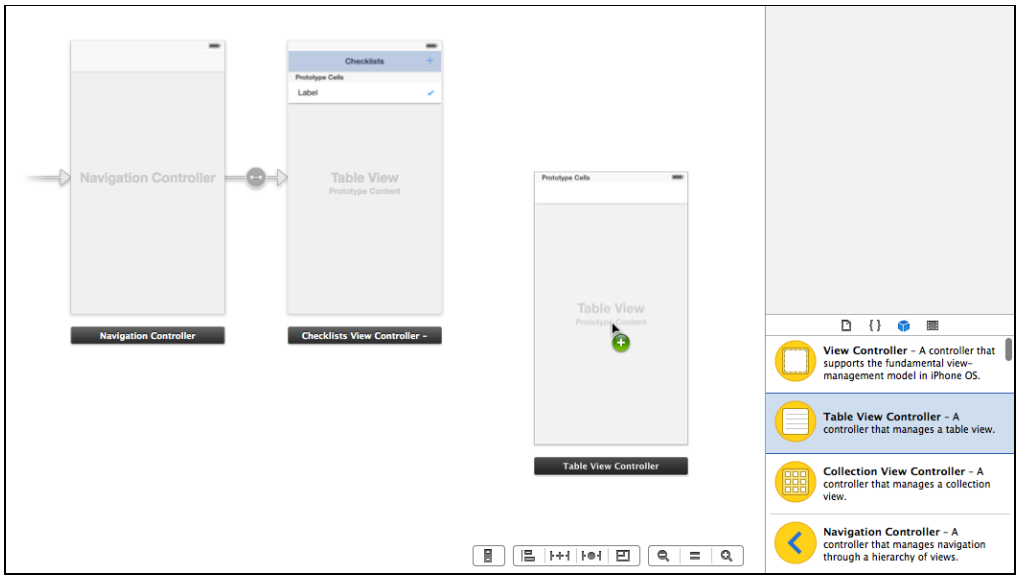
The to-do list for this section:

- 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.

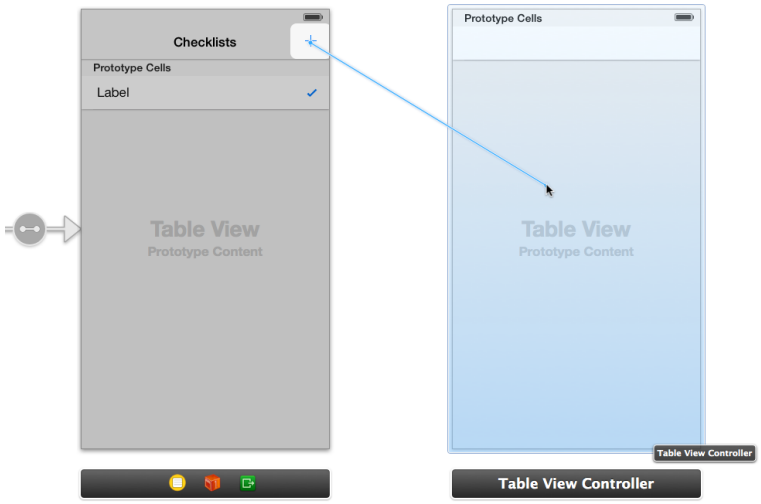
➤ Go to the Objects Library and drag a new **Table View Controller** (not a regular view controller) into the storyboard canvas.

You may need to zoom out to fit everything properly. Either use the loupe icons at the bottom of the screen or double-click the mouse on the canvas.



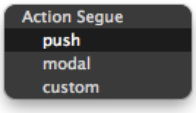
Dragging a new Table View Controller into the canvas

› With the new view controller in place, zoom back in and select the Add button from the Checklists View Controller. **Ctrl-drag** to the new view controller.



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

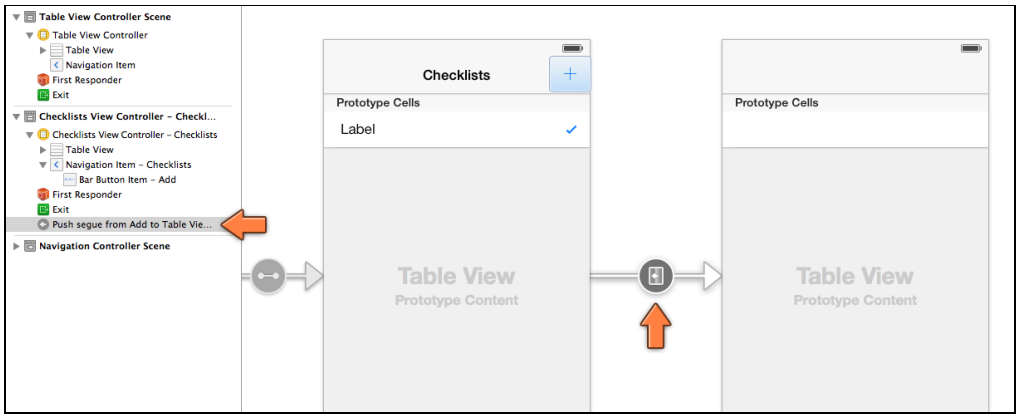
Let go of the mouse and a menu named **Action Segue** pops up:



The Action Segue popup

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

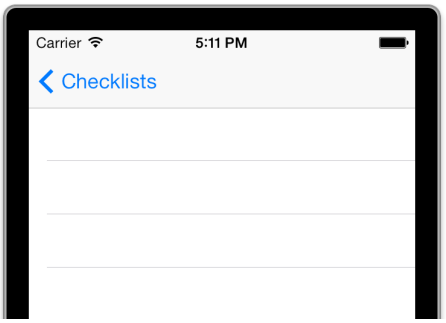
This type of connection is named a **segue** (if you're not a native English speaker, that is pronounced "seg-way" like the strange scooters that you can stand on).



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.

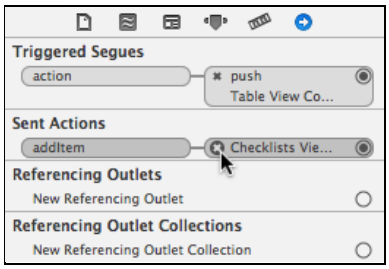


The screen that shows up after you press the Add button

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**.

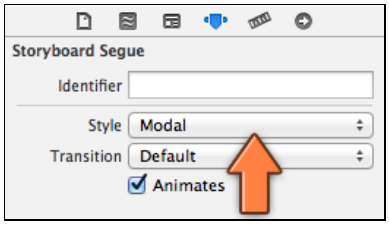


Removing the addItem action from the Add button

Notice that the 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 **Style**: **Modal**.



Changing the segue style to Modal

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.

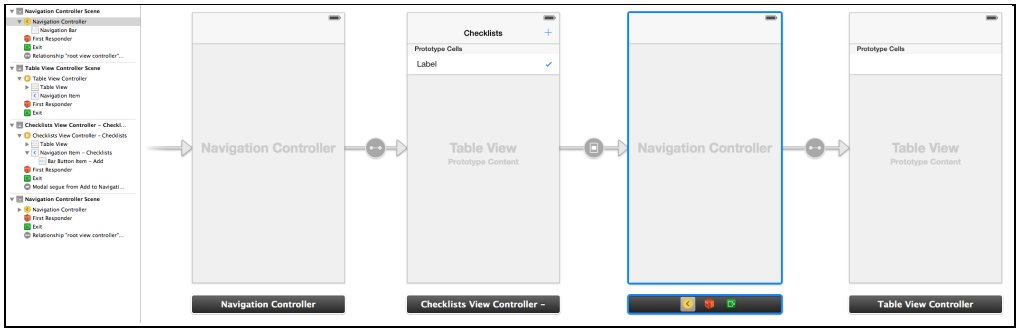
When you do, you'll notice that you no longer have a way to go back to the previous screen. Eek!

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 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**.

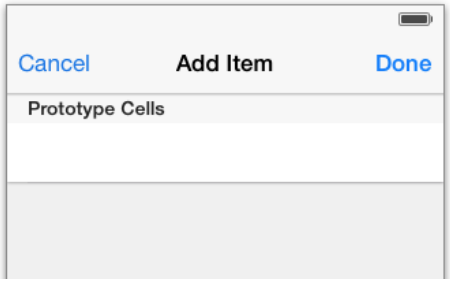
Now the storyboard looks like this:



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.

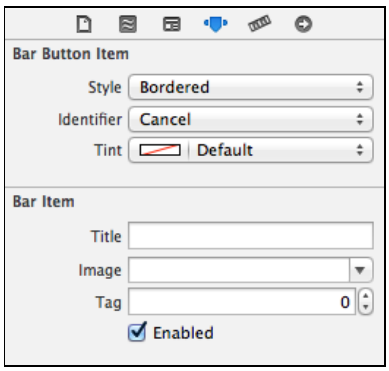
› Double-click the navigation bar in the right-most view controller to edit its title and change it to **Add Item**. Drag two **Bar Button Items** into the navigation bar, one in the left slot and one in the right slot.



The navigation bar items for the new screen

› In the **Attributes inspector** for the left button choose **Identifier**: **Cancel**. For the right button choose **Done** for both the **Identifier** and **Style** attributes.

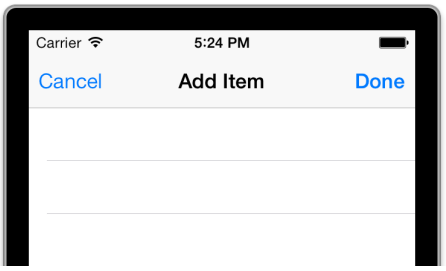
You don't have to type anything into the button's Title field. The Cancel and Done buttons are built-in button types that automatically use the proper text.



Cancel and Done are built-in button types

You could also choose **Identifier: Custom** and type the text "Cancel" or "Done" into the button. However, there is an advantage to choosing these predefined Cancel and Done buttons. If your app runs on an iPhone where the language is set to something other than English, the standard 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

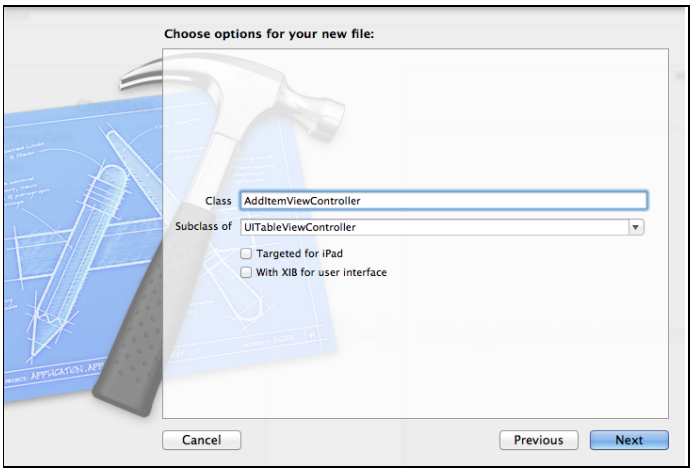
Making your own view controller object

The Cancel and Done buttons should close the Add Item screen and return the app to the main screen. 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 `ChecklistsViewController` because that is not the view controller you're dealing with here. Instead, you have to make a new view controller object 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 and choose **New File...** Choose the **Objective-C class** template. In the next step, choose the following options:

- Class: **AddItemViewController**
- Subclass of: **UITableViewController**
- Targeted for iPad: Uncheck this
- With XIB for user interface: Uncheck this



Choosing the options for the new view controller

Note: Make sure the "Subclass of" field is set to UI**Table**ViewController, not just "UIViewController"!

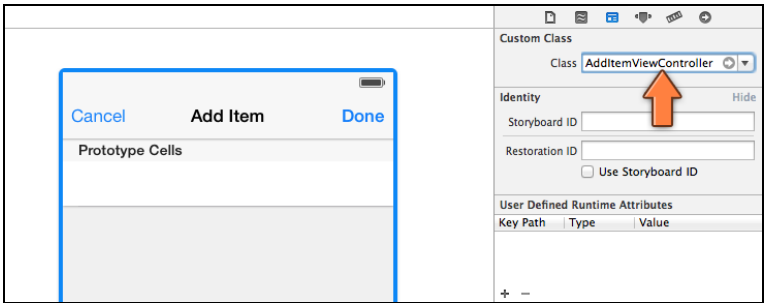
This adds two files to the project, **AddItemViewController.h** and **.m**. However, it does not change the storyboard.

› Change **AddItemViewController.h** to add the two action methods:

```
#import <UIKit/UIKit.h>

@interface AddItemViewController : UITableViewController

- (IBAction)cancel;
- (IBAction)done;

@end
```

› 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. Also 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 fat blue border). A common mistake is to select the table view and change that.

With the Class field set, you can hook up the Cancel bar button to the `cancel` action and the Done bar button to the `done` action.

➤ **Ctrl-drag** from the bar buttons to anywhere else in that view controller's scene (for example, the status bar) and pick the proper action from the popup menu.

The final step is to implement the action methods in **AddItemViewController.m**. However, the Xcode template put a lot of stuff in this file that you don't need. The template assumes you'll fill this in before you run the app again. If you try to run the app right now, Xcode will give many warnings. So let's get rid of that placeholder code first.

➤ In **AddItemViewController.m**, delete everything from the following line until `@end` (but not the `@end` line itself!):

```
#pragma mark – Table view data source
```

The lines you just deleted included placeholders for the `numberOfRowsInSection`, `cellForRowAtIndexPath` and `didSelectRowAtIndexPath` methods that you've seen before plus a few other data source and delegate methods for the table view. You won't need them for this particular view controller.

IMPORTANT! Do not skip this step. If you do not remove these methods, then the Add Item screen will not work properly.

➤ Add the new `cancel` and `done` actions at the bottom of **AddItemViewController.m**, as always before `@end`:

```
- (IBAction)cancel
{
  [self.presentingViewController
              dismissViewControllerAnimated:YES completion:nil];
}

- (IBAction)done
{
  [self.presentingViewController
              dismissViewControllerAnimated:YES completion:nil];
}
```

This tells the "presenting view controller", which is the view controller that presented this modal screen, to close the screen with an animation. If you're wondering which of the four view controllers is the presenting one, it's the

`UINavigationController` that contains the `ChecklistsViewController`, i.e. the one on the far left in the Storyboard.

➤ 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

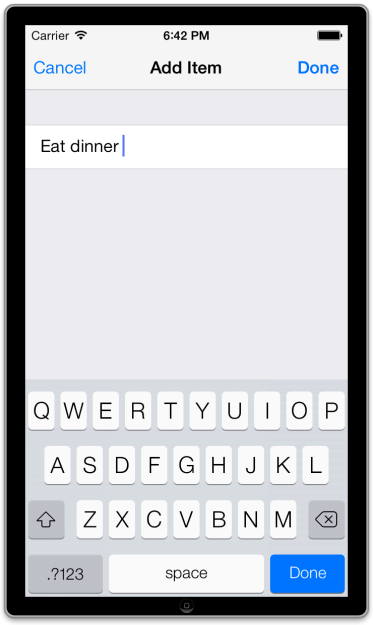
I've been saying that one view controller represents one screen, but here you actually have two view controllers for each screen. The app's main screen consists of the `ChecklistsViewController` inside a navigation controller, and the Add Item screen is composed of the `AddItemViewController` that sits inside its own 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. The container essentially "wraps around" these screens. It's just the frame that contains the view controllers that do the real work, which are known as the "content" controllers.

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 fullscreen 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.

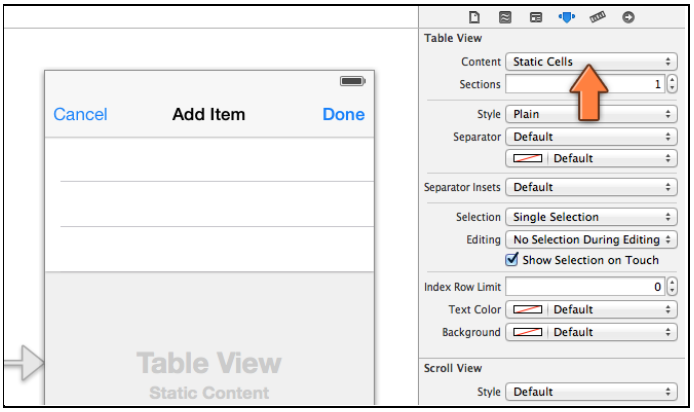
Static table cells

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**.

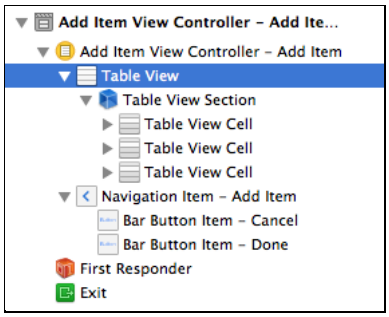


Changing the table view to static cells

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 properties on the view controller.

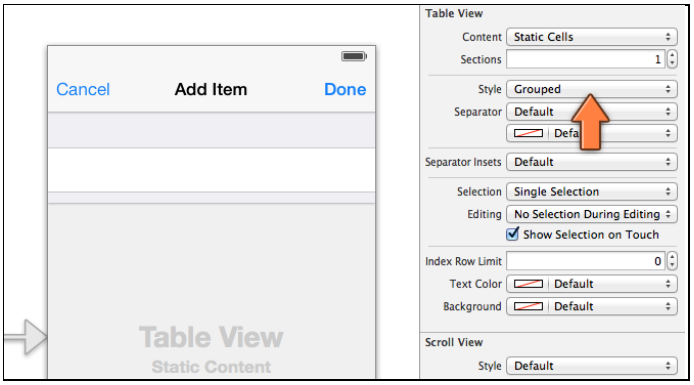
As you can see in the outline pane on the left, the table view now has a Table View Section object hanging under it, and three Table View Cells in that section.

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



The table view has a section with three static cells

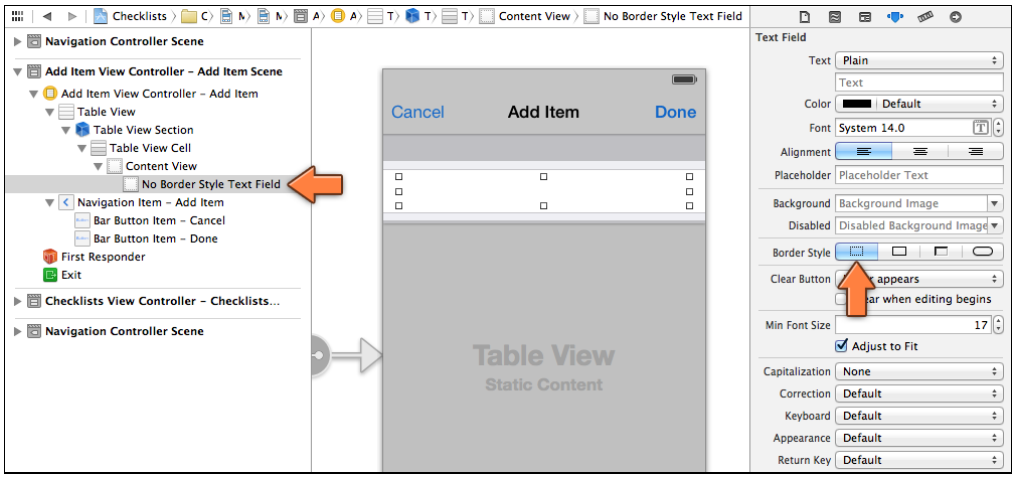
› Click the Table View again and in the **Attributes inspector** set its **Style** to **Grouped**. That gives us the look we want.



The table view with grouped style

Inside the table view cell you'll add a text field component that lets the user type text.

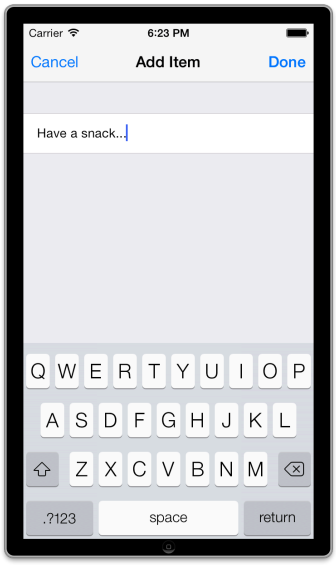
› Drag a **Text Field** object into the cell and size it up nicely. In the **Attributes inspector** for the text field, set the **Border Style** to **none**:



Adding a text field to the table view cell

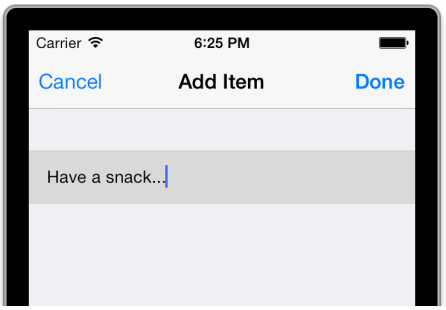
➤ Run the app and press the + button to open the Add Item screen. Tap on the cell and you'll see the keyboard slide in from the bottom of the screen.

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

But look what happens when you tap just outside the text field's area, but still in the cell:



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 the **AddItemViewController.m** file, add the following method at the bottom:

```
- (NSIndexPath *)tableView:(UITableView *)tableView
            willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  return nil;
}
```

This is another table view delegate method. When the user taps in a row, the table sends the delegate a `willSelectRowAtIndexPath` message that says: "Hi delegate, I am about to select this particular row." By returning `nil`, the delegate answers: "Sorry, but you're not allowed to!"

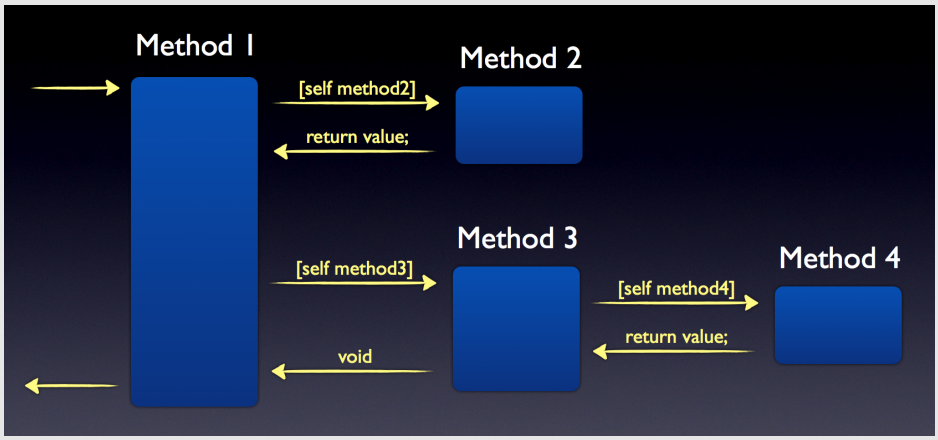
There is one more thing you need to do to prevent the row from going gray. It's already impossible to select the row, as you've just told the table view you won't allow it. However, the cell also has a Selection Color property. This is set to Blue by default (for historical reasons this is still called Blue even though the color is actually gray). Even if you make it impossible for the row to be selected, sometimes UIKit still briefly draws it 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 blue.

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 datatype that is specified in front of the method name. For example, `tableView:numberOfRowsInSection:` must return an `NSInteger` value as its name begins with `(NSInteger)`:

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
    return 1;
}
```

`NSInteger` is another name for `int`, so the statement "return 1" is perfectly valid because 1 is an integer number.

If instead you were to write,

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
    return @"1";
}
```

then the compiler would give an error message as `@"1"` is a string, not an `NSInteger`. To a human reader they look similar and you'd easily understand the intent, but Objective-C isn't that tolerant. Datatypes have to match or it just isn't allowed.

Your most recent version of this method looks like this:

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
    return [_items count];
}
```

That is also a valid return statement because the `count` method from `NSArray` returns an `NSUInteger` value. The differences between `NSInteger`, `NSUInteger` and `int` are not very interesting. The important thing is that Objective-C can easily convert between them as they all represent whole numbers (integers).

The `tableView:cellForRowAtIndexPath:` method is supposed to return a `UITableViewCell` object:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
       dequeueReusableCellWithIdentifier:@"ChecklistItem"];

  . . .

  return cell;
}
```

The difference with the previous examples is that `int` and `NSInteger` are not objects but `UITableViewCell` is. You can tell the difference by the `*`. A name followed by an asterisk means that you're dealing with an object. The others are so-called *primitive types*. The next tutorial explains more about the differences between objects and primitive types. For now, just keep an eye out for the `*`.

The `tableView:willSelectRowAtIndexPath:` method is supposed to return an `NSIndexPath` object. However, you can also make it return "nil", which means no object.

```
- (NSIndexPath *)tableView:(UITableView *)tableView
            willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    return nil;
}
```

When a method is expected to return an object, you can return `nil` as well. That doesn't mean `nil` is always a proper response. For example, if you were to return `nil` from `cellForRowAtIndexPath`, the app would crash. Some methods don't like it when you return `nil` instead of an actual object, but for others it is OK. How do you know which is the case? You can find that in the documentation of the method in question.

In the case of `willSelectRowAtIndexPath`, the iOS documentation says:

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

The documentation for `cellForRowAtIndexPath` says:

"Return Value: An object inheriting from UITableViewCell that the table view can use for the specified row. An assertion is raised if you return nil."

In other words, this method must always return a proper table view cell object.

"An assertion is raised if you return nil" means that returning `nil` instead of a valid `UITableViewCell` object will crash the app on purpose because you're doing something you're not supposed to. An *assertion* is a special debugging tool that is used to check that your code always does something valid. If not, the app will crash with a helpful error message. You'll see more of this later when we talk about finding bugs – and squashing them.

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

```
- (void)viewDidLoad
```

and:

```
- (IBAction)cancel
```

The term "void" means: this method does not pass a value back to the caller. `IBAction` is a synonym for `void` but as you know it's also a special symbol that lets Interface Builder know that this method can be hooked up to a button or other control. Because `IBAction` means the same as `void`, action methods never return a value.

If you forget to return a value from a method that expects to return something, then Xcode will give you the following warning: "Control reaches end of non-void function". The app will still run, but there's a big chance it will crash at some point, as whoever calls that method is expecting a to receive a value but the method is not returning any. As always, pay attention to the warnings of Xcode!

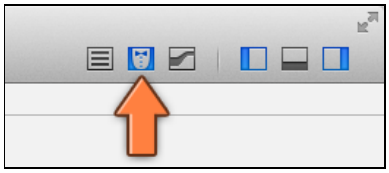
Reading the text from the text field

You now 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 a property.

When you added outlets in the previous tutorial, I told you to type in the `@property` declaration 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 into your source code file.

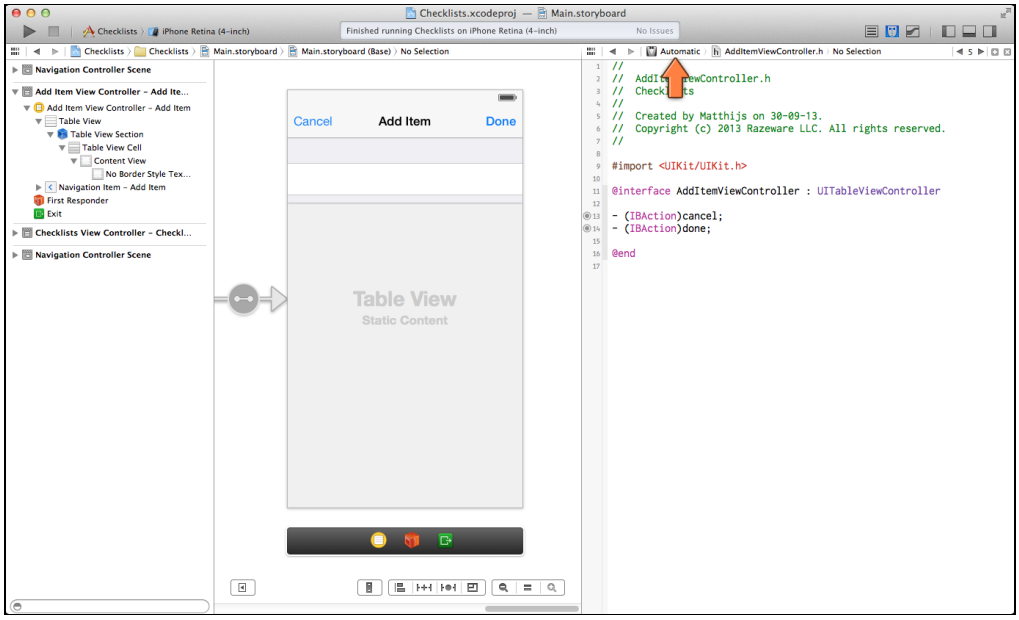
› First, go to the storyboard and then open the **Assistant editor**, using the toolbar button:



Click the toolbar button to open the Assistant editor

This may make the screen a little crowded – there are now five horizontal panels open –so 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**:



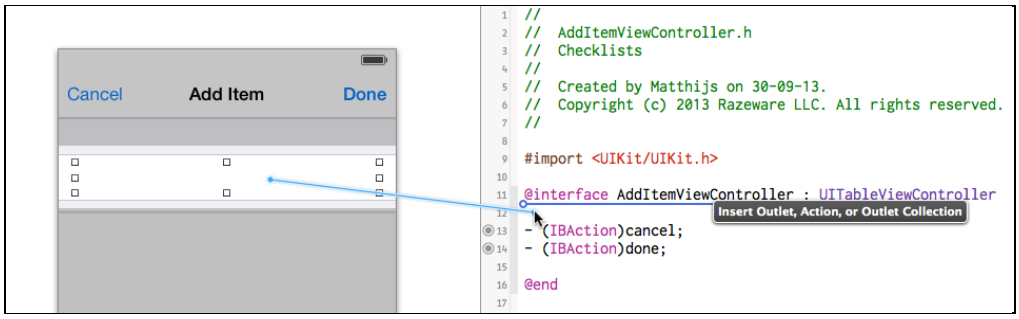
The Assistant editor

The Assistant editor should have automatically chosen the **AddItemViewController.h** file for you. "Automatic" means the Assistant editor

figures out what other file is related to the one you're currently editing. If you're editing the Storyboard, the related file is the selected view controller's .h file.

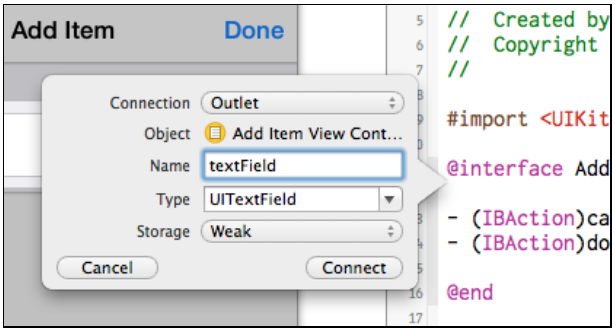
(Sometimes Xcode can be a little dodgy here. If it shows you something other than AddItemViewController.h, then click in the Jump Bar to select that file.)

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



Ctrl-dragging from the text field into the .h 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

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

The **AddItemViewController.h** file now looks like this:

```
#import <UIKit/UIKit.h>

@interface AddItemViewController : UITableViewController
```

```
@property (weak, nonatomic) IBOutlet UITextField *textField;

- (IBAction)cancel;
- (IBAction)done;

@end
```

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 `NSLog()` messages show up). This is a quick way to verify that you can actually read what the user typed.

➤ In **AddItemViewController.m,** change the `done` action to:

```
- (IBAction)done
{
  NSLog(@"Contents of the text field: %@", self.textField.text);

  [self.presentingViewController
             dismissViewControllerAnimated:YES completion:nil];
}
```

➤ 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 open the Debug pane with a message like this:

```
Checklists[1165:207] Contents of the text field: Hello, world!
```

Great, so that works. `NSLog()` should be an old friend by now. You've seen the `%d` and `%f` format specifiers before, which were used for integer values and floating-point values (decimals), respectively. The `%@` specifier is used to print out the value of an object, in this case the contents of the text field's `text` property.

Polishing it up

Before you will write the code to take this 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, `viewWillAppear:`, to **AddItemViewController.m**:

```
- (void)viewWillAppear:(BOOL)animated
```

```
{
  [super viewWillAppear:animated];

  [self.textField becomeFirstResponder];
}
```

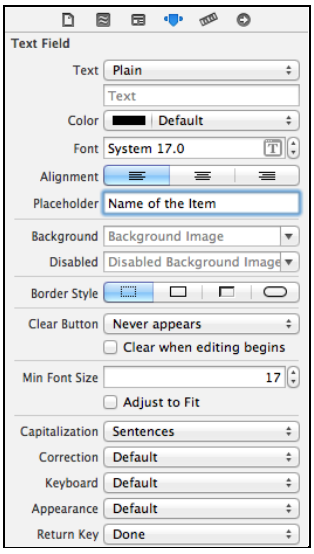
The view controller receives the `viewWillAppear` message just before it becomes visible. That is a perfect time to make the text field active. You do this by sending it the `becomeFirstResponder` message. If you've done programming on other platforms, this is often called "giving the control focus". 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.

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

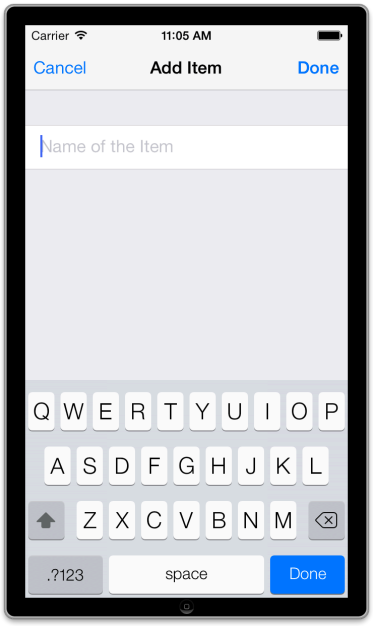
There are several options here that let you configure the keyboard that appears when the text field becomes active. If this were a field that only allowed numbers, for example, you would set the Keyboard to Number Pad. If it were an email

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.

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



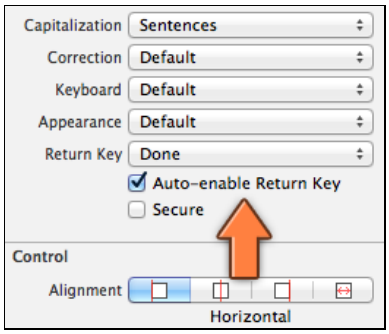
The keyboard now has a big blue Done button

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, 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.

You've seen delegates before with `UITableView`. The text field object, `UITextField`, also has a delegate. These are two different delegates and you make the view controller play both roles. Later in this tutorial you'll add even more delegates.

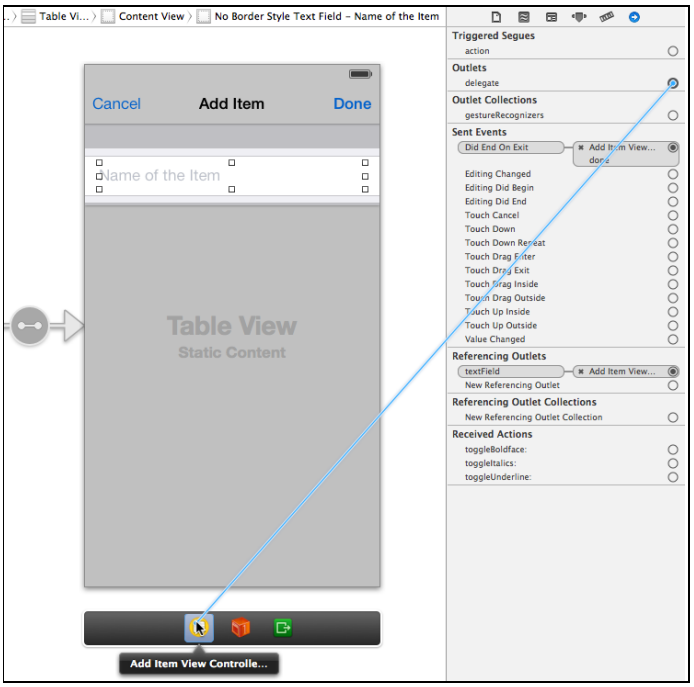
▸ Add `<UITextFieldDelegate>` to **AddItemViewController.h**'s `@interface` declaration (this goes all on one line):

```
@interface 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 icon in the dock:



Drag from the Connections inspector to connect the text field delegate

You also have to add a property 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.h** is visible in the newly opened pane. **Ctrl-drag** from the Done bar button to AddItemViewController.h and let go. Name the new outlet `doneBarButton`.

This adds the following property:

```
@property (weak, nonatomic) IBOutlet UIBarButtonItem
                                          *doneBarButton;
```

› Add the following to **AddItemViewController.m**, at the bottom:

```
- (BOOL)textField:(UITextField *)theTextField
        shouldChangeCharactersInRange:(NSRange)range
        replacementString:(NSString *)string
{
  NSString *newText = [theTextField.text
    stringByReplacingCharactersInRange:range withString:string];

  if ([newText length] > 0) {
    self.doneBarButton.enabled = YES;
  } else {
    self.doneBarButton.enabled = NO;
  }
```

```
  return YES;
}
```

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

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

```
NSString *newText = [theTextField.text
    stringByReplacingCharactersInRange:range withString:string];
```

The `shouldChangeCharactersInRange` 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). So 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` local variable.

Then you check if the new text is empty by looking at its length, and enable or disable the Done button accordingly:

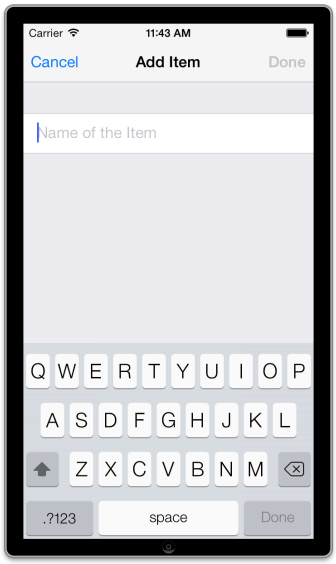
```
if ([newText length] > 0) {
  self.doneBarButton.enabled = YES;
} else {
  self.doneBarButton.enabled = NO;
}
```

➤ 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:

```
- (BOOL)textField:(UITextField *)theTextField
        shouldChangeCharactersInRange:(NSRange)range
        replacementString:(NSString *)string
{
  NSString *newText = [theTextField.text
    stringByReplacingCharactersInRange:range withString:string];

  self.doneBarButton.enabled = ([newText length] > 0);

  return YES;
}
```

You replaced the if-statement by the line:

```
self.doneBarButton.enabled = ([newText length] > 0);
```

Earlier you did:

```
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
}
```

In both cases 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 `enabled` property to `YES`. If the condition is false, you set the `enabled` property to `NO`.

Notice that these sentences are basically saying: if the condition is `YES` then `enabled` becomes `YES` but if the condition is `NO` then `enabled` becomes `NO`. In other words, you always set the `enabled` property to the result of the condition: `YES` or `NO`.

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

```
self.doneBarButton.enabled = the result of the condition;
```

which in Objective-C reads as follows:

```
self.doneBarButton.enabled = ([newText length] > 0);
```

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

```
self.doneBarButton.enabled = [newText length] > 0;
```

However, I find this to be 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, `self.doneBarButton.enabled` becomes `YES`; otherwise it becomes `NO`.

You can fit this into a single statement because the *relational operators* all return `YES` if the condition is true and `NO` if the condition is false. These are the relational operators in Objective-C:

- `>` 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 = YES;
} else {
  something = NO;
}
```

then you can write it simply as:

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

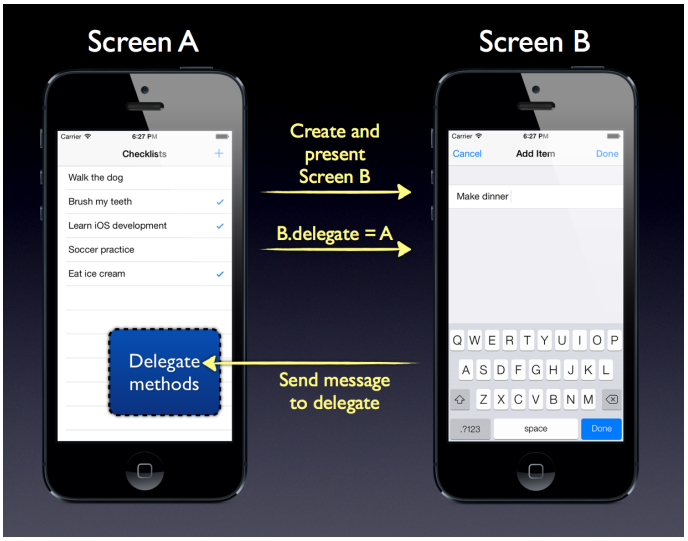
In practice it doesn't really matter which version you use. I prefer the shorter one; that's what the pros do. Just remember that relational operators such as == and > always return `YES` or `NO`, so the extra `if` really isn't necessary.

Adding new ChecklistItems

You now have an Add Item screen 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? To do this you will have to make your own delegate.

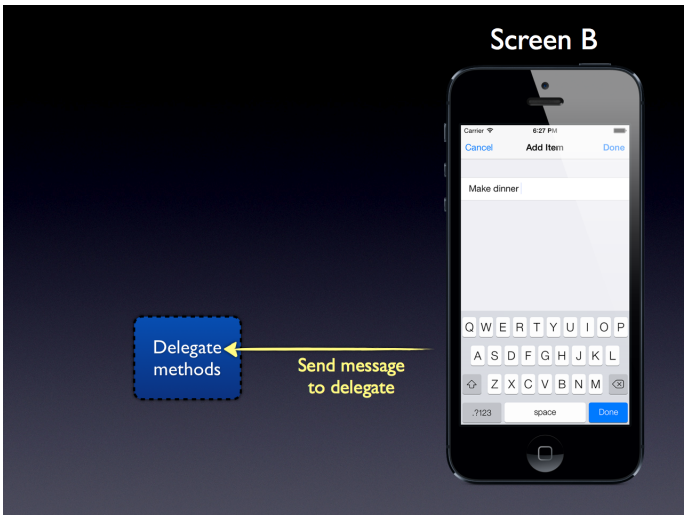
You've already seen delegates in a few different places: The table view has a delegate that responds to taps on the rows. The text field has a delegate that you use to validate the length of the text. In the Bull's Eye tutorial you used a delegate to listen to the alert view. And the app also has something named the `ChecklistsAppDelegate` (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 and at some point screen B needs to communicate back to screen A, for example when it closes. The solution is to make A a delegate of B so screen 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 object-oriented 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 to the `ChecklistsViewController`, without it having to know anything about this object.

› At the top of **AddItemViewController.h**, add this in between the `#import` and `@interface` lines:

```
@class AddItemViewController;
@class ChecklistItem;

@protocol AddItemViewControllerDelegate <NSObject>

- (void)addItemViewControllerDidCancel:
                            (AddItemViewController *)controller;

- (void)addItemViewController:
                            (AddItemViewController *)controller
        didFinishAddingItem:(ChecklistItem *)item;

@end
```

This defines the `AddItemViewControllerDelegate` protocol. You should recognize the lines inside the `@protocol ... @end` block as method declarations.

Protocols

In Objective-C, 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 have instance variables and it 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 methods listed in the `AddItemViewControllerDelegate` protocol are `addItemViewControllerDidCancel:` and `addItemViewController:didFinishAddingItem:`. Delegates often have very long method names!

If you make the `ChecklistsViewController` conform to this protocol, then it must implement these two methods. The trick is that from then on you can refer to the `ChecklistsViewController` using the protocol name. That is done using the following syntax:

```
id <AddItemViewControllerDelegate> delegate;
```

The variable `delegate` is now a reference to some object that implements the methods of this protocol. You can send messages to the object in the `delegate` variable, without knowing what kind of object it really is. Of course, you know it is the `ChecklistsViewController` but `AddItemViewController` doesn't need to be aware of that. All it sees is `delegate`.

It is customary for the delegate methods to have a reference to their owner as the first (or only) parameter. That's why you pass along the `AddItemViewController` object to both methods. This is not required but still a good idea. For example, it may happen that a table view delegate is the delegate for more than one table view. In that case, it needs to be able to distinguish between those two table views. That's why the table view delegate methods contain a reference to the `UITableView` object that sent the notification.

If you've programmed in other languages before, you may recognize protocols as being very similar to "interfaces". When the designers of the Objective-C language added protocols they couldn't use the term interface to describe protocols as this keyword was already used to declare objects (the `@interface` line in the .h files). Therefore `@protocol` it is. It's only a name; the concept is the same.

Notice the line that says `@class ChecklistItem;`. This tells the delegate protocol about the `ChecklistItem` object. In the past you've used `#import` to let one object know about other objects. So what's the difference between these two?

- The line `@class ChecklistItem` simply says to the compiler: if you see the name `ChecklistItem` then that's an object we're going to be using. It doesn't tell the compiler exactly what that object does, just that it's an object. You'll use `@class` mostly in .h files because at that point the compiler doesn't need to know what the object's properties and methods are, just that it exists. This is also known as a *forward declaration*.

- `#import "ChecklistItem.h"` literally adds the contents of the **ChecklistItem.h** file into the current file when it is being compiled. More about this in the next tutorial, but it means that when you use `#import`, the compiler knows everything about that object. If you want to access the properties of an object or call any of its methods, you need to use `#import`.

You need this forward declaration for the `ChecklistItem` object because you are using that object in the `AddItemViewControllerDelegate` protocol, as a parameter for the method `addItemViewController:didFinishAddingItem:`. Likewise for the forward declaration of `@class AddItemViewController`.

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

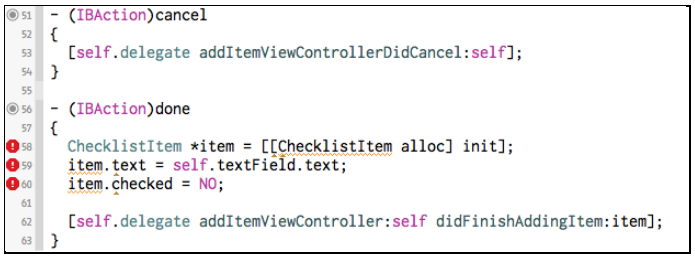
➤ Add this inside the `@interface` block, below the other properties:

```
@property (nonatomic, weak) id <AddItemViewControllerDelegate>
                                                    delegate;
```

➤ In **AddItemViewController.m**, replace the `cancel` and `done` actions with the following:

```
- (IBAction)cancel
{
  [self.delegate addItemViewControllerDidCancel:self];
}

- (IBAction)done
{
  ChecklistItem *item = [[ChecklistItem alloc] init];
  item.text = self.textField.text;
  item.checked = NO;

  [self.delegate addItemViewController:self
                                    didFinishAddingItem:item];
}
```

As soon as you add this code, Xcode shows errors in the gutter:



Xcode does not know what ChecklistItem means

This happens because inside **AddItemViewController.m** the compiler doesn't know yet what a `ChecklistItem` object is. You still need to import its definition.

➤ Add the following line at the top of the file:

```
#import "ChecklistItem.h"
```

Now the error messages will disappear.

Let's look at the changes you made to the `cancel` and `done` action methods. When the user taps the Cancel button, you send the `addItemViewControllerDidCancel` message to the delegate. You do something similar for the Done button, except that the message is `addItemViewController:didFinishAddingItem:` and you pass along a new `ChecklistItem` object.

If you were to run the app now, the Cancel and Done buttons would no longer appear to work. (Try it out!) That is because you haven't told the Add Item screen yet who its delegate is. That means the `self.delegate` property is `nil` and the messages aren't being sent to anyone; there is no one listening for them.

> **Tip:** If you've programmed in other languages before, you may be afraid of the dreaded "null pointer dereference". In Objective-C it is perfectly valid to send a message to `nil`. This will not crash your app. You don't have to add an `if (self.delegate != nil)` check before sending a message to the delegate. The only thing you don't want to send messages to are deallocated objects (so-called zombies), but everyone knows you don't mess with zombies.

First you need to make the `ChecklistsViewController` suitable to be a delegate for `AddItemViewController`.

➤ Change **ChecklistsViewController.h** to:

```
#import <UIKit/UIKit.h>
#import "AddItemViewController.h"

@interface ChecklistsViewController : UITableViewController
                                    <AddItemViewControllerDelegate>

- (IBAction)addItem;

@end
```

The `#import` line for AddItemViewController.h is necessary to load the definition of the `AddItemViewControllerDelegate`, otherwise you cannot use it. The change to the `@interface` line tells the compiler that `ChecklistsViewController` now conforms to the `AddItemViewControllerDelegate` protocol. Anytime you see something in

between < > brackets on an `@interface` line, that means the object implements a particular protocol.

➤ Add the implementations of the protocol's methods at the bottom of **ChecklistsViewController.m**:

```
- (void)addItemViewControllerDidCancel:
                              (AddItemViewController *)controller
{
  [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)addItemViewController:
                              (AddItemViewController *)controller
        didFinishAddingItem:(ChecklistItem *)item
{
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

Currently these methods simply close the Add Item screen. Once you have this working you'll add the code to make the new `ChecklistItem` object and add it to the table view. Note that this is what the `AddItemViewController` used to do itself in its `cancel` and `done` actions. You've simply moved that responsibility to the delegate now.

In review, 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 out the messages:

1. Define a delegate `@protocol` for object B.
2. Give object B a property for that delegate protocol.
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.
4. Make object A conform to the delegate protocol. It should put the name of the protocol in its `@interface` line and implement the methods from the protocol.
5. Tell object B that object A is now its delegate.

This means there is one more thing you need to do: tell `AddItemViewController` that the `ChecklistsViewController` is now its delegate. The proper place to do that when you're using storyboards is in the `prepareForSegue:sender:` method.

➤ Add this method to **ChecklistsViewController.m**:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
```

```
  if ([segue.identifier isEqualToString:@"AddItem"]) {

    // 1
    UINavigationController *navigationController =
                                segue.destinationViewController;

    // 2
    AddItemViewController *controller =
            (AddItemViewController *)
                        navigationController.topViewController;

    // 3
    controller.delegate = self;
  }
}
```

The `prepareForSegue` 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. `prepareForSegue` allows you to give data to the new view controller before it will be displayed. Usually you'll do that by setting its properties.

This is what `prepareForSegue` does, step-by-step:

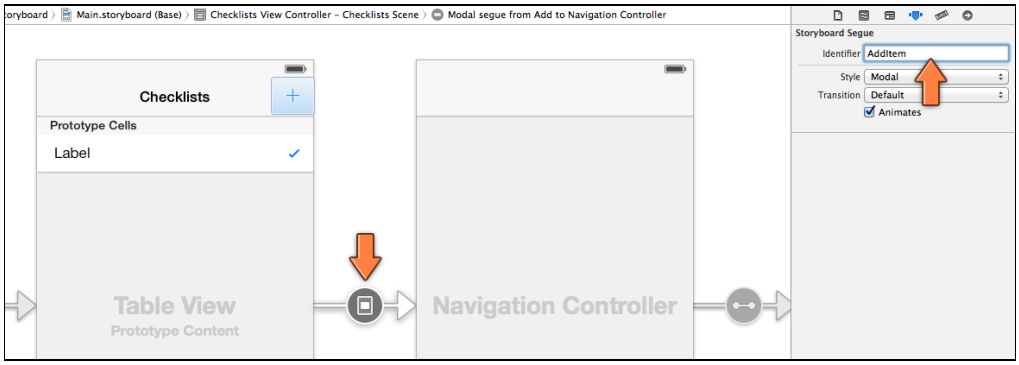
1. The new view controller can be found in `segue.destinationViewController`. In this app, the destination is not `AddItemViewController` but the navigation controller that embeds it.
2. To get the `AddItemViewController` object, you can look at the navigation controller's `topViewController` property. This property refers to the screen that is currently active inside the navigation controller.
3. Once you have a reference to the `AddItemViewController` object, you set its `delegate` property to `self` and the connection is complete. `ChecklistsViewController` is now the delegate of `AddItemViewController`. It took some work, but you're all set now.

You may wonder why you did this:

```
if ([segue.identifier isEqualToString:@"AddItem"]) {
    . . .
}
```

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.

➤ Open the storyboard and select the segue between the Checklists 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.

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, `ChecklistsViewController`. Currently the delegate simply closes the Add Item screen, but you'll soon make it do more.

Equal or not equal

You may be wondering why you did not use the == operator to check if the segue identifier was equal to the text "AddItem", in the following manner:

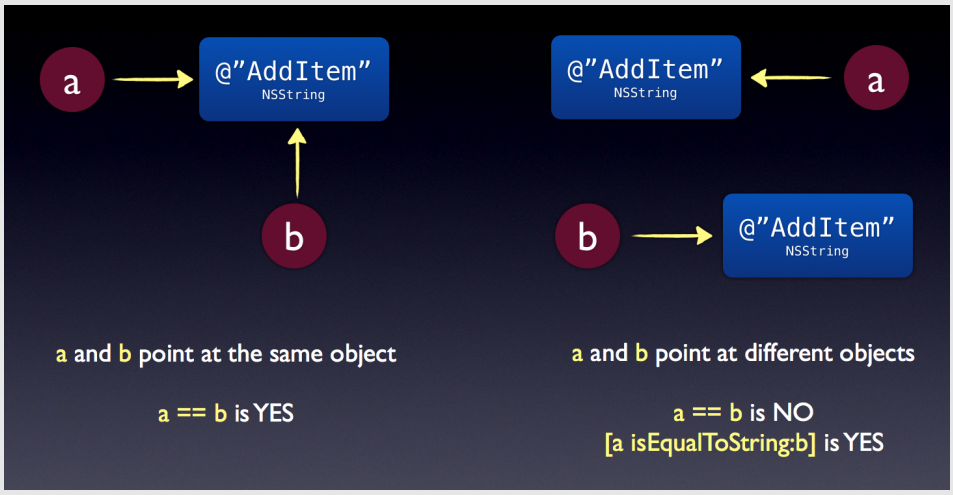
```
if (segue.identifier == @"AddItem") {
    . . .
}
```

This gets into the topic of *object identity*, or what does it mean for two objects to be equal.

If you use ==, you're checking whether two variables refer to the exact same object. That would be wrong in this case, as the literal text `@"AddItem"` and the value from `segue.identifier` are most likely two separate string objects.

However, both of these string objects may have the same value – they both may contain the text `@"AddItem"`. To compare the values of these objects but not the objects themselves, you use the `isEqual` method, or more specifically for strings, `isEqualToString`.

The difference between using == and `isEqualToString` to compare two strings:



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 with `joe1 == joe2` then the result would be `NO`, as they're not the same person. But `[joe1.name isEqualToString:joe2.name]` would be `YES`.

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 `YES`.

You can now add the new `ChecklistItem` to the data model and table view. Finally!

➤ Change the implementation of the `didFinishAddingItem` delegate method in **ChecklistsViewController.m** to the following:

```
- (void)addItemViewController:
                              (AddItemViewController *)controller
        didFinishAddingItem:(ChecklistItem *)item
{
  NSInteger newRowIndex = [_items count];
  [_items addObject:item];

  NSIndexPath *indexPath = [NSIndexPath
                       indexPathForRow:newRowIndex inSection:0];
  NSArray *indexPaths = @[indexPath];
  [self.tableView insertRowsAtIndexPaths:indexPaths
            withRowAnimation:UITableViewRowAnimationAutomatic];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

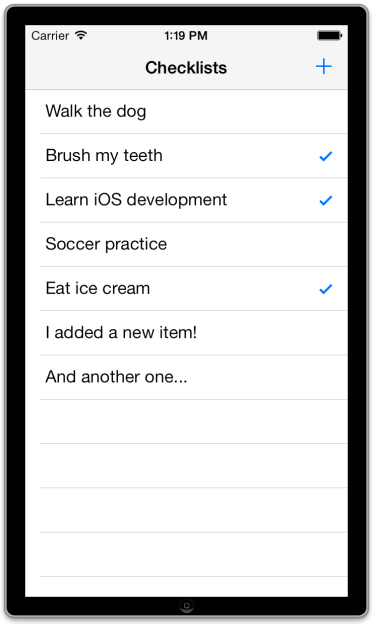
This is largely the same as what you did in `addItem` before. In fact, I simply copied the contents of the `addItem` action and pasted them into this delegate method. The

only difference is that you no longer create the `ChecklistItem` object yourself; this now 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 the `addItem` action from the **ChecklistsViewController.h** and **.m** files 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, under Sent Actions. Nothing should be connected there.)

➤ 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.

Delegates seem like a lot of work!

Why go through all this effort making a delegate protocol when you simply could have given `AddItemViewController` a property that points directly to `ChecklistsViewController`? That would have looked something like this:

```
// in the .h file
@interface AddItemViewController

@property (nonatomic, weak) ChecklistsViewController
                                    *checklistsViewController;
```

```
@end

// in the .m file
#import "ChecklistsViewController.h"

@implementation AddItemViewController

- (IBAction)done
{
  // directly call a method from ChecklistsViewController
  [self.checklistsViewController
                          addItemWithText:self.textField.text];
}

@end
```

It's true that this saves some typing, but it also shackles these two objects together. As a general design 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).

When you give `AddItemViewController` a direct reference to `ChecklistsViewController`, then you can never open the Add Item screen from somewhere else in the app. You won't actually do this in the Checklists app, but it's not uncommon for one screen to be accessible from multiple places. Using a delegate helps to abstract the dependency from screen B on screen A.

In addition, the delegate protocol defines a contract between screen B and any screens that wish to use it. Screen A doesn't have to expose any methods that it might prefer to keep hidden from other objects. By making it a delegate of screen B, screen A doesn't have to put any methods in its public `@interface` beyond those from the delegate protocol.

Anytime you want one part of your app to notify another part about something, usually in order to update the screen, you want to use delegates. It's the iOS way. (Actually, there is another, less well-known way: using so-called *unwind* segues. You will see those in the next tutorial.)

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 later 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 instead let's re-use the Add Item screen and make it capable of editing an existing `ChecklistItem` object. When the user presses Done you will update the text in that object and tell the delegate about it so that it can update the label of the corresponding table view cell.

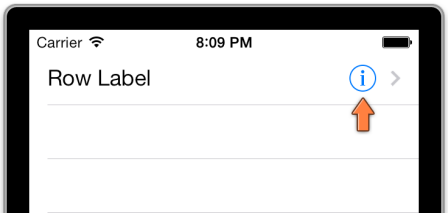
Exercise: Which changes would you need to make to the Add Item screen to enable it to edit existing items? □

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 item's text into the text field.
4. When the user presses Done, you should not add a new item 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. 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 you'll make it so that 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 left-most section 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.

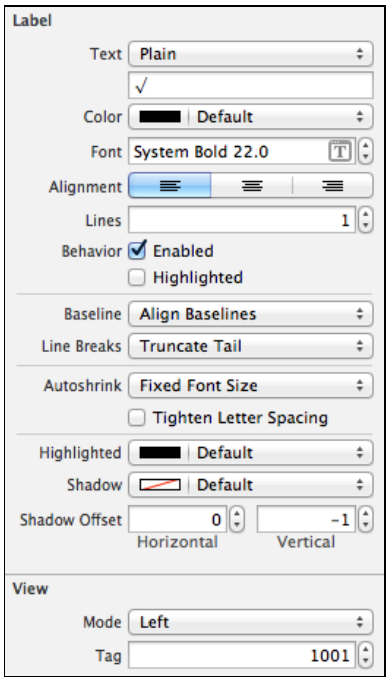
› Go to the table view cell in the storyboard and in the **Attributes inspector** set its **Accessory** to **Detail Disclosure**.

Instead of the checkmark you'll now see a chevron (>) and a blue 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:

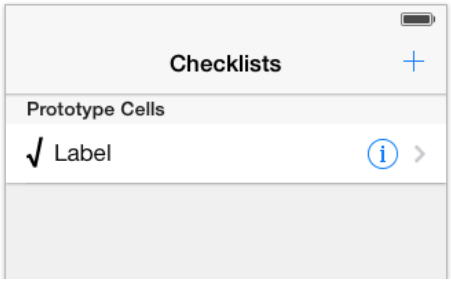
- Text: √ (you can type this with **Alt/Option+V**)
- Font: System Bold, size 22
- Tag: 1001

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



Attributes for the checkmark label

The design of the prototype cell now looks like this:



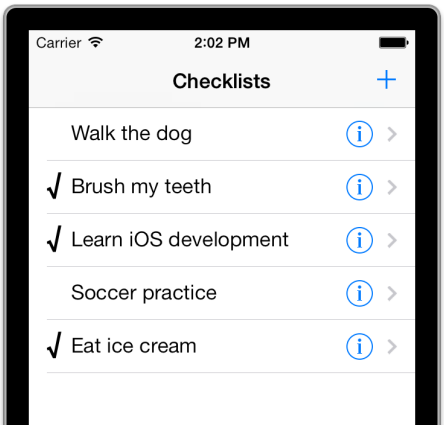
The new design of the prototype cell

➤ In **ChecklistsViewController.m**, change `configureCheckmarkForCell` to:

```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell
                  withChecklistItem:(ChecklistItem *)item
{
  UILabel *label = (UILabel *)[cell viewWithTag:1001];

  if (item.checked) {
    label.text = @"√";
  } else {
    label.text = @"";
  }
}
```

Instead of changing the cell's `accessoryType` property, this now changes the text in the new label.

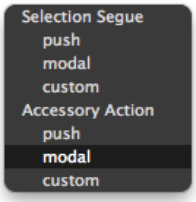
➤ Run the app and you'll see that the checkmark has moved to the left. There is now a blue detail disclosure button on the right. Tapping the row still toggles the checkmark, but tapping the blue button doesn't.



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 make a segue for a disclosure button.

➤ Open the storyboard. Select the table view cell and **Ctrl-drag** to the Navigation Controller next door. From the popup, choose **modal** 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. For the app to make a distinction between the 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 chevron button will open the Add Item screen. But the Cancel and Done buttons won't work because you haven't set the delegate yet. So far you only set the delegate in `prepareForSegue` for when you tap the + button and perform the "AddItem" segue, but not for this new "EditItem" segue.

Before you fix the delegate business, you shall make the Add/Edit Item screen capable of editing existing `ChecklistItem` objects.

➤ Add a new property for a `ChecklistItem` object below the other properties in **AddItemViewController.h**:

```
@property (nonatomic, strong) ChecklistItem *itemToEdit;
```

This property contains the existing `ChecklistItem` object that the user will be editing. But when adding a new to-do item, the `itemToEdit` property will be `nil`. That is how the view controller will make the distinction between adding and editing.

➤ Change the `viewDidLoad` in **AddItemViewController.m** method to the following:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.itemToEdit != nil) {
    self.title = @"Edit Item";
    self.textField.text = self.itemToEdit.text;
  }
}
```

Remember that `viewDidLoad` is called when the view controller is created, but before it is shown on the screen. That gives you time to put the user interface in order. In

editing mode, when `self.itemToEdit` is not `nil`, you change the title in the navigation bar to "Edit Item". You do this by changing the `self.title` property. The navigation controller looks for this property and automatically changes the title in the navigation bar. You also set the text in the text field to the value from the item's `text` property.

The `AddItemViewController` is now capable of recognizing when it needs to edit an item. If the `self.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 `prepareForSegue`, of course! That's the ideal place for putting values into the properties of the new screen before it becomes visible.

➤ Change `prepareForSegue` in **ChecklistsViewController.m** to the following:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"AddItem"]) {
    . . .

  } else if ([segue.identifier isEqualToString:@"EditItem"]) {

    UINavigationController *navigationController =
                                segue.destinationViewController;

    AddItemViewController *controller =
                  (AddItemViewController *)
                        navigationController.topViewController;

    controller.delegate = self;

    NSIndexPath *indexPath = [self.tableView
                                          indexPathForCell:sender];
    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 controller's `delegate` property so you're notified when the user taps Cancel or Done.

This is the interesting new bit:

```
NSIndexPath *indexPath = [self.tableView
                                        indexPathForCell:sender];
controller.itemToEdit = _items[indexPath.row];
```

The `sender` 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 to find the row number by looking up the corresponding index-path, and with the row number you can obtain the `ChecklistItem` object to edit.

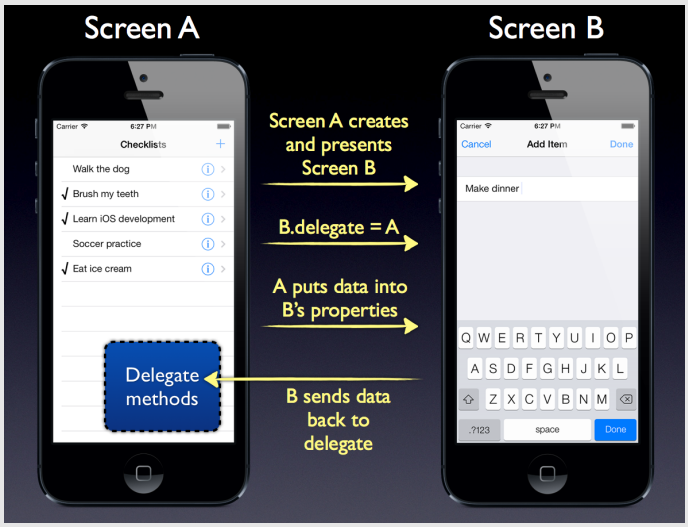
Sending data between the view controllers

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 note that here you're actually passing a piece of data from screen A to screen B, namely the `ChecklistItem` to edit.

This data transfer works two ways: if screen A opens screen B, then A can give B the data it needs. You simply make a property for this data on screen B and then screen A puts something into this property right before it makes screen B visible.

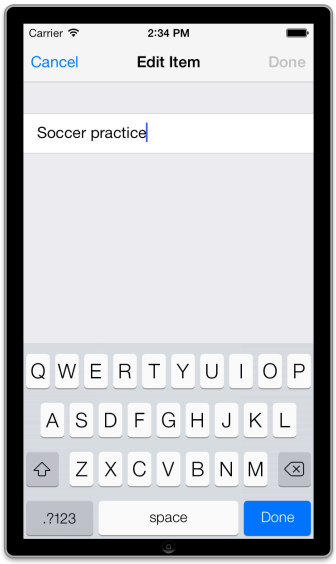
To pass data back from B to A you would use a delegate.

This illustration shows how screen A sends data to B by putting it into B's properties, and Screen B sends data back to the delegate:



You're going to do this a few more times in this lesson, just to make sure you don't forget it. :-)

With these steps done, you can now run the app. If you tap the + button, the Add Item screen works as before. However, if you tap the accessory button on an existing row, the screen that opens is now named Edit Item and it already contains the text of that item.



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.m** to fix this:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.itemToEdit != nil) {
    self.title = @"Edit Item";
    self.textField.text = self.itemToEdit.text;
    self.doneBarButton.enabled = YES;
  }
}
```

You can simply always enable the Done button because 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 now 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.h**:

```
- (void)addItemViewController:
                              (AddItemViewController *)controller
        didFinishEditingItem:(ChecklistItem *)item;
```

The full protocol now looks like this:

```
@protocol AddItemViewControllerDelegate <NSObject>

- (void)addItemViewControllerDidCancel:
                             (AddItemViewController *)controller;

- (void)addItemViewController:
                             (AddItemViewController *)controller
        didFinishAddingItem:(ChecklistItem *)item;

- (void)addItemViewController:
                              (AddItemViewController *)controller
        didFinishEditingItem:(ChecklistItem *)item;

@end
```

There is method that is invoked when the user presses Cancel and two methods for when the user presses Done. If they are adding a new item, `didFinishAddingItem` is called, but if they were editing an existing item then `didFinishEditingItem` is called instead. By using different methods the delegate (your view controller) can make a distinction between those two situations.

➤ In **AddItemViewController.m**, change the `done` method to:

```
- (IBAction)done
{
  if (self.itemToEdit == nil) {
    ChecklistItem *item = [[ChecklistItem alloc] init];
    item.text = self.textField.text;
    item.checked = NO;

    [self.delegate addItemViewController:self
                                 didFinishAddingItem:item];

  } else {
    self.itemToEdit.text = self.textField.text;
    [self.delegate addItemViewController:self
                         didFinishEditingItem:self.itemToEdit];
  }
}
```

First this checks whether the `itemToEdit` property contains an object. If not, then the user is adding a new item and you do the stuff you did before. The new part is this:

```
self.itemToEdit.text = self.textField.text;
[self.delegate addItemViewController:self
                          didFinishEditingItem:self.itemToEdit];
```

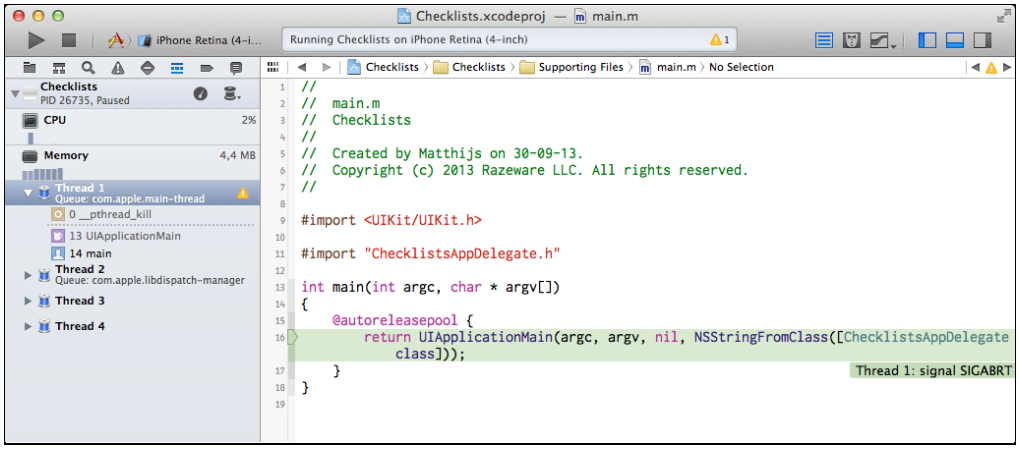
Nothing too spectacular here. You put the text from the text field into the existing `ChecklistItem` object from `itemToEdit` and then call the new delegate method.

(Note that you pass along the object from `itemToEdit` with the delegate method. This isn't strictly necessary as the delegate could simply look at the `itemToEdit` property to find that object. I chose to do it this way to be consistent with `didFinishAddingItem` because that message also sends along a `ChecklistItem` object.)

If you were to run the app now, it will crash. (Go ahead, try it out.) Once you press Done, the following will appear in your debug area:

```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException',
reason: '-[ChecklistsViewController addItemViewController:
didFinishEditingItem:]: unrecognized selector sent to instance
0xab04950'
```

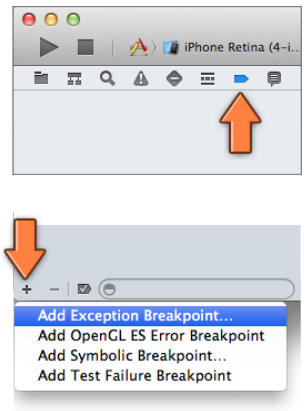
The Xcode window has switched to the debugger and points out which line caused the crash:



Xcode isn't being very helpful here

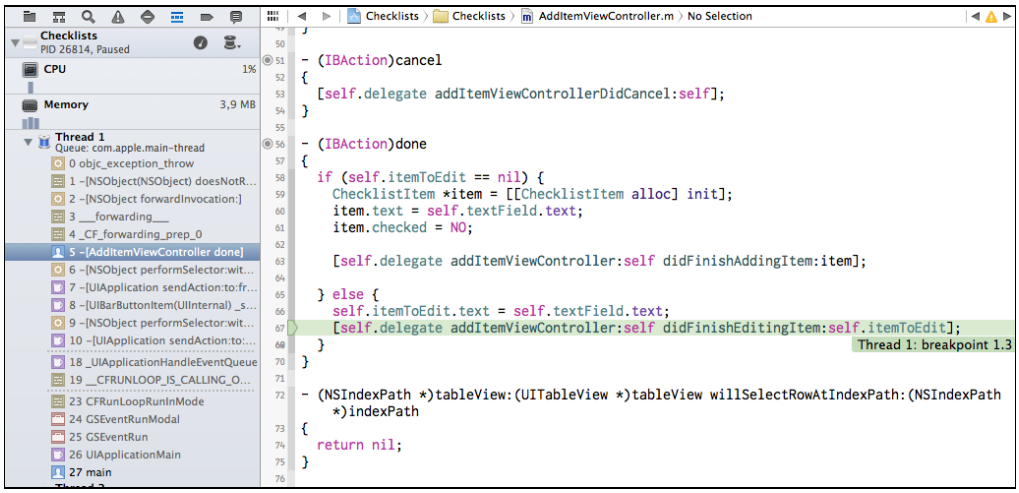
Xcode points at the **main.m** source file as the cause for the crash, but that's a little misleading. If this happened to you too, then you need to enable the **Exception Breakpoint**.

› Switch to the **Breakpoint navigator** and click the **+** button at the bottom:



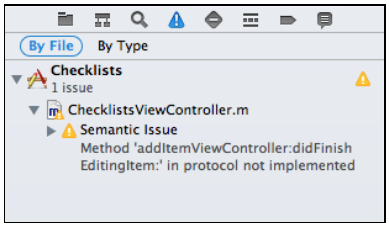
Adding the Exception Breakpoint

Now try it again. Run the app, edit an existing item and press Done. This time Xcode points at the correct line, inside `AddItemViewController done`:



The app crashes on the code you just added

More about debugging later, but I wanted to show you this error as it is very common to get this when you're using delegates. Xcode actually warned you about this problem in the **Issue navigator**:



Xcode warns about incomplete implementation

Xcode complains that `ChecklistsViewController` has an "incomplete implementation" because a method from the protocol wasn't implemented. That is also the reason for the crash: unrecognized selector sent.

A **selector** is term Objective-C uses for the name of a method, so this warning means the app tried to call a method named `addItemViewController:didFinishEditingItem:` that doesn't actually exist anywhere.

That is not so strange because you only added this method to the delegate protocol but did not actually tell the view controller, the actual delegate, what to do with it.

➤ Add the following to **ChecklistsViewController.m** and the crash will be history:

```
- (void)addItemViewController:
                              (AddItemViewController *)controller
        didFinishEditingItem:(ChecklistItem *)item
{
  NSInteger index = [_items indexOfObject:item];
  NSIndexPath *indexPath = [NSIndexPath indexPathForRow:index
                                              inSection:0];
  UITableViewCell *cell = [self.tableView
                                cellForRowAtIndexPath:indexPath];

  [self configureTextForCell:cell withChecklistItem:item];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

The `AddItemViewController` already updated the `ChecklistItem` object with the new text but you still need to update the table view. You look for the corresponding cell in the table and tell it to update its label using the `configureTextForCell` method you wrote earlier.

➤ Run the app again and verify that editing items now works.

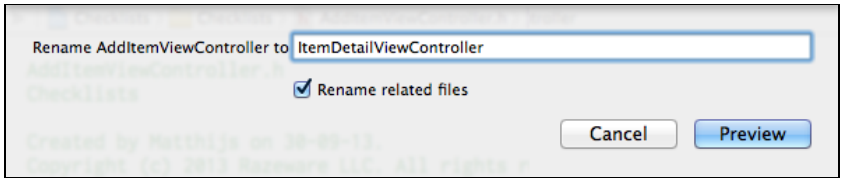
Refactoring the code

At this point you have an app that shows a list of items. You can toggle the checkmark on and off by tapping the rows. You can add new items and edit existing items using the Add/Edit Item screen and you can delete items by swiping the rows. 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 items and edit items. I propose you rename it to `ItemDetailViewController`.

➤ Go to **AddItemViewController.h** and click in the `@interface` line so that the blinking cursor is on the word `AddItemViewController`. From the Xcode menubar at the top of the screen choose **Edit → Refactor → Rename...**

Xcode will now ask you for the new name:



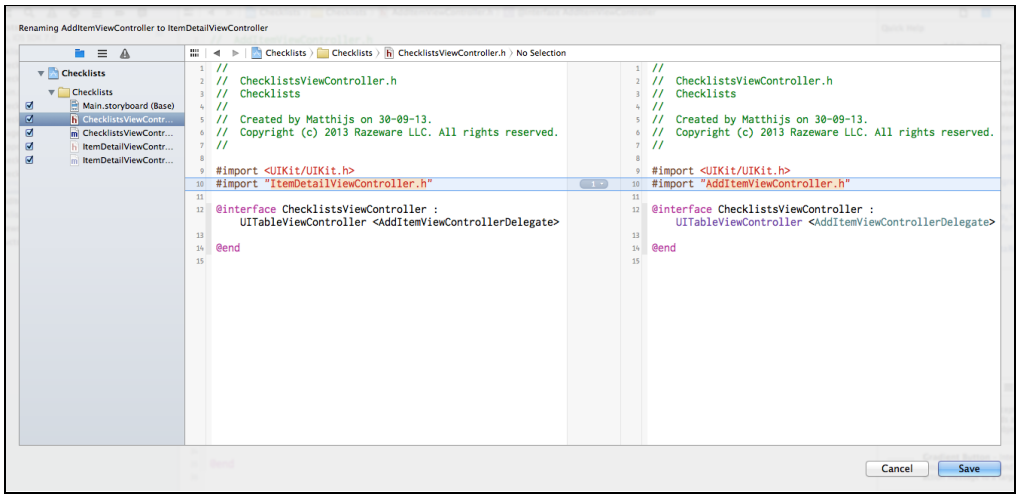
Changing the name of AddItemViewController

› Type **ItemDetailViewController** for the new name and check **Rename related files**.

Tip: If Xcode gives the error message "Wait for indexing and try again", then stop the running app first.

› Press **Preview**. Xcode opens a screen that shows the files that will change. Click on a filename to see the changes that will be made inside of that file.

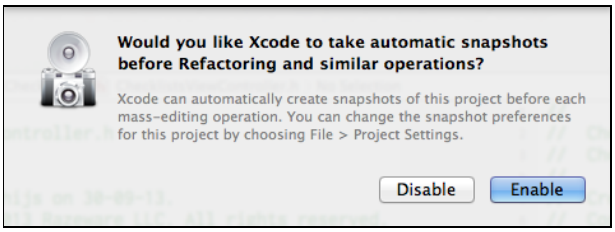
You'll see that Xcode will simply rename everything from `AddItemViewController` to `ItemDetailViewController`. It's always smart to check what Xcode is going to do.



Xcode gives a preview of which files it will change

› Click **Save** to let Xcode do its thing.

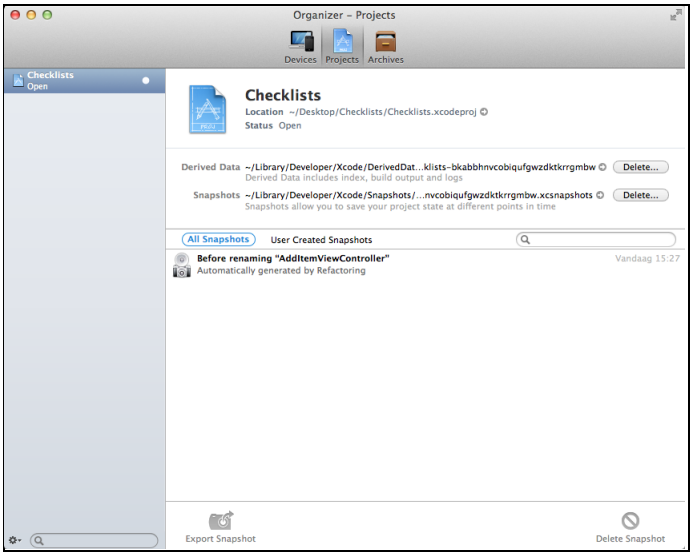
Xcode will now ask you if you want to enable automatic snapshots. A snapshot is a copy of your entire project for safekeeping. It is probably a good idea to enable this. If something goes wrong, you can always go back to an earlier snapshot.



Enable automatic snapshots for simple backups of your project

› Click **Enable** and wait a few seconds for Xcode to complete the operation.

In case you're curious (or something went wrong!) you can find the snapshots in the **Organizer** window, under the **Projects** tab:



The Organizer window lists the snapshots for your project

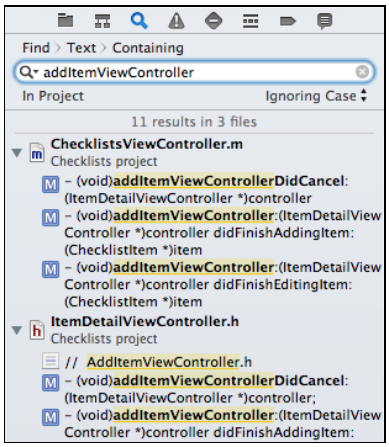
You can also use the **File → Restore Snapshot...** option from the Xcode menu bar. If at some point you wish to make a manual snapshot, use **File → Create Snapshot**.

You're not done yet with these refactorings.

› Repeat this process to rename the `AddItemViewControllerDelegate` protocol to `ItemDetailViewControllerDelegate`. You better get used to those long names!

Unfortunately, Xcode isn't able to automatically rename the methods from the delegate protocol for you, so you'll have to do this manually.

› Switch to the **Search navigator** and type: **addItemViewController**. This searches through the entire project for that text.



Using the Search navigator to find the methods to change

› You will have to change:

`addItemViewControllerDidCancel:` into `itemDetailViewControllerDidCancel:`

and,

`addItemViewController:` into `itemDetailViewController:`

After these changes the `@protocol` in **ItemDetailViewController.h** now has these methods:

```
@protocol ItemDetailViewControllerDelegate <NSObject>

- (void)itemDetailViewControllerDidCancel:
                        (ItemDetailViewController *)controller;

- (void)itemDetailViewController:
                        (ItemDetailViewController *)controller
        didFinishAddingItem:(ChecklistItem *)item;

- (void)itemDetailViewController:
                        (ItemDetailViewController *)controller
        didFinishEditingItem:(ChecklistItem *)item;

@end
```

› You'll also need to change these method names in **ItemDetailViewController.m** and **ChecklistsViewController.m**.

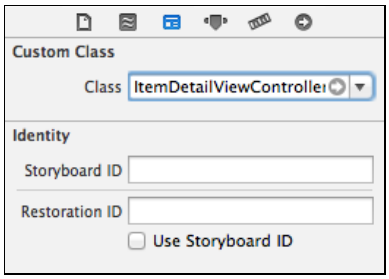
I always repeat the search afterwards to make sure I didn't skip anything by accident.

› Press ⌘**+B** to compile the app. If you made all the changes without any mistakes, the app should build without errors.

There is one more thing you need to verify. Depending on your version of Xcode, the Refactor tool may not be smart enough to also change the view controller inside the storyboard. If that happens, the storyboard still thinks the view controller is named `AddItemViewController` and the app will crash with the following message:

```
Checklists[2248:207] Unknown class AddItemViewController in Interface
Builder file.
```

› Open the storyboard and select the Add Item View Controller. In the **Identity inspector**, change **Class** to **ItemDetailViewController**:



Don't forget to change the storyboard as well

Because you made quite a few changes all over the place, it's a good idea to do a *clean* build 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 do a clean build once in a while.

› From Xcode's menubar choose **Product** → **Clean**. When the clean is done choose **Product** → **Build** (or simply press the Run button). If there are no issues, then run the app again and test the various features just to make sure everything still works!

> **Tip:** If renaming gives you problems, then double-check your spelling. Objective-C is case-sensitive, so it considers "itemDetailViewController" and "ItemDetailViewController" to be two completely different words.

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 is a little messy, then you're absolutely right. You started out with one design but as you were developing it 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 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 to you how each part works. That's not how software development happens in the real world. So 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 chapter, 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 than meticulous up-front planning and provides better results.

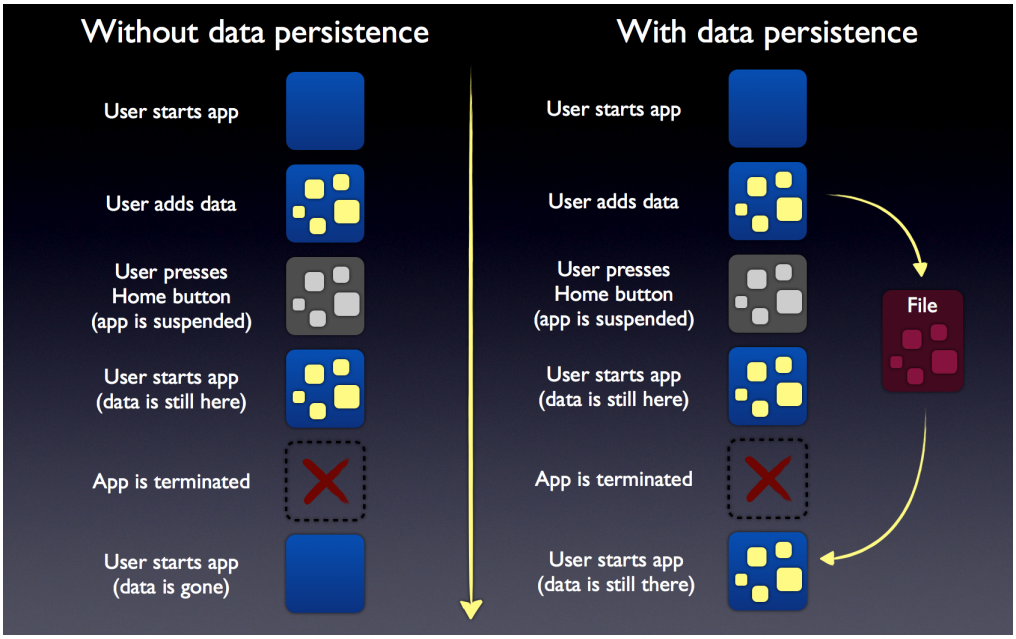
Saving and loading the checklist items

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). You can also delete the five default items from the list but they keep reappearing after a new launch. That's not how a real app should behave, of course.

Thanks to the multitasking features of iOS, an app stays in memory when you close it. 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, 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, then it cannot break any other part of the system.

Your apps can store files in the so-called "Documents" directory. You are guaranteed this directory is always available 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 contents of the Documents folder are left untouched. So any data the app has saved into this folder stays there even if the app is updated.

Let's look at how this works.

▸ Add the following methods to **ChecklistsViewController.m**, above `viewDidLoad`:

```
- (NSString *)documentsDirectory
{
  NSArray *paths = NSSearchPathForDirectoriesInDomains(
                   NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths firstObject];
  return documentsDirectory;
}

- (NSString *)dataFilePath
{
  return [[self documentsDirectory]
            stringByAppendingPathComponent:@"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 you will use to store the checklist items. This file is named **Checklists.plist** and it lives inside the Documents directory.

You use the `NSString` method `stringByAppendingPathComponent` to build a proper file system path to Checklists.plist. You can call this method because the return value of `[self documentsDirectory]` is also an `NSString`. Recall that `NSString` objects are immutable, so this method will create a new string object with the full path to the file.

You could also have constructed the full path like this:

```
- (NSString *)dataFilePath
{
  return [NSString stringWithFormat:@"%@/Checklists.plist",
                                    [self documentsDirectory]];
}
```

This adds "Checklist.plist" to the Documents directory path, separated by a forward slash. However, I prefer to use `stringByAppendingPathComponent` since that frees me from worrying about whether to use a forward slash or a backward slash, what to do if there already is a slash in the folder name, and many other tiny concerns. The

built-in objects from iOS come with a lot of useful helper methods like these and it's often better to use them instead of trying to do things on your own.

➤ Add the following two `NSLog()` statements to `viewDidLoad`:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  NSLog(@"Documents folder is %@", [self documentsDirectory]);
  NSLog(@"Data file path is %@", [self 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 /Users/matthijs/Library/Application Support/
iPhone Simulator/7.0/Applications/A0C05CC9-2798-4693-830C-
7B5F5CB86AEA/Documents

Data file path is /Users/matthijs/Library/Application Support/
iPhone Simulator/7.0/Applications/A0C05CC9-2798-4693-830C-
7B5F5CB86AEA/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):

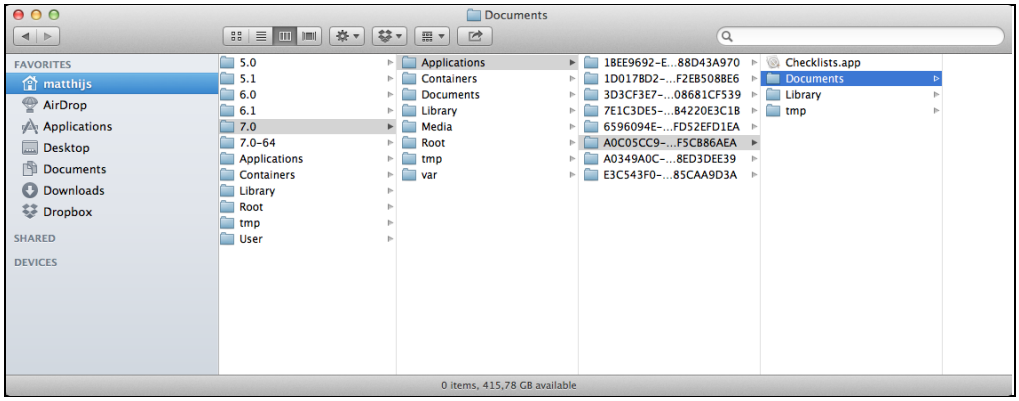
```
Documents folder is /var/mobile/Applications/FDD50B54-9383-4DCC-9C19-
C3DEBC1A96FE/Documents

Data file path is /var/mobile/Applications/FDD50B54-9383-4DCC-9C19-
C3DEBC1A96FE/Documents/Checklists.plist
```

The name of the folder that contains the app is "A0C05CC9-2798-4693-830C-7B5F5CB86AEA" (on the Simulator) and "FDD50B54-9383-4DCC-9C19-C3DEBC1A96FE" (on the device). This is a random ID that Xcode (or iTunes) 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. 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.

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.



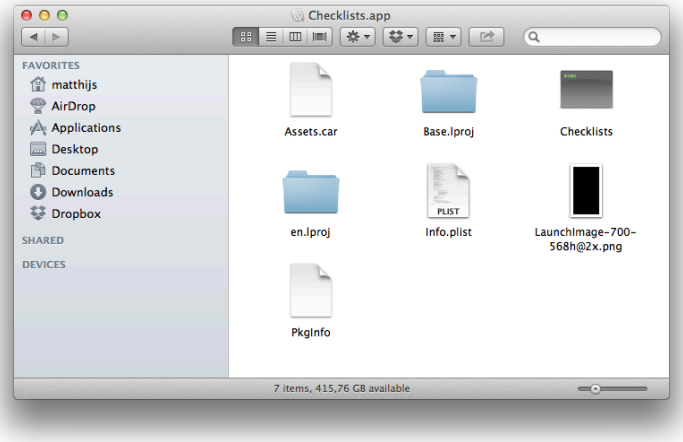
The app's directory structure in the Simulator

> **Tip:** Are you using OS X Lion (10.7) or Mountain Lion (10.8) and you cannot find the Library folder in your home directory? To fix this, open Terminal and type the following command:
>
> ```
> chflags nohidden ~/Library
> ```
>
> You can also 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:

- Checklists.app, which is your application bundle.
- 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. This 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. The OS will clear out this folder from time to time.

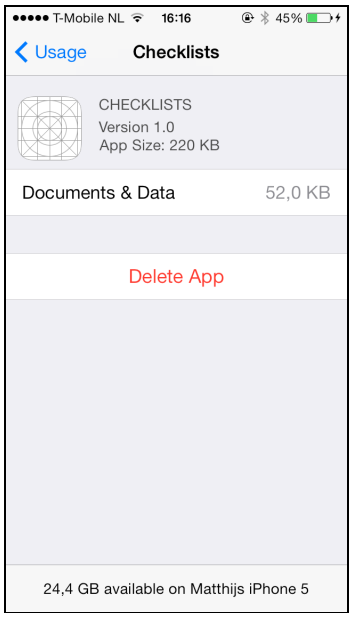
If you're curious about what is exactly in the application bundle, then right click its filename and choose **Show Package Contents**. The app bundle is really a folder although Finder pretends it isn't. With Show Package Contents you can see what is inside that folder.



The contents of the application bundle

Later we'll discuss in more detail what is going on here.

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** and tap 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

You are going to write code that will save 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 will add the code that is required to load this list.

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 for the file named Checklists-Info.plist). "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, 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). This process of converting objects to files and back again is also known as **serialization**.

I like to think of this whole process as freezing objects. You take a living object and freeze it so that it is now suspended in time. You store that frozen object into a file 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.

➤ Add the following method to **ChecklistsViewController.m**, below `dataFilePath`:

```
- (void)saveChecklistItems
{
  NSMutableData *data = [[NSMutableData alloc] init];
  NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
                               initForWritingWithMutableData:data];
  [archiver encodeObject:_items forKey:@"ChecklistItems"];
  [archiver finishEncoding];
  [data writeToFile:[self dataFilePath] atomically:YES];
}
```

This method takes the contents of the `_items` array and in two steps converts this 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, which happens in the `ItemDetailViewControllerDelegate` methods.

▸ Make the following changes in these methods inside **ChecklistsViewController.m**:

```
- (void)itemDetailViewController:
                          (ItemDetailViewController *)controller
        didFinishAddingItem:(ChecklistItem *)item
{
  . . .

  [self saveChecklistItems];

  [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)itemDetailViewController:
                          (ItemDetailViewController *)controller
        didFinishEditingItem:(ChecklistItem *)item
{
  . . .

  [self saveChecklistItems];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

▸ Let's not forget the swipe-to-delete function:

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
     forRowAtIndexPath:(NSIndexPath *)indexPath
{
  [_items removeObjectAtIndex:indexPath.row];

  [self saveChecklistItems];

  NSArray *indexPaths = @[indexPath];
  [tableView deleteRowsAtIndexPaths:indexPaths
            withRowAnimation:UITableViewRowAnimationAutomatic];
}
```

▸ And toggling the checkmark on a row on or off:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  . . .

  [self saveChecklistItems];

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

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: '-[ChecklistItem
encodeWithCoder:]: unrecognized selector sent to instance 0x6a26810'
```

The Xcode debugger points to this line as the culprit, inside the `saveChecklistItems` method:

```
[archiver encodeObject:_items forKey:@"ChecklistItems"];
```

You've already seen the "unrecognized selector" error message before. This means you forgot to implement a certain method. In this case, the missing method appears to be `encodeWithCoder` on the `ChecklistItem` object – that's what the error 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 `NSMutableArray` object but it doesn't know anything about `ChecklistItem`. So you have to help it out a bit.

▸ Change the `@interface` line in **ChecklistItem.h**:

```
@interface ChecklistItem : NSObject <NSCoding>
```

Recall that < > means that an object conforms to a protocol. In this case you're adding the `NSCoding` protocol to `ChecklistItem`.

▸ Add the following to **ChecklistItem.m**:

```
- (void)encodeWithCoder:(NSCoder *)aCoder
{
  [aCoder encodeObject:self.text forKey:@"Text"];
  [aCoder encodeBool:self.checked forKey:@"Checked"];
```

```
}
```

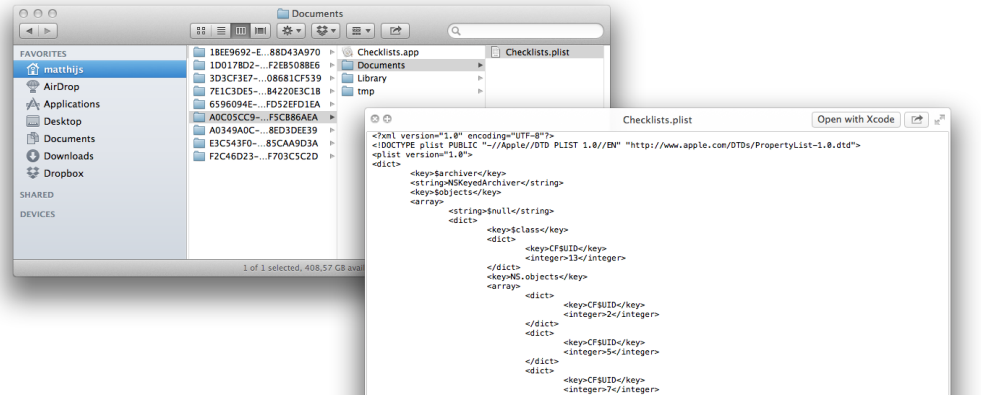
This is the missing method from the unrecognized selector error. When `NSKeyedArchiver` tries to encode the `ChecklistItem` object it will send it an `encodeWithCoder` message.

Here you simply say: a `ChecklistItem` has an object named "Text" that contains the value of the property `self.text` and a boolean named "Checked" that contains the value of `self.checked`. Just these two lines are enough to make the coder system work.

› Run the app again and tap a row to toggle a checkmark. The app didn't crash? Good!

Note: At this point Xcode may give a warning that the implementation of `ChecklistItem` is incomplete. That's OK, you will add the missing method shortly.

› Go to the Finder window that has the app's Documents directory open. (If you forgot where that was, it's in the Library folder in your home directory and then Application Support/iPhone Simulator/7.0/Applications/weird number/Documents.)



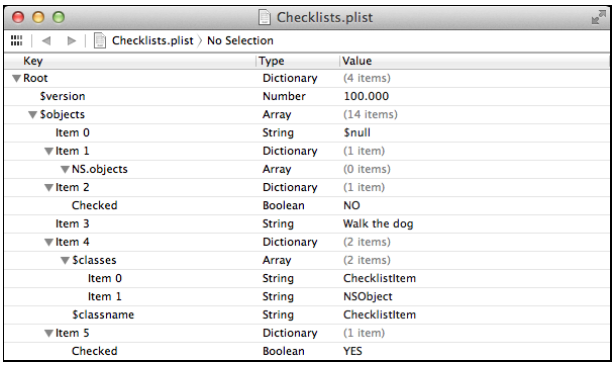
The Documents directory now contains a Checklists.plist file

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 to you. 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. If you expand some of the rows you can see that this file was made by `NSKeyedArchiver` and that the names of the `ChecklistItem`s are also in there. But exactly how everything fits together, I have no idea.

Loading the file

Saving is all well and good but pretty useless by itself so let's 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 may have noticed Xcode is complaining that **ChecklistItem.m** does not implement another method from the `NSCoding` protocol, `initWithCoder`. That is the method for unfreezing the objects from the file.

➤ Add the following directly above the `encodeWithCoder` method:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super init])) {
    self.text = [aDecoder decodeObjectForKey:@"Text"];
    self.checked = [aDecoder decodeBoolForKey:@"Checked"];
  }
  return self;
}
```

Inside `initWithCoder` you do the opposite from `encodeWithCoder`. 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 `self.text` property. Likewise for the boolean "Checked" value and `self.checked`.

Loading the Checklists.plist file will be done by an `NSKeyedUnarchiver` object. That unarchiver does the following behind the scenes to create the `ChecklistItem` objects:

```
ChecklistItem *item = [[ChecklistItem alloc]
                              initWithCoder:someDecoderObject];
```

But when you created the `ChecklistItem`s by hand, you did this:

```
ChecklistItem *item = [[ChecklistItem alloc] init];
```

The difference is that `NSKeyedUnarchiver` will not use the regular `init` method but the one named `initWithCoder`. It is not uncommon for objects to have more than one init method. They are always named "initSomething", which is a mandatory convention. But which one is used depends on the circumstances.

You use the regular `init` method for creating `ChecklistItem` objects when the user presses the + button and fills in the Add Item screen, and you use `initWithCoder` to restore `ChecklistItem`s that were saved to disk.

Init methods

Method names beginning with the word "init" are special in Objective-C. You only use them when you're creating new objects. First you `alloc` the object to reserve a chunk of memory big enough to hold all of the object's data (its instance variables), followed by a call to an `init` method to initialize the object so that it is ready for use.

The implementations of these init methods, whether they're just called `init` or `initWithCoder` 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:

```
- (id)init
{
  self = [super init];
  if (self) {

    // Initialization code here.

  }
  return self;
}
```

First you call `[super init]` to initialize this object's superclass and then assign the result to `self`. If you're coming from another programming language and this looks weird to you, well, that's Objective-C for you.

If you haven't done any object-oriented programming at all, then 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 assigning the result from `[super init]` to `self`, you have to look at the value of `self` to determine that it is not `nil`:

```
if (self) {
    // Initialization code here.
}
```

The statement `if (self)` is shorthand for: `if (self != nil)`.

In the `initWithCoder` method from `ChecklistItem`, you used a common Objective-C idiom:

```
if ((self = [super init])) {
   . . .
}
```

This combines these two lines into one:

```
self = [super init];
if (self != nil) {
  . . .
}
```

If you wanted to be fully explicit, you'd write it as:

```
if ((self = [super init]) != nil) {
   . . .
}
```

This still calls `[super init]`, assigns it to `self` and then checks if `self` is not `nil`, except it does it all inside the if-statement.

Note the double pair of parentheses. If you were to write it like this,

```
if (self = [super init]) {
   . . .
}
```

then Xcode complains. It is not sure whether you meant to make the assignment or whether you intended to compare the value of `self` to the return value of `[super init]`, in which case you should have used two equals signs:

```
if (self == [super init]) {
   . . .
}
```

That doesn't really make sense here, as you definitely want to assign the result of `[super init]` to `self`, not compare the two. The extra pair of parentheses is used to make the intention clear to the compiler.

It doesn't really matter which approach you use to write this. They are all equivalent:

```
self = [super init];
if (self) { . . . }


self = [super init];
if (self != nil) { . . . }


if ((self = [super init])) { . . . }


if ((self = [super init]) != nil) { . . . }
```

As long as you don't forget to do it!

It can happen that the call to `[super init]` returns `nil` in which case the initialization for the object's superclass failed and this object cannot be used. The object has already been destroyed at that point and there is nothing left to do but bail out. That's why you use the if-statement.

Fortunately, that doesn't happen a lot. An instance where this could happen is if your object tries to load an image but the image is not available and without it your object cannot function. Then you'd destroy the object and return `nil`. That is exactly what the `UIImage` object does if you give it the name of an image file that is not present in your application bundle.

If `self` is not `nil`, then the superclass was properly initialized and you can perform your own initialization. For example, in `ChecklistItem`'s `initWithCoder` method you initialize the object by reading the values from the `NSCoder` object and stuffing them into `ChecklistItem`'s properties.

Finally, you return `self`. That is necessary so the caller can assign the new object to a variable.

In summary, this is what an init method needs to do:

1. Call `[super init]` and assign the result to `self`.

2. Check whether `self` is `nil`. If so, then exit this method right away and return `nil` to the caller.

3. If `self` was not `nil`, do additional initialization if necessary. Usually this means giving properties and instance variables their initial values. By default, objects are `nil`, `int`s are 0 and `BOOL`s are `NO`. If you want to give these variables different initial values, then this is the place to do so.

4. Return `self` to the caller.

You don't always need to provide an init method. If your init method doesn't need to do anything – if there are no properties or instance variables to fill in – then you can leave it out completely and the compiler will provide one for you.

That's why you can call `[[ChecklistItem alloc] init]`, even though there is no method named `init` in **ChecklistItem.m** (only `initWithCoder`).

The implementation of `ChecklistItem` is complete, as it can now bring back to life objects that were serialized (or frozen) into the plist file. But you still have to write the code that will actually load this plist. That happens in `ChecklistsViewController`.

Until now you've done the initialization of the data model in `viewDidLoad`. That was convenient for the purposes of the tutorial but not always correct. `viewDidLoad` is the place where you should set up your views, i.e. the visible user elements from your app. Data model objects should be initialized earlier because typically the data model tends to live longer than the view. Now that you know all about init methods, let's create the data model in `ChecklistsViewController`'s init method.

A table view controller, like many objects, has more than one init method. There is:

- `initWithCoder`, for view controllers that are automatically loaded from a storyboard
- `initWithNibName`, 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)
- `initWithStyle`, for table view controllers that you manually want to create without using a storyboard or nib

This view controller comes from a storyboard, so you'll use `initWithCoder` to create the data model and load the plist file. Yup, that's actually the same 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 **ChecklistsViewController.m**, add the `initWithCoder` method above `viewDidLoad`:

```
- (id)initWithCoder:(NSCoder *)aDecoder
```

```
{
  if ((self = [super initWithCoder:aDecoder])) {
    [self loadChecklistItems];
  }
  return self;
}
```

Notice the same pattern: This is an init method so you call `super` and assign the result to `self`. If `self` is not `nil` you will do stuff, and finally you return a reference to `self`. The only difference is that you don't call `[super init]` but `[super initWithCoder]`. That ensures the rest of the view controller is properly unfrozen from the storyboard file.

▸ Add the `loadChecklistItems` method above `initWithCoder`:

```
- (void)loadChecklistItems
{
  NSString *path = [self dataFilePath];
  if ([[NSFileManager defaultManager] fileExistsAtPath:path]) {

    NSData *data = [[NSData alloc] initWithContentsOfFile:path];
    NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
                                     initForReadingWithData:data];

   _items = [unarchiver decodeObjectForKey:@"ChecklistItems"];
    [unarchiver finishDecoding];

  } else {
    _items = [[NSMutableArray alloc] initWithCapacity:20];
  }
}
```

Let's go through this. There are basically two courses that this method can take:

```
NSString *path = [self dataFilePath];
if ([[NSFileManager defaultManager] fileExistsAtPath:path]) {

  . . .

} else {
  _items = [[NSMutableArray alloc] initWithCapacity:20];
}
```

You first put the results of `[self dataFilePath]` in a temporary variable named `path`. You use the path name more than once in this method so having it available

in a local variable instead of calling `dataFilePath` several times over is a small optimization.

Then you check whether the file actually exists and decide what happens based on that.

If there is no Checklists.plist then there are obviously no `ChecklistItem` objects to load. In that case, you go to the `else` section and create an empty `NSMutableArray`. That is exactly what you used to do in `viewDidLoad`, so this shouldn't be too surprising. This is what happens when the app is started up for the very first time.

When the app does find a Checklists.plist file, you don't have to make the array yourself. Instead, you'll load the entire array and its contents from the Checklists.plist file:

```
NSData *data = [[NSData alloc] initWithContentsOfFile:path];
NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
                                 initForReadingWithData:data];

_items = [unarchiver decodeObjectForKey:@"ChecklistItems"];
[unarchiver finishDecoding];
```

This is essentially the reverse of what `saveChecklistItems` does. First you load the contents of the file into an `NSData` object. Then you create an `NSKeyedUnarchiver` object (note: this is an *un*archiver) and ask it to decode that data into the `_items` array. This creates an `NSMutableArray` and populate it with exact copies of the `ChecklistItem` objects that were frozen into this file.

➤ You can now remove the code that created fake items from `viewDidLoad`:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
}
```

There is currently nothing left to do in `viewDidLoad` (other than calling `super`), but you'll give it something new to do soon enough.

➤ 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.

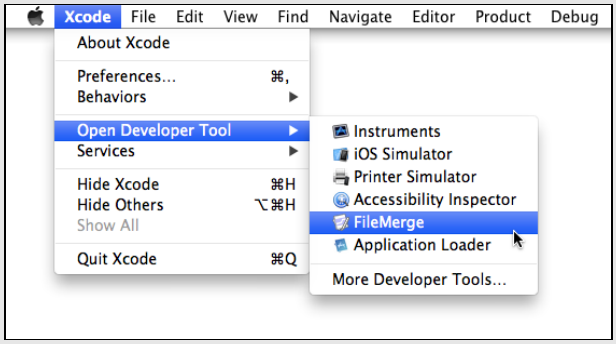
➤ Stop the app again. Go to the Finder window with the Documents folder and remove the Checklists.plist file. Run the app once more. You should now have an empty screen. Add an item and notice that the Checklists.plist file re-appears.

Awesome! You've written an app that not only lets you add and edit data, but that also persists that 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.

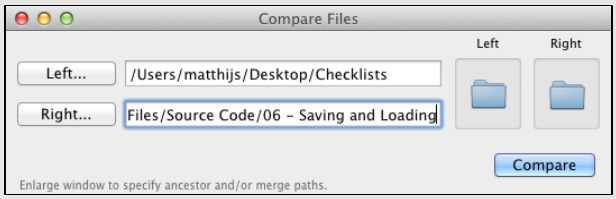
You can find the project files for the app up to this point under **06 - Saving and Loading** in the tutorial's Source Code folder.

Using FileMerge to compare files

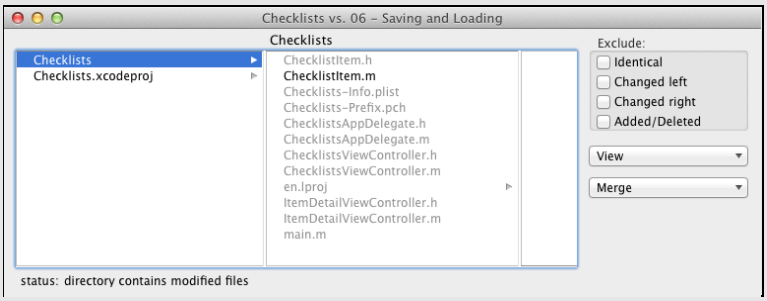
You can compare your own work with my version of the app using the FileMerge tool. You open this tool from the Xcode menu:



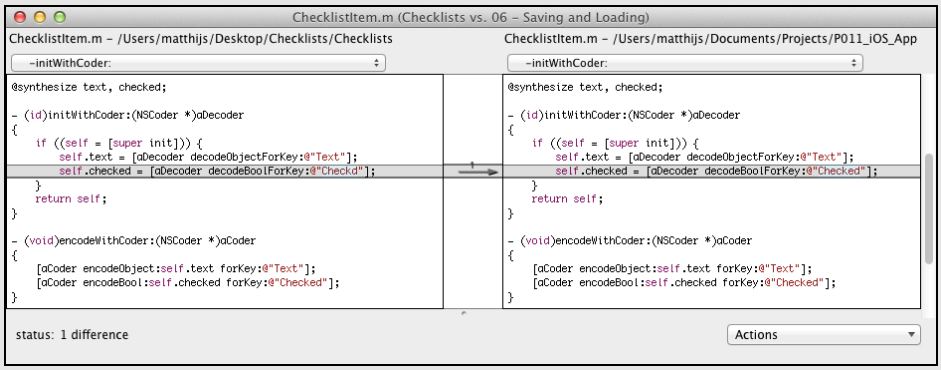
You give FileMerge two files or two folders to compare:



After working hard for a few seconds or so, FileMerge tells you what is different:



Double-click on a filename from the list and FileMerge shows 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" between your own files and the ones from the Source Code folder to see if you can find any anomalies.

Just to make sure you truly get everything you've done so far, you will now expand the app with some new features that more or less repeat what you just did. You'll also add cool stuff such as local notifications.

Adding 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: `AllListsViewController` that shows all the user's lists and `ListDetailViewController` that allows adding a new list and editing the name and icon of an existing list.

The app's main screen is currently the `ChecklistsViewController`. This file was created by Xcode as part of the Single View Application template and was named after the Class Prefix you chose when you first created the project. In light of the changes you're about to make, the plural form "Checklists" no longer makes sense as this screen only shows the to-do items that belong to a single checklist.

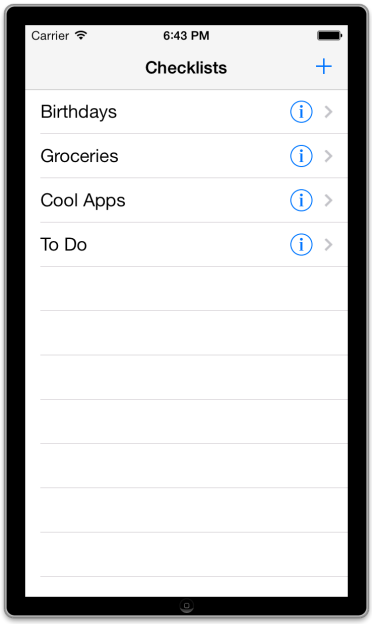
› Use Xcode's **Refactor** tool to rename this object to `ChecklistViewController` (drop the "s" after "Checklist"). Don't forget to change the storyboard as well!

When you're done, do a clean build and run the app to make sure it all still works.

The All Lists screen

You will first add the new `AllListsViewController`. This will now become the 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**. Under Cocoa Touch choose the **Objective-C class** template. Press **Next** and name the new file **AllListsViewController**, subclass of **UITableViewController**. Make sure the other options are unchecked.

The default template for this file needs some work before you can run the app. As a first step, you'll put some fake data in the table view just to get it up and running. 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 expand on what you have and put in the real data.

› In **AllListsViewController.m**, remove the `numberOfSectionsInTableView` method.

➤ Change the `numberOfRowsInSection` method to:

```
- (NSInteger)tableView:(UITableView *)tableView
            numberOfRowsInSection:(NSInteger)section
{
  return 3;
}
```

➤ Change `cellForRowAtIndexPath` to:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"Cell";

  UITableViewCell *cell = [tableView
              dequeueReusableCellWithIdentifier:CellIdentifier];

  if (cell == nil) {
    cell = [[UITableViewCell alloc]
                      initWithStyle:UITableViewCellStyleDefault
                    reuseIdentifier:CellIdentifier];
  }

  cell.textLabel.text = [NSString stringWithFormat:
                                      @"List %d", indexPath.row];
  return cell;
}
```

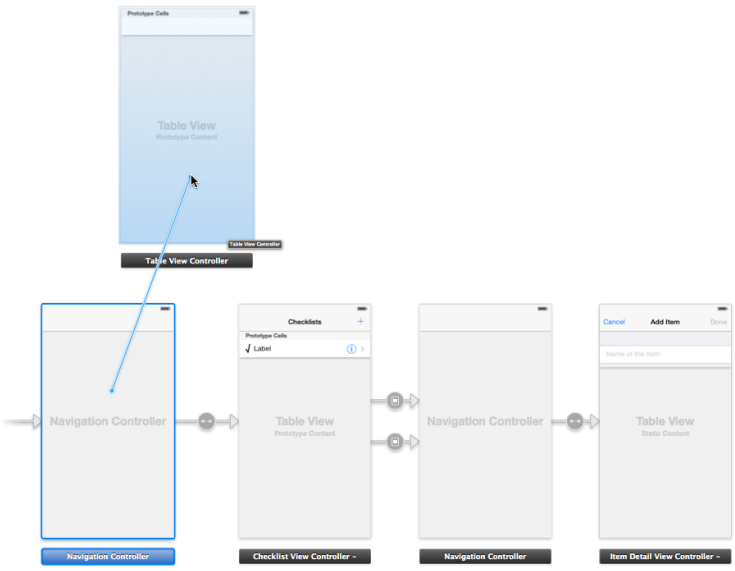
This is very similar to what the template already put in that method, except that you're also putting some text in the cells.

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.

➤ **Ctrl-drag** from the very first navigation controller to this new table view controller. From the popup menu choose **Relationship Segue - root view controller**.

This will break the connection that existed between the navigation controller and the Checklist View Controller so that "Checklists" is no longer the app's main screen.



Ctrl-drag from the navigation controller to the new table view controller

➤ Select the new table view controller and set its **Class** in the **Identity inspector** to **AllListsViewController**.

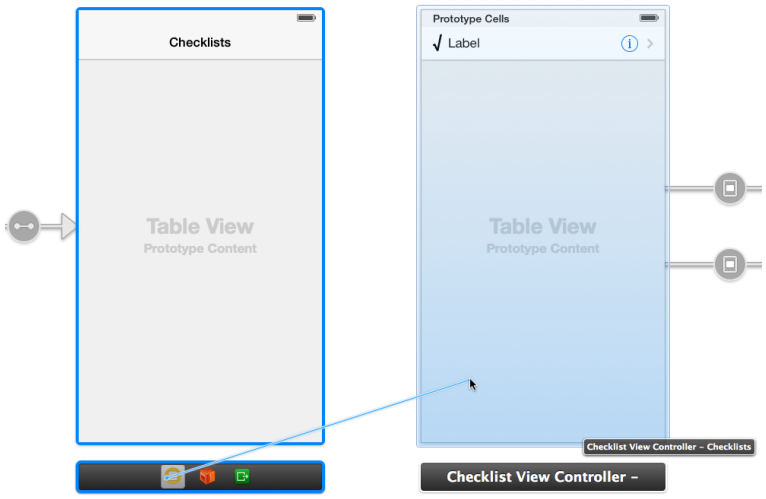
➤ Double-click the view controller's navigation bar and change its title to **Checklists**.

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.

Just for the fun of it, 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.

➤ **Ctrl-drag** from the All Lists View Controller icon in the dock (or the outline pane) into the Checklist View Controller and create a **push** 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. Note that this new segue isn't attached to any button or table view cell. There is nothing on the screen that you can tap or otherwise interact with in order to trigger this segue. That means you have to perform it programmatically.

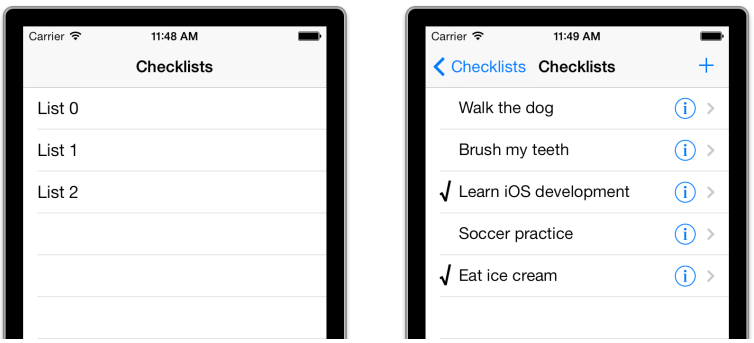
➤ Click on the new segue to select it, go to the **Attributes inspector** and give it the identifier **ShowChecklist**. The **Style** should be **Push** because you're pushing the Checklist View Controller onto the navigation stack when performing this segue.

➤ In **AllListsViewController.m**, add the `didSelectRowAtIndexPath` method:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [self performSegueWithIdentifier:@"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 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 `performSegueWithIdentifier` 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).

If you tap on a row, the familiar `ChecklistViewController` slides into the screen. You can tap the Checklists back button in the top-left to go back to the main list. Now you're truly using the power of the navigation controller!

> **Note:** If the app crashes for you at this point, then make sure the `cellForRowAtIndexPath` method in **AllListsViewController.m** says this,
>
> ```
> UITableViewCell *cell = [tableView
>             dequeueReusableCellWithIdentifier:CellIdentifier];
> ```
>
> Instead of this:
>
> ```
> UITableViewCell *cell = [tableView
>              dequeueReusableCellWithIdentifier:CellIdentifier
>              forIndexPath:indexPath];
> ```
>
> Xcode's autocomplete makes it really easy to mistakenly select the second one, but that is not the method you want to use here!

You're going to duplicate most of the functionality from the Checklist View Controller for this new All Lists screen. You'll add a + button at the top that lets users add a new checklist, you'll do swipe-to-delete, and you'll let users edit the name of the checklist from a disclosure button. Of course, you 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 **Objective-C class** template, subclass of **NSObject**, and name it **Checklist**.

This adds the files Checklist.h and Checklist.m to the project.

➤ Give **Checklist.h** a `name` property:

```
#import <Foundation/Foundation.h>
```

```
@interface Checklist : NSObject

@property (nonatomic, copy) NSString *name;

@end
```

Next, you'll give `AllListsViewController` an array that will store these new `Checklist` objects.

➤ Add to **AllListsViewController.m**, below the other import:

```
#import "Checklist.h"
```

➤ Add a new instance variable block with an array named `_lists` that will hold the `Checklist` objects:

```
@implementation AllListsViewController
{
  NSMutableArray *_lists;
}
```

As a first step you will fill this list with test data, which you'll do from the `initWithCoder` method. Remember that this method is automatically invoked by UIKit as it loads the view controller from the storyboard.

➤ Still in **AllListsViewController.m**, add the `initWithCoder` method:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder]))
  {
    _lists = [[NSMutableArray alloc] initWithCapacity:20];

    Checklist *list;

    list = [[Checklist alloc] init];
    list.name = @"Birthdays";
    [_lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"Groceries";
    [_lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"Cool Apps";
    [_lists addObject:list];
```

```
    list = [[Checklist alloc] init];
    list.name = @"To Do";
    [_lists addObject:list];
  }
  return self;
}
```

Notice that the file already has an `initWithStyle` method. Remove that method as this app won't be using it. Since the view controller is part of a storyboard, it will always be initialized using `initWithCoder`.

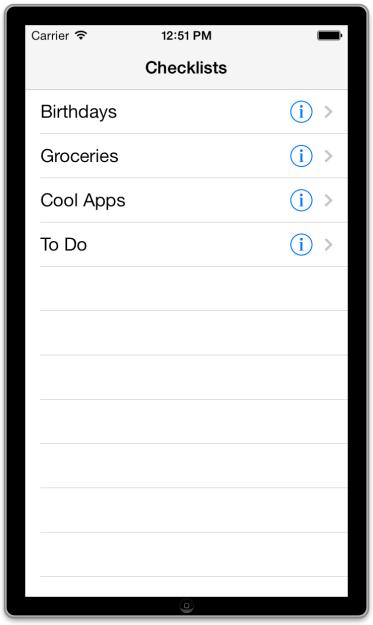
› Change the `numberOfRowsInSection` method to return the number of objects in the new array:

```
- (NSInteger)tableView:(UITableView *)tableView
            numberOfRowsInSection:(NSInteger)section
{
  return [_lists count];
}
```

› Finally, change `cellForRowAtIndexPath` to create the cells for the rows:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"Cell";

  UITableViewCell *cell = [tableView
             dequeueReusableCellWithIdentifier:CellIdentifier];

  if (cell == nil) {
    cell = [[UITableViewCell alloc]
                      initWithStyle:UITableViewCellStyleDefault
                    reuseIdentifier:CellIdentifier];
  }

  Checklist *checklist = _lists[indexPath.row];
  cell.textLabel.text = checklist.name;
  cell.accessoryType =
                UITableViewCellAccessoryDetailDisclosureButton;

  return cell;
}
```

› Run the app. It looks like this:



The table view shows Checklist objects

The many ways to make table view cells

This `cellForRowAtIndexPath` method does a lot more than the one from `ChecklistViewController`. There you just did the following to obtain a new table view cell:

```
UITableViewCell *cell = [tableView
            dequeueReusableCellWithIdentifier:@"ChecklistItem"];
```

But here you have a whole chunk of code to accomplish the same:

```
static NSString *CellIdentifier = @"Cell";

UITableViewCell *cell = [tableView
              dequeueReusableCellWithIdentifier:CellIdentifier];

if (cell == nil) {
  cell = [[UITableViewCell alloc]
                      initWithStyle:UITableViewCellStyleDefault
                    reuseIdentifier:CellIdentifier];
}
```

The call to `dequeueReusableCellWithIdentifier` 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.

There are four ways that you can make table view cells:

1. Using prototype cells. This is a probably the simplest and quickest way. You did this in `ChecklistViewController`.

2. Using static cells. You did this for the Add/Edit Item screen. This is 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 (`cellForRowAtIndexPath` and so on).

3. By hand, what you did above. This is how you were supposed to do it before iOS 5. 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 most of the flexibility.

4. Using a *nib* file. A nib (also known as a XIB) is like a storyboard but it only contains a single view controller or in this case a single custom `UITableViewCell` object. This is very similar to using prototype cells, except that you can do it outside of a storyboard.

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 but 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 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 `dequeueReusableCellWithIdentifier`. 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!

The data model now consists of the `_lists` array from `AllListsViewController` and the `_items` array from `ChecklistViewController`, and the `Checklist` and `ChecklistItem` objects that they respectively contain.

You may have noticed that when you tap the name of a checklist, the Checklist screen slides into view but it currently always shows the same to-do items, regardless of which row you tap on. Each checklist should really have its own list of

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.

▸ Change **ChecklistViewController.h** to the following:

```
#import <UIKit/UIKit.h>
#import "ItemDetailViewController.h"

@class Checklist;

@interface ChecklistViewController : UITableViewController
                                <ItemDetailViewControllerDelegate>

@property (nonatomic, strong) Checklist *checklist;

@end
```

You've added a property for a `Checklist` object, simply named `checklist`. Don't forget the forward declaration `@class Checklist` at the top or Xcode will complain that `ChecklistViewController` doesn't know anything about the `Checklist` object.

▸ At the top of **ChecklistViewController.m**, add an import statement to load the complete definition of the `Checklist` object:

```
#import "Checklist.h"
```

▸ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  self.title = self.checklist.name;
}
```

This changes the title of the screen, which is shown in the navigation bar, to the name of the `Checklist` object.

Now you have to give this `Checklist` object to the `ChecklistViewController` when the segue is performed.

▸ In **AllListsViewController.m**, update `didSelectRowAtIndexPath` to the following:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  Checklist *checklist = _lists[indexPath.row];
```

```
    [self performSegueWithIdentifier:@"ShowChecklist"
                              sender:checklist];
}
```

As before, you use `performSegueWithIdentifier` 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 `prepareForSegue`, which you still need to write.

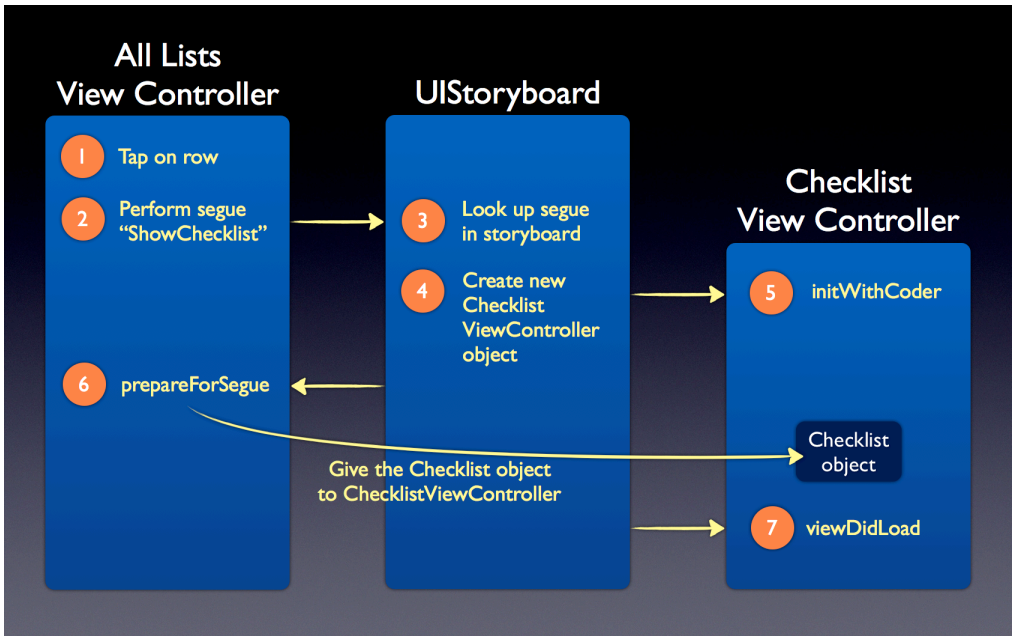
▸ Add the following method below `didSelectRowAtIndexPath`:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"ShowChecklist"]) {
    ChecklistViewController *controller =
                                segue.destinationViewController;
    controller.checklist = sender;
  }
}
```

You've seen this method before. `prepareForSegue` is called by the storyboard right before the segue happens. Here you get a chance to set the properties of the new view controller before it will become visible.

You need to give it 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 before `ChecklistViewController`'s view is loaded, so its `viewDidLoad` method (the one that you just changed) can set the title of the screen accordingly.



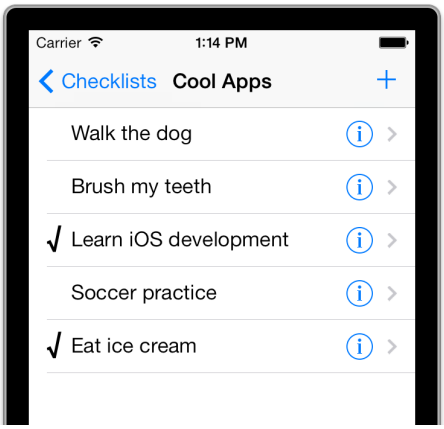
The sequence of events involved in performing a segue

➤ Finally, add an import at the top of **AllListsViewController.m**, so the compiler can find the `ChecklistViewController` object:

```
#import "ChecklistViewController.h"
```

That should do it.

➤ 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 (also known as a **pointer**) to that object, so any changes the user makes to it 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`.

Adding and editing checklists

Let's quickly add the Add Checklist / Edit Checklist screen. This is going to be yet another `UITableViewController`, this time 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, a `UITableViewController` subclass named **ListDetailViewController**.

➤ Change **ListDetailViewController.h** to:

```
#import <UIKit/UIKit.h>

@class ListDetailViewController;
@class Checklist;

@protocol ListDetailViewControllerDelegate <NSObject>

- (void)listDetailViewControllerDidCancel:
                        (ListDetailViewController *)controller;

- (void)listDetailViewController:
                         (ListDetailViewController *)controller
        didFinishAddingChecklist:(Checklist *)checklist;

- (void)listDetailViewController:
                         (ListDetailViewController *)controller
        didFinishEditingChecklist:(Checklist *)checklist;

@end

@interface ListDetailViewController : UITableViewController
                                          <UITextFieldDelegate>

@property (nonatomic, weak) IBOutlet UITextField *textField;
@property (nonatomic, weak) IBOutlet UIBarButtonItem
                                                 *doneBarButton;

@property (nonatomic, weak) id
                    <ListDetailViewControllerDelegate> delegate;

@property (nonatomic, strong) Checklist *checklistToEdit;
```

```
- (IBAction)cancel;
- (IBAction)done;

@end
```

This may seem like a lot of code all at once, but I simply took the contents of **ItemDetailViewController.h** and changed the names. Also, instead of a property for a `ChecklistItem` you're now dealing with a `Checklist`.

➤ To the top of **ListDetailViewController.m** add:

```
#import "Checklist.h"
```

➤ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.checklistToEdit != nil) {
    self.title = @"Edit Checklist";
    self.textField.text = self.checklistToEdit.name;
    self.doneBarButton.enabled = YES;
  }
}
```

➤ And add the `viewWillAppear` method:

```
- (void)viewWillAppear:(BOOL)animated
{
  [super viewWillAppear:animated];

  [self.textField becomeFirstResponder];
}
```

➤ Throw away everything below the `#pragma mark - Table view data source` line and replace it with:

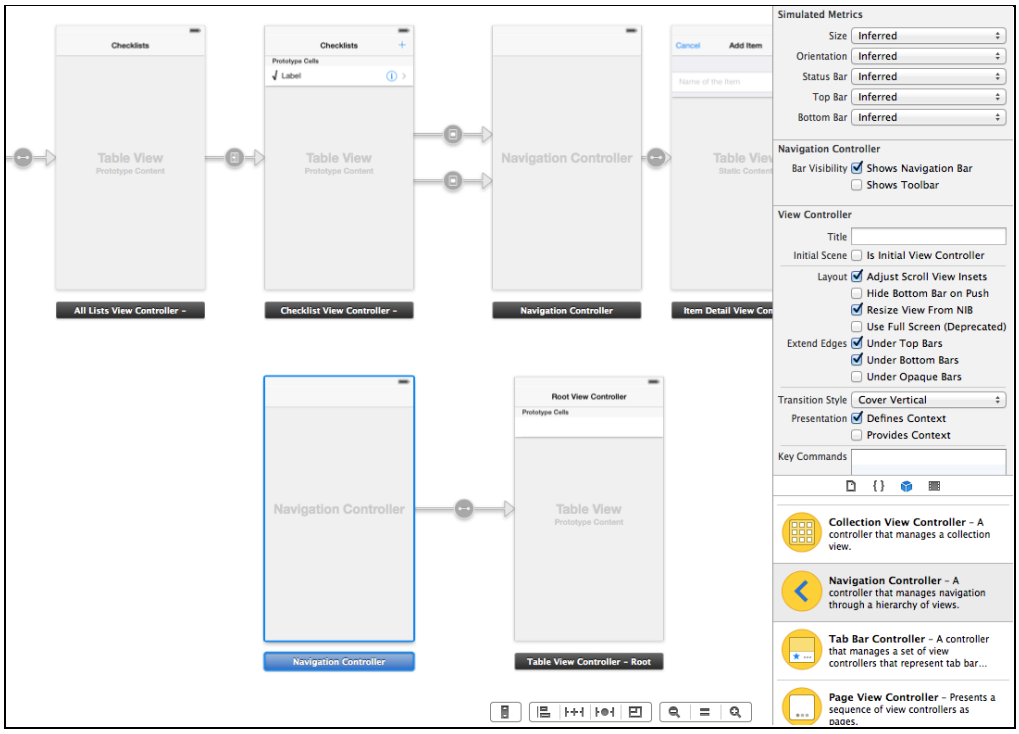
```
- (IBAction)cancel
{
  [self.delegate listDetailViewControllerDidCancel:self];
}

- (IBAction)done
{
  if (self.checklistToEdit == nil) {
    Checklist *checklist = [[Checklist alloc] init];
```

```
    checklist.name = self.textField.text;

    [self.delegate listDetailViewController:self
                           didFinishAddingChecklist:checklist];

  } else {
    self.checklistToEdit.name = self.textField.text;
    [self.delegate listDetailViewController:self
                didFinishEditingChecklist:self.checklistToEdit];
  }
}

- (NSIndexPath *)tableView:(UITableView *)tableView
           willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  return nil;
}

- (BOOL)textField:(UITextField *)theTextField
       shouldChangeCharactersInRange:(NSRange)range
       replacementString:(NSString *)string
{
  NSString *newText = [theTextField.text
                       stringByReplacingCharactersInRange:range
                       withString:string];
  self.doneBarButton.enabled = ([newText length] > 0);
  return YES;
}
```

Again, this is what you did in `ItemDetailViewController` but now for `Checklist` objects instead of `ChecklistItem` objects.

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

› Delete the "Root View Controller" that is attached to the new navigation controller. Interface Builder automatically added this second view controller but you don't need it.

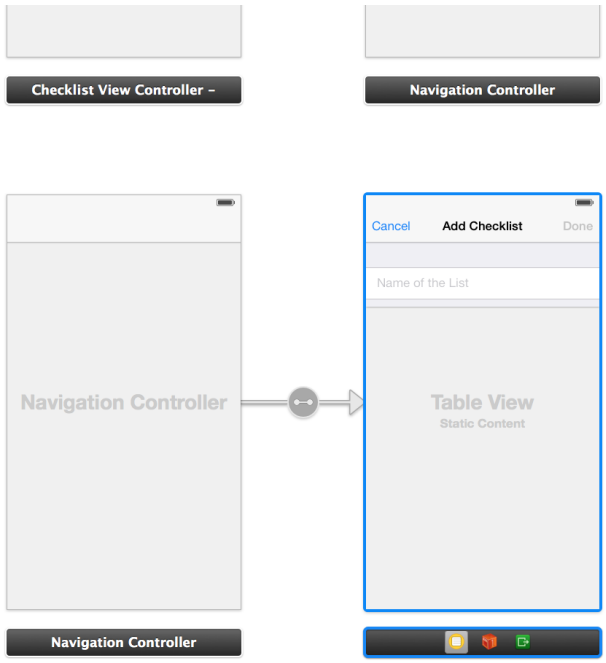
› Select the existing **Item Detail View Controller**. Press ⌘**+D** to create a duplicate. Move the duplicate next to the new navigation controller. The navigation bar with the Cancel and Done buttons has disappeared from this duplicate but that's no problem.

› **Ctrl-drag** from the navigation controller into this second Item Detail View Controller and choose **Relationship Segue – root view controller**. Now it is hooked up again and the navigation bar with the Cancel/Done buttons has reappeared.

› Select the clone of the Item Detail View Controller and go to the **Identity inspector**. Change its class to **ListDetailViewController**.

› Change the navigation bar title to **Add Checklist** and the placeholder text in the text field to **Name of the List**.

This completes the steps for converting this view controller to the Add / Edit Checklist screen:



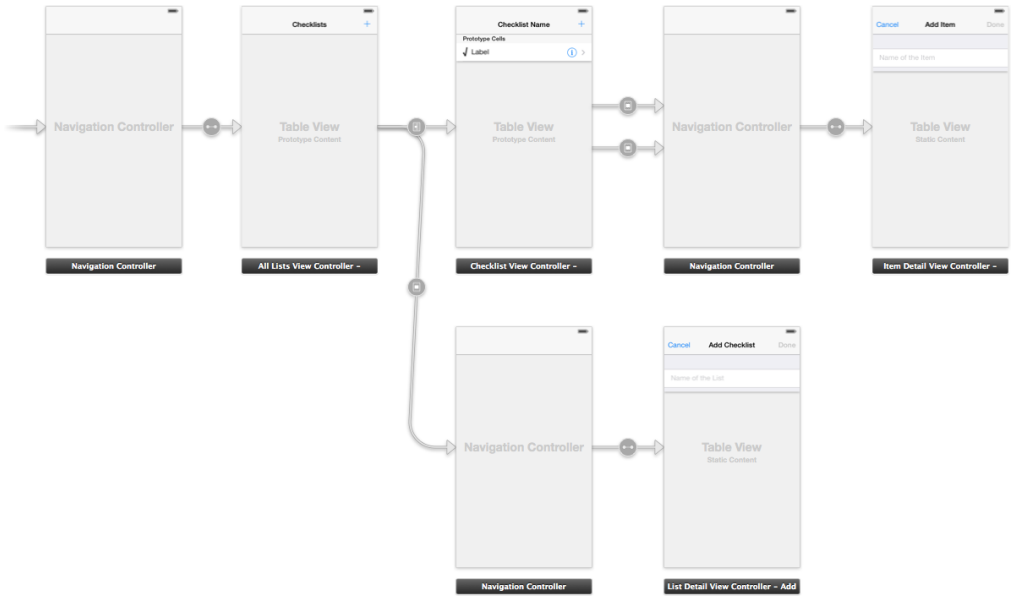
The List Detail View Controller is now hooked up to the new navigation controller

➤ Go to the **All Lists View Controller** and drag a **Bar Button Item** into its navigation bar. Change it into an **Add** button. **Ctrl-drag** from this button to the navigation controller below to add a new **modal** segue.

➤ Click on the new segue and name it **AddChecklist**.

➤ Just so you don't get confused as to which screen does what, change the title of the Checklist View Controller from "Checklists" to **Checklist Name**.

Your storyboard should now look like this:



The full storyboard

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 `@interface` line. You do this in **AllListsViewController.h**:

```
#import <UIKit/UIKit.h>
#import "ListDetailViewController.h"

@interface AllListsViewController : UITableViewController
                                <ListDetailViewControllerDelegate>

@end
```

› In **AllListsViewController.m**, first extend `prepareForSegue` to:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"ShowChecklist"]) {
    ChecklistViewController *controller =
                                segue.destinationViewController;
    controller.checklist = sender;
  }
  else if ([segue.identifier isEqualToString:@"AddChecklist"]) {
```

```
    UINavigationController *navigationController =
                                segue.destinationViewController;

    ListDetailViewController *controller =
                    (ListDetailViewController *)
                            navigationController.topViewController;

    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.m**, implement the following delegate methods.

```
- (void)listDetailViewControllerDidCancel:
                            (ListDetailViewController *)controller
{
  [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)listDetailViewController:
                            (ListDetailViewController *)controller
        didFinishAddingChecklist:(Checklist *)checklist
{
  NSInteger newRowIndex = [_lists count];
  [_lists addObject:checklist];

  NSIndexPath *indexPath = [NSIndexPath
                        indexPathForRow:newRowIndex inSection:0];

  NSArray *indexPaths = @[indexPath];
  [self.tableView insertRowsAtIndexPaths:indexPaths
              withRowAnimation:UITableViewRowAnimationAutomatic];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

```
- (void)listDetailViewController:
                          (ListDetailViewController *)controller
        didFinishEditingChecklist:(Checklist *)checklist
{
  NSInteger index = [_lists indexOfObject:checklist];
  NSIndexPath *indexPath = [NSIndexPath indexPathForRow:index
                                              inSection:0];

  UITableViewCell *cell = [self.tableView
                                cellForRowAtIndexPath:indexPath];
  cell.textLabel.text = checklist.name;

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

None of this code should surprise you. It's exactly what you did before but now for the `ListDetailViewController` and `Checklist` objects. These methods are called when the user presses Cancel or Done inside the new Add/Edit Checklist screen.

➤ Also add the table view data source method that allows the user to delete checklists:

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
  [_lists removeObjectAtIndex:indexPath.row];

  NSArray *indexPaths = @[indexPath];
  [tableView deleteRowsAtIndexPaths:indexPaths
            withRowAnimation:UITableViewRowAnimationAutomatic];
}
```

➤ Run the app. Now you can add new checklists and delete them again, but 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 `accessoryButtonTappedForRowWithIndexPath` method to **AllListsViewController.m**. This method comes from the table view delegate protocol and the name is hopefully obvious enough to guess what it does.

```
- (void)tableView:(UITableView *)tableView
        accessoryButtonTappedForRowWithIndexPath:
                                            (NSIndexPath *)indexPath
{
  UINavigationController *navigationController =
       [self.storyboard instantiateViewControllerWithIdentifier:
                        @"ListNavigationController"];

  ListDetailViewController *controller =
                (ListDetailViewController *)
                          navigationController.topViewController;

  controller.delegate = self;

  Checklist *checklist = _lists[indexPath.row];
  controller.checklistToEdit = checklist;

  [self presentViewController:navigationController animated:YES
                   completion:nil];
}
```

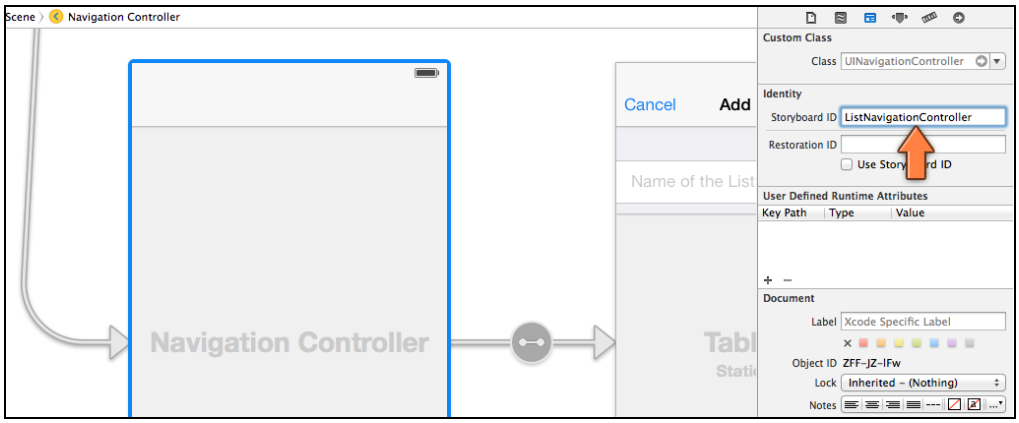
Inside this method you create the view controller object for the Add/Edit Checklist screen and show it ("present" it) on the screen. This is roughly equivalent to what a segue would do behind the scenes. The view controller is embedded in a storyboard and you have to ask the storyboard object to load it.

Where did you get that storyboard object? As it happens, each view controller has a `self.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 call to `instantiateViewControllerWithIdentifier` takes an identifier string, `@"ListNavigationController"`. 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 so that 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 type **ListNavigationController** into the field **Storyboard ID**:



Setting an identifier on the navigation controller

➤ That should do the trick. Now run the app and tap some detail disclosure buttons.

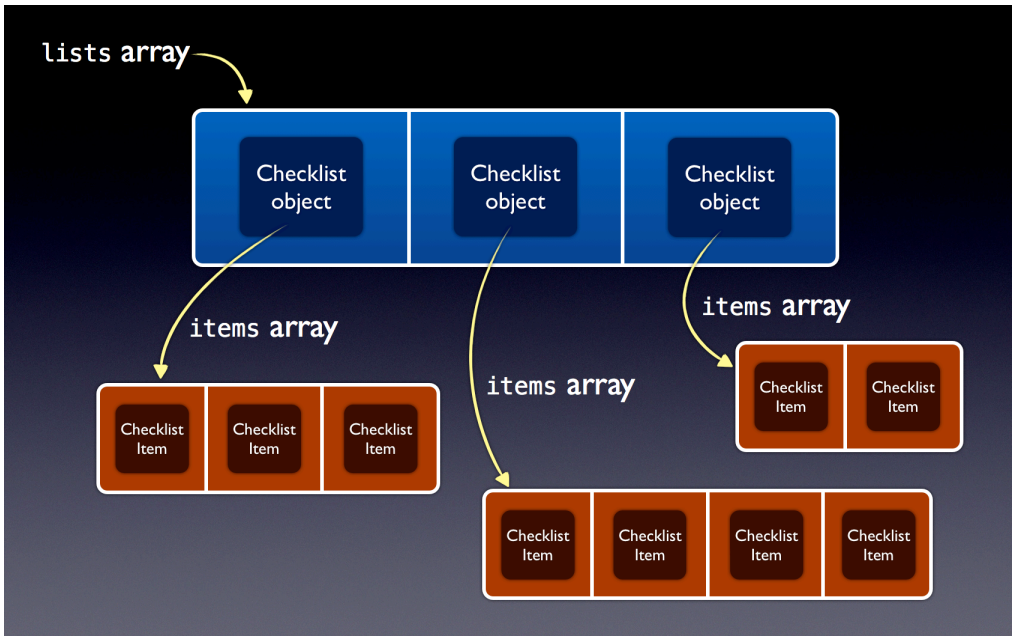
Exercise: Set the **ListNavigationController** identifier on the List Detail View Controller instead and see what happens when you run the app. □

You can find the project files for the app up to this point under **07 - Lists** in the tutorial's Source Code folder.

Putting to-do items into the checklists

This 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 a `_lists` array that contains the `Checklist` objects, but each of these `Checklists` will have its own array of `ChecklistItem` objects.

▸ Add a new property to **Checklist.h**:

```
@property (nonatomic, strong) NSMutableArray *items;
```

▸ Give **Checklist.m** an `init` method:

```
- (id)init
{
  if ((self = [super init])) {
    self.items = [[NSMutableArray alloc] initWithCapacity:20];
  }
  return self;
}
```

The `Checklist` object now contains the array of `ChecklistItem` objects. Initially, that array is empty.

Earlier you fixed `prepareForSegue` in **AllListsViewController.m** so that when you tap on a row in the main screen, the app segues into the `ChecklistViewController` and the `Checklist` object that belongs to that row is passed along.

However, 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.

▸ Make the following changes in **ChecklistViewController.m**:

```
- (NSInteger)tableView:(UITableView *)tableView
            numberOfRowsInSection:(NSInteger)section
{
  return [self.checklist.items count];
}
```

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  . . .

  ChecklistItem *item = self.checklist.items[indexPath.row];

  . . .
}
```

```
- (void)tableView:(UITableView *)tableView
       didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  . . .

  ChecklistItem *item = self.checklist.items[indexPath.row];

  . . .
}
```

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
  [self.checklist.items removeObjectAtIndex:indexPath.row];

  . . .
}
```

```
- (void)itemDetailViewController:
                          (ItemDetailViewController *)controller
       didFinishAddingItem:(ChecklistItem *)item
```

```
{
  NSInteger newRowIndex = [self.checklist.items count];
  [self.checklist.items addObject:item];

  . . .
}
```

```
- (void)itemDetailViewController:
                            (ItemDetailViewController *)controller
        didFinishEditingItem:(ChecklistItem *)item
{
  NSInteger index = [self.checklist.items indexOfObject:item];

  . . .
}
```

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
    . . .

    controller.itemToEdit = self.checklist.items[indexPath.row];

    . . .
}
```

Anywhere it said `_items` you have changed it to say `self.checklist.items` instead.

➤ Delete the following methods from **ChecklistViewController.m**:

- `(NSString *)documentsDirectory`
- `(NSString *)dataFilePath`
- `(void)saveChecklistItems`
- `(void)loadChecklistItems`
- `(id)initWithCoder:(NSCoder *)aDecoder`

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, though. 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 4 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 in a different place.

- Remove the lines that call `saveChecklistItems`.
- Also remove the `_items` instance variable so that this,

```
@implementation ChecklistViewController
{
  NSMutableArray *_items;
}
```

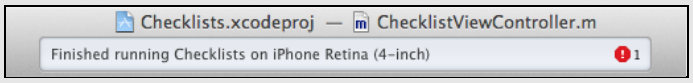
simply becomes again:

```
@implementation ChecklistViewController
```

Since there are no instance variables anymore, the `{ }` brackets are no longer necessary. Note that unlike regular statements, the `@implementation` line never ends in a semicolon!

Xcode errors

When Xcode detects a problem it shows a warning or error icon at the top of the window:



When you fix the problem, this error icon may not immediately go away. Xcode is pretty smart about detecting any changes you make, but it doesn't always pick up on everything. At times that may be a bit confusing. After all, you just fixed the problem but Xcode still complains about it.

Just press Run to launch the app or press ⌘+B to do a build without running the app. If there are truly still errors or warnings then Xcode will tell you.

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 `initWithCoder` method you already put fake `Checklist` objects into the `_lists` array. It's time to add something new to this method.

- At the top of the **AllListsViewController.m** file, add an import:

```
#import "ChecklistItem.h"
```

- Change the `initWithCoder` method to the following:

```
- (id)initWithCoder:(NSCoder *)aDecoder
```

```
{
  if ((self = [super initWithCoder:aDecoder]))
  {
    _lists = [[NSMutableArray alloc] initWithCapacity:20];

    Checklist *list;

    list = [[Checklist alloc] init];
    list.name = @"Birthdays";
    [_lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"Groceries";
    [_lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"Cool Apps";
    [_lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"To Do";
    [_lists addObject:list];

    for (Checklist *list in _lists) {
      ChecklistItem *item = [[ChecklistItem alloc] init];
      item.text = [NSString stringWithFormat:
                                    @"Item for %@", list.name];
      [list.items addObject:item];
    }

  }
  return self;
}
```

Only the highlighted bit is new. This introduces something you haven't seen before in these tutorials: the `for`-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 (`UIButton`,

`ChecklistItem`) and even others can store collections of objects (`NSMutableArray`).

* 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 an `if` with many `else ifs`.

* 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 group functionality (methods) and data (variables) into objects.

* The ability to repeat a set of statements more than once. This is what the `for` statement does. There are several other ways to perform repetitions as well: `while` and `do - while`. 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 these concepts, then you're well on your way to becoming a software developer.

Let's go through that `for` loop line-by-line:

```
for (Checklist *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:

```
ChecklistItem *item = [[ChecklistItem alloc] init];
item.text = [NSString stringWithFormat:
                                  @"Item for %@", list.name];
[list.items addObject:item];
```

This shouldn't be too unfamiliar. 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`-statement will look at the `_lists` array again and sees that there are three more `Checklist` objects in that list. So it puts the next one, Groceries, into the `list` variable and the process repeats. This time the text is "Item for Groceries", which will be 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 an item "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.

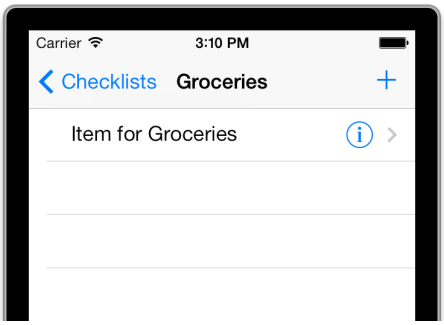
Using loops will often save you a lot of time. You could have written this code as follows:

```
Checklist* list;
ChecklistItem *item;

list = _lists[0];
item = [[ChecklistItem alloc] init];
item.text = @"Item for Birthdays";
[list.items addObject:item];

list = _lists[1];
item = [[ChecklistItem alloc] init];
item.text = @"Item for Groceries";
[list.items addObject:item];

list = _lists[2];
item = [[ChecklistItem alloc] init];
item.text = @"Item for Cool Apps";
[list.items addObject:item];

list = _lists[3];
item = [[ChecklistItem alloc] init];
item.text = @"Item for To Do";
[list.items addObject:item];
```

That's a little repetitive, which is a good sign it's better to use a loop. And what 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.m**, above `initWithCoder`:

```
- (NSString *)documentsDirectory
{
  NSArray *paths = NSSearchPathForDirectoriesInDomains(
                     NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths firstObject];
  return documentsDirectory;
}

- (NSString *)dataFilePath
{
  return [[self documentsDirectory]
            stringByAppendingPathComponent:@"Checklists.plist"];
}

- (void)saveChecklists
{
  NSMutableData *data = [[NSMutableData alloc] init];
  NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
                               initForWritingWithMutableData:data];
  [archiver encodeObject:_lists forKey:@"Checklists"];
  [archiver finishEncoding];
  [data writeToFile:[self dataFilePath] atomically:YES];
```

```
}

- (void)loadChecklists
{
  NSString *path = [self dataFilePath];
  if ([[NSFileManager defaultManager] fileExistsAtPath:path]) {
    NSData *data = [[NSData alloc] initWithContentsOfFile:path];
    NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
                                     initForReadingWithData:data];
    _lists = [unarchiver decodeObjectForKey:@"Checklists"];
    [unarchiver finishDecoding];
  } else {
    _lists = [[NSMutableArray alloc] initWithCapacity:20];
  }
}
```

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.

▸ Change `initWithCoder` to:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    [self loadChecklists];
  }
  return self;
}
```

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.h**:

```
@interface Checklist : NSObject <NSCoding>
```

▸ Add the following methods to **Checklist.m**:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super init])) {
    self.name = [aDecoder decodeObjectForKey:@"Name"];
    self.items = [aDecoder decodeObjectForKey:@"Items"];
  }
  return self;
```

```
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
  [aCoder encodeObject:self.name forKey:@"Name"];
  [aCoder encodeObject:self.items forKey:@"Items"];
}
```

Both the `name` and `items` properties are objects (`NSString` and `NSMutableArray`) so you use `decodeObjectForKey:` and `encodeObject:forKey:` to load and save them.

Important! Do not throw away the regular `init` method from **Checklist.m**! This object needs both `init` and `initWithCoder` to function properly. The regular `init` method is used when the users adds a new checklist to the app, while `initWithCoder` is used to load the existing checklists when the app starts up. Without the regular `init` method the app will crash when you try to add new to-do items to this checklist, because in that case the `self.items` array never is allocated and remains `nil`.

➤ 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 the following error message:

```
*** Terminating app due to uncaught exception 'NSRangeException',
reason: '*** -[NSMutableIndexSet addIndexesInRange:]: Range {2147483647,
1} exceeds maximum index value of NSNotFound - 1'
```

The line where the crash occurred was:

```
[self.tableView insertRowsAtIndexPaths:indexPaths
             withRowAnimation:UITableViewRowAnimationAutomatic];
```

That is a very strange error message and 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 error is that somehow the code manages to load the old file, even though its format is all wrong and no longer corresponds to the

new data model. This puts the table view into a bad state. Any subsequent operations on the table view will cause the 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 big section devoted to debugging techniques in a later tutorial because it's inevitable that you'll introduce bugs in your code and 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.

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: added a new item, deleted an item, or toggled a checkmark. That all used to happen in Checklist View Controller. However, you moved the saving logic into `AllListsViewController`. So how do you make sure changes made in `ChecklistViewController` get saved now?

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 *child-parent dependency* and you've been trying hard to avoid those.

You may think: ah, I could use a delegate for this. True – and if you thought that indeed then 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 started, 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 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. On iOS 4 and better the app will simply be suspended in the background when that happens. There are also 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 wipes the suspended apps 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 and 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 **ChecklistsAppDelegate.m**, you'll see the methods:

```
- (void)applicationDidEnterBackground:
                                    (UIApplication *)application
```

and:

```
- (void)applicationWillTerminate:(UIApplication *)application
```

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 `saveChecklist` 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.

➤ At the top of **ChecklistsAppDelegate.m**, add:

```
#import "AllListsViewController.h"
```

➤ Above `applicationDidEnterBackground`, add this new method:

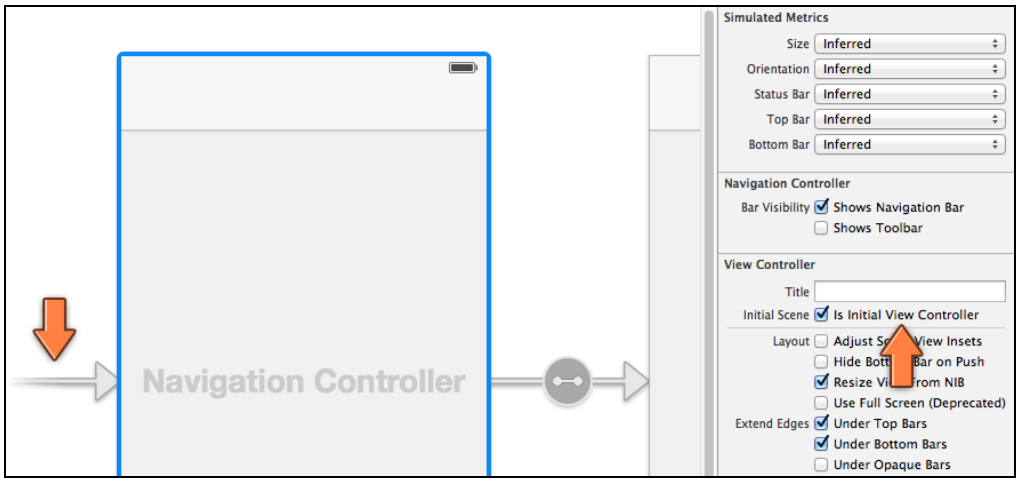
```
- (void)saveData
{
  UINavigationController *navigationController =
       (UINavigationController *)self.window.rootViewController;
  AllListsViewController *controller =
                          navigationController.viewControllers[0];
  [controller saveChecklists];
}
```

➤ Change the `applicationDidEnterBackground` and `applicationWillTerminate` methods to:

```
- (void)applicationDidEnterBackground:
                                    (UIApplication *)application
{
  [self saveData];
}

- (void)applicationWillTerminate:(UIApplication *)application
{
  [self saveData];
}
```

The `saveData` method looks at the `self.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).

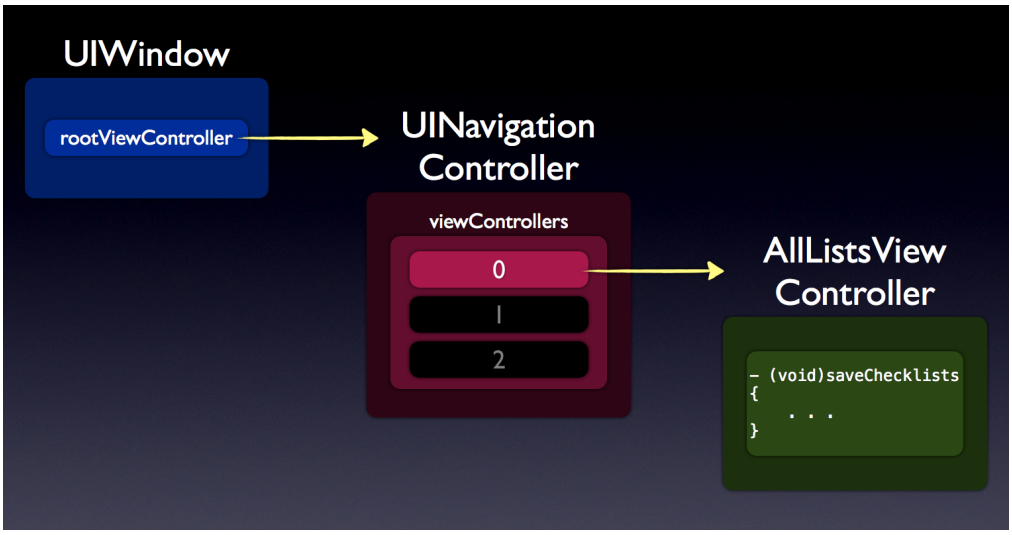
Normally you don't need to do anything with your `UIWindow`, but in this case you ask it for its `rootViewController`. This is the very first view controller from the storyboard, the navigation controller all the way over on the left. You can see this in Interface Builder because this navigation controller has the **Is Initial View Controller** box checked and a big arrow pointing at it:



The left-most navigation controller is the window's root view controller

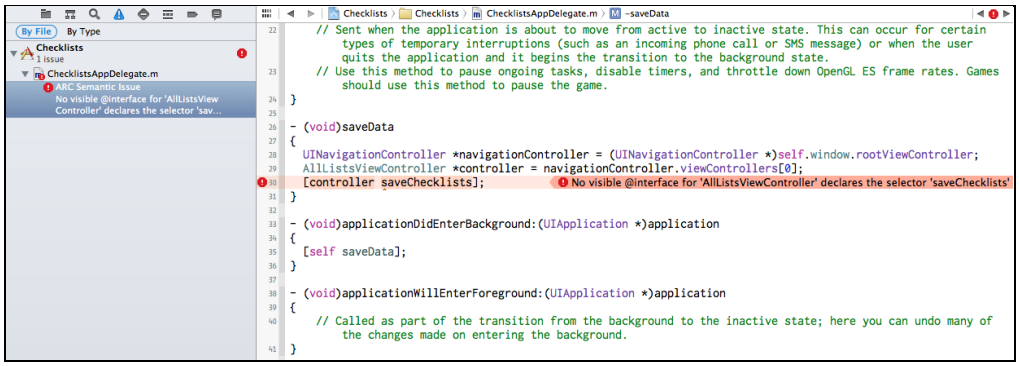
Once you have the navigation controller, you can find the `AllListsViewController` and then call its `saveChecklists` method. Unfortunately, the `UINavigationController` does not have a "rootViewController" property of its own, so you have to look into its `viewControllers` array to find the bottom one.

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 very well be the `ChecklistViewController`. You don't want to send the `saveChecklists` message to that screen – it no longer has a method to handle that message and the app will crash!



From the root view controller to the AllListsViewController

There is a small problem with the changes you've made so far: Xcode gives the error "No visible @interface for 'AllListsViewController' declares the selector 'saveChecklists'". How can this be when you've definitely added the `saveChecklists` method to the `AllListsViewController` object?



Xcode error: you're calling a method on AllListsViewController that it doesn't have

We haven't talked much about the distinction between interface and implementation yet, but what an object shows on the outside is different from what it has on the inside. That's done on purpose because its internals – the so-called *implementation details* – are not interesting to the user of the object. So you hide as much as possible inside the object and only show a few things on the outside.

For example, instance variables are necessary for the implementation only, not for the users of the object. That's the reason you put them in the `@implementation` section and not in the `@interface` section.

Simply put, the .h file is the interface of an object and the .m file is its implementation.

You've only added the `saveChecklists` method to the **AllListsViewController.m** file, inside the `@implementation` section. That means it can be used only by this object itself and no one else. Other objects cannot see this method. It is usually a good idea to hide methods unless other objects need to be able to use them.

To make the `saveChecklists` method accessible to other objects, you need to add its **signature** to the .h file.

› Open **AllListsViewController.h** and add the line:

```
- (void)saveChecklists;
```

The complete .h file now looks like this:

```
#import <UIKit/UIKit.h>
#import "ListDetailViewController.h"

@interface AllListsViewController : UITableViewController
                                <ListDetailViewControllerDelegate>

- (void)saveChecklists;

@end
```

Because you've added `saveChecklists` to the object's `@interface` section, other objects can now use it.

➤ Run the app, add some checklists, add items to those lists, and set some checkmarks. Then press the **Home** button on the simulator to make the app go to the background.

Look inside the app's Documents folder using Finder. There is now 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, first tap the Home button to make the app go into the background and then press Stop. If you don't press Home first, you'll lose your data.

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 there is still code in `AllListsViewController` for loading and saving the Checklists.plist file, that really belongs in the data model as well.

I prefer to create a top-level `DataModel` object for many of my apps. For this app, the `DataModel` object 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, **Objective-C class** template, subclass of **NSObject**. Save as **DataModel**.

➤ Change **DataModel.h** to the following:

```
#import <Foundation/Foundation.h>

@interface DataModel : NSObject

@property (nonatomic, strong) NSMutableArray *lists;

- (void)saveChecklists;

@end
```

You've added a `lists` property and the `saveChecklist` method. `DataModel` will be taking over these responsibilities from `AllListsViewController`.

➤ Inside **DataModel.m**, add the following methods:

```
- (NSString *)documentsDirectory
{
  NSArray *paths = NSSearchPathForDirectoriesInDomains(
                   NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths firstObject];
  return documentsDirectory;
}

- (NSString *)dataFilePath
{
  return [[self documentsDirectory]
            stringByAppendingPathComponent:@"Checklists.plist"];
}

- (void)saveChecklists
{
  NSMutableData *data = [[NSMutableData alloc] init];
  NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
                               initForWritingWithMutableData:data];
  [archiver encodeObject:self.lists forKey:@"Checklists"];
  [archiver finishEncoding];
  [data writeToFile:[self dataFilePath] atomically:YES];
}

- (void)loadChecklists
{
  NSString *path = [self dataFilePath];
  if ([[NSFileManager defaultManager] fileExistsAtPath:path]) {
    NSData *data = [[NSData alloc] initWithContentsOfFile:path];
    NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
                                     initForReadingWithData:data];
    self.lists = [unarchiver decodeObjectForKey:@"Checklists"];
    [unarchiver finishDecoding];
  } else {
    self.lists = [[NSMutableArray alloc] initWithCapacity:20];
  }
}
```

I simply cut these lines out of `AllListsViewController`, so make sure they are no longer in that file. (Note that the array is now a property, so you should access it as `self.lists` instead of `_lists`).

Also remove the `initWithCoder` method from **AllListsViewController.m**.

➤ Also add an init method to **DataModel.m**:

```
- (id)init
{
  if ((self = [super init])) {
    [self loadChecklists];
  }
  return self;
}
```

This makes sure that, as soon as the `DataModel` object is created, it will attempt to load Checklists.plist.

➤ Change **AllListsViewController.h** to:

```
#import <UIKit/UIKit.h>
#import "ListDetailViewController.h"

@class DataModel;

@interface AllListsViewController : UITableViewController
                                <ListDetailViewControllerDelegate>

@property (nonatomic, strong) DataModel *dataModel;

@end
```

You have removed the `saveChecklists` method and added the `dataModel` property.

You should already have removed the `documentsDirectory`, `dataFilePath`, `saveChecklists` and `loadChecklists` methods from **AllListsViewController.m**.

➤ Also remove the `_lists` instance variable.

➤ Add an import for the new `DataModel` object:

```
#import "DataModel.h"
```

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.

➤ Make the following changes to `AllListsViewController`:

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  return [self.dataModel.lists count];
```

```
}
```

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  . . .

  Checklist *checklist = self.dataModel.lists[indexPath.row];

  . . .
}
```

```
- (void)tableView:(UITableView *)tableView
      didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  Checklist *checklist = self.dataModel.lists[indexPath.row];

  . . .
}
```

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
  [self.dataModel.lists removeObjectAtIndex:indexPath.row];

  . . .
}
```

```
- (void)tableView:(UITableView *)tableView
          accessoryButtonTappedForRowWithIndexPath:
                                        (NSIndexPath *)indexPath
{
  . . .

  Checklist *checklist = self.dataModel.lists[indexPath.row];

  . . .
```

```
}
```

```
- (void)listDetailViewController:
                            (ListDetailViewController *)controller
        didFinishAddingChecklist:(Checklist *)checklist
{
  NSInteger newRowIndex = [self.dataModel.lists count];
  [self.dataModel.lists addObject:checklist];

  . . .
}
```

```
- (void)listDetailViewController:
                            (ListDetailViewController *)controller
        didFinishEditingChecklist:(Checklist *)checklist
{
  NSInteger index = [self.dataModel.lists
                                        indexOfObject:checklist];
  . . .
}
```

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 its `self.dataModel` property.

But where does this `DataModel` object get created? There is no place in the code that currently does `[[DataModel alloc] init]`.

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 all the view controllers that need to use it.

▸ In **ChecklistsAppDelegate.m**, first add the requisite import:

```
#import "DataModel.h"
```

▸ Add a new instance variable:

```
@implementation ChecklistsAppDelegate
{
  DataModel *_dataModel;
```

```
}
```

▸ Simplify the `saveData` method to just this:

```
- (void)saveData
{
  [_dataModel saveChecklists];
}
```

If you run the app now, it will crash once you add a new checklist because `_dataModel` is still `nil`. The best place to create the `DataModel` instance is in the `application:didFinishLaunchingWithOptions:` method, which gets called as soon as the app starts up:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  _dataModel = [[DataModel alloc] init];

  UINavigationController *navigationController =
       (UINavigationController *)self.window.rootViewController;

  AllListsViewController *controller =
                         navigationController.viewControllers[0];

  controller.dataModel = _dataModel;

  return YES;
}
```

First this creates the `DataModel` object and then it finds the `AllListsViewController` (by looking in the storyboard) in order to set its `dataModel` property.

▸ 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.

Using NSUserDefaults to remember stuff

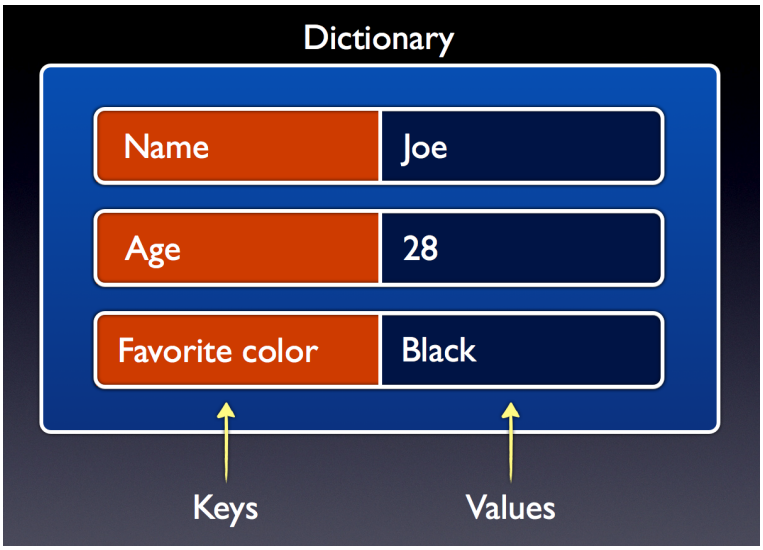
You now have an app that lets you create checklists and add to-do items to those lists. All of this data is saved to long-term storage so even if the app gets terminated, nothing is lost. There are some user interface improvements you can make, though.

Imagine the user is on the Birthdays checklist and presses the Home button to switch to another app. The Checklists app is now suspended. Suppose that at some point the app gets terminated. When the user reopens the app 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 you 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 `NSUserDefaults` object.

`NSUserDefaults` 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 Objective-C are handled by the `NSDictionary` and `NSMutableDictionary` objects. 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. This 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.

`NSUserDefaults` isn't a true dictionary, but it acts like one. When you insert new values into `NSUserDefaults`, 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 `NSUserDefaults`, 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:

- On the segue from the main screen (`AllListsViewController`) to the checklist screen (`ChecklistViewController`), you write the row index of the selected checklist into `NSUserDefaults`. This is how you'll remember which checklist was selected. You could have saved the name of the checklist instead of the row index, but what would happen then if two checklists have the same name? Unlikely, but not impossible. Using the row index guarantees that you'll always select the proper one.
- When the user presses the back button to return to the main screen, you have to remove this value from `NSUserDefaults` 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 or a positive number (unless you use a huge number such as 1000000; 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, so that makes it easy to spot during debugging.
- If the app starts up and the value from `NSUserDefaults` isn't -1, then 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.m**, change `didSelectRowAtIndexPath` to the following:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [[NSUserDefaults standardUserDefaults]
              setInteger:indexPath.row forKey:@"ChecklistIndex"];

  Checklist *checklist = self.dataModel.lists[indexPath.row];

  [self performSegueWithIdentifier:@"ShowChecklist"
                            sender:checklist];
}
```

In addition to what this method did before, you now store the index of the selected row into `NSUserDefaults` under the key "ChecklistIndex".

To recognize whether the user presses the back button on the navigation bar, you have to set a delegate for 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 **AllListsViewController.h** `@interface` line:

```
@interface 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.m**:

```
- (void)navigationController:
                   (UINavigationController *)navigationController
       willShowViewController:(UIViewController *)viewController
       animated:(BOOL)animated
{
  if (viewController == self) {
    [[NSUserDefaults standardUserDefaults] setInteger:-1
                                               forKey:@"ChecklistIndex"];
  }
}
```

This method is called whenever the navigation controller will slide to a new screen. If the back button was pressed, then the new view controller is `AllListsViewController` itself and you set the "ChecklistIndex" value in `NSUserDefaults` to -1, meaning that no checklist is currently selected.

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.m**:

```
- (void)viewDidAppear:(BOOL)animated
{
  [super viewDidAppear:animated];

  self.navigationController.delegate = self;

  NSInteger index = [[NSUserDefaults standardUserDefaults]
                                integerForKey:@"ChecklistIndex"];

  if (index != -1) {
    Checklist *checklist = self.dataModel.lists[index];

    [self performSegueWithIdentifier:@"ShowChecklist"
                              sender:checklist];
  }
```

```
}
```

This method is called after the view controller has become visible.

First, the view controller makes itself the delegate for the navigation controller and then checks `NSUserDefaults` whether it has to perform the segue.

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 `Checklist` object into the `sender` parameter of `performSegueWithIdentifier`.

Note that `!=` means: not equal. It is the opposite of the `==` operator. (Some languages use `<>` for not equal but that won't work in Objective-C.)

> I'm actually guilty of a bit of trickery 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 just once when the app starts, so why did you put this logic in `viewDidAppear`?
>
> The very first time that `AllListsViewController`'s screen becomes visible you do not want the `willShowViewController` delegate method to be called, 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.
>
> When the user presses the back button, the navigation controller will call `willShowViewController` before `viewDidAppear`. Because the value of "ChecklistIndex" will now always be -1, `viewDidAppear` does not trigger a segue again.
>
> 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 worry about it. Let it sink in and soon enough it will all start to make sense. Even better, probe around in the code. Change things around to see what the effect is. That's the quickest way to learn!

▸ Run the app and go to a checklist screen. Press the simulator's Home button, followed by Stop to quit the app.

Tip: You need to press the Home button because `NSUserDefaults` may not immediately save its settings to disk and therefore you may lose your changes if you kill the app from within Xcode.

▸ Run the app again and you'll notice that Xcode immediately switches to the screen where you were last at. Cool, huh!

➤ Now do the following: Stop the app and delete it from the Simulator. You can either reset the whole simulator from its menu (**iOS Simulator → Reset Contents and Settings**) or hold down the app icon until it starts to wiggle and then delete it just as you would on your iPhone.

Then run the app again from within Xcode and watch it crash:

```
*** Terminating app due to uncaught exception 'NSRangeException',
reason: '*** -[__NSArrayM objectAtIndex:]: index 0 beyond bounds for
empty array'
```

The app crashes in `viewDidAppear` on the line:

```
Checklist *checklist = self.dataModel.lists[index];
```

What's going on here? Apparently the value of the `index` variable is 0, even though there should be nothing in `NSUserDefaults` yet because this is a fresh install of the app. It didn't write anything in the "ChecklistIndex" key yet.

It turns out that `NSUserDefaults`'s `integerForKey` 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 `NSUserDefaults` returns -1 if the "ChecklistIndex" key isn't set, because to this app -1 means: show the main screen instead of a specific checklist. Fortunately, `NSUserDefaults` 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 above `init` inside **DataModel.m**:

```
- (void)registerDefaults
{
  NSDictionary *dictionary = @{ @"ChecklistIndex" : @-1 };

  [[NSUserDefaults standardUserDefaults]
                                  registerDefaults:dictionary];
}
```

This creates a new `NSDictionary` object and adds the value -1 for the key "ChecklistIndex". `NSUserDefaults` will use the values from this dictionary if you ask it for a key but it cannot find anything under that key.

➤ Change `init` to call this new method:

```
- (id)init
{
```

```
  if ((self = [super init])) {
    [self loadChecklists];
    [self registerDefaults];
  }
  return self;
}
```

➤ 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 `NSUserDefaults` throughout the code. In fact, lets move all of the `NSUserDefaults` stuff into `DataModel`.

➤ Add the following methods to the bottom of **DataModel.m**, before `@end`:

```
- (NSInteger)indexOfSelectedChecklist
{
  return [[NSUserDefaults standardUserDefaults]
                                integerForKey:@"ChecklistIndex"];
}

- (void)setIndexOfSelectedChecklist:(NSInteger)index
{
   [[NSUserDefaults standardUserDefaults]
                     setInteger:index forKey:@"ChecklistIndex"];
}
```

You're doing this so the rest of the code won't have to worry about `NSUserDefaults`. The other objects just have to call the proper methods on `DataModel`.

Hiding implementation details is an important Object-Oriented Programming principle. If you decide later that you want to store these settings somewhere else, for example in a database, 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.

You need to add these method names to **DataModel.h** too, otherwise the other objects cannot use them.

➤ Add the method signatures to **DataModel.h**:

```
- (NSInteger)indexOfSelectedChecklist;
- (void)setIndexOfSelectedChecklist:(NSInteger)index;
```

➤ Update the code in **AllListsViewController.m** to use these new methods:

```
- (void)viewDidAppear:(BOOL)animated
{
```

```
  [super viewDidAppear:animated];

  self.navigationController.delegate = self;

  NSInteger index = [self.dataModel indexOfSelectedChecklist];
  if (index != -1) {
    Checklist *checklist = self.dataModel.lists[index];
    [self performSegueWithIdentifier:@"ShowChecklist"
                              sender:checklist];
  }
}
```

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [self.dataModel setIndexOfSelectedChecklist:indexPath.row];

  Checklist *checklist = self.dataModel.lists[indexPath.row];

  [self performSegueWithIdentifier:@"ShowChecklist"
                            sender:checklist];
}
```

```
- (void)navigationController:
                  (UINavigationController *)navigationController
       willShowViewController:(UIViewController *)viewController
       animated:(BOOL)animated
{
  if (viewController == self) {
    [self.dataModel setIndexOfSelectedChecklist:-1];
  }
}
```

▸ 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 press the Home button, the new checklist and its item were not saved to Checklists.plist. However, there is a (small) chance that

`NSUserDefaults` 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).

`NSUserDefaults` will save its changes at indeterminate times so it could have saved before you terminated the app. (This is especially true when you implement local notifications later in this tutorial, where you force `NSUserDefaults` to save its changes every time you add a new to-do item. Then the app is guaranteed to crash at this point.)

➤ Run the app again and – if you're lucky? – it will crash with:

```
Checklists[1124:707] *** Terminating app due to uncaught exception
'NSRangeException', reason: '*** -[__NSArrayM objectAtIndex:]: index 1
beyond bounds [0 .. 0]'
```

The problem is that `NSUserDefaults` and the contents of Checklists.plist are out-of-sync. `NSUserDefaults` 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. Under normal circumstances the user would press the Home button at some point and as the app goes into the background it will save both Checklists.plist and `NSUserDefaults` and everything is in sync again. However, the OS can always decide to terminate the app and then this 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 get because often you have no idea what the user did to make it happen. It's good to stick to the practice of *defensive programming*, to 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 `viewDidAppear` to:

```
- (void)viewDidAppear:(BOOL)animated
{
  [super viewDidAppear:animated];

  self.navigationController.delegate = self;

  NSInteger index = [self.dataModel indexOfSelectedChecklist];

  if (index >= 0 && index < [self.dataModel.lists count]) {

    Checklist *checklist = self.dataModel.lists[index];
```

```
    [self performSegueWithIdentifier:@"ShowChecklist"
                               sender:checklist];
  }
}
```

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 will prevent `self.dataModel.lists[index]` from asking for an object 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 that the app doesn't 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, then 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 `NSUserDefaults` 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 in the book *iOS 6 by Tutorials*.

The first-run experience

Let's use `NSUserDefaults` 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 the screen 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 `NSUserDefaults` whether this is the first time the user runs the app. If it is, then you create a new `Checklist` object. You can perform all of this logic inside `DataModel`.

▸ Add the following import to the top of **DataModel.m**:

```
#import "Checklist.h"
```

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:

```
- (void)registerDefaults
{
  NSDictionary *dictionary = @{
    @"ChecklistIndex" : @-1,
    @"FirstTime" : @YES
    };

  [[NSUserDefaults standardUserDefaults]
                                    registerDefaults:dictionary];
}
```

"FirstTime" acts as a boolean value because it's either yes or no (true or false). The value of "FirstTime" needs to be `YES` if this is the first time the app runs after a fresh install.

Primitive values vs. objects

Dictionaries cannot contain primitive values such as `int` and `BOOL`, only objects. The same thing goes for arrays. If you want to put an `int` or `BOOL` value into a dictionary or array, you have to convert it into a so-called `NSNumber` object first.

I have briefly mentioned the difference between primitive datatypes and objects a few times before. In some programming languages everything is an object; in Objective-C *almost* everything is an object. There is a cost associated with using objects and for certain simple operations, such as doing arithmetic with whole numbers, it's easier and faster to do this with primitive values instead.

You can tell primitive types and objects apart by the `*` that follows their name. Only objects have this asterisk. In addition, you do not use `[[alloc] init]` to create primitive values.

> Sometimes you need to convert between primitive types and objects. To put an `int` or `BOOL` value into a dictionary, you need to stuff it into an `NSNumber` object first. You do that with the notation `@value`, such as `@-1` or `@YES`, or by calling the method `[NSNumber numberWithInt:]` or `[NSNumber numberWithBool:]`.
>
> The other way around is possible too: to get an integer value out of an `NSNumber`, you'd do `[number intValue]`. For a `BOOL` that is `[number boolValue]`.
>
> If you're still confused about the difference between primitive values and objects, then rest assured, a more detailed explanation is forthcoming in the next tutorial.

➤ Still in **DataModel.m**, add the `handleFirstTime` method above `init`:

```
- (void)handleFirstTime
{
  BOOL firstTime = [[NSUserDefaults standardUserDefaults]
                                      boolForKey:@"FirstTime"];

  if (firstTime) {

    Checklist *checklist = [[Checklist alloc] init];
    checklist.name = @"List";

    [self.lists addObject:checklist];
    [self setIndexOfSelectedChecklist:0];

    [[NSUserDefaults standardUserDefaults] setBool:NO
                                            forKey:@"FirstTime"];
  }
}
```

Here you check `NSUserDefaults` for the value of the "FirstTime" key. Interestingly enough, you can simply forget about the whole `NSNumber` thing and ask the `NSUserDefaults` directly for a boolean value. Converting to an `NSNumber` was only necessary when you registered the defaults. (That's because `NSUserDefaults` isn't a true dictionary object but only acts like one.)

If the "FirstTime" value is `YES`, 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 call `setIndexOfSelectedChecklist` to make sure the app will automatically segue to this new checklist in `AllListsViewController`'s `viewDidAppear`. Finally, you set "FirstTime" to `NO`, so this bit of code won't be executed again the next time the app starts up.

➤ Call this new method from `init`:

```
- (id)init
{
  if ((self = [super init])) {
    [self loadChecklists];
    [self registerDefaults];
    [self handleFirstTime];
  }
  return self;
}
```

▸ Remove the app from the Simulator and run it again from Xcode. Because it's the first time you run the app (at least from the app's perspective), it will automatically create a new checklist named List and switch to it.

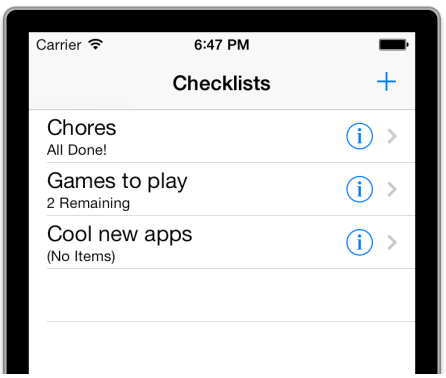
You can find the project files for the app up to this point under **09 - NSUserDefaults** in the tutorial's Source Code folder.

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 name to **Checklist.h**:

```
- (int)countUncheckedItems;
```

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.

➤ Add the implementation of the `countUncheckedItems` method to **Checklist.m**, including an import for `ChecklistItem` or else Xcode won't let you call `item.checked` on the item objects.

```
#import "ChecklistItem.h"

. . .

- (int)countUncheckedItems
{
  int count = 0;
  for (ChecklistItem *item in self.items) {
    if (!item.checked) {
      count += 1;
    }
  }
  return count;
}
```

This method loops through the `ChecklistItem` objects from the `items` array. If the `item` object has its `checked` property set to `NO`, you increment the local variable `count` by 1. When you've looked at all the objects, you return the value of this count to the caller.

Remember that the `!` operator negates the result. So if `item.checked` is `YES`, then `!item.checked` will make it `NO`. You should read it as "if not `item.checked`".

➤ Go to **AllListsViewController.m** and change `cellForRowAtIndexPath` to:

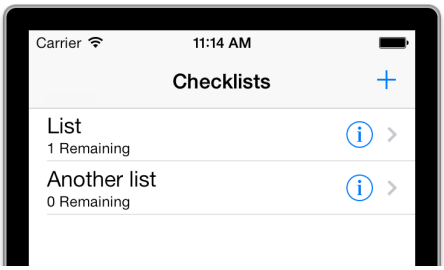
```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"Cell";

  UITableViewCell *cell = [tableView
              dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[UITableViewCell alloc]
                      initWithStyle:UITableViewCellStyleSubtitle
                    reuseIdentifier:CellIdentifier];
  }

  Checklist *checklist = self.dataModel.lists[indexPath.row];
```

```
  cell.textLabel.text = checklist.name;
  cell.accessoryType =
                UITableViewCellAccessoryDetailDisclosureButton;

  cell.detailTextLabel.text = [NSString stringWithFormat:
              @"%d Remaining", [checklist countUncheckedItems]];

  return cell;
}
```

Most of the code stays the same, except you now use `UITableViewCellStyleSubtitle` instead of `UITableViewCellStyleDefault`. 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.

You call the `countUncheckedItems` method on the `Checklist` object and put the count into a new string that you place into the `detailTextLabel`.

➤ 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.

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 is 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, 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 return? It would simply set a new text on the cell's `detailTextLabel` in all cases.

This 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.m** and add the `viewWillAppear` method to do the following:

```
- (void)viewWillAppear:(BOOL)animated
{
  [super viewWillAppear:animated];

  [self.tableView reloadData];
}
```

Don't confuse this method with `viewDidAppear`. The difference is in the verb: *will* versus *did*. `viewWillAppear` is called before `viewDidAppear`.

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 (a-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. When people talk about "the Facebook API" or "the Google API", then 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 `cellForRowAtIndexPath` 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 and thanks to the call to `reloadData` the app will update all of the table cells, including the `detailTextLabels`.

Reloading all of the cells may be a little overkill but in this situation you can get away with it. It's unlikely the All Lists screen will contain many rows (say, less than 100) 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 `cellForRowAtIndexPath` to:

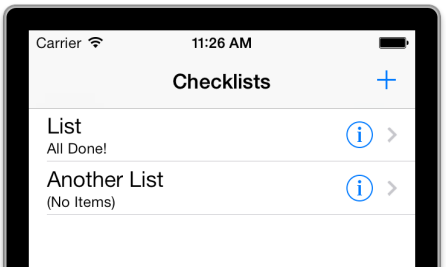
```
int count = [checklist countUncheckedItems];
if (count == 0) {
  cell.detailTextLabel.text = @"All Done!";
} else {
  cell.detailTextLabel.text = [NSString stringWithFormat:
                                       @"%d Remaining", count];
}
```

Exercise: Now update the label to say "No Items" when the list is empty. □

Answer:

```
int 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 = [NSString stringWithFormat:
                                       @"%d Remaining", count];
}
```

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. So you need to look at the total number of items as well 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!"

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 list.

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 `didFinishAddingChecklist` and `didFinishEditingChecklist`.

➤ Change these methods to the following:

```
- (void)listDetailViewController:
                            (ListDetailViewController *)controller
        didFinishAddingChecklist:(Checklist *)checklist
{
  [self.dataModel.lists addObject:checklist];

  [self.dataModel sortChecklists];
  [self.tableView reloadData];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

```
- (void)listDetailViewController:
                            (ListDetailViewController *)controller
        didFinishEditingChecklist:(Checklist *)checklist
{
  [self.dataModel sortChecklists];
  [self.tableView reloadData];

  [self dismissViewControllerAnimated:YES 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 `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 held 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.

➤ Add its signature to **DataModel.h**:

```
- (void)sortChecklists;
```

➤ And the full implementation in **DataModel.m**:

```
- (void)sortChecklists
{
  [self.lists sortUsingSelector:@selector(compare:)];
}
```

The datatype of the `self.lists` property is `NSMutableArray`. This object has a `sortUsingSelector` method that is really easy to use. A **selector** is the name of a method. Here you tell the `lists` array that it should be sorted using the `compare:` method. This method is not defined on the array itself but on the objects it contains, the `Checklists`.

The sort algorithm will call `[Checklist compare]` to see how the `Checklist` objects relate to one another. Because the sorting algorithm doesn't really know anything about your `Checklist` objects – or what it means for one `Checklist` to come before another – you have to help it out by providing this method.

➤ Add the `compare` method to **Checklist.m**:

```
- (NSComparisonResult)compare:(Checklist *)otherChecklist
{
```

```
  return [self.name localizedStandardCompare:
                                          otherChecklist.name];
}
```

That's all you need to do. To compare two `Checklist` objects, you're only looking at the name of this `Checklist` object versus the name of the `otherChecklist` object. The `name` property is an `NSString`, which already has a very convenient comparison method, `localizedStandardCompare`.

This method will compare the two `name` objects 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.

So `NSMutableArray`'s `sortWithSelector` method will repeatedly ask one `Checklist` object how it compares to another `Checklist` object and then shuffle them around until the array is sorted. Inside that `Checklist` object's `compare` method, you simply compare the `name` properties of the two objects. If you wanted to sort on other criteria all you have to do is change the compare method.

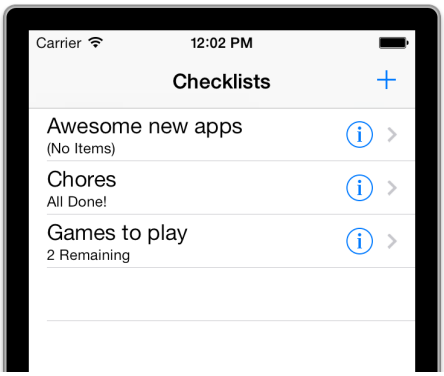
Dynamic method name resolution (using selectors)

If you're wondering why you didn't add the signature for the `compare:` method to **Checklist.h**, then here's a little secret: you could do this but it's not necessary.

If you were calling `[checklist compare]` directly in your code then you would indeed need to declare this method in the Checklist.h file. But here you're using a *selector*, which will resolve the method name at **runtime** (i.e. when the app is running in the Simulator or on the device) rather than at compile-time.

For this type of dynamic method name resolution you don't need to add methods to the .h file, as the .h file isn't used for this. It's also a little more dangerous: you can call a selector on an object that doesn't exist. You've already seen that this makes the app crash with an "unrecognized selector sent to instance xxx" error message.

➤ Run the app and add some new checklists. Change their names and notice that the list is always sorted alphabetically.

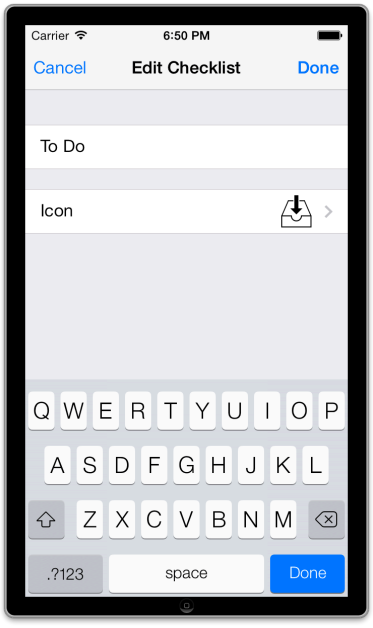


New checklists are always sorted alphabetically

Adding icons to the checklists

Because I can't get enough of view controllers and delegates, let's add a new setting to the `Checklist` object that lets you choose an icon. I really want to cement these principles in your mind.

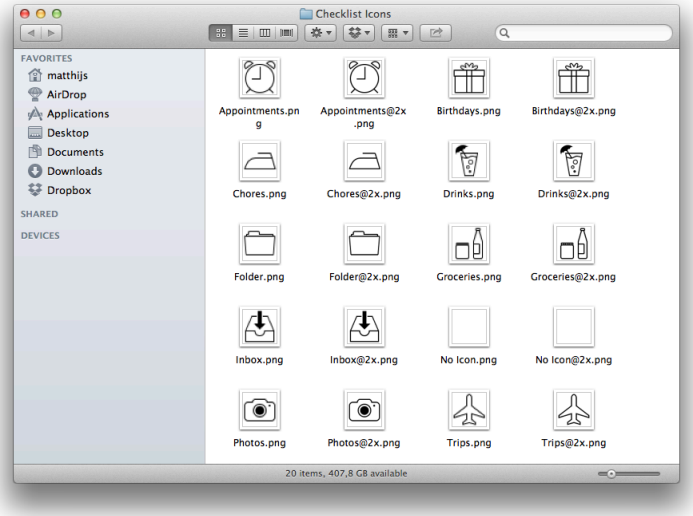
When you're done, the Add/Edit Checklist screen will look like this:



You can assign an icon to a checklist

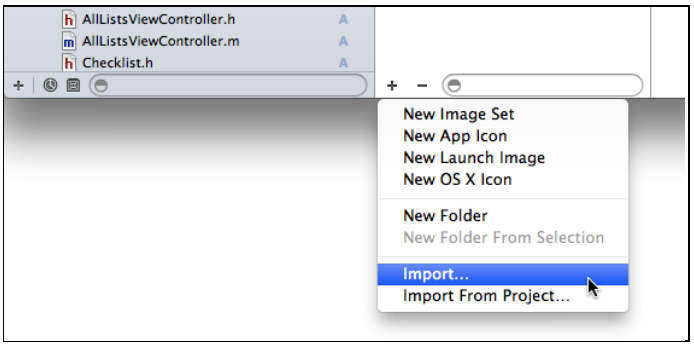
You will add a row to the Add/Edit Checklist screen that opens a new screen that lets you pick an icon. This icon picker is a new view controller. You won't show it modally this time but push it on the navigation stack so it slides into the screen.

The Resources folder for this tutorial contains a folder named **Checklist Icons** with a selection of PNG images that depict different categories.



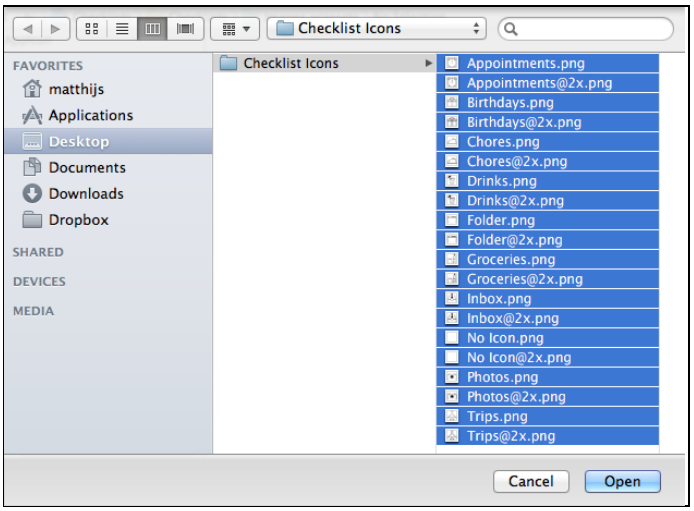
The various checklist icon images

› Add the images from this folder to the asset catalog. Select **Image.xcassets** in the project navigator, click the **+** button at the bottom and choose **Import...**



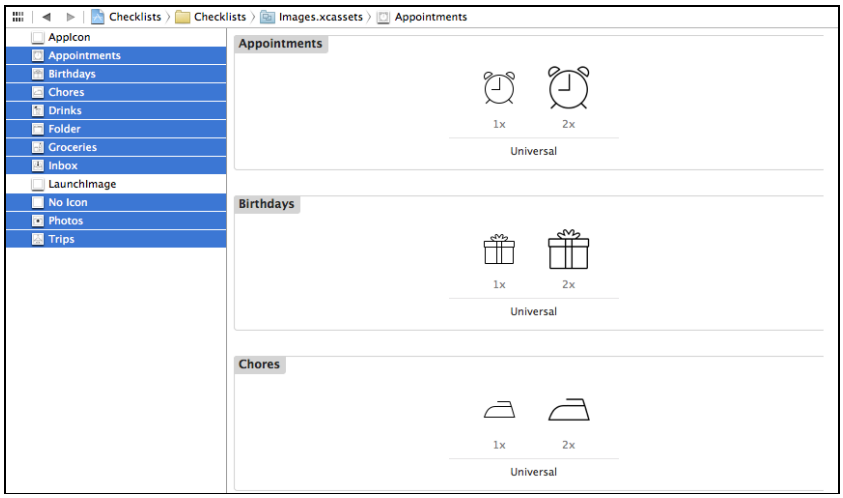
Choosing the option to import new images into the asset catalog

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

Notice that each image comes with a 1x version for low-resolution devices and a 2x version for devices with Retina screens. As I pointed out in the previous tutorial you only need the 1x graphics for apps that are universal or iPad-only (or if you want to support iOS versions before 7.0), but it won't hurt to include them either.

➤ Add the following property to **Checklist.h**:

```
@property (nonatomic, copy) NSString *iconName;
```

➤ Inside **Checklist.m**, extend `initWithCoder` and `encodeWithCoder` to respectively load and save this icon name in the Checklists.plist file:

```
- (id)initWithCoder:(NSCoder *)aDecoder
```

```
{
  if ((self = [super init])) {
    self.name = [aDecoder decodeObjectForKey:@"Name"];
    self.items = [aDecoder decodeObjectForKey:@"Items"];
    self.iconName = [aDecoder decodeObjectForKey:@"IconName"];
  }
  return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
  [aCoder encodeObject:self.name forKey:@"Name"];
  [aCoder encodeObject:self.items forKey:@"Items"];
  [aCoder encodeObject:self.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 this object; otherwise it won't get saved to the plist file.

➤ Just for testing, update the `init` method to the following:

```
- (id)init
{
  if ((self = [super init])) {
    self.items = [[NSMutableArray alloc] initWithCapacity:20];
    self.iconName = @"Appointments";
  }
  return self;
}
```

This will give all 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 the icon.

➤ Change `cellForRowAtIndexPath` in **AllListsViewController.m** to put the icon into the table view cell:

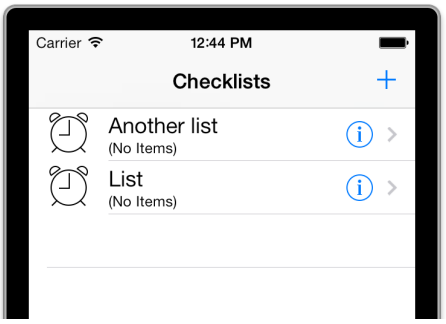
```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  . . .

  cell.imageView.image = [UIImage
                                imageNamed:checklist.iconName];

  return cell;
```

```
}
```

Cells using the standard type `UITableViewCellStyleSubtitle` 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 uninstall the app from the Simulator because you've modified the file format again (you added the "IconName" in `init/encodeWithCoder`). 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 now have an icon

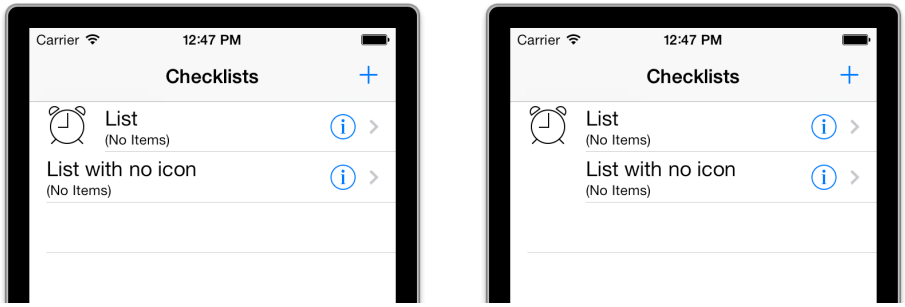
Satisfied that this works, you can now change `Checklist`'s `init` to give each `Checklist` object an icon named "No Icon" by default.

➤ In **Checklist.m**, change the `init` method to:

```
- (id)init
{
  if ((self = [super init])) {
    self.items = [[NSMutableArray alloc] initWithCapacity:20];
    self.iconName = @"No Icon";
  }
  return self;
}
```

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 `self.iconName` to `nil` instead, then the image view in the table view cell would remain empty and the text would align with the left margin of the screen. But 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 file for a `UITableViewController` subclass to the project. Name it **IconPickerViewController**.

➤ Change **IconPickerViewController.h** to:

```
#import <UIKit/UIKit.h>

@class IconPickerViewController;

@protocol IconPickerViewControllerDelegate <NSObject>

- (void)iconPicker:(IconPickerViewController *)picker
                              didPickIcon:(NSString *)iconName;

@end

@interface IconPickerViewController : UITableViewController

@property (nonatomic, weak) id
                    <IconPickerViewControllerDelegate> delegate;

@end
```

➤ In **IconPickerViewController.m**, add an instance variable to hold the array of icons:

```
@implementation IconPickerViewController
{
  NSArray *_icons;
}
```

➤ Also change `viewDidLoad`:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
```

```
  _icons = @[
    @"No Icon",
    @"Appointments",
    @"Birthdays",
    @"Chores",
    @"Drinks",
    @"Folder",
    @"Groceries",
    @"Inbox",
    @"Photos",
    @"Trips"];
}
```

The instance variable `_icons` is an `NSArray` 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 `NSArray` (instead of an `NSMutableArray`) because the user cannot add or delete icons.

Because this new view controller is a `UITableViewController`, you have to implement the data source methods for the table view.

➤ Remove the existing data source stuff from the source file and replace it with:

```
- (NSInteger)tableView:(UITableView *)tableView
            numberOfRowsInSection:(NSInteger)section
{
  return [_icons count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
                dequeueReusableCellWithIdentifier:@"IconCell"];

  NSString *icon = _icons[indexPath.row];
  cell.textLabel.text = icon;
  cell.imageView.image = [UIImage imageNamed:icon];

  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.

Convenience constructors

Earlier you've seen that new objects are created using a combination of `alloc` and `init`:

```
Checklist *checklist = [[Checklist alloc] init];
```

But here you're creating a new `UIImage` object using a different method:

```
cell.imageView.image = [UIImage imageNamed:icon];
```

No `alloc` or `init` in sight. What gives? This form is called a **convenience constructor**. You can actually also write it as:

```
image = [[UIImage alloc] initWithContentsOfFile:. . .];
```

For most intents and purposes these two forms are equivalent. They both allocate and initialize a new `UIImage` object.

Another example that you've been using quite a bit:

```
NSString *string = [NSString stringWithFormat:
                                @"I ate %d ice cream today", 3];
```

This can also be written as:

```
NSString *string = [[NSString alloc] initWithFormat:
                                @"I ate %d ice cream today", 3];
```

So why are there two approaches to the same thing? For convenience, mostly. Using `imageNamed` and `stringWithFormat` saves you from typing `alloc`. There is also a historical reason that has to do with memory management, but that went out the door with the arrival of iOS 5 and Automatic Reference Counting (ARC).

› Open the storyboard. Drag a new **Table View Controller** from the Object Library and place it next to the List Detail View Controller.

› 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.

➤ Select 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**.

➤ Add an **Image View** to the right of the cell. Make it 36 × 36 points big.

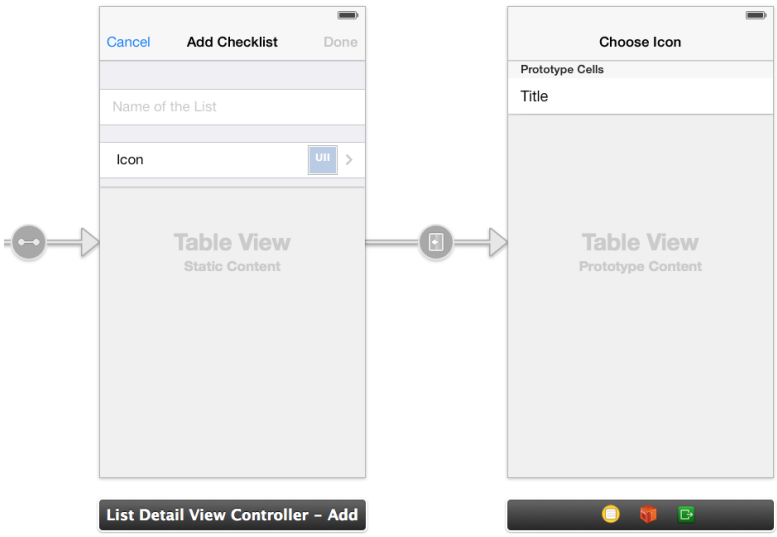
➤ Use the **Assistant Editor** to add an outlet property for this image view to **ListDetailViewController.h** and name it `iconImageView`.

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 (pick **Selection Segue – push**). 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**.

This part of the storyboard should now look like this:



The Icon Picker view controller in the storyboard

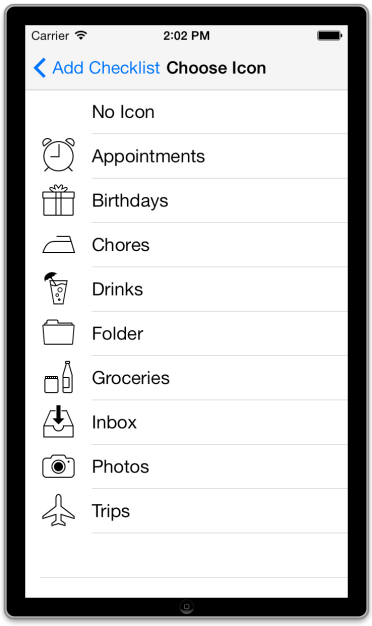
➤ In **ListDetailViewController.m**, change `willSelectRowAtIndexPath` to:

```
- (NSIndexPath *)tableView:(UITableView *)tableView
             willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (indexPath.section == 1) {
```

```
    return indexPath;
  } else {
    return nil;
  }
}
```

This is necessary otherwise you cannot tap the 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 on 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`. 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 and that tapping it will open the Choose Icon screen. The icon picker should show a list of icons. You can press the back button to go back.



The icon picker screen

Note: If tapping the Icon row does not bring up the icon picker, then make sure the table view's Selection attribute is not set to No Selection. It should be Single Selection, but Interface Builder may have messed this up when you duplicated this view controller earlier in the tutorial.

Selecting an icon doesn't do anything yet. You have to hook up the icon picker to the Add/Edit Checklist screen through its own delegate protocol.

➤ Add the following to **ListDetailViewController.h**:

```
#import "IconPickerViewController.h"

. . .

@interface ListDetailViewController : UITableViewController
        <UITextFieldDelegate, IconPickerViewControllerDelegate>
```

▸ Add an instance variable in **ListDetailViewController.m**:

```
@implementation ListDetailViewController
{
  NSString *_iconName;
}
```

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. To do this, you need to add the `initWithCoder` method as this is the method that is used to initialize this view controller (since it's being loaded from a storyboard).

▸ Add a new `initWithCoder` method to **ListDetailViewController.m**:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    _iconName = @"Folder";
  }
  return self;
}
```

This sets the `_iconName` variable to `@"Folder"`. This is only necessary for new `Checklists`, which get the Folder icon by default.

▸ You can get rid of the `initWithStyle` method as it's not being used for anything.

▸ Update `viewDidLoad` to the following:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.checklistToEdit != nil) {
```

```
    self.title = @"Edit Checklist";
    self.textField.text = self.checklistToEdit.name;
    self.doneBarButton.enabled = YES;
    _iconName = self.checklistToEdit.iconName;
  }

  self.iconImageView.image = [UIImage imageNamed:_iconName];
}
```

This has two new lines: If the `checklistToEdit` property is not `nil`, then you copy the `Checklist` object's icon name into the `_iconName` instance variable. You also load the icon into a new `UIImage` object and set it on the `iconImageView` so it shows up in the Icon row.

You hooked up the Add/Edit Checklist screen to the `IconPickerViewController` with a push segue named "PickIcon". You need to implement `prepareForSegue` in order to tell the `IconPickerViewController` that this screen is now its delegate.

➤ Add the following method to the bottom of **ListDetailViewController.m**:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"PickIcon"]) {
    IconPickerViewController *controller =
                                segue.destinationViewController;
    controller.delegate = self;
  }
}
```

This should have no surprises for you.

Finally, you implement the delegate callback method to remember the name of the chosen icon.

➤ Add to the bottom of **ListDetailViewController.m**:

```
- (void)iconPicker:(IconPickerViewController *)picker
       didPickIcon:(NSString *)theIconName
{
  _iconName = theIconName;
  self.iconImageView.image = [UIImage imageNamed:_iconName];

  [self.navigationController popViewControllerAnimated:YES];
}
```

This puts the name of the chosen icon into the `_iconName` variable and also updates the image view with a new image.

You don't call `dismissViewController` here but `popViewControllerAnimated` because the Icon Picker is on the navigation stack (you used segue style "push" instead of "modal").

➤ Change the `done` action so that it puts the chosen icon name into the `Checklist` object when the user closes the screen:

```
- (IBAction)done
{
  if (self.checklistToEdit == nil) {
    Checklist *checklist = [[Checklist alloc] init];
    checklist.name = self.textField.text;
    checklist.iconName = _iconName;

    [self.delegate listDetailViewController:self
                           didFinishAddingChecklist:checklist];

  } else {
    self.checklistToEdit.name = self.textField.text;
    self.checklistToEdit.iconName = _iconName;

    [self.delegate listDetailViewController:self
                didFinishEditingChecklist:self.checklistToEdit];
  }
}
```

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.m**:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  NSString *iconName = _icons[indexPath.row];
  [self.delegate iconPicker:self didPickIcon:iconName];
}
```

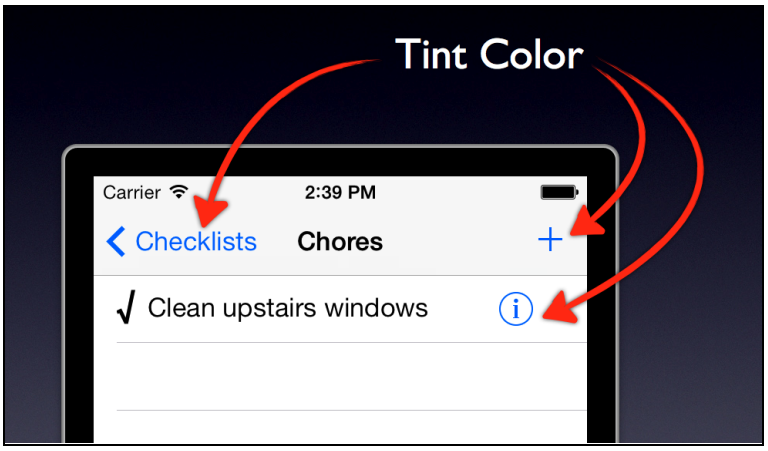
And that's it. You can now set icons on the `Checklist` objects. Try it out!

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.

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.

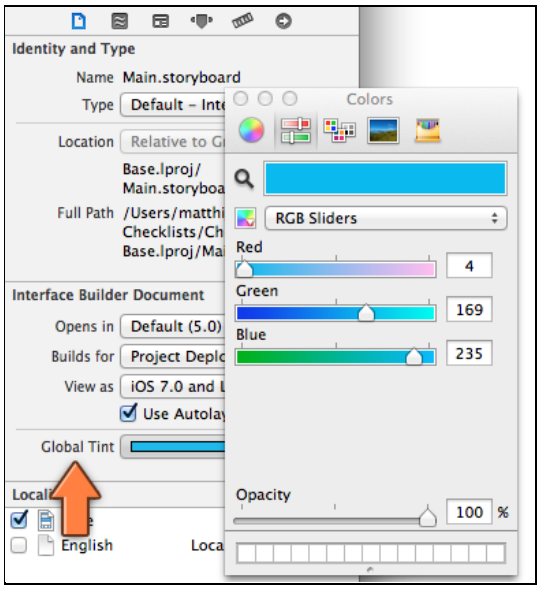
Even though the app uses the stock visuals, there is a simple trick to give the app its own personality: changing the **tint color**. The tint color is what UIKit uses to indicate that things can be interacted with, such as buttons. The default tint color is a medium blue.



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.



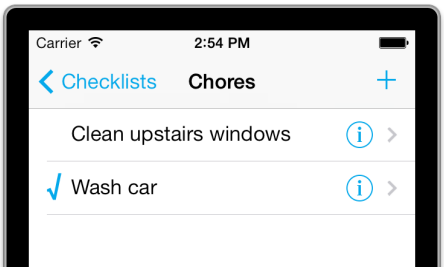
Changing the Global Tint color for the storyboard

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 `configureCheckmarkForCell` in **ChecklistViewController.m**:

```
label.textColor = self.view.tintColor;
```

➤ Run the app. It already looks a lot more interesting:



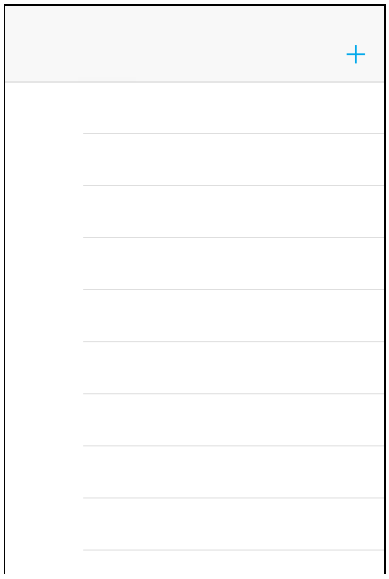
The tint color makes the app less plain looking

No app is complete without an icon and a launch image, so you'll finish up by adding those. 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 (**Images.xcassets**). Recall that icons go into the **AppIcon** section. Simply drag them from the Finder window into the slots.

➤ Add the launch images to the asset catalog, under **LaunchImage**. You can find the launch images for this app with this tutorial's Resources. The **Default-568h@2x.png** image goes into the **R4** slot.

The launch image simply has a navigation bar without a title but with the Add button, and an empty table view. This will give the illusion the app's UI has already been loaded but that the data hasn't been filled in yet.



The launch image for this app

To make this launch image, I ran the app in the Simulator and chose **File → Save Screen Shot**. This puts a new PNG file on the Desktop. I then opened this image in Photoshop and simply trimmed out the stuff it doesn't need. I also blanked out the status bar portion of the image. The iPhone will draw its own status bar on top anyway.

When you now run the app, instead of a black screen it immediately shows an empty table view with a navigation bar on top. It makes the app look more professional – and faster!

> **Note:** So far you've probably been running the app on the 4-inch Simulator or a 4-inch device (iPhone 5 or better). Because this app uses only table view controllers it should run equally well on 3.5-inch devices. Try it out. The table views will automatically resize to fit the screen. That isn't the case with regular view controllers, and in the next tutorials you'll learn how to deal with the differences in screen sizes between the various iPhone models.

You can find the project files for the app up to this point under **10 - UI Improvements** in the tutorial's Source Code folder.

Bonus feature: local notifications

I hope you're still with me! We have discussed in great detail view controllers, navigation controllers, storyboards, segues, tables 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**. 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'll 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. Push allows 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".

› Open **ChecklistsAppDelegate.m** and add the following code to the method `application:didFinishLaunchingWithOptions:`

```
NSDate *date = [NSDate dateWithTimeIntervalSinceNow:10];

UILocalNotification *localNotification =
                              [[UILocalNotification alloc] init];
localNotification.fireDate = date;
localNotification.timeZone = [NSTimeZone defaultTimeZone];
localNotification.alertBody = @"I am a local notification!";
localNotification.soundName =
                             UILocalNotificationDefaultSoundName;

[[UIApplication sharedApplication]
                    scheduleLocalNotification:localNotification];
```

Recall that the `didFinishLaunchingWithOptions` method is called when the app starts up. You create a new local notification here and tell it to fire 10 seconds after the app has started.

A local notification is scheduled in the future using an `NSDate` object, which specifies a certain date and time. You use the `dateWithTimeIntervalSinceNow` convenience constructor to create an `NSDate` object that points at a time 10 seconds into the future.

When you create the `UILocalNotification` object you give it the `NSDate` object as its "fire date". You also set the time zone, so the system automatically adjusts the fire date when the device travels across different time zones (for you frequent flyers).

Local notifications can appear in different ways. Here you set a text so that an alert message will be shown when the notification fires. You also set a sound. Finally, you tell the `UIApplication` object to schedule the notification.

> A word on `UIApplication`. You haven't used this object before, but every app has one and it deals with application-wide functionality. You always have to provide a delegate object for `UIApplication` that will handle messages that concern the app as a whole, such as `applicationDidEnterBackground` that you've seen earlier. In this app, the delegate for `UIApplication` is the `ChecklistsAppDelegate` object. The Xcode project templates always provide an app delegate object for you. You won't directly use `UIApplication` a lot, except for special features such as local notifications.

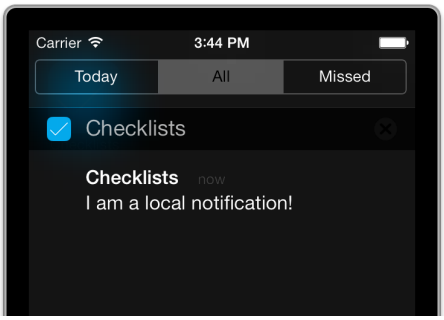
➤ Add the following method to **ChecklistsAppDelegate.m**:

```
- (void)application:(UIApplication *)application
 didReceiveLocalNotification:(UILocalNotification *)notification
{
  NSLog(@"didReceiveLocalNotification %@", notification);
}
```

This method will be invoked when the local notification is posted and the app is still running or in a suspended state in the background. You won't do anything here except log a message to the debug pane. For some apps it makes sense to react to the notification, for example to show a message to the user or to refresh the screen.

➤ Run the app. Immediately after it has started, press the Home button on the Simulator (or your device if you're running this on your iPhone). Wait 10 seconds.

After 10 seconds a message should pop up in Notification Center:



The local notification message

➤ Tap the notification to go back to the app.

The debug area shows that `didReceiveLocalNotification` is called with the notification object. It displays something like this:

```
Checklists[42887:a0b] didReceiveLocalNotification
<UIConcreteLocalNotification: 0x8dc6000>{fire date = Thursday, October
3, 2013 at 3:45:34 PM Central European Summer Time, time zone =
Europe/Amsterdam (GMT+2) offset 7200 (Daylight), repeat interval = 0,
repeat count = UILocalNotificationInfiniteRepeatCount, next fire date =
(null), user info = (null)}
```

Why did I want you to press the Home button? iOS will only show an alert with the notification message if the app is not currently active.

➤ Stop the app and run it again. Now don't press Home and just wait.

After 10 seconds you should see the log message for `didReceiveLocalNotification` in the debug area but no alert is shown. (The notification does show up in Notification Center, though.) When your app is active and in the foreground, it is supposed to handle any fired notifications in its own manner.

All right, now you know that it works, you should restore **ChecklistsAppDelegate.m** to its former state because you don't really want to schedule a new notification every time the user starts the app.

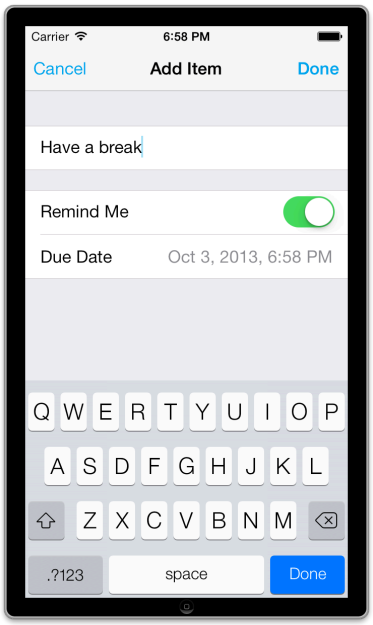
➤ Change the `didFinishLaunchingWithOptions` method back to the way it was.

You can keep the `didReceiveLocalNotification` 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 (an `NSDate` object) 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` 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 will 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 list 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.

`UIApplication` has a method `cancelLocalNotification` that allows you to cancel a notification that was previously scheduled. That method takes a `UILocalNotification` object. Somehow you must associate the `ChecklistItem` object with a `UILocalNotification` in order to be able to cancel that notification.

It is tempting to put the `UILocalNotification` object in `ChecklistItem`, so you always know what it is, but imagine what happens when the app goes to the background. In that case you save the `ChecklistItem` object to the Checklists.plist file – but what about the `UILocalNotification` object? As it happens the `UILocalNotification` conforms to the `NSCoding` protocol so you could serialize it along with the `ChecklistItem` object into the plist file. However, that is asking for trouble.

These `UILocalNotification` objects are owned by the operating system, not by your app. When the app starts again, it is very well possible that iOS uses different objects to represent the same notifications. You cannot unfreeze these objects from the plist file and expect iOS to recognize them. So let's not store the `UILocalNotification` objects directly.

What will work better is to give the `UILocalNotification` a reference to the associated `ChecklistItem`. Each local notification has an `NSDictionary` named `userInfo` that you can use to store your own values.

You will not use this dictionary to store the `ChecklistItem` object itself, for the same reason as above: when the app closes and later starts again, it will get new `ChecklistItem` objects. Even though they look and behave exactly the same as the old `ChecklistItem`s (because you froze and unfroze them), they are likely to be placed elsewhere in memory and the references inside the `UILocalNotification`s will be broken.

Instead of direct references, you will use a numeric identifier. You will give each `ChecklistItem` object a unique numeric ID. 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.

You'll save this ID in the Checklists.plist file and also store it in the `userInfo` dictionary of the `UILocalNotification`. Then you can easily find the notification when you have the `ChecklistItem` object, or the `ChecklistItem` object when you have the notification. This will work even after the app has terminated and all the original objects have long been destroyed.

➤ Make these changes to **ChecklistItem.h**:

```
@property (nonatomic, copy) NSDate *dueDate;
@property (nonatomic, assign) BOOL shouldRemind;
@property (nonatomic, assign) NSInteger itemId;
```

Note that you spelled it `itemId`, not `itemID`. That's a stylistic thing; I just like it better that way. You also did not simply call it "id" because `id` is a special keyword in Objective-C – and it has nothing to do with identifiers. For example, you've already seen the `id` keyword when you created your own delegate protocols.

You have to extend `initWithCoder` and `encodeWithCoder` in order to be able to load and save these new properties along with the `ChecklistItem` objects.

➤ Change these methods in **ChecklistItem.m**:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super init])) {
    self.text = [aDecoder decodeObjectForKey:@"Text"];
    self.checked = [aDecoder decodeBoolForKey:@"Checked"];
    self.dueDate = [aDecoder decodeObjectForKey:@"DueDate"];
    self.shouldRemind = [aDecoder
                              decodeBoolForKey:@"ShouldRemind"];
    self.itemId = [aDecoder decodeIntegerForKey:@"ItemID"];
  }
  return self;
}
```

```
- (void)encodeWithCoder:(NSCoder *)aCoder
{
  [aCoder encodeObject:self.text forKey:@"Text"];
  [aCoder encodeBool:self.checked forKey:@"Checked"];
  [aCoder encodeObject:self.dueDate forKey:@"DueDate"];
  [aCoder encodeBool:self.shouldRemind forKey:@"ShouldRemind"];
  [aCoder encodeInteger:self.itemId forKey:@"ItemID"];
}
```

That takes care of saving and loading existing objects, but you still have to assign an ID to new objects.

➤ Add an regular `init` method to **ChecklistItem.m**:

```
- (id)init
{
  if (self = [super init]) {
    self.itemId = [DataModel nextChecklistItemId];
  }
  return self;
}
```

This asks the `DataModel` object for a new item ID whenever the app creates a new `ChecklistItem` object. Because you haven't used the `DataModel` object before in this source file, you have to import it.

➤ Add the import line at the top of the file:

```
#import "DataModel.h"
```

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.

➤ First, add the method signature to **DataModel.h**:

```
+ (NSInteger)nextChecklistItemId;
```

➤ Add the implementation to **DataModel.m**:

```
+ (int)nextChecklistItemId
{
  NSUserDefaults *userDefaults = [NSUserDefaults
                                        standardUserDefaults];

  NSInteger itemId = [userDefaults
                              integerForKey:@"ChecklistItemId"];

  [userDefaults setInteger:itemId + 1
                    forKey:@"ChecklistItemId"];

  [userDefaults synchronize];
  return itemId;
}
```

You're using your old friend `NSUserDefaults` again. This method gets the current "ChecklistItemId" value from `NSUserDefaults`, adds one to it, and writes it back to `NSUserDefaults`. It returns the previous value to the caller.

The method also does `[userDefaults synchronize]` to force `NSUserDefaults` 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 `ChecklistItem`s to get the same ID.

➤ Add a default value for "ChecklistItemId" to the `registerDefaults` method:

```
- (void)registerDefaults
{
  NSDictionary *dictionary = @{
    @"ChecklistIndex" : @-1,
    @"FirstTime" : @YES,
```

```
    @"ChecklistItemId" : @0,
    };

  [[NSUserDefaults standardUserDefaults]
                                    registerDefaults:dictionary];
}
```

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,

```
+ (int)nextChecklistItemId
```

and not:

```
- (int)nextChecklistItemId
```

then I'm glad you're paying attention. :-) Using the `+` instead of the `–` means that you can call this method without having a reference to the `DataModel` object.

Remember, you did,

```
self.itemId = [DataModel nextChecklistItemId];
```

instead of:

```
self.itemId = [self.dataModel nextChecklistItemId];
```

This is because `ChecklistItem` objects do not have a reference to the `DataModel` object, i.e. they don't have a `self.dataModel` property. You could certainly give them such a reference, but I decided that using a **class method** was easier.

The name of a class method begins with a `+` (plus sign). This kind of method applies to the class as a whole. So far you've been using **instance methods**. They begin with a `–` (minus sign) 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 a `+` 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 but it's very well possible that, if you were to develop this app further, it would make more sense to use the references 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.

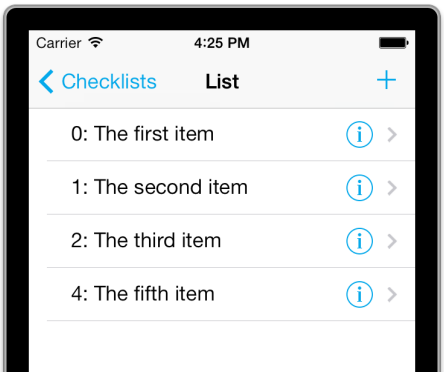
▸ In **ChecklistViewController.m**, change the `configureTextForCell` method to:

```
- (void)configureTextForCell:(UITableViewCell *)cell
       withChecklistItem:(ChecklistItem *)item
{
  UILabel *label = (UILabel *)[cell viewWithTag:1000];

  //label.text = item.text;

  label.text = [NSString stringWithFormat:
                            @"%d: %@", item.itemId, item.text];
}
```

I have commented out the original line because you want to put that back later. The new line uses `stringWithFormat` to add the to-do item's `itemId` property into the text.

▸ Before you run the app, make sure to delete it from the Simulator first. You have changed the format of the Checklists.plist file again and reading an incompatible file may cause weird crashes.

▸ Run the app and add some checklist items. Each new item should get a unique identifier. Press Home (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 `configureTextForCell` the way it is for the time being; that will come in handy with testing the notifications.)

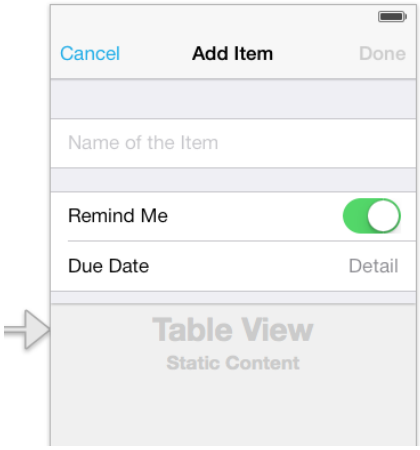
➤ Add the following outlets to **ItemDetailViewController.h**:

```
@property (nonatomic, weak) IBOutlet UISwitch *switchControl;
@property (nonatomic, weak) IBOutlet UILabel *dueDateLabel;
```

➤ 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**.

➤ Also drag a **Switch** control into the cell. Hook this switch up to the `switchControl` outlet on the view controller. (I would have preferred to call this outlet simply "switch", but that is a reserved keyword in the Objective-C language.)

➤ 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**.

➤ 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.)

Let's write the code for this.

➤ Add a new `_dueDate` instance variable to **ItemDetailViewController.m**:

```
@implementation ItemDetailViewController
{
  NSDate *_dueDate;
}
```

➤ Change the `viewDidLoad` method to the following:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.itemToEdit != nil) {
    self.title = @"Edit Item";
    self.textField.text = self.itemToEdit.text;
    self.doneBarButton.enabled = YES;
    self.switchControl.on = self.itemToEdit.shouldRemind;
    _dueDate = self.itemToEdit.dueDate;
  } else {
    self.switchControl.on = NO;
    _dueDate = [NSDate date];
  }

  [self updateDueDateLabel];
}
```

If you already have an existing `ChecklistItem` object, you set the switch control on or off, depending on the value of the object's `shouldRemind` property. If the user is adding a new item, you always set the switch to off.

For a new item, the due date is right now, or `[NSDate date]`. That might not make much sense because by the time you have completed the rest of the fields and pressed Done, that due date will be 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.

➤ The `updateDueDateLabel` method is new. Add it to the file:

```
- (void)updateDueDateLabel
{
  NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
  [formatter setDateStyle:NSDateFormatterMediumStyle];
  [formatter setTimeStyle:NSDateFormatterShortStyle];
  self.dueDateLabel.text = [formatter stringFromDate:_dueDate];
}
```

To convert the `NSDate` value to text, you use the `NSDateFormatter` 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 `NSDate` 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 `NSDateFormatter` is that it takes the current locale into consideration so the time will look good to the user no matter where he is on the globe.

➤ The last thing to change in this file is the `done` action. Change it to:

```
- (IBAction)done
{
  if (self.itemToEdit == nil) {
    ChecklistItem *item = [[ChecklistItem alloc] init];
    item.text = self.textField.text;
    item.checked = NO;
    item.shouldRemind = self.switchControl.on;
    item.dueDate = _dueDate;

    [self.delegate itemDetailViewController:self
                        didFinishAddingItem:item];

  } else {
    self.itemToEdit.text = self.textField.text;
    self.itemToEdit.shouldRemind = self.switchControl.on;
    self.itemToEdit.dueDate = _dueDate;

    [self.delegate itemDetailViewController:self
                       didFinishEditingItem:self.itemToEdit];
  }
}
```

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.

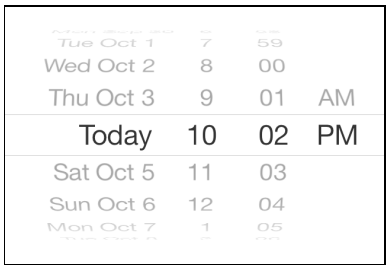
> **Note:** Maybe you're wondering why you're using an instance variable for the `_dueDate` but not for `shouldRemind`. You don't need one for `shouldRemind` because it's easy to get the state of the switch control: you just look at its `on` property, which is either `YES` or `NO`. However, it is hard to read the chosen date back out of the `dueDateLabel`, because the label stores text (an `NSString`), not an `NSDate`. So it's easier to keep track of the chosen date separately in an `NSDate` instance variable.

▸ 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 press the Home button 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.

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 in the built-in Calendar app.



The UIDatePicker component

▸ Add a new instance variable to **ItemDetailViewController.m**, to keep track of whether the date picker is currently visible:

```
@implementation ItemDetailViewController
{
  NSDate *_dueDate;
  BOOL _datePickerVisible;
}
```

▸ Add the `showDatePicker` method:

```
- (void)showDatePicker
{
  _datePickerVisible = YES;

  NSIndexPath *indexPathDatePicker = [NSIndexPath
                                  indexPathForRow:2 inSection:1];
```

```
  [self.tableView insertRowsAtIndexPaths:@[indexPathDatePicker]
                        withRowAnimation:UITableViewRowAnimationFade];
}
```

This sets the new instance variable to `YES`, and then it tells the table view to insert a new row below the Due Date cell. This new row will contain the `UIDatePicker` component.

The question is: where does the cell for this new row come from? Normally you would implement the `cellForRowAtIndexPath` 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 `cellForRowAtIndexPath`. If you look in **ItemDetailViewController.m** 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 `cellForRowAtIndexPath` method:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  // 1
  if (indexPath.section == 1 && indexPath.row == 2) {

    // 2
    UITableViewCell *cell = [tableView
           dequeueReusableCellWithIdentifier:@"DatePickerCell"];

    if (cell == nil) {
      cell = [[UITableViewCell alloc]
                      initWithStyle:UITableViewCellStyleDefault
                      reuseIdentifier:@"DatePickerCell"];

      cell.selectionStyle = UITableViewCellSelectionStyleNone;

      // 3
      UIDatePicker *datePicker = [[UIDatePicker alloc]
          initWithFrame:CGRectMake(0.0f, 0.0f, 320.0f, 216.0f)];

      datePicker.tag = 100;
      [cell.contentView addSubview:datePicker];

      // 4
      [datePicker addTarget:self action:@selector(dateChanged:)
                   forControlEvents:UIControlEventValueChanged];
```

```
    }
    return cell;

  // 5
  } else {
    return [super tableView:tableView
                      cellForRowAtIndexPath:indexPath];
  }
}
```

Here is what this method does, step-by-step:

1. Check whether this is the index-path for the row with the date picker. If not, jump to step 5.
2. Ask the table view whether it already has the date picker cell. If not, then create a new one. The selection style is "none" because you don't want to show a selected state for this cell when the user taps it.
3. Create a new `UIDatePicker` component. It has a tag (100) so you can easily find this date picker later.
4. Tell the date picker to call the method `dateChanged:` whenever the user changes the date. You have seen how to connect action methods from Interface Builder; this is how you do it from code. The `UIDatePicker`'s Value Changed event now triggers the `dateChanged` method. Because this method does not exist yet, Xcode shows a warning on this line.
5. For any index-paths that are not the date picker cell, 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 `numberOfRowsInSection`:

```
- (NSInteger)tableView:(UITableView *)tableView
          numberOfRowsInSection:(NSInteger)section
{
  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 `heightForRowAtIndexPath` method:

```
- (CGFloat)tableView:(UITableView *)tableView
            heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (indexPath.section == 1 && indexPath.row == 2) {
    return 217.0f;
  } else {
    return [super tableView:tableView
                            heightForRowAtIndexPath:indexPath];
  }
}
```

So far the cells in your table views all had the same height (44 points), but this is not a requirement. By providing the `heightForRowAtIndexPath` method you can give each cell its own height. The `UIDatePicker` component is 216 points tall, plus 1 point for the separator line, makes 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 `didSelectRowAtIndexPath`.

➤ Add that method:

```
- (void)tableView:(UITableView *)tableView
         didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [self.tableView deselectRowAtIndexPath:indexPath
                                animated:YES];

  [self.textField resignFirstResponder];

  if (indexPath.section == 1 && indexPath.row == 1) {
    [self 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.m** already has a `willSelectRowAtIndexPath` method that always returns `nil`, causing taps on all rows to be ignored.

➤ Change `willSelectRowAtIndexPath` to:

```
- (NSIndexPath *)tableView:(UITableView *)tableView
              willSelectRowAtIndexPath:(NSIndexPath *)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.

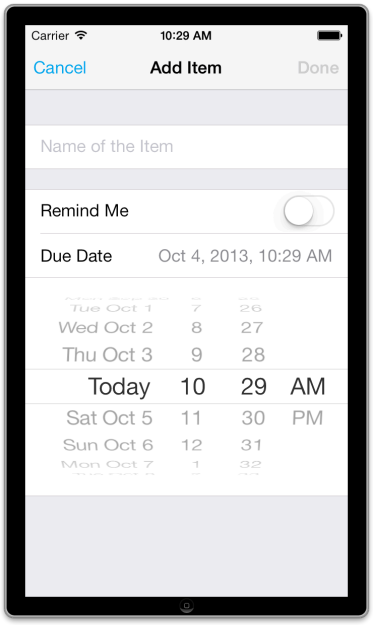
Whoops. The app crashes. After some investigating, it turns out that when you override the data source for a static table view cell, you also need to provide the delegate method `indentationLevelForRowAtIndexPath`. That's not a method you'd typically use but because you're messing with the data source for a static table view you need to override it. I told you this was a little tricky.

➤ Add the `indentationLevelForRowAtIndexPath` method:

```
- (NSInteger)tableView:(UITableView *)tableView
      indentationLevelForRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (indexPath.section == 1 && indexPath.row == 2) {

    NSIndexPath *newIndexPath = [NSIndexPath indexPathForRow:0
                                    inSection:indexPath.section];

    return [super tableView:tableView
                indentationLevelForRowAtIndexPath:newIndexPath];

  } else {
    return [super tableView:tableView
                 indentationLevelForRowAtIndexPath:indexPath];
  }
}
```

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 is 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.

➤ Run the app again. This time the date picker cell shows up where it should:



The date picker appears in a new cell

However, if you interact with the date picker, the app crashes again. That's because `UIDatePicker` attempts to call the `dateChanged` method whenever its value changes (the Value Changed event), but you haven't added that method yet.

➤ Add the `dateChanged` method to **ItemDetailViewController.m**:

```
- (void)dateChanged:(UIDatePicker *)datePicker
{
  _dueDate = datePicker.date;
  [self 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.

➤ 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 time.

➤ Add the following lines to the bottom of `showDatePicker`:

```
UITableViewCell *datePickerCell = [self.tableView
                    cellForRowAtIndexPath:indexPathDatePicker];

UIDatePicker *datePicker = (UIDatePicker *)
                               [datePickerCell viewWithTag:100];
```

```
  [datePicker setDate:_dueDate animated:NO];
```

This locates the `UIDatePicker` component in the new cell, and gives it the proper date.

➤ Verify that this works. Click on the (i) button from an existing to-do item, preferably one you made a while ago, and make sure that the date picker shows the same date and time as the Due Date label.

Speaking of the date 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 built-in Calendar app does).

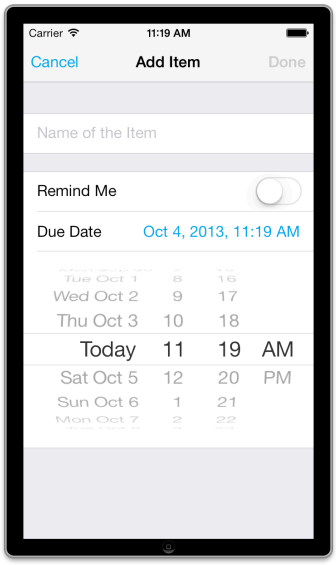
➤ Change `showDatePicker` one last time:

```
- (void)showDatePicker
{
  _datePickerVisible = YES;

  NSIndexPath *indexPathDateRow = [NSIndexPath
                                   indexPathForRow:1 inSection:1];
  NSIndexPath *indexPathDatePicker = [NSIndexPath
                                   indexPathForRow:2 inSection:1];

  UITableViewCell *cell = [self.tableView
                          cellForRowAtIndexPath:indexPathDateRow];
  cell.detailTextLabel.textColor =
                                    cell.detailTextLabel.tintColor;

  [self.tableView beginUpdates];
  [self.tableView insertRowsAtIndexPaths:@[indexPathDatePicker]
                  withRowAnimation:UITableViewRowAnimationFade];
  [self.tableView reloadRowsAtIndexPaths:@[indexPathDateRow]
                  withRowAnimation:UITableViewRowAnimationNone];
  [self.tableView endUpdates];

  UITableViewCell *datePickerCell = [self.tableView
                     cellForRowAtIndexPath:indexPathDatePicker];
  UIDatePicker *datePicker = (UIDatePicker *)
                                  [datePickerCell viewWithTag:100];
  [datePicker setDate:_dueDate animated:NO];
}
```

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, which obviously won't win it many favorable reviews.

➤ Add the new `hideDatePicker` method:

```
- (void)hideDatePicker
{
  if (_datePickerVisible) {
    _datePickerVisible = NO;

    NSIndexPath *indexPathDateRow = [NSIndexPath
                                     indexPathForRow:1 inSection:1];
    NSIndexPath *indexPathDatePicker = [NSIndexPath
                                     indexPathForRow:2 inSection:1];

    UITableViewCell *cell = [self.tableView
                         cellForRowAtIndexPath:indexPathDateRow];
    cell.detailTextLabel.textColor = [UIColor
                                colorWithWhite:0.0f alpha:0.5f];

    [self.tableView beginUpdates];
    [self.tableView reloadRowsAtIndexPaths:@[indexPathDateRow]
                  withRowAnimation:UITableViewRowAnimationNone];
    [self.tableView
```

```
                  deleteRowsAtIndexPaths:@[indexPathDatePicker]
                   withRowAnimation:UITableViewRowAnimationFade];
    [self.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 `didSelectRowAtIndexPath` to toggle between the visible and hidden states:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [self.tableView deselectRowAtIndexPath:indexPath
                                animated:YES];

  [self.textField resignFirstResponder];

  if (indexPath.section == 1 && indexPath.row == 1) {

    if (!_datePickerVisible) {
      [self showDatePicker];
    } else {
      [self 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:

```
- (void)textFieldDidBeginEditing:(UITextField *)textField
{
  [self hideDatePicker];
}
```

And with that you have a cool inline date picker!

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 declaration to **ChecklistItem.h**:

```
- (void)scheduleNotification;
```

▸ Add the method itself to **ChecklistItem.m**:

```
- (void)scheduleNotification
{
  if (self.shouldRemind &&
      [self.dueDate compare:[NSDate date]] != NSOrderedAscending)
  {
    NSLog(@"We should schedule a notification");
  }
}
```

This compares the due date on the item with the current date. If the due date is in the past, then the `NSLog()` 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.

▸ Change the `done` action in **ItemDetailViewController.m**:

```
- (IBAction)done
{
  if (self.itemToEdit == nil) {
    ChecklistItem *item = [[ChecklistItem alloc] init];
    item.text = self.textField.text;
    item.checked = NO;
    item.shouldRemind = self.switchControl.on;
    item.dueDate = _dueDate;

    [item scheduleNotification];

    [self.delegate itemDetailViewController:self
                        didFinishAddingItem:item];

  } else {
    self.itemToEdit.text = self.textField.text;
    self.itemToEdit.shouldRemind = self.switchControl.on;
```

```
    self.itemToEdit.dueDate = _dueDate;

    [self.itemToEdit scheduleNotification];

    [self.delegate itemDetailViewController:self
                        didFinishEditingItem:self.itemToEdit];
  }
}
```

Only the lines with `[scheduleNotification]` are new.

➤ 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 an `NSLog` ("We should schedule a notification") in the debug area.

Now that you've verified the method is called in the proper place, let's actually schedule a new `UILocalNotification` object. First consider the case of a new to-do item being added.

➤ In **ChecklistItem.m**, change `scheduleNotification` to:

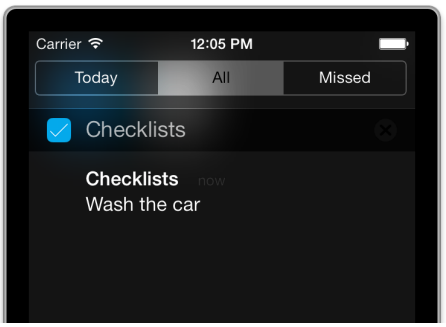
```
- (void)scheduleNotification
{
  if (self.shouldRemind &&
     [self.dueDate compare:[NSDate date]] != NSOrderedAscending)
  {

    UILocalNotification *localNotification =
                            [[UILocalNotification alloc] init];
    localNotification.fireDate = self.dueDate;
    localNotification.timeZone = [NSTimeZone defaultTimeZone];
    localNotification.alertBody = self.text;
    localNotification.soundName =
                           UILocalNotificationDefaultSoundName;

    localNotification.userInfo = @{
                                  @"ItemID" : @(self.itemId) };

    [[UIApplication sharedApplication]
                    scheduleLocalNotification:localNotification];

    NSLog(@"Scheduled notification %@ for itemId %d",
```

```
                                localNotification, self.itemId);
    }
}
```

You've seen this code before. It creates a `UILocalNotification` object. This time, however, it uses the `ChecklistItem`'s `dueDate` and `text`. You also add a `userInfo` dictionary with the item's ID as the only contents. That is how you'll be able to find this notification later in case you need to cancel it.

➤ Test it out. Run the app, add a new checklist item, set the due date a minute into the future, press Done and tap the Home button on the Simulator.

Wait one minute (patience…) and the notification should appear. Pretty cool!



The local notification in Notification Center

> The date picker doesn't show you seconds but they still are there (just watch the `NSLog` output). If you set the due date to 10:16 PM when it's currently 10:15:54 PM, you'll have to wait until exactly 10:16:54 for the event to fire. It would probably be a better user experience if you always set the seconds to 0, but that's a topic for another day.

That takes care of the case where you're adding a new notification. There are two situations left: the user edits an existing item and 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 to the top of `scheduleNotification` in **ChecklistItem.m**:

```
- (void)scheduleNotification
{
  UILocalNotification *existingNotification = [self
                                        notificationForThisItem];
  if (existingNotification != nil) {
    NSLog(@"Found an existing notification %@",
                                           existingNotification);
    [[UIApplication sharedApplication]
                  cancelLocalNotification:existingNotification];
  }

  . . .
```

This calls a new method `notificationForThisItem`, which you'll add in a second. If that method returns a valid `UILocalNotification` object (i.e. not `nil`), then you dump some debug info using `NSLog()` and then ask the `UIApplication` object to cancel this notification.

➤ Add the new `notificationForThisItem` method:

```
- (UILocalNotification *)notificationForThisItem
{
  NSArray *allNotifications = [[UIApplication sharedApplication]
                                    scheduledLocalNotifications];

  for (UILocalNotification *notification in allNotifications) {

    NSNumber *number = [notification.userInfo
                                         objectForKey:@"ItemID"];
```

```
    if (number != nil && [number integerValue] == self.itemId) {
      return notification;
    }
  }
  return nil;
}
```

This asks `UIApplication` for a list of all scheduled notifications. Then it loops through that list and looks at each notification one-by-one. It should have an "ItemID" value inside the `userInfo` dictionary. If that value exists and equals the `self.itemID` property, then you've found a notification that belongs to this `ChecklistItem`. If none of the local notifications match, or there aren't any to begin with, the method returns `nil`.

This is a common pattern that you'll see in a lot of code. Something returns an array of items and you loop through the array to find the first item that matches what you're looking for, in this case the item ID. Once you've found it, you can exit the loop.

➤ Run the app and try it out. Add a to-do item with a due date a few days into the future. A new notification will be scheduled. Edit the item and change the due date. The old notification will be removed and a new one scheduled for the new date. You can tell that this happens from the `NSLog()` output.

➤ Edit the to-do item again but now set the switch to OFF. The old notification will be removed and no new notification will be scheduled. Edit again and don't change anything; no new notification will be scheduled because the switch is still off. This 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 `dealloc` 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.m**:

```
- (void)dealloc
{
  UILocalNotification *existingNotification = [self
                                        notificationForThisItem];
  if (existingNotification != nil) {

    NSLog(@"Removing existing notification %@",
```

```
                                                existingNotification);
    [[UIApplication sharedApplication]
                    cancelLocalNotification:existingNotification];
  }
}
```

That's all you have to do. The `dealloc` method will be invoked when you delete an individual `ChecklistItem` but also when you delete a whole `Checklist` (because then all its `ChecklistItem`s will be destroyed as well, as the array they are in is deallocated).

➤ Run the app and try it out. First schedule some notifications far into the future (so they won't be fired when you're testing) and then remove that to-do item or its entire checklist. You should now see a message in the debug area.

Once you're convinced everything works, you can remove the `NSLog()` statements. They are only temporary for debugging purposes. You probably don't want to leave them in the final app. (They 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.

Exercise: Put the due date in a label on the table view cells under the text of the to-do item. □

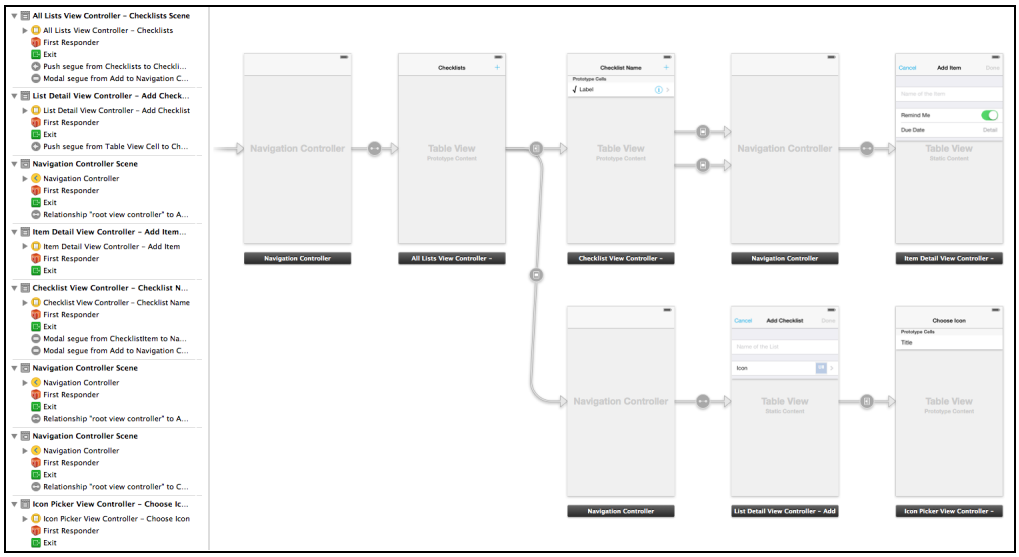
Exercise: Sort the to-do items list based on the due date. This is similar to what you did with the list of `Checklist`s except that now you're sorting `ChecklistItem` objects and you'll be comparing `NSDate` objects instead of `NSString`s. (`NSDate` does not have a `localizedStandardCompare` method but it does have a regular `compare`). □

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 I'm doing. If not, then 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.

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 Objective-C 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!

You can find the full source code of the app in the **Source Code** folder for this tutorial.