Exercise 1 – Objective-C 101

Objective

To re-familiarise you with Objective-C with some simple exercises in which you will create and manipulate some basic Objective-C classes.

Overview

Objective-C allows you to define classes, implement inheritance and extend classes using categories and protocols. In this exercise, we will uses all of those techniques.

Estimated Duration

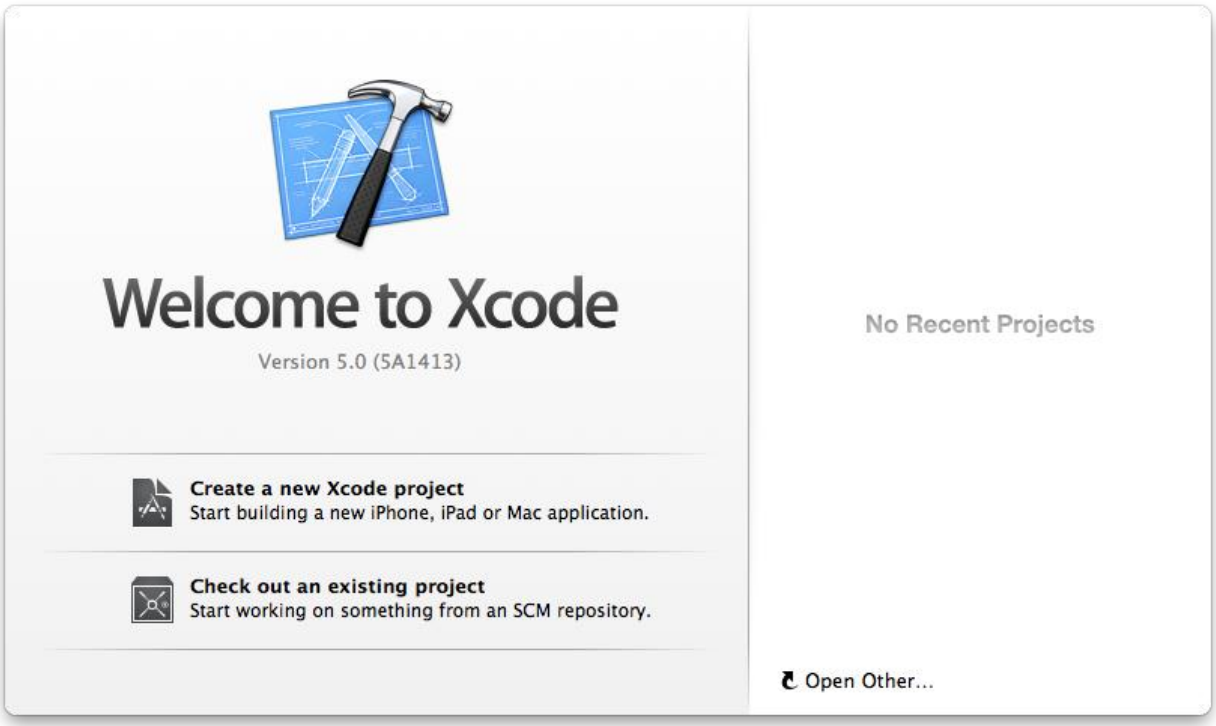
30 minutes.

Step by Step Instructions

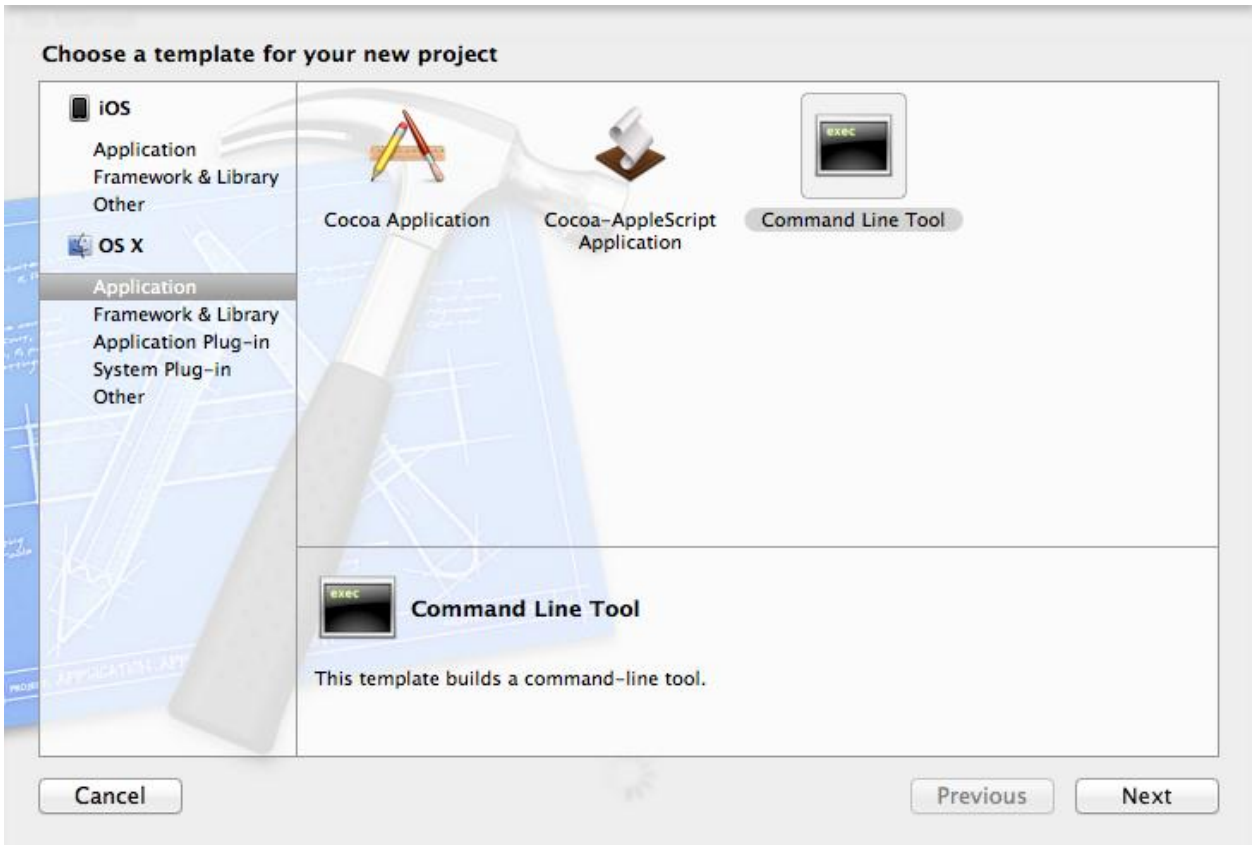
Step 1 – Creating an Xcode project

1. Start XCode which you will find in the dock bar at the bottom of the screen. The icon will look like this: 

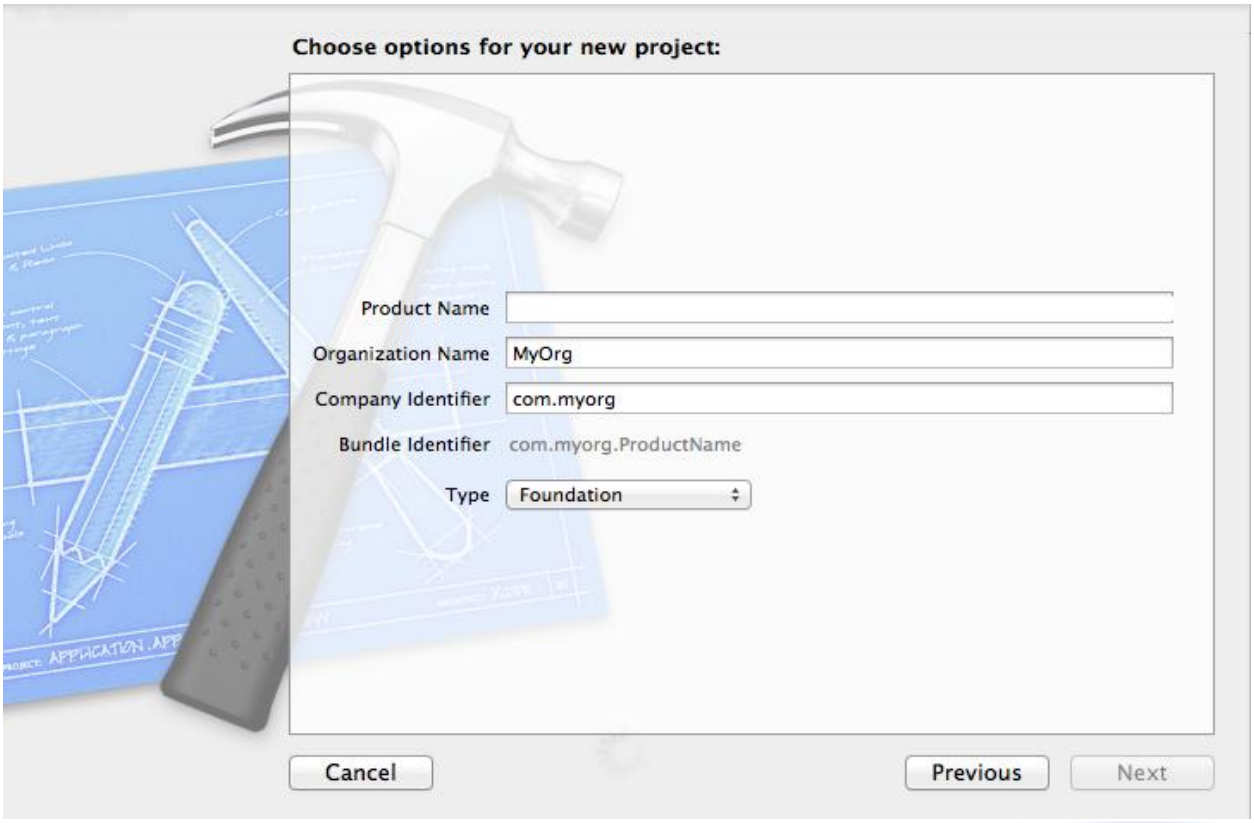
XCode will load and the “Choose a project template screen” may display:



XCode will load and the “Choose a project template screen” will be displayed:



- In the **Mac OS X** section choose – **Application** which will display a list of possible Mac application types
- Select **Command Line Tool** and click the **Next** button to display the project options screen



Product options allows you to enter the product name, in this case Exercise1, and a company identifier. The combination of company identifier and product name is used to create a unique application (bundle) identifier.

- Enter a Product Name and Company Identifier e.g. your backwards url
- In the **Type** dropdown choose Foundation as we want to write Objective C
- Click **Next**

You will now be asked to choose the folder location where you want to create the project.

- From the Favourite folders list choose **Documents\QAIOSDEV\Exercise\01 Objective-C 101\Starter**
- Press **Next** and your project will be created

Step 2 – Creating a Class

The next step is to create a new class. You do that from the menu option File -> New -> New File (or pressing ⌘N) and selecting an Objective-C class. Make sure you create a sub class of NSObject.

1. Create a new class called **Publication** and add the following instance variables (within curly braces)
    a. title (NSString)
    b. publisher (NSString)
    c. author (NSString)
    d. year of publication (int)
    e. pages (int)
    f. published (BOOL)
2. Make **published** a private ivar and the others public

It should look something like this when you are done:

```
@interface Publication : NSObject
{
    @public
    NSString * title;
    NSString * publisher;
    NSString * author;
    int yearOfPublication;
    int pages;
    @private
    BOOL published;
```

```
}
@end
```

3. Add code to main.m to create an instance of Publication (Inside the @autoreleasepool brackets { … })
    a. Make sure you import Publication.h
    b. Assign values to the public values
4. Add an NSLog statement to write the public ivars for your publication instance to the output window.
5. Run and test your application. The code should look similar to:

```
Publication * pub=[[Publication alloc] init];


pub->title=@"Test";
pub->author=@"Chris";
pub->yearOfPublication=2011;
pub->pages=345;
pub->publisher=@"Anyone";


NSLog(@"Publication %@ was written by %@ and published in %i.
It has %i pages and was published by %@.", pub->title,pub-
>author,pub->yearOfPublication,pub->pages,pub->publisher);
```

Step 3 – Writing methods

6. Add a method called **publish** that takes an int parameter for year of publication and returns void.
7. Set the year of publication and the published flag.
8. Make sure you add a definition of your publish method to the header file.
9. Add an NSLog statement inside the publish method indicating “Publication Published”.
10. Call the publish method on your publication instance in main.m.
11. Detect if the publication is published and output different messages for published and unpublished publications.
12. Run and test your application.

You should have written:

```
-(void)publish:(int)year
{
```

```
        _yearOfPublication=year;
        _published=YES;
        NSLog(@"Publication published");


}
```

Your header file (.h) should have the following (outside of the curly braces).

```
-(void)publish:(int)year;


@end
```

When you call a method, you use square brackets so the following method call could be made in main.m

```
[pub publish:1998];
```

Step 4 – Writing additional methods (if you have time)

13. Add a class method called create that
    a. Returns an instance of a Publication
    b. Add its definition to the header file
14. Replace where you create an instance Publication (using alloc init) in main.m with a call to the create method of Publication.

Step 5 – Adding a Custom Initialiser

15. Add a custom initialiser method to the publication class called initWithTitle:andYear.

    It should look like this:

```
-(id)initWithTitle:(NSString*)ptitle andYear:(int)year
{
        if (self=[super init])
        {
                title=ptitle;
                yearOfPublication=year;
        }
        return self;
}
```

16. Create an instance of a publication but this time use alloc/initWithTitle to create it.

Exercise 2 – Objective-C 102

Objective

To add inheritance, categories and the use of protocols and delegates to your knowledge.

Estimated Duration

30 minutes.

Step by Step Instructions

Step 1 – Properties

1. Open project Exercise 2.xcodeproj in **Documents\QAIOSDEV\Exercise\02 Objective-C 102\Starter**
2. Edit Publication.h
3. Replace all of your ivars with properties
4. Make **published** and **yearOfPublication** read only properties.
5. Set values for all of the writable properties using either the dot syntax or method syntax.
6. Open main.m and add an NSLog statement to write the properties for your publication instance to the output window.
7. Run and test your application.

Your properties definition should appear in your header file and look similar to:

```
@interface Publication : NSObject
@property (nonatomic,strong) NSString * title;
@property (nonatomic,strong) NSString * author;
@property (nonatomic,strong) NSString * publisher;
@property (nonatomic,readonly,assign) int yearOfPublication;
@property (nonatomic,assign) int   pages;
@property (nonatomic,readonly, assign) BOOL  published;
@end
```

To access your properties you probably wrote something like this:

```
Publication * pub=[[Publication alloc] init];
pub.title=@"Test"; // dot setter syntax
[pub setAuthor:@"Chris"] method setter syntax;
```

```
NSLog(@"Title:%@, Author:%@",[pub title], pub.author);
```

Step 2 – Inheritance

8. Create two new classes Article and Book both of which inherit from Publication
    a. When you create the new Class (File->New File->Objective-C class) set the subclass to Publication
9. Book should have additional properties for
    a. Isbn
    b. Price
10. Article should have a property for
    c. Journal
11. Override the publish method for both classes and add NSLog statements indicating that either “Book” or “Article” “ was published”.
12. Also call the publish method on the base class from the derived class publish method. You will need to use the **super** keyword to reference the base class.
13. In main.m create instances of **Book** and **Article** and call **publish** on both.

Step 3 – Categories

Categories can be used to extend existing classes and to illustrate that we will add logging capabilities to the Publication class.

Using File -> New ->New File-> Objective-C Category:

- Create a category called Logger for class Publication
- Add a method called **log** that calls NSLog to output all of the publication details to the output window
- Make sure you add the definition for the log method to your header (.h) file
- In main.m import the newly created category definition file (.h)
- Call log on both instances of **Book** and **Article**
- Run and test the application

Step 4 - Protocols and Delegates (If you have time)

Let’s make publications fire a published event when they are published. To do that we need to set up the **Publication** class to hold a reference to class instances it will fire events to. We also need to define a protocol to describe the methods that will be fired in the other classes.

14. Create a new protocol called **PublicationDelegate**. (Menu option File->New->New File).
15. Define a required void method called published with a single parameter of type **Publication**. You will need to add a forward declaration for the Publication class using.

```
@class Publication;

@protocol PublicationDelegate

….

@end
```

Next we need to add a property to hold a reference to the class instance to which we will fire publication events.

16. Open Publication.h
17. Add a delegate property for the protocol

It should look something like this. Make sure you synthesize it!

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

Next we need to raise an event when a publication is published.

- In the **publish** method of your Publication class add the following line

```
[self.delegate published:self];
```

The [self.delegate bit means we are accessing the delegate property. We are then calling the published method passing the current class. So when a publication is published it raises the published event method and passes itself as a parameter.

All we need now is an object that it going to subscribe to these published events.

- Create a class called **LibraryMonitor**
- Import the **Publication** and the **PublicationDelegate**
- Add the **PublicationDelegate** protocol to the interface definition

```
@interface LibraryMonitor:NSObject <PublicationDelegate>
```

- In LibraryMonitors.m file implement the published method from the protocol and add an NSLog statement to it confirming which book has been published and the year

Almost there now. We just need to hook everything up in main.m

- Open main.m
- Create an instance of LibraryMonitor main.m (Import LibraryMonitor)
- Assign the LibraryMonitor instance to the delegate property of both your article and book instance
- Run and test

When publish is called the Published method should fire on the LibraryMonitor class indicating that you have hooked up your classes for delegation.

Exercise 3 – Views and View Controllers – Part 1

Objective

To help you understand how an iOS app works under the hood. By creating and app using pure code from scratch you will begin to understand how iOS apps actually work.

Overview

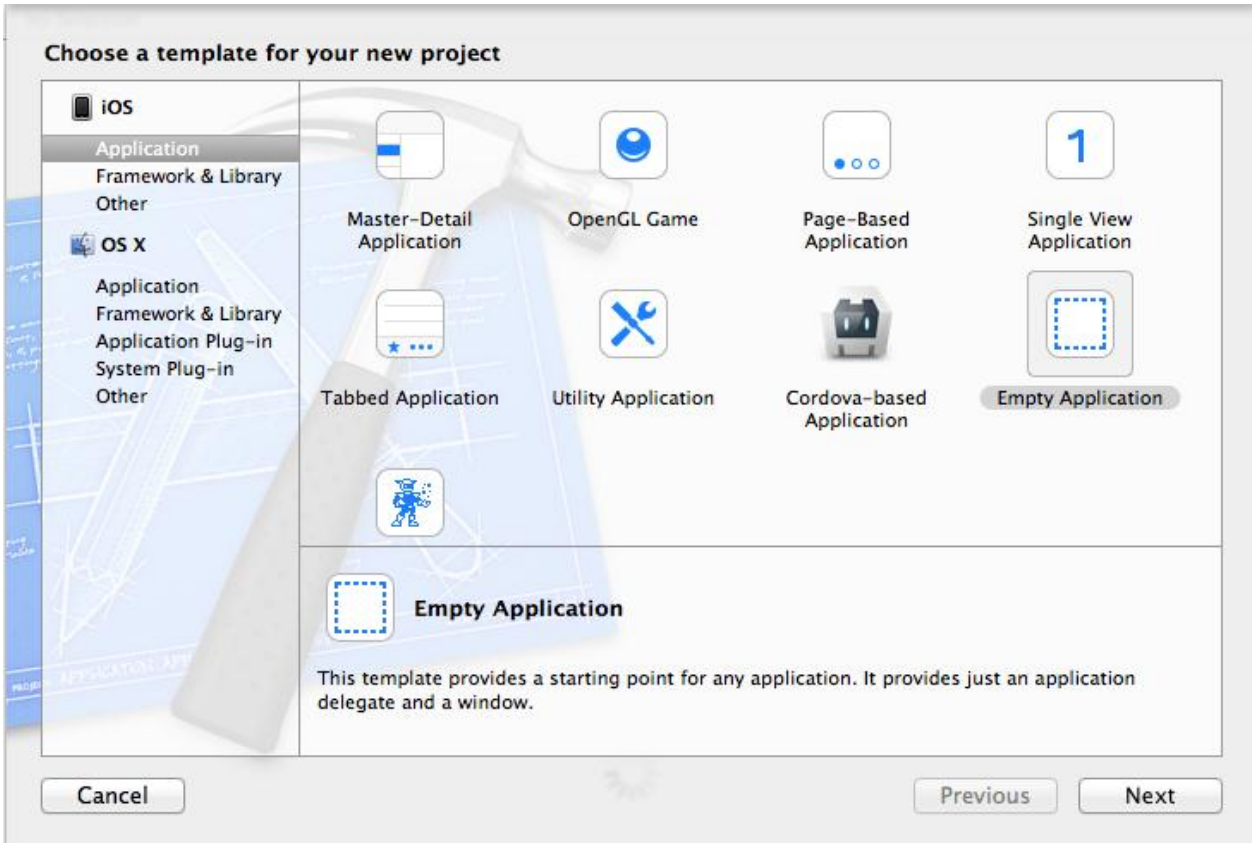
You will create an iOS app from the ground up using view controllers, navigation controllers and tab bar controllers.

Estimated Duration

15 Minutes

Step 1 – Creating an iOS application project

1. Create a new project in XCode but this time from the iOS section choose Application and then Empty Application as below.



2. Choose **Device Family** iPhone.
3. Save it to folder **Documents\QAIOSDEV\Exercise\03 Views and View Controllers\Starter**. Call it **CodedUIInterface**.

An empty application just contains the basic files required for an iOS application. It won't do anything other than start and display a blank window.

4. In the Supporting File folder, open main.m

Here is the application's start point, just like any Objective-C app. It uses the iOS UIKit framework function UIApplicationMain to start the application by passing it the name of the class to use to send applications events to. This is the AppDelegate class and is located in AppDelegate.h and AppDelegate.m.

5. Open AppDelegate.h

Notice it has a single property to hold the applications window and it supports the UIApplicationDelegate protocol that defines the application event methods.

6. Open AppDelegate.m and have a look at some of the methods it implements.

The implemented methods are part of UIApplicationDelegate and are called by the application when application specific events occur.

The key method is **application:didFinishLaunchingWithOptions**, which is basically an app loaded event.

application:didFinishLaunchingWithOptions, creates the main application window to the size of the screen. It sets the window's background colour and makes it the primary output window bringing it to the front so that it displays.

We need to create a specialist view called a UILabel (inherits from UIView). UILabels are used all the time in iOS apps as, you guessed it, labels.

7. Open AppDelegate.m and locate **didFinishLauchingWithOptions**
8. Add the following lines of code just after the alloc/init of **self.window**.

```
CGRect pos=CGRectMake(50, 100, 150, 25);

UILabel *lbl =[[UILabel alloc] initWithFrame:pos];
```

Here you're creating a label using a CGRect structure to specify the position and size of the label.

CGRectMake function has the following spec and returns a C struct.

```
CGRect CGRectMake(CGFloat x, CGFloat y, CGFloat width, CGFloat
height)
```

Next we need to add some text to the label.

9. Set the labels **text** property to "Hello World" or whatever you want.

Finally, we need to add the label to the window. To do that call **addSubView** which is a common method to add views to windows or other views.

```
[self.window addSubView:lbl];
```

10. Run and test the application.

"Hello World" should display on the screen.

11. Change the UILabel to a UITextField and so that you can see its border add the following line.

```
lbl.borderStyle=UITextBorderStyleRoundedRect;
```

12. Run and test the application.

You should now be able to enter information into the text field.

If you have time

13. Create a View Controller by choosing File->New-File
14. Choose Cocoa Touch - Object-C Class and create a class called MyViewController to be a subclass of UIViewController
15. Remove the code you just added to the AppDelegate and place it in the **viewDidLoad** method of MyViewController
16. Change **[self.window addSubView** to **[self.view addSubView**
17. In your AppDelegate.m
    a. Import MyViewController
    b. In didFinishLaunchingWithOptions
        i. Create an instance of MyViewController
        ii. Assign the instance to self.window.rootViewController
18. Test your application. It should still run but this time in the control of a view controller.

Exercise 3 – Views and View Controllers – Part 2

Objective

Let's make our life easier and use Interface Builder to build a user interface.

Overview

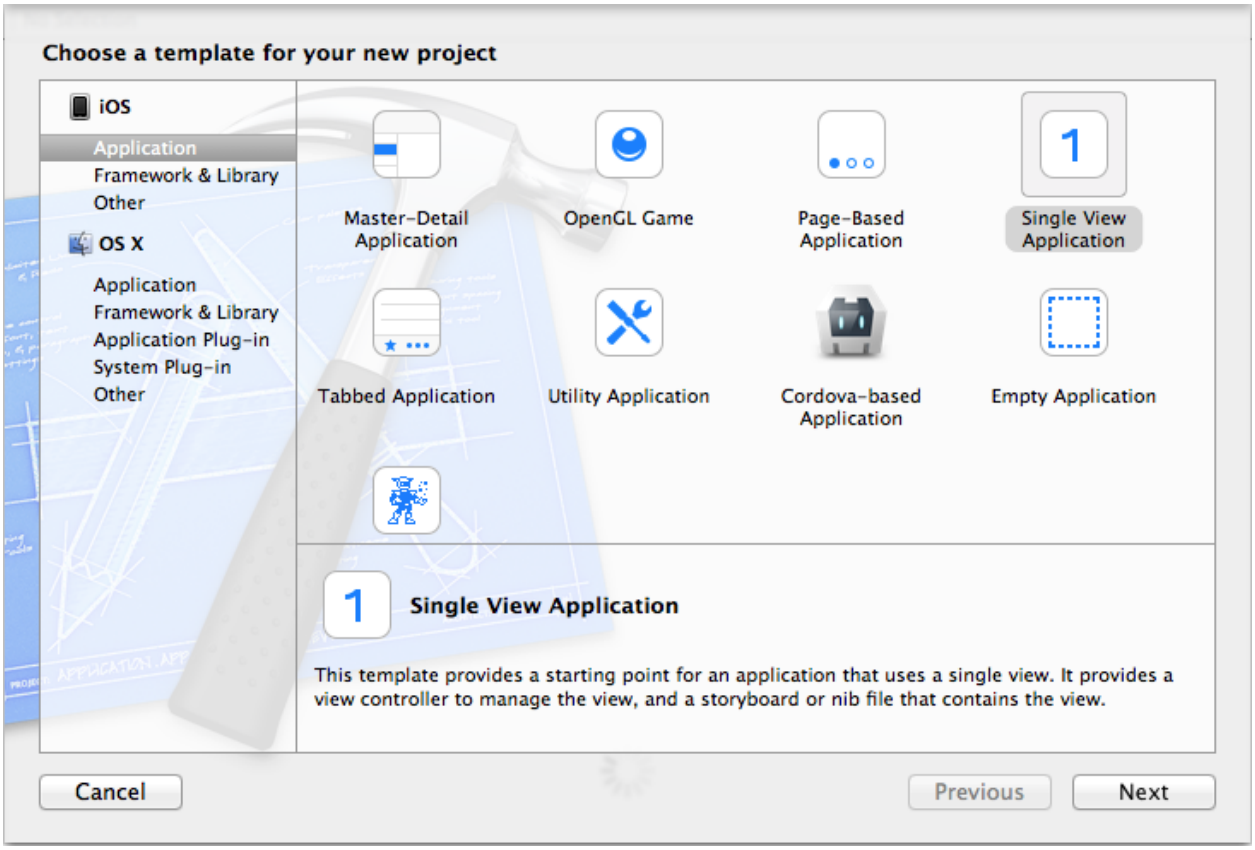
You will create a user interface using Interface Builder and will define a storyboard describing two scenes and a segue to link them together.

Estimated Duration

45 minutes

Step 1 – Creating an iOS application project

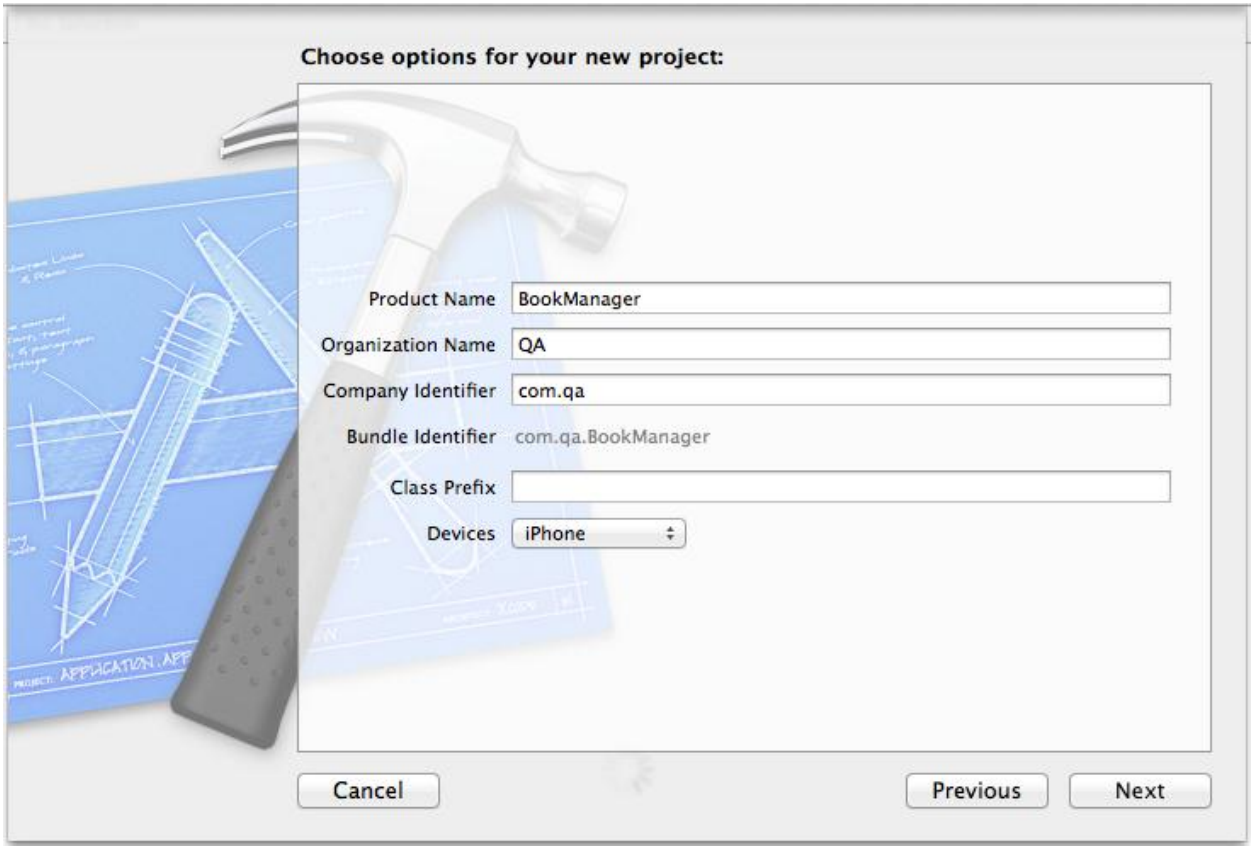
1. Create a new iOS project as before but this time create a **Single View Application**



2. In the next window enter:
    a. Project name: BooksManager
    b. Organization Name: QA (or your company's name)
    c. Company Identifier: com.qa (or your company's backwards domain)

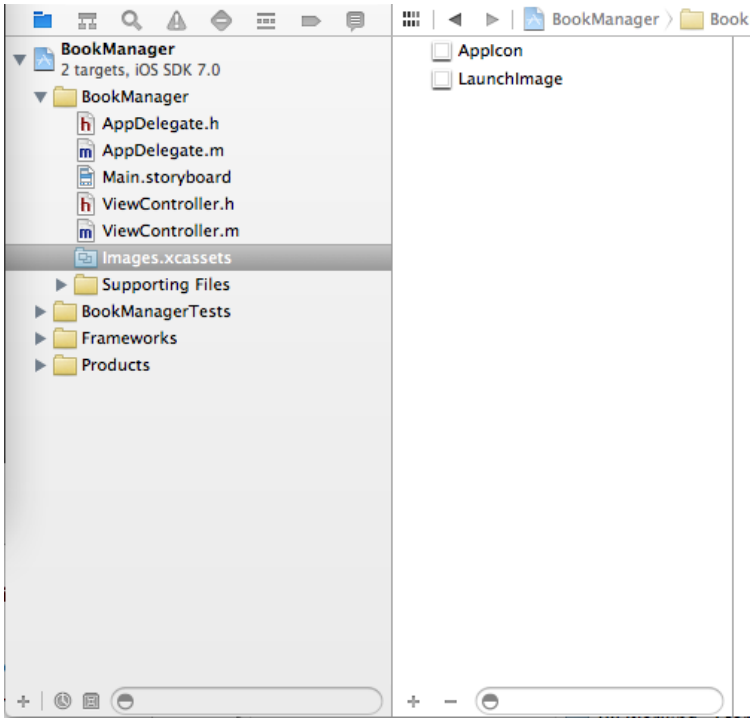
d. Class Prefix should be blank

e. Devices should be iPhone



3. Create the project in QAIOSDEV\Exercise\03 Views and View Controllers\Starter

4. In the project navigator window, select Image.xcassets and press the + button at the bottom of the Image assets window and choose **Import**



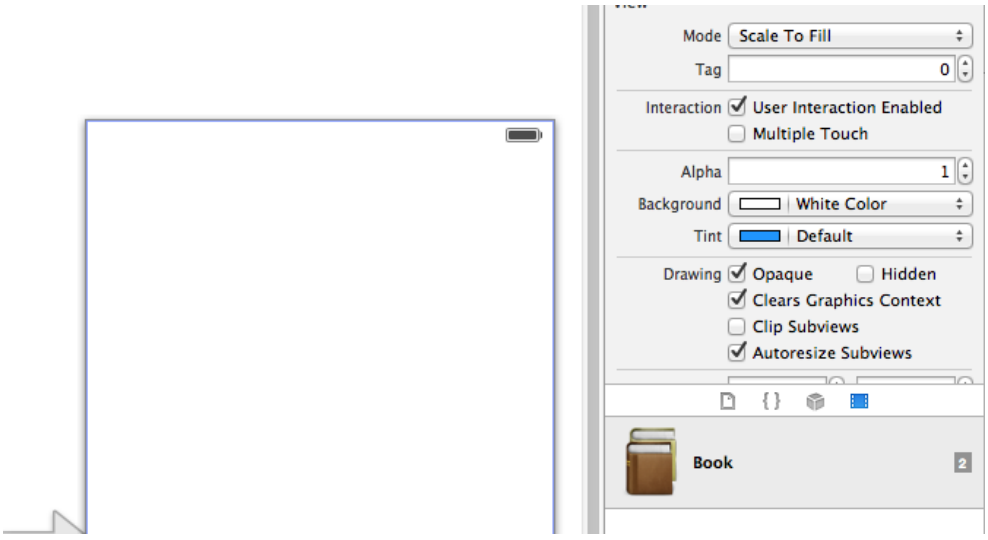
a. From directory QAIOSDEV\Exercise\03 Views and View Controllers\ **Include** add files
    i. Book.png
    ii. Book@2x.png

5. Click on the BookManager folder
    a. Select **File->Add Files to BookManager**
    b. Select From directory QAIOSDEV\Exercise\03 Views and View Controllers\ **Include** files
        i. Book.h, Book.m
        ii. Publication.h, Publication.m

Step 2 – Creating a Scene

Next we will use the default storyboard to create an initial scene.

6. Select **Main.storyboard**
7. Notice a scene has already been created by default
8. Make sure the right hand Utilities Window is displayed
9. Click on the Media Library button and you should see the book image you just added



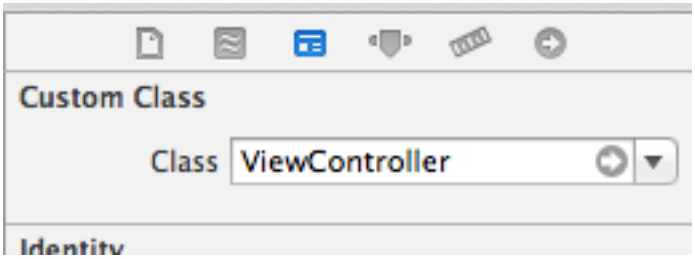
10. Drag the book image onto the scene near to top left corner

Next we need to map our scene to a view controller class.

11. Chose **File->New->File** and create an Objective-C class that is a subclass of UIViewController and call it BookSummaryViewController

12. Now, from you scene press the yellow control at the bottom of the scene

13. In the Inspector window (right side) click the third tool bar button

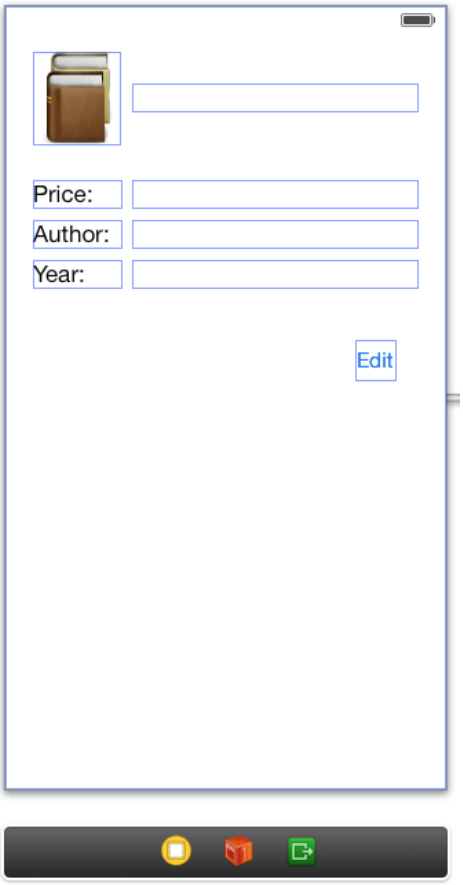


14. Change the class name in the Class field (above) to be **BookSummaryViewController**

Whenever you add a scene, you always need to create a new view controller class to act as its handler and then link the two together.

We can now add some other UI to our app.

15. Using the Object Library (press button next to the Media Library you just used) add UILabels to give the following scene layout to display book info.

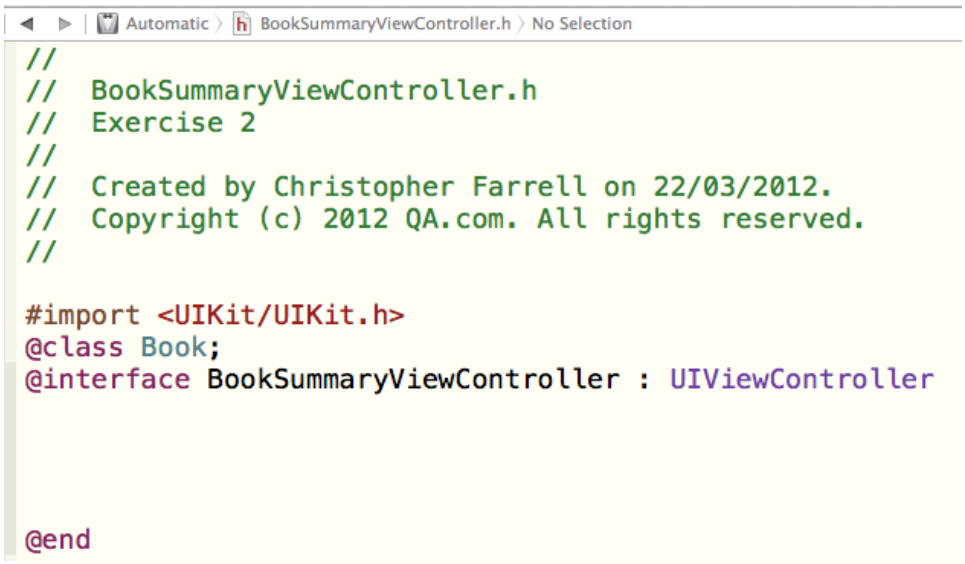


16. Also add a Button control and label it **Edit**.

Step 3 – Connecting as Scene

Once the UI has been laid out you know need to connect it up to your view controller class.

17. From the toolbar click on the Assistant Editor button to reveal the code window for the selected scene
18. Make sure BookSummaryViewController.h is selected in the jump bar



19. For each of your labels select, press **CTRL** and drag over to the code window then let go somewhere in between the @interface and @end markers. A popup should appear.
20. In the text box within the popup type in the name of the label.
21. For the label next to the image call it titleLabel
22. The other labels should be named similarly based on the caption of the label they are next to e.g.
    a. priceLabel
    b. authorLabel
    c. yearLabel

Notice each time a property was created.

23. Select BookSummaryViewController.m
24. Edit method viewDidLoad

25. Inside viewDidLoad fill in some book details for each of the label properties

```
self.titleLabel.text=@"Xcode for Beginners";
self.priceLabel.text=@"25.99";
self.authorLabel.text=@"Fred Bloggs";
self.yearLabel.text=@"2012";
```

Run and test the application

Step 4 – Adding a Segue

We will now create a segue form the book summary to another view controller

26. From the Object Library drag a View Controller onto the storyboard (to the right of the existing one).
27. Create a new ViewController class called EditBookViewController and link the scene and class together (see earlier)
28. Now select the Edit button on the BookSummaryViewController, press **CTRL** and drag the mouse over to the EditBookViewController and let go of the mouse.
29. A popup should appear that will ask you if you want a push, modal or custom segue. Choose **push.**

Push segues need to work with navigation controllers so to get this to work you need to introduce one.

30. Select BookSummaryViewController
31. From the Xcode menu choose Editor->EmbedIn->Navigation Controller
32. That's it.
33. Run and test. You should be able to segue between view controllers.

You might notice that the UI is disappearing behind the status bar. To avoid that happening

34. Select BooksSummaryViewController
35. Reveal Attributes Inspector (fourth button from left in utilities window)
36. Uncheck **Extend Edges – Under Top Bars** checkbox

You can design your apps in iOS7 to fill the screen and that includes extending under the status bar and tab bar if one is there.

Step 5 – If you have time

So we are performing a segue between view controllers but how do you pass data between them?

In this step you will pass a Book class from BookSummaryViewController to EditBookSummaryViewController.

37. Duplicate the BookSummaryViewController UI in EditBookSummaryViewController but replace the labels used to display book data with UITextFields.
38. Import Books.h into BookSummaryViewController.m and EditBookSummaryViewController .h
39. Add a property to EditBookSummaryViewController of type Book called thisBook
40. In viewDidLoad of EditBookSummaryViewController assign the book detail fields from the thisBook property to your UI text fields so they are displayed on the screen.
41. In BookSummaryViewController handle method **prepareForSegue**
    a. Retrieve the destinationViewController and cast it to EditBookSummaryViewController
    b. Create an instance of a Book, assign data to its fields from your UILabel properties
        i. Convert strings to doubles using string method doubleValue
        ii. Convert strings to integers using string method intValue
    c. Assign your book instance to the **thisBook** property of the destination EditBookSummaryViewController. This may help.

```
self.titleTextField.text=self.thisBook.title;

self.priceTextField.text=[NSString
stringWithFormat:@"%.2f",self.thisBook.price] ;

self.authorTextField.text=self.thisBook.author;

self.yearTextField.text=[NSString
stringWithFormat:@"%i",self.thisBook.year] ;
```

42. Run and test. You should be passing book data from one view controller to another.

Exercise 4 – Table Views

Objective

In this exercise, you will use table view controllers. By the end, you will have learned how to use Interface builder to create complex user interface flows with minimal code.

Estimated Duration

1 hour.

Step 1 – Creating a Storyboard

1. Open iOS project in **QAIOSDEV\Exercise\04 Table Views\Starter\BookManager\BookManager.xcodeproj**
2. Click on Main.storyboard
3. Notice it has a single scene from the previous exercise for editing books

We want to now display a list of books to be edited and to do that we will use a table view.

4. From the Object Library tab in the Library window on the left side. (menu View->Utilities->Show Object Library) click and drag the table view controller icon onto the storyboard.

5. Now create a new Objective-C class called **BookListViewController** that is a subclass of **UITableViewController**. (Use File->New->File and choose Objective-C class as you have previously)
6. Using the Identity inspector, set the custom class of the table view controller you added to the storyboard to **BookListViewController**.

We've now created a storyboard scene and connected it to a view controller class ready for coding.

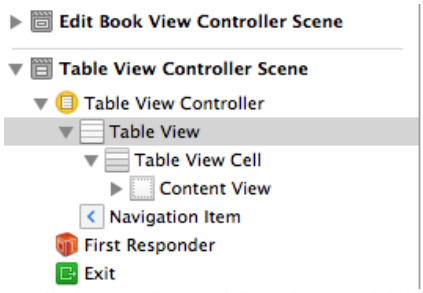
Step 2 – Configuring the storyboard

Next, we need to set up the storyboard flow. The first thing we need to do is identify the initial view controller. At the moment it's the one with the arrow pointing to it which should be the EditBookViewController scene.

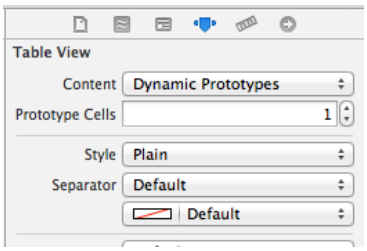
7. Click on and drag the arrow to the BookListViewController scene
8. Click on BookListViewController scene press **CTRL** and drag your mouse across to EditBookViewController scene and let go.
9. Choose a push segue.
10. Use the Editor menu to embed BookListViewController in a navigation controller as you did in the previous exercise.

Step 3 – Setting up the Table Cell

11. Expand the document outline using the button on bottom left hand corner of the storyboard



12. Click on the table view within the table view controller. This will select the table view
13. Now view the table view's properties in the attributes inspector



Notice its Content is set to Dynamic prototypes. This means the table view will be populated dynamically from data we provide in our BookListViewController class.

14. Select the Table View Cell and again view its attributes. Notice Style is set to custom. Custom allows us to create a custom cell layout and style.
15. Change style to **Subtitle**. So that we can use a predefined style.
16. Set **Identifier** to **BookCell**. We will use the identifier later when creating cell instances.

We are now ready to populate our table view.

Step 4 – Retrieving Data

17. Open BookListViewController.m
18. Import Book.h
19. Setup an instance variable (ivar) to hold an array of books. Add it to the @interface BookListViewController in between { } and call it _computerBooks.

It should look like this:

```
#import "BookListViewController.h"
#import "Book.h"
@interface BookListViewController ()
{
     NSMutableArray * _computerBooks;
}
@end
@implementation BookListViewController
```

20. Add the following code (bold) to the **viewDidLoadMethod**

```
- (void)viewDidLoad
{
    [super viewDidLoad];
     _computerBooks=[Book getComputerBooks];
     [self.tableView reloadData];
}
```

reloadData is a method on a table view that forces it to refresh itself.

21. You now need to tell the table view how many sections it should display and how many rows should be displayed in each section
    a. Find method numberOfSectionsInTableView and return 1
    b. Find method tableView: numberOfRowsInSection and return _computerBooks.count

Step 5 – Formatting a Cell

The final piece of the jigsaw is to provide a method that actually generates the cells.

22. Find method tableView:cellForRowAtIndexPath
23. Change the CellIdentifier string from "Cell" to the one we specified earlier "BookCell"

24. After the comment // Configure the cell you need to
    a. Retrieve the correct book from _computerBooks array using the parameter indexPath.row into a local variable
    b. Fill in the cell's textLabel, detailtext, and imageView

The following code should help:

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

{

    static NSString *CellIdentifier = @"BookCell";

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];


    // Configure the cell...

    Book *book=_computerBooks[indexPath.row];

     cell.textLabel.text=book.title;

     cell.detailTextLabel.text=book.author;

     cell.imageView.image=[UIImage imageNamed:@"Book"];

    return cell;

}
```

25. Run and test the application. You should see a list of books.

Step 6 – Creating a segue (If you have time)

We now need to add editing functionality to the app. To do that we need to handle the segue between the two views so that we can pass the selected book.

26. Add method prepareForSegue to class BookListViewController

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

This method is called just before a storyboard segue, so is the ideal place to pass data. The segue parameter has a property **destinationViewController**, which is the view controller being navigated to.

27. Import EditBookViewController.h
28. Within **prepareForSegue,** assign segue.destinationViewController to a local variable of type EditBookViewController.
29. Get the selected index using [self.tableView indexPathForSelectedRow].row;
30. Retrieve the corresponding book from the _computerBooks array using the selected index

31. Assign the retrieved Book to the **thisBook** property of EditBookViewController.

Your code should look something like this:

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

{

 EditBookViewController *vc= segue.destinationViewController;

 NSIndexPath *idx=[self.tableView indexPathForSelectedRow];

 vc.thisBook=_computerBooks[idx.row];

}
```

32. Run and test the application. It should build and run as before with no errors. This time it should edit the book.

Step 7 – Adding update features (If you have even more time)

Add a Bar Button Item to the navigation bar of EditBookViewController and create an event handler for it for the TouchUpInside event. The event handler should update the fields of the thisBook Book instance with the data held in the UI. It should then tell the navigation controller to pop the current view controller off the stack.

The event handler should look something like this:

```
- (IBAction)donePressed:(id)sender {
     self.thisBook.title=self.titleTextField.text;
     self.thisBook.price=[self.priceTextField.text doubleValue];
     self.thisBook.author=self.authorTextField.text;
     self.thisBook.year=[self.yearTextField.text intValue];
     [self.navigationController popViewControllerAnimated:YES];
}
```

You will also need to add the following method to BookListViewController to ensure the changes appear in the list:

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

{

     [super viewDidAppear:animated];

     [self.tableView reloadData];


}
```

Exercise 5 – Extending the User Interface

Objective

In this exercise, you will extend the existing books application UI.

Estimated Duration

45 minutes.

Overview

We will use some additional controls and techniques to extend the user interface.

Step 1 – Setting the tintColor

Let's start by choosing a tintColor for the app. This will help give the app a distinctive style and will be used as the key action color.

1. Open iOS project **Documents\QAIOSDEV\Exercise\05 Extending the User Interface\Starter\BookManager\ BookManager.xcodeproj**
2. Open AppDelegate.m and set the window tintColor to a color of your choice

```
self.window.tintColor=[UIColor purpleColor];
```

This tintColor will be inherited by all child views. When you navigate to the Edit Books scene, the Back button will be in your chosen tint color.

Step 2 – Enhancing the UI

3. Open Main.storyboard

Notice the Edit Book scene has changed. It's now a table view controller. If you inspect its table view you will see that it's set to "Static Cells" rather than "Dynamic Prototype" which is what we used in the last exercise. Static Cells means we design the table view in place, statically, most of which is already done.

There is one space left and that will be for the publication date.

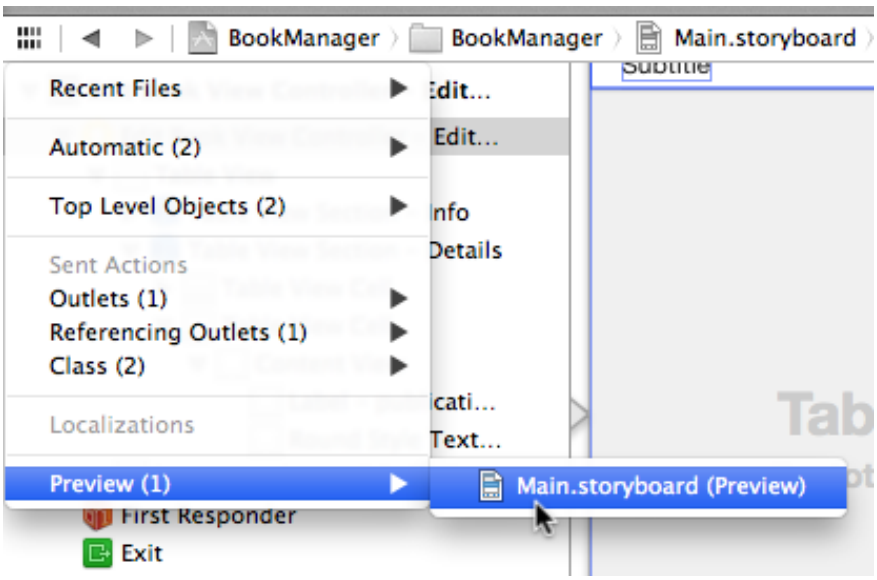
4. In the table cell underneath the price cell, add a label and text field for **publication.**
5. Open the Assistant Editor window, make sure EditBookViewController.h is open and then CTRL drag from the text field to the editor window and create a property called **publicationDateTextField.**
6. Run and test the app. Edit a book.

You should be able to edit a book but the publication date won't display.

7. Press CMD + LEFT ARROW to rotate the screen. Oops!

Sounds like a job for Auto Layout.

8. Xcode will give you a little help here if you press CTRL ALT then, from the jump bar as below, select Preview Main.storyboard.



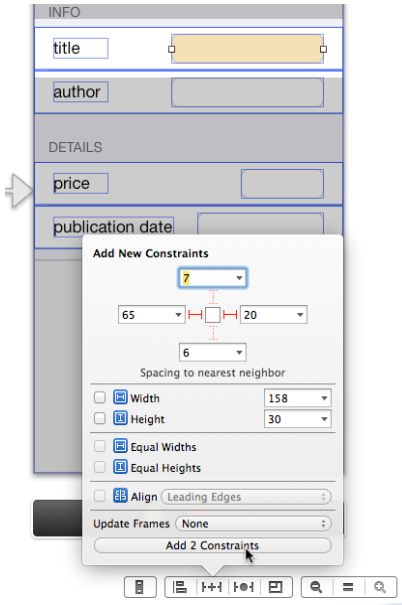
A preview window will display that shows you how the scene will look on a device.

Press the arrow at the bottom of the preview window and the device will rotate.

We can use this to help us with layout.

9. Click on the title text field then click the Pin button on the constraints bar (at bottom of the storyboard)

10. You can set the spacing to the text fields nearest neighbour (either the edge of the cell or the label)
11. Click on all four faint red lines to indicate you want to add those constraint then press Add 4 Constraints



12. Now select both the label and text field, so that we can vertically align them in the cell. Click on the Align button.

13. Tick the Vertical Center in Container check box and press Add 2 Constraints
14. Finally, select the label and press the Pin button again then anchor the label to the left side by clicking the left horizontal faint red bar.
15. Because the label will shrink to fit the text, the layout is inconsistent so select both the label and text field then press the Resolve Auto Layout Issues button.

16. Select Update Frames and the layout will adjust the label and become consistent.

You have now added constraints to allow the first cell to adjust to changing UI conditions.

A quick way to add a set of basic constraints is to ask Interface Builder to do it for you.

17. Select the next table cell. Make sure the Content View is selected. (Click on cell then click on it again or select it from Document Outline)
18. Press the Resolve Auto Layout Issues button and select Add Missing Constraints in Table View Cell
19. Repeat for the other cells

Notice in the Document Outline window, the list of constraints for each control is held under the Constraints collection.

20. Test the UI now rotates and adjusts satisfactorily.

To get the layout working exactly the way we want it, its likely we will need to configure a control's Content Compression Resistance priority (on Size inspector) and manage the priority of some of the constraints. By managing these priorities, we can exactly describe what should be compressed and what shouldn't relative to each other in a layout.

Step 3 – Setting style and type

Let's now setup styling within the app.

21. Set the font of each of the labels in the Edit Book scene to style Headline.
22. Set the font of each of the text fields to style Body
23. Set the place holder text for each test field
24. Set the text alignment of each text field to right align

25. Set the keyboard of the price text field to be Decimal Pad

Because we are using Dynamic type, when the user changes Text Size in Settings->General area of iOS our app can now respond. Try it.

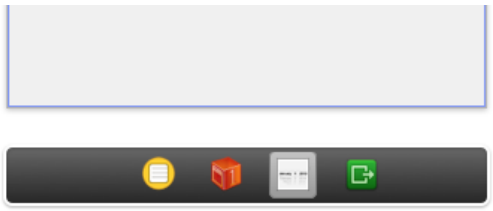
Take a look at EditBookViewController.m. Notice in viewDidLoad there is a call to NSNotificationCenter (we will cover this later in the course). It sets up a notification for when text size has changed. When it does, we fire the method **contentSizeChanged** and use that to reset the font size for our controls using [UIFont **preferredFontForTextStyle:**

It all works except for our added publication date. If you have time later on update this method to allow publication label and text field to respond to text size changes.

Step 4 – Date Handling

Up to now, to keep things simple, we have just been capturing the publication year. We now need to start capturing the actual publication date Open EditBookViewController.m

26. Drag a Date Picker onto the Edit Book's scene bar



27. Click on the added date picker and set its Mode property in Attributes Inspector to Date

28. Press CTRL and drag the date picker on the scene bar onto the Assistant Editor of EditBookViewController.h and create a property called datePicker.

29. Edit method **viewDidLoad**

30. Assign **self.datePicker** to the **inputView** property of the publicationDateTextField field (holds the published date)

The **inputView** property of a text field allows you to replace the keyboard as the field's input mechanism. We are collecting dates from this field so it makes sense.

31. Run and test the application. When you touch the Published field the date picker should appear.

When the date in the picker changes it should update the date field.

32. Press CTRL and drag/drop from the date picker on the scene bar onto the Assistant Editor of EditBookViewController.m and create a ValueChanged event handler method called **dateChanged**

33. Add the following line of code:

```
self.publicationDateTextField.text=[self.datePicker
toString:@"dd MMM YYYY"];
```

34. Add this line of code to viewDidLoad

35. Now you need to alter the donePressed method so when the Done button is pressed it will update self.thisBook.publicationDate with the selected date.

    a. Add the line update the book instance's publication date

```
self.thisBook.publicationDate=self.datePicker.date;
```

Step 5 – Validating text Input (if you have time)

The next thing to do is add some validation to the text input and to keep things simple let's just implement the following rules:

- Book titles should <= 35 characters long

To do that, we need to be able to see text entry before it is displayed in the control, which means we need to handle the text fields UITextFieldDelegate protocol.

36. Open file EditBookViewController.h and add a reference to the class definition for UITextFieldDelegate protocol.

37. Open EditBookViewController.m and implement method

```
-(BOOL)textField:(UITextField*)textField
shouldChangeCharactersInRange:(NSRange)range
replacementString:(NSString*)string
```

38. Add code to calculate what the length of the new string would if the change was allowed to complete.

To calculate this, you need to now the length of the original text (**textField.text.length** plus the length of the new characters (**string.length**). You also need subtract the length of characters that will be replaced or deleted by the change (**range.length**).

The result will be the length of the new string and if it exceed 35 then you need to return NO from the method which will prevent the change taking place in the text field.

Your code should look something like this:

```
-(BOOL)textField:(UITextField *)textField
shouldChangeCharactersInRange:(NSRange)range
replacementString:(NSString *)string

{

  int newLength = [textField.text length]+[string length] -
range.length;

     if (textField==self.titleTextField && newLength> 35) return
NO;

     return  YES;

}
```

We need to tell the title field to delegate its UITextFieldDelegate event method calls to our view controller, EditBookViewController class.

39. Click on the book title field, press **ctrl** and drag the mouse down to the Files Owner icon (bottom, right icon on scene info bar). When the popup appears select **delegate.**

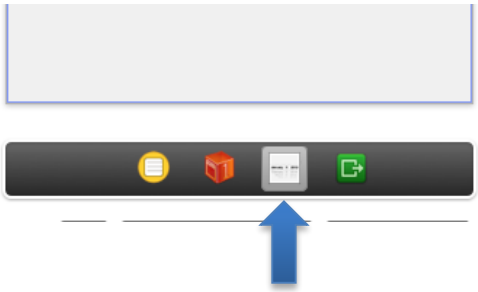
40. Run and test the application. You should be restricted to 35 chars for a title.

Step 6 – Unwind Segue (If you have even more time)

When the Done button is pressed replace the donePressed event method with an unwind segue.

You will need to:

41. Remove the viewWillAppear event in BookListViewController
42. Create a segue method in BookListViewController to handle the unwind segue
    a. Call [self.tableView reloadData] why is this better?
43. **Make sure you add the prototype for your segue method in BookListViewController.h**
44. Create the unwind segue by dragging from the Done button to Exit handler on the scene bar.



45. You will also need to handle method **shouldPerformSegueWithIdentifier**. This will return true if a segue is allowed to take place. We can use it to update the book as we did in **donePressed.**

```
-(BOOL)shouldPerformSegueWithIdentifier:(NSString *)identifier
sender:(id)sender
{


    self.thisBook.title=self.titleTextField.text;
    self.thisBook.price=[self.priceTextField.text doubleValue];
    self.thisBook.author=self.authorTextField.text;
    self.thisBook.publicationDate=self.datePicker.date;


    return YES;
}
```

Exercise 7 – Working with Stored Data

Objective

In this exercise, we will take a break from the Books App and write some code to save and retrieve data to and from file stores.

Overview

We will start by getting to grips with NSFileManager and how we can store any kind of data by creating and loading files. Next, we will look at how to store information more simply using NSUserDefaults to store key value pairs. Finally, we will use SQLite to store data in a SQLite database.

Estimated Duration

40 minutes.

Step 1 – File Management

1. Open iOS project in **Documents\QAIOSDEV\Exercise\07 Working with Stored Data\Starter\Exercise 7\Exercise 7.xcodeproj**
2. Select AppDelegate.m and edit method **didFinishLaunchingWithOptions**.
3. Declare an **NSFileManager** variable and assign a defaultManager instance to it.
4. Using the **NSFileManager** instance, call **URLsForDirectory** to retrieve the URL for the Documents directory (**NSDocumentsDirectory**) in domain **NSUserDomainMask** and assign it to an NSURL variable.

This method returns an array of one so you need to call **lastObject** to get the actual URL.

5. Append the name **Data.csv** to the NSURL to create a file URL. (URLByAppendingPathComponent)
6. Using the **NSFileManager** instance determine if the file at this URL exists
7. If it doesn't carry out the following steps
    a. Create a comma separated string of data and produce an NSData object from it e.g.

```
NSData *data=[@"2009-10-22, 10.78, 98.5; 2009-10-29,
11.28, 92.3" dataUsingEncoding:NSUTF8StringEncoding];
```

    b. Using the **NSFileManager** instance call **createFileAtPath** with the following parameters:
        - File Path - Path of the data file (you need to return the actual file path from the URL first, using the **path** method of NSURL)
        - Data - NSData stream
        - Attributes - nil

This method returns YES if it succeeds.

c. If successful use NSLog to output a success message to the output window

8. Run and test the application

The application should run and the success message should display in the output window.

At this point, your code should look something like this:

```
NSURL *url = [[mgr URLsForDirectory:NSDocumentDirectory
                 inDomains:NSUserDomainMask] lastObject];
NSURL * file = [url URLByAppendingPathComponent:@"Data.csv"];
if (![mgr fileExistsAtPath:[file path]])
{
   NSData *data=[@"2009-10-22, 10.78, 98.5; 2009-10-29, 11.28,
     92.3" dataUsingEncoding:NSUTF8StringEncoding];
   if ([mgr createFileAtPath:[file path] contents:data
       attributes:nil])
   {
       NSLog(@"File %@ Created", file);
   }
}
```

9. Add an else clause to the file doesn't exist if statement. This clause will load the Data.csv file if it already exists.

10. Using the **NSFileManager** instance, call the **contentAtPath** method passing it the **file** path to load the contents of the file into an NSData variable.

11. To convert the NSData instance to a string use the following code

```
NSString *str=[[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding];
```

12. Use an NSLog statement to output the file contents to the output window

13. Run and test the application

Your application should run and the contents of the file should be displayed in the output window.

Step 2 – Storing Data in the User Defaults Database

For simple configuration data, storing data in a file isn't the easiest place to keep it. For that purpose iOS has a user defaults database available that gives you quick and easy access to key/value type data. It's great for storing user preferences information.

14. Select file MainStoryboard.storyboard.

Notice there is a Tab Bar controller and two view controllers one of which has a three part segmented control on it stored in property **colorSettings**. We will use this view controller to set the background color of the view controller but also store the preference so that it is remembered for the next time the app reloads.

15. Select SettingsViewController.m
16. Edit the **settingChanged** method. This method is called when a user chooses a different color using the segmented control.
17. Retrieve an instance of **NSUserDefaults** to a variable using class method **standardUserDefaults**.
18. Using the NSUserDefaults variable call **setValue:forKey** passing
    a. An NSNumber wrapping the value of **self.colorSettings.selectedSegmentIndex**.
    b. forKey – "backGroundColor"

You have to wrapper the **selectedSegmentIndex** in an object wrapper as **setValue** is expecting an **id** type.

19. Call **synchronize** on the NSDefaults instance to save.

Your code should look something like this:

```
NSUserDefaults *defaults=[NSUserDefaults standardUserDefaults];
[defaults setValue:[NSNumber
      numberWithInt:self.colorSettings.selectedSegmentIndex]
      forKey:@"backGroundColor"];
[defaults synchronize];
```

20. Edit **viewDidLoad**
21. Retrieve an instance of **NSUserDefaults** to a variable using class method **standardUserDefaults**.
22. Call **valueForKey** on the NSUserDefaults variable and assign it to an NSNumber variable called **retrievedColor**.
23. If it's not nil
    a. Set the **intValue of retrievedColor** to **self.colorSettings.selectedSegmentIndex**
    b. Set **self.chosenColor** property the intValue of **retrievedColor**
24. Run and test the application

The app should save and remember the stored background color for the settings tab only in between stopping and starting the app.

Step 3 – If You Have Time

Take a look at the KeyChainSecret class. It's an abstraction of the key chain API allowing for the secure storage and retrieval of password strings.

25. Replace the code you have written for the user defaults with calls to the KeyChainSecret class
26. Use the add and update methods to store the selected color. You will need to detect if the key exists using the exists method
27. Retrieve the stored key using secretForKey

Exercise 9 – Asynchronous Programming

Objective

In this exercise, you will explore the asynchronous programming options in iOS and look at both Grand Central Dispatch and Dispatch Queues as well as the Objective-C based Operation Queues.

So you can focus on just asynchronous programming. This exercise is independent from the Books project.

Overview

Although iOS allows you to create and manage threads, it is generally discouraged in favour of Apple's asynchronous programming model which uses queues to manage the processing of asynchronous tasks. It's a much simpler and more scalable solution than managing threads yourself.

Estimated Duration

40 minutes.

Step 1 – Concurrent (Global) Dispatch Queues

1. Open iOS project in **Documents\QAIOSDEV\Exercise\10 Asynchronous Programming\Starter\Exercise 9\Exercise 9.xcodeproj**
2. Build and Run the application. Notice it has two tab button on the tab bar, one for Dispatch Queues, the other for Operation Queues.

On each view, there are for labels containing a number and a Start button. We will use these controls to create asynchronous counters.

3. Open DispatchViewController.m

Each of the labels containing a number are linked to properties

- display1
- display2
- display3
- display4

The button press is connected to method **startPressed**.

The first thing we want to do is set up an asynchronous counter for each of the labels so they independently count up to 10 pausing for one second in between, at the same time (concurrently).

4. Edit method **startDisplayConcurrent**
5. Retrieve the global dispatch queue for the default priority and assign it to a **dispatch_queue_t** type variable called **queue**.

6. Call function **dispatch_async** passing it the **queue** and add a block section. Make sure you close the block section with **;**
7. Within the block section write a for loop that
    a) loops ten times
    b) Sets the text property of the **lbl UILabel parameter** to the value of the loop integer (convert it to a string first). You will need to do this from the main thread!
    c) Call **sleep(1);** to sleep for one second each iteration around the loop

Your code should look something like this:

```
dispatch_queue_t queue;
queue=dispatch get global queue(DISPATCH QUEUE PRIORITY DEFAULT,
0);
dispatch_async(queue, ^{


  for (int i=0;i<11;i++)
  {
     dispatch_async(dispatch_get_main_queue(), ^{
       lbl.text=[[NSNumber numberWithInt:i] stringValue];
     });
     sleep(1);
  }
});
```

8. Edit method **startPressed**
    a) Add a call to **startDisplayConcurrent** passing **self.display1**
    b) Add a call to **startDisplayConcurrent** passing **self.display2**
    c) Add a call to **startDisplayConcurrent** passing **self.display3**
    d) Add a call to **startDisplayConcurrent** passing **self.display4**
9. Run and test the application. Each of the labels should increment at the same time up to 10.

Step 2 – Serial Dispatch Queues

Serial dispatch queues execute tasks in sequence in first in first out sequence. You can use them to control access to resources as an alternative to locking.

In this exercise, we want to set four tasks running on a serial queue each of which will run in turn rather than together.

If you look in the @interface section of DispatchViewController.m, you will see an instance of a **dispatch_queue_t** has been declared called **_serialQueue.** We will use this to hold the serial queue.

10. Edit method **startDisplaySerial**
11. If **_serialQueue** is nil, create a dispatch queue, give it a label of **qa.com.counter** (C string) and assign it to **_serialQueue**
12. Call **dispatch_async** as before and implement a for loop to increment the label every second using **_serialQueue**

At this point your code should look something like this:

```
if (_serialQueue==nil)
{
  _serialQueue=dispatch_queue_create("qa.com.counter", NULL);
}
dispatch_async(_serialQueue, ^{

  for (int i=0;i<11;i++)
  {
   dispatch_async(dispatch_get_main_queue(), ^{
        lbl.text=[[NSNumber numberWithInt:i] stringValue];
   });
   sleep(1);
  }
});
```

13. Edit method **startPressed**
14. Replace calls to **startDisplayConcurrent** with calls to **startDisplaySerial**
15. Run and test the application. Each counter should increment in turn.

Step 3 – Operation Queues

Where dispatch queues are C function implementations, Operations are Objective-C based implementations. In this exercise, we will reproduce the functionality in step 1 this time using **NSOperationQueue** to run operations concurrently and **NSBlockOperation** to define them.

16. Open file OperationViewController.m
17. Look at the **@interface** section and notice an **NSOperationQueue** has been defined called **_queue**. It is created in **viewDidLoad**

We need to implement method **createLabelOperation.** It returns an NSOperation and takes a UILabel as a parameter. It will increment a counter in the label just as with the previous steps. This time it will do it using a type of NSOperation executing concurrently on **_queue**.

18. Edit method **createLabelOperation**
19. Create an **NSBlockOperation** using method **blockOperationWithBlock.**
20. Inside the block body implement a **for loop**
    a) Set the text property of the UILabel **lbl** with the loop integer.
    b) Ensure you update the label on the main thread but use an operation queue rather than a dispatch queue to do it.
    c) Make sure the loop sleeps for one second per iteration
21. Return the NSBlockOperation from the method

At this point your code should look something like this.

```
NSBlockOperation *block=[NSBlockOperation
blockOperationWithBlock:^{
    for (int i=0;i<11;i++)
    {
   [[NSOperationQueue mainQueue] addOperationWithBlock:^{
         lbl.text=[[NSNumber numberWithInt:i] stringValue];
   }];
   sleep(1);
    }
}];
```

22. Edit method **startPressed**
23. Call **createLabelOperation** four times
    a) Pass parameter **self.display1** and assign return to an NSOperation called op1
    b) Pass parameter **self.display2** and assign return to an NSOperation called op2
    c) Pass parameter **self.display3** and assign return to an NSOperation called op3
    d) Pass parameter **self.display4** and assign return to an NSOperation called op4
24. Add the four NSOperations to an NSArray. We want to execute them all at once
25. Pass the array to method **addOperations** of NSOperationQueue **_queue** and set **waitUntilFinished** parameter to NO.

26. Run and test the application. Make sure you press the **Operation Q** tab button (Blue Background). All four labels should increment together.

Step 4 – Operation Queue Dependencies (If you have time)

Modify the **startPressed** method and use the **addDependency** method on an NSOperation to force each of the operations to execute sequentially.

Only one label should increment at a time.

Exercise 11 – Accessing Web Data Services

Objective

In this exercise, you will use Objective-C and the frameworks libraries to run a query against Google's geocoding API. This will return a JSON feed that you will parse using the built in JSON parser and display locations on a map.

You will type in a postcode or a place name and then look it up using the geocoding API. The returned JSON will be parsed to retrieve its latitude and longitude. Eventually you will use this data to add a placemark to the MapView.

Estimated Duration

45 minutes.

Step 1 – Querying Google's Api

1. Open iOS project in **QA\iOS\Code\12 Animation\Starter\Exercise 11\Exercise 11.xcodeproj**
2. Build and run the application for either iPhone or iPad
3. Notice, the app displays a Map and a search bar. To allow mapping and location services, we needed to add the following framework libraries to the project:
    a) CoreLocation.framework
    b) MapKit.framework

    You don't need to do this.
4. Open ViewController.h Notice it's a view controller and has two properties for the
    a) mapView (MKMapView)
    b) searchBar (UISearchBar)
5. The search bar works with delegation so we are implementing the UISearchBarDelegate
6. Open ViewController.m.
7. Look for method **searchBarSearchButtonClicked**.
8. This is a delegate method that fires when the user presses Search on the keyboard. We are simply calling the method **findAddress.**
9. Go to the **findAddress** method

findAddress has setup a URL that queries the Google API passing whatever was typed into self.searchBar.text.

10. Using the NSData class retrieve the data at the url specified in the NSURL variable url – (hint dataWithContentsOfURL)
11. Return it to an NSData object called **data**.

a) Check **data** isn't nil and then add the following code

```
CLLocationCoordinate2D coord=[self parseFeed:data];

[self addLocation:coord zoom:YES withTitle:self.searchBar.text];
```

parseFeed will eventually parse the JSON data and return a coordinate location. At the moment, if you take a look at it, it just outputs the returned **data** object as a string of JSON.

12. Run and Test the application. Type in you postcode and press Search. You should see JSON data for you query appear in the output window. The map won't relocate at the moment!

Step 2 – Parsing the data

We know from the twitter data format that it's a JSON feed and that the top-level object is a dictionary.

It has the following format (simplified), starting with a dictionary with two key value pairs containing basic feed information for results and status. The **results** key value pair is the one we are interested in because the value contains an array that holds the data.

```
{"results" :
[
    {
       "address_components" : [..],
       "formatted_address" : "…",
       "geometry":
       {
            "bounds":{…},
            "location":
             {
                  "lat":51.3123,
                "lon":-0.0744
             }
       }
    }
],
"status":"OK"
}
```

So **results** contains an array with one item in it that is a dictionary.

This dictionary contains a **geometry** key that we need to retrieve. **geometry** itself is a dictionary and contains a **location** key value pair that contains the latitude and longitude values.

We need to navigate down to the **lat, lon** key value pairs to retrieve the location data for plotting on the map.

13. Edit method **parseFeedData**
14. Using the **NSJSONSerialization** class parse the NSData parameter **data** using method **JSONObjectWithData**
    a) Pass **options:NSJSONReadingAllowFragments**
    b) Pass **error:&error** where error has been declared of type NSError.
15. Return an NSDictionary from this method call to a variable called **feed**
16. Check the error object. If it's not nil, there has been an error
17. Return an NSArray from **feed** by calling **objectForKey:@"results"** to give us the results dictionary into a variable called **resultsDictionary**
18. From **resultsDictionary**, retrieve the **geometry** dictionary using **objectForKey** into an NSDictionary variable called **geometry.**
19. From **geometry**, retrieve the **location** dictionary into an NSDictionary variable called **location.**
20. Once we have the **location** dictionary, we can get at the **lat** and **lon** dictionary items within using objectForKey as before. Return them to two NSStrings called **lat** and **lon** respectively.
21. Now return a CLLocationCoordinate2D using function CLLocationCoordinate2DMake passing the lat and lon variables converted to floats using the **floatValue** method.

At this point your code should look something like this:

```
NSError *error;

NSDictionary *feed = [NSJSONSerialization
JSONObjectWithData:data options:NSJSONReadingAllowFragments
error:&error];

if(error)

{

     NSLog(@"Error:%@",error.localizedDescription);

     return kCLLocationCoordinate2DInvalid;

}


NSArray *resultsArray=[feed objectForKey:@"results"];

NSDictionary *resultsDictionary=[resultsArray objectAtIndex:0];
```

```
NSDictionary *geometry=[resultsDictionary
objectForKey:@"geometry"];


NSDictionary *location=[geometry objectForKey:@"location"];

NSString *lat=[location objectForKey:@"lat"];

NSString *lon=[location objectForKey:@"lng"];

return CLLocationCoordinate2DMake([lat floatValue], [lon
floatValue]);
```

22. Run your application. It should now cause the map to navigate to the correct location. **findAddress** calls **addLocation,** which adds a placemark to the requested location on the map.

Step 3 – Going Asynchronous with Blocks (if you have time)

Let's improve things by making the call to Google asynchronous.

23. Go to method **findAddress**
24. Remove all code within, after the creation of the NSURL variable url
25. Create an instance of **NSURLRequest** using method **requestWithURL**, passing **url** as a parameter
26. Setup a call to class method **sendAsynchronousRequest** on class **NSURLConnection** (it's a block method).
    a) Pass your NSURLRequest variable as the first parameter
    b) Pass **[NSOperationQueue mainQueue]** as the queue parameter – so the completion handler runs in the main thread.
    c) Add a completion handler that checks for errors
        i. To check the status code of NSURLResponse you will have to cast it to NSHTTPURLResponse
    d) Within the completionHandler block, add the following code:

```
CLLocationCoordinate2D coord=[self parseFeed:data];

[self addLocation:coord zoom:YES withTitle:self.searchBar.text];
```

27. At this stage, your code should look something like this:

```
NSURL *url=[NSURL URLWithString:sUrl];
NSURLRequest *request=[NSURLRequest requestWithURL:url];
[NSURLConnection sendAsynchronousRequest:request
     queue:[NSOperationQueue mainQueue]
     completionHandler:^(NSURLResponse * resp, NSData * data,
                         NSError *error) {
     NSHTTPURLResponse *response=(NSHTTPURLResponse*)resp;
     if(!error && response.statusCode==200 && data )
     {

          CLLocationCoordinate2D coord=[self parseFeed:data];
          [self addLocation:coord zoom:YES
                  withTitle:self.searchBar.text];
     }

}];
```

28. Test the app still works but this time it will be asynchronous.

Step 4 – Going Asynchronous using delegates

29. Open ViewController.h
30. Notice the class implements protocol NSURLConnectionDataDelegate. We will use methods from this protocol to handle delegated web access.
31. Open ViewController.m
32. Edit method **findAddress**
33. Remove the call to the block method sendAsynchronousRequest.
34. Add a call to the class variable _connection (of type NSURLConnection) **cancel** method. This cancel any currently executing downloads.
35. Now, create an instance of an NSURLRequest assigning it to _**connection** and passing
    a) **initWithRequest:request**
    b) **delegate:self**
36. Data retrieval will begin immediately
37. Passing self allows NSURLRequest to do call-backs such as:
    a) connection:didReceiveData – Data fragment was received

b) connection:didFailWithError – Error occurred

c) connection:didReceiveResponse – Response has started arriving

d) connectionDidFinishLoading – Response has completed

38. Each of these methods have been added to ViewController.m. You need to add code to complete the data download

39. In **didReceiveData** you can append the NSData parameter data to the class level NSMutableData variable called _**buffer**. This will store the data as it arrives

40. In **didFailWithError** just output the error.localizedDescription.

41. In **didReceiveResponse** assign a new NSMutableData object to _**buffer**

42. In **connectionDidFinishLoading** add the following code:

```
CLLocationCoordinate2D coord=[self parseFeed:_buffer];
[self addLocation:coord zoom:YES withTitle:self.searchBar.text];
```

43. Run the application and ensure it still performs geocoding.