SKILL DEVELOPMENT FOR
MOBILE GAME & APPLICATION

মোবাইল গেইম ও এ্যাপ্লিকেশন

এর দক্ষতা উন্নয়ন শীর্ষক প্রকল্প

এডুয়েড

DIGITAL BANGLADESH
Skilled | Equipped | DigitalReady

ICT DIVISION
FUTURE IS HERE

Time Schedule

Module	Time
Module 1: Introduction to iOS, SDK and Tools	(6 Hours.)
Introduction to iOS, SDK and Tools	
Module 2: Objective-C	(15 Hour.)
Introduction to Objective-C	
Data Types and Variables	
Module 3: iOS App Architecture	(6 Hours.)
Model-View-Controller-MVC	
View Controllers	
Storyboard	
Outlet	
Action	
Module 4: Auto Layout	(12 Hours)
Constraints, insufficient and Conflicting Constraints	
Misplaced Views	
Content Hugging and Compression Resistance	
Size Classes	
Module 5: ARC and Object Initialization	(12 Hours)
Automatic Reference Counting (ARC)	
Object Initialization	
Module 6: Storyboards	(10 Hours)
Scenes	
Segues	
Moving Data between Controllers	
Navigation Controller	
Module 7: Table Views	(8 Hours)
Introduction	
Data Source and Delegate	
Simple app using the module taught	
Module 8: Protocols, Categories & Blocks	(12 Hours)
Protocols	
Categories	
Blocks	
Module 09: Networking	(16 h Hours.)
URL Loading System	

Background Execution

Module 15: Notifications.....(6 Hours.)

Local Notification

Push Notification

Module 16: Core Location Framework.....(5 Hours)

Basics

MapKit Framework

Simple App using the module taught

Module 17: Unit Testing .....(5 Hours.)

Introduction

XCTest Framework

Xcode Services

Module 18: SpriteKit .....(8 Hours.)

Introduction to SpriteKit

Simple game using SpriteKit

Module 19: Revenue.....(6 Hours.)

AdMob Integration

In App Purchase

iAD Integration

Module 20: Advanced iOS Concepts.....(5 Hours.)

iBeacon

Localization

Module 21: Deployment.....(8 Hours.)

Uploading to AppStore

Details of iTunes Connect, App ID, Provisioning, Certificate

Generating IPA and Testing over TestFlight

Module 22: Project Work .....(25 Hours)

2 Completed projects have to be completed and delivered for every student (First one is a group project and the last one is the individual project)

Module 1: Introduction to iOS, SDK and Tools

Topic: Introduction to iOS, SDK and Tools

Chapter Overview:

IOS stands for iPhone operating system. It is a proprietary mobile operating system of apple for its handheld. It supports Objective-C, C, C++, Swift programming language. It is based on the Macintosh OS X. iPhone, ipod and iPad all comes with IOS.

In this guide, we'll walk you through everything you'll need to set up and talk about iOS Overview, iOS features, iOS History, iOS Stack and Setting development Environment also Getting Familiar with XCODE.

Learning Outcome:

Learning how to develop software can be one of the most intimidating prospects for any computer enthusiast, and with the growing saturation of applications in mobile marketplaces, it is becoming increasingly difficult to get your work noticed. That's what this series is for, helping you learn iOS development from a conceptual perspective. No prior knowledge of computer programming will be necessary. Over the coming weeks, we'll examine the iPhone's ability to deliver immersive, intuitive content, a unique opportunity for both developers and consumers.

iOS

iOS (formerly iPhone OS) is a mobile operating system created and developed by Apple Inc. exclusively for its hardware. It is the operating system that presently powers many of the company's mobile devices, including the iPhone, iPad, and iPod Touch.

iOS Overview

The operating system that runs on iPad, iPhone, and iPod touch devices. The operating system manages the device hardware and provides the technologies required to implement native apps. The operating system also ships with various system apps, such as Phone, Mail, and Safari, which provide standard system services to the user.

iOS features

Multitasking, Camera Integration, Phone, E-Mail, Safari and Messages, Passbook, Apple Pay, Interactive Notifications, SQLite, Core Data, Social Framework, Apple Maps, iBeacons, Location Services, iCloud, In-App Purchase, Accelerometer, Gyroscope, iAd Rich Media Ads

iOS History:

The first iPhone's operating system had no official name, but Steve Jobs said it was a version of OS X onstage in 2007 because it shared code with the Mac's operating system.

Fig: First iPhone launching by Steve Jobs

The first iPhone's OS was pretty basic. No App Store. No folders. No multitasking. It was still revolutionary because it introduced the concept of "multi-touch" screens to the world. Now every smartphone has a touchscreen.

Fig: Steve Jobs holding iPhone

It also combined "An iPod, a phone, and an Internet communicator," as Jobs explained.

Fig: iPhone

It wasn't until a year later (in 2008) that the iPhone added the App Store in Phone OS 2. Billions of downloads later, the App Store has arguably been the most important part of the iPhone's success.

Fig: iPhone

There were some important software updates a year later in Phone OS 3: push notifications, multimedia messages (this was before iMessage), Spotlight search, cellular data tethering, and the ability to copy and paste.

Fig: Messaging in iPhone

The iPhone 4's release in 2010 came with two big software features: app multitasking and FaceTime. It also introduced the concept of app folders. Apple dropped the "Phone" and started calling its mobile software just "iOS 4."

Fig: iPhone Apps

iOS 5 was a big update in 2012. It introduced iMessage, Siri, iCloud, and Notification Center.

Fig: iPhone Launching Ceremony

It wasn't until iOS 6 in 2012 that Apple parted ways with Google Maps for its own mapping service, which was widely panned at first.

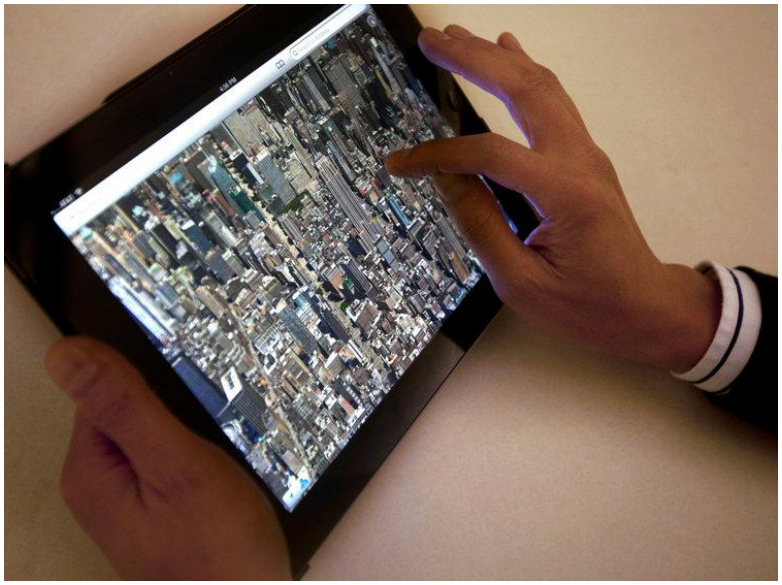


Fig: iPhone Apps in use

iOS 7 marked a complete visual redesign. There are other additions, like AirDrop and Control Center, but the biggest change by far was the overall aesthetic of the operating system.

Fig: New iPhone

2014's iOS 8 was mostly about under the hood improvements. Apple opened up keyboards and Notification Center widgets to app developers. It also started letting apps talk to each other in more advanced ways.

Fig: iPhone Apps View

A later version of iOS 8 introduced Apple Pay.

Fig: iPhone Pay (Apple Pay)

Apple Music was unveiled to the world alongside iOS 9 last summer.

Fig: iPhone Music

Besides Apple Music, iOS 9 didn't bring much new stuff to the table. Transit directions were added to Maps, Siri became a little smarter with contextually aware tech called Proactive, and the Notes app got upgraded with some new features.

Fig: iOS OS launching ceremony

Apple is announcing iOS 10 at its big developer conference on June 13. Stay tuned for what's next.

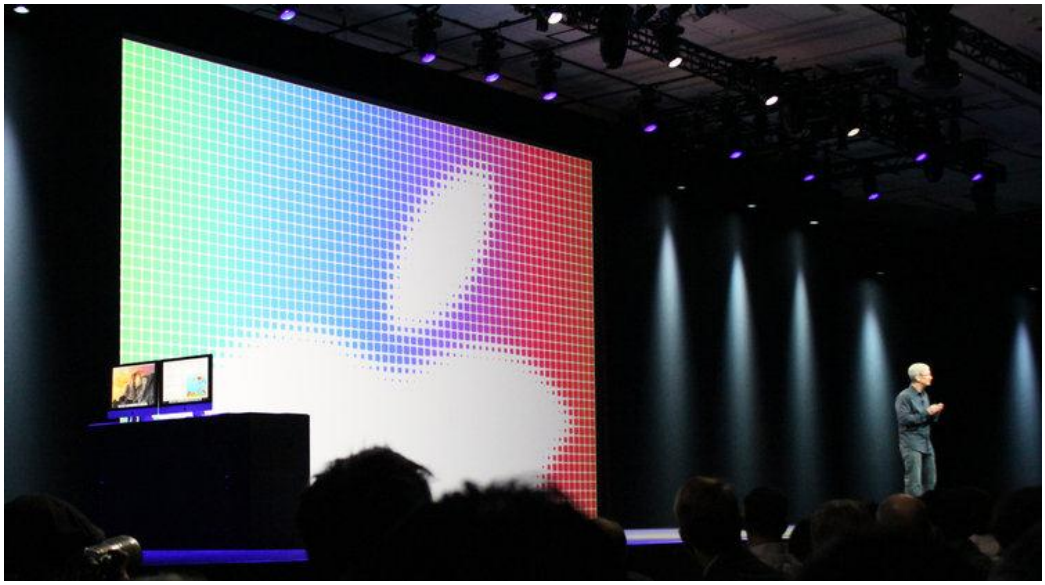


Fig: iOS OS launching ceremony

2007 - iPhone OS 1, 2008 - iPhone OS 2, 2009 - iPhone OS 3, January 2010 - iPhone OS 3.2, June 2010-WWDC-iOS 4, June 2011 - WWDC – iOS 5, June 2012 - WWDC - iOS 6, June 2013 - WWDC - iOS 7, June 2014 - WWDC – iOS 8, June 2015 - WWDC – iOS 9, June 2016 - WWDC – iOS 10, 2017 - WWDC – iOS 11

SDK

Software Development Kit (SDK or devkit) is typically a set of software development tools that allows the creation of applications for a certain software package, software framework, hardware platform, computer system, video game console, operating system, or similar development platform

Alternative Technology:

iOS is a mobile optimized variant of Mac OS X created by Apple Inc. It is distributed with all iPod touch, iPhone, and iPad devices, and only occupies about 500 MB of storage space.

There are three distinct approaches to iOS development:

1. ***Web Application Development***

   The original iPhone OS 1.0 required all non-Apple applications to be web-based and executed within the Mobile Safari web browser. Because Mobile Safari does not support plugins like Adobe Flash or Microsoft Silverlight, this meant that all third party applications were originally written in HTML, CSS, and JavaScript. This method of development remains a viable option today, especially for applications that must be accessible on a wide range of devices or for development teams with an aversion to Mac OS X and Objective-C.

2. ***Native Application Development***

   With the release of iPhone OS 2.0 and the introduction of the iPhone SDK and iTunes App Store, developers have been encouraged to write native applications for the iPhone using Objective-C and Xcode. Native applications are compiled binaries that are installed and executed on the user's device. These applications are granted considerable access to the device hardware, and only native application binaries can be distributed through the iTunes App Store. Because the iPhone OS runs on iPod touch, iPhone, and iPad devices, most applications can be built for all three devices with only minor code variations, though there are significant advantages to optimizing your application for the much larger iPad screen.

3. ***Hybrid Application Development***

   It is also possible to combine the above approaches and create iPhone applications that are installed on a user's device, are written primarily in HTML, CSS, and JavaScript, and are released through the iTunes App Store. Such applications are growing in popularity thanks to open-source libraries like QuickConnect and platforms like PhoneGap, AppCelerator, and rhomobile.

Mobiletuts+ will cover all of the above methods of iPhone OS development, but this tutorial will focus on native application development with the official iPhone SDK using Objective-C and Xcode. This is Apple's recommended method of building native iPhone OS applications for distribution in the iTunes App Store.

Scope of Implementation: In App Store

iOS SDK Tools

Getting Familiar with XCODE

Xcode is an integrated development environment (IDE) for macOS containing a suite of software development tools developed by Apple for developing software for macOS, iOS, watchOS, and tvOS. Xcode provides everything you need to kick start your app development. It already bundles the latest version of iOS SDK (short for Software Development Kit), a built-in source code editor, graphic user interface (UI) editor, debugging tools and many more. Most importantly, Xcode comes with an iPhone (or iPad) simulator so you can test your app even without the physical devices. The latest stable release is version 9.3 and is available via the Mac App Store free of charge for macOS High Sierra and macOS Sierra users

Fig: Xcode interface

Part of being a great software developer is mastering your IDE. You will become more efficient in programming, just by knowing how your environment works. That's why I would like to dedicate some time around the new Xcode 9. This is a perfect tutorial if you are a beginner in iOS Development. Let's start . . .

Left Sidebar

On the top left sidebar, we can see many tabs available. This is probably the most used area as it contains many key features that Xcode offers called navigators. I would start explaining them one by one (from left to right)

Fig: Left Sidebar

1. **Project Navigator (⌘ + 1)**– your file manager. Whatever you need to do with your files it's done here. This navigator enables you to add, remove, edit or group files. Always keep this tab in focus.
2. **Source Control Navigator (⌘ + 2) –** A super-useful navigator meant for source control. It has an integrated support for GitHub accounts, which enables you to manage your repositories directly from your sidebar, and push changes to the cloud without having to use other tools. This is only available from Xcode 9.
3. **Symbol Navigator (⌘ + 3) –** This navigator will enable you to quickly jump to a specific method or a property definition**.** Instead of going through your files to find the method you are looking for, just click on the file you need and then click the desired method or property definition. Useful for files that contain lots of lines of code. You can display the symbols in a hierarchical or flat list.
4. **Find Navigator (⌘ + 4) –** Something that I use quite often. This makes a global search for the given text and returns results that are matching. You can also include various filters in the search.

5. **Issue Navigator (⌘ + 5) –** It's a place that keeps all the errors and warnings that appeared as a result of your coding. Errors are shown with a red color, while warnings are yellow. It will provide you with a detailed log of what is going on. You can display Buildtime or Runtime issues.
6. **Test Navigator (⌘ + 6) –** Is used for running the written test cases. This navigator is a shortcut for your **XCTest's**.
7. **Debug Navigator (⌘ + 7)** – Whenever the app crashes, this tab gets automatically opened. It will provide you with exact lines where the app has stopped and provide you with a reason in the console. Also, you can find useful information about your app's Memory, CPU, Disk and Network consumption.
8. **Breakpoint Navigator (⌘ + 8) –** Another tab that I use a lot for debugging**.** From here, you can easily set breakpoints and monitor their activities. I really love how clean this feature is.
9. **Report Navigator (⌘ + 9) –** Control your continuous integration from here. You should create a bot, and it will provide you access to a detailed information about your bots and the integrations performed on the server.

Right Sidebar

Moving on the other side of Xcode. On the right side, you can see another sidebar with various tabs. But, unlike the left sidebar, this one contains a different set of tabs (depending on your location). On this side, the tabs are called inspectors.

Swift/Objective-C File

If you click on a Swift/Objective-C file you will see the following tabs.

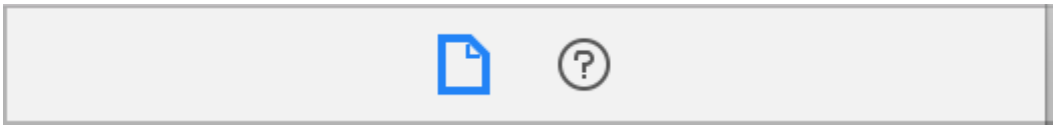


Fig: Right Sidebar

You will use these tabs to less than 10% of your projects, but let's cover them.

1. **File Inspector –** this inspector provides basic details and settings about the selected file. It's divided into three main sections.
   - **Identity and Type** – provides you with an information where the file is located in the directory and gives you a possibility to open it in Finder. Also, you can set the location of the file whether it should be an absolute or relative path.
   - **Target Membership** – if your file doesn't get recognized in the project, the first place you should search is here. This tells the project that the file belongs to its target. Make sure your target is always checked, otherwise the file won't be found.
   - **Text Settings –** Personally, I have never used this part. You can set up indentation and other text settings from here. These settings are only for the selected file.
2. **Quick Help Inspector** – provides you with a documentation for a selected class. For example, find a String class inside your file, and move the cursor to that word. A quick explanation about the class will appear.

Storyboard File

If you click on a .storyboard file, in the right corner you will see a new set of tabs.

Fig: Storybaord

1. **File Inspector —** this inspector contains the same details that I have explained above, just with a few new sections.
    - **Interface Builder Document –** enables the Auto Layout, Trait Variations, and Safe Area. Also, you can set the Global Tint color for the whole storyboard and decide if you want to use the selected UIViewController as a Launch Screen.
    - **Localization –** shows all your localization files. You can enable/disable languages that you want to appear in the storyboard.

2. **Quick Help Inspector –** same info as above.

3. **Identity Inspector –** controls the identity of the UIViewController. By identity we understand, assigning a custom class, providing a storyboard ID so you can access the view controller via code, and User Defined Runtime Attributes where you can add various styling properties, instead of adding them via code (example: layer.cornerRadius).

4. **Attributes Inspector –** this inspector is used for adjusting the properties of the selected object. Each object contains its own set of properties. For example, the UILabelcontains settings like adjusting text, text color, font, background color etc. You can also add your own properties, by using @IBInspectable.

5. **Size Inspector –** I think the name explains it all. Anything related to the sizing of the object can be found here. Despite the x, y, width and height values you can find the Auto Layout constraints, or Autoresizing if you aren't using Auto Layout.

6. **Connections Inspector –** used for communication between the code and the view controller using @IBOutlet and assigning actions via @IBAction.

I will go through the .xcassets file which can be found in any project. It's used for storing your assets.

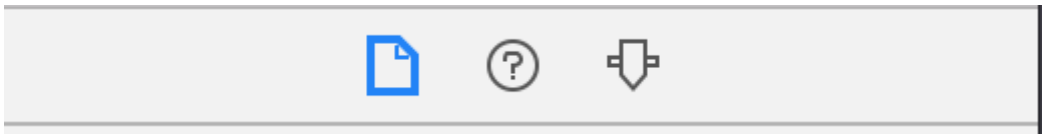


Fig: Connections Inspector

Two of the three tabs are already familiar to you from the previous points above.

- **Attributes Inspector** – from here, you can adjust the properties for the selected asset. Most common properties are image compression, rendering type, device support, scale (vector or individual) etc.

Header

Next on the list is the header. We are going a little bit above the left and right sidebar, at the very top of Xcode.

Fig: Header

Left Side

You can find actions like run and stop your project from building, managing your target's schemes, or picking and downloading a new simulator.

Right Side

A place where you can apply different options to the editor. I would start explaining them from left to right.

1. **Standard Editor –** this control represents the default view of the editor.
2. **Assistant Editor –** By using the assistant editor, your coding area will split into two parts. The left side represents your file, and the right side represents only your method definitions. What it does is, it ignores the code written inside your methods, and just shows you the method definitions. Useful if your class contains lots of code.
3. **Version Editor –** Splits your coding area and adds a replica of your class which tells you what you have changed from the last commit. This is used for source control.

The next three icons that are located in the top right corner, represent the showing and hiding of the sidebars and the debug area. It is used if you lack space on your screen (i.e. working on a MacBook Pro 13″ screen) and want to give more focus on the coding area.

Project Settings

We have finished exploring the coding area. Now let's proceed with something more complex. Something that doesn't involves code. We will go through the project settings. I am sure you have seen this screen before, so I will start explaining the tabs directly. I will explain only the most used ones.

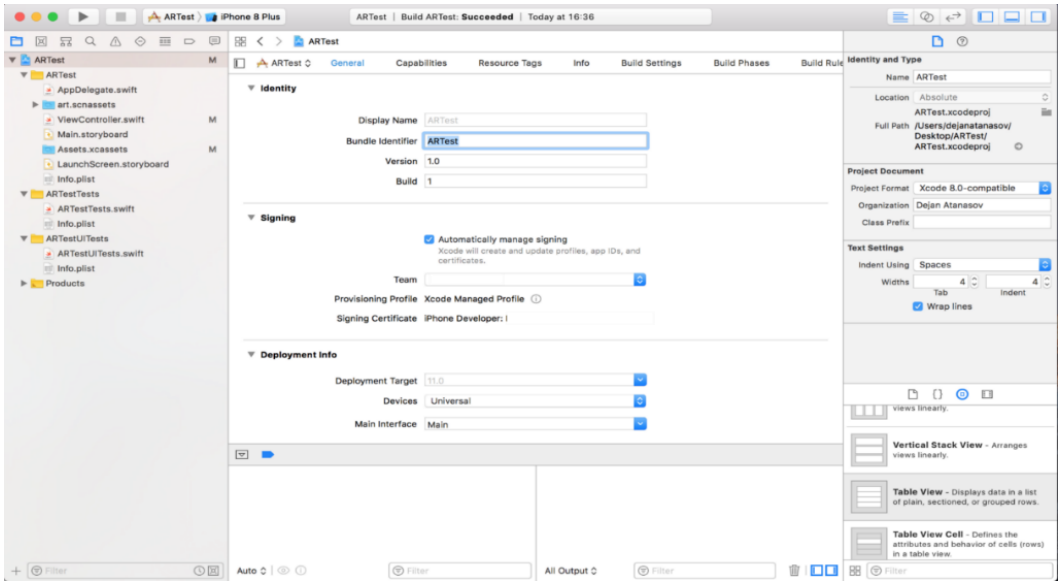


Fig: Project Settings

1. **General –** the name explains it all. Here, you can find settings that are general for the specific target.

- **Identity –** controls the app name, bundle identifier, version number and build. number,
- **Signing –** a section that handles the provisioning profiles. Basically, it signs the app for sending versions of the app for testing or production.
- **Deployment Info –** from here, you can add minimal iOS version support, decide if the app is going to be Universal or only iPhone or only IPad, and add the default Storyboard file.
- **App Icons and Launch Images –** assign app icon and splash screen assets.

2. **Capabilities –** contains switches with various services that you can use. By default, all the services are OFF and you need to activate what you need, by going to your **App ID**. So if you want to use Push Notifications, Background Services or **In-App Purchases**, make sure that the switch is set to ON.

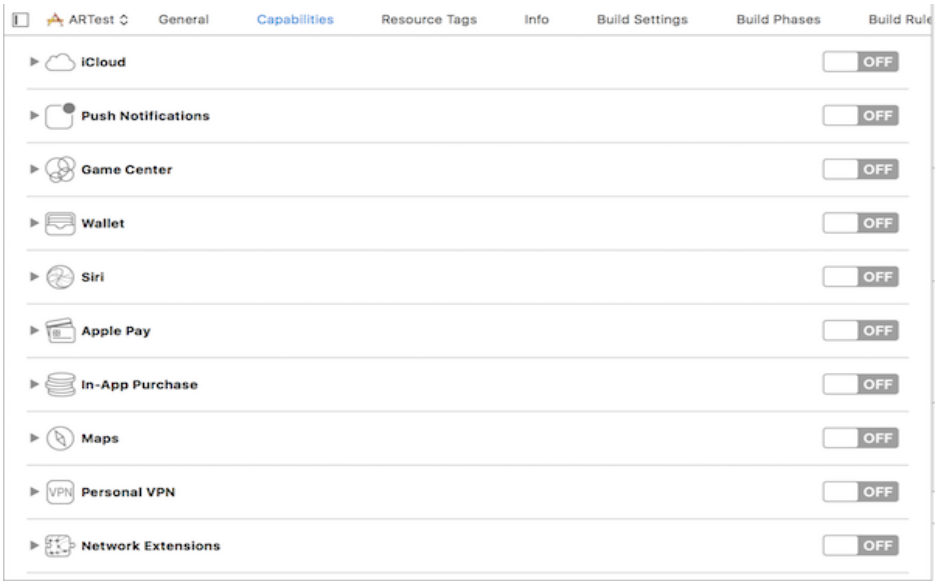


Fig: Capabilities

3. **Resource Tags –** you can assign tags to resources, and they will all end up here. I have never used this feature..

4. **Info –** this tab shows the properties from your active **.plist** file. From here, you can add/edit/remove the property information.

5. **Build Settings –** the official Apple documentation explains this part well… *"A build setting is a variable that contains information about how a particular aspect of a product's build process should be performed. For example, the information in a build setting can specify which options Xcode passes to the compiler."*

Build Settings contains many settings that are related to the build process. From assigning provisioning profiles to adding directory paths for 3rd party libraries. Each target has its own Build Settings.

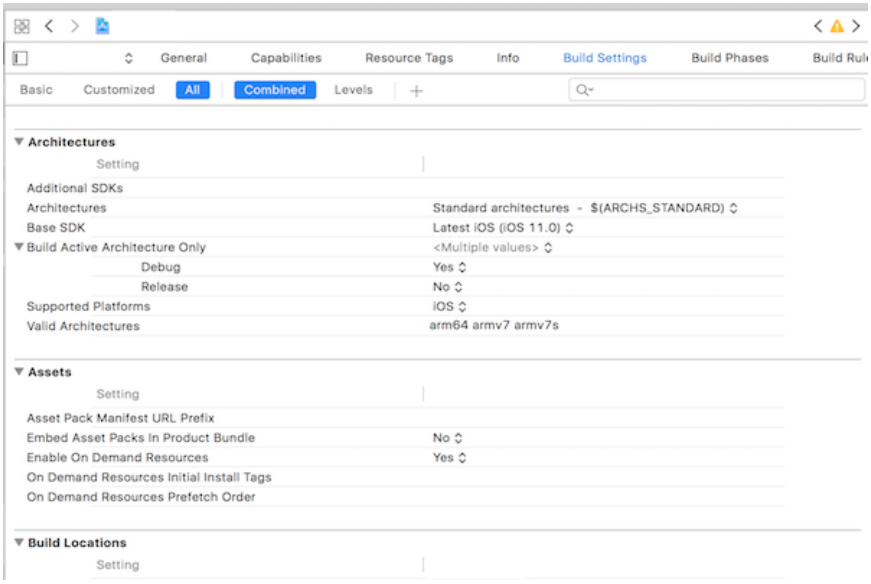


Fig: Build Settings

6. **Build Phases –** provides a list of all the files that will be included in the compile. All the frameworks, assets, .swift files, storyboards will be shown here, with an option to add or remove. Also, there is a Run Script feature for adding a custom shell script.

7. **Build Rules –** Xcode gives you a possibility to write your own rules. If you got decent scripting skills you can write almost anything. For example, converting .css files into .c, or add formatting rules to the text inside the .rtf file.

Practice:

Step 1. Launch Xcode and Create Your Project

Xcode is an integrated development environment (IDE) that combines the multiple tools necessary to build an iPhone program into one application. Launch Xcode now by searching for it in spotlight or using finder to navigate to the default install path of **/Developer/Applications/Xcode**.

Select "Create a new Xcode project" from the dialogue box that appears.

Fig: Create Xcode Project

Select “Application” under iOS in the left column. Familiarize yourself with the various types of application templates available in the content panel. This tutorial will be a simple, single view app, so select the “View-based Application” icon and click “Choose.”

Fig: Template

Enter the text “FortuneCrunch” into the “Save As” field to name our project and click “Save.”

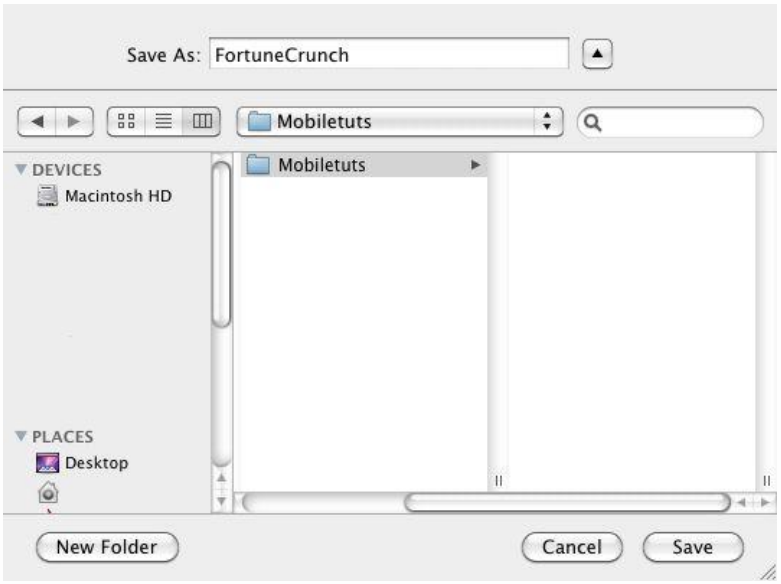


Fig: Save

The FortuneCrunch project template should now be displayed on screen.

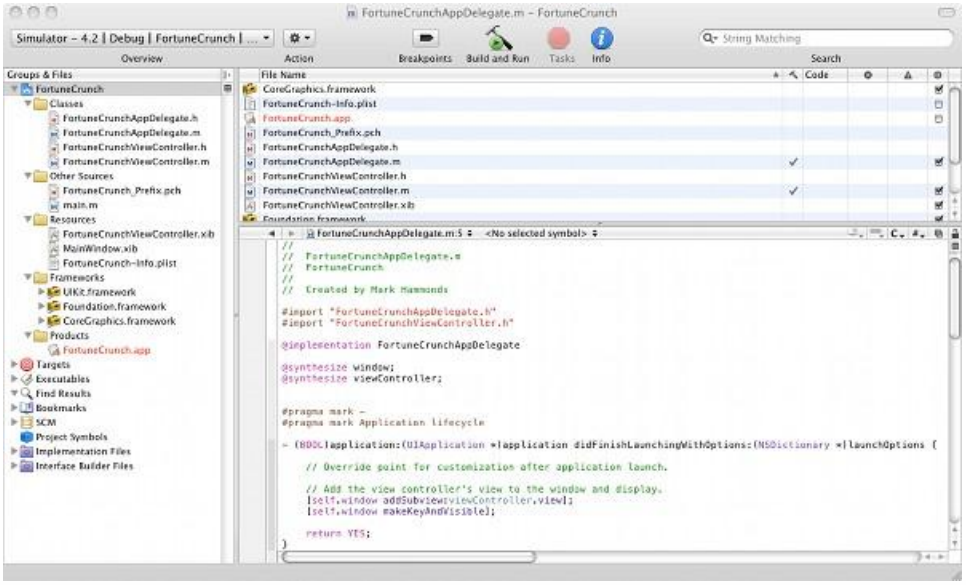


Fig: Project Template

It is important to realize that this starting point is a fully executable program. Go ahead and use the iPhone simulator to run the template by clicking **"Build and Run"** or by selecting **Build > Build and Run** from the menu bar.

Fig: Build and Run

Though the default application template is incredibly drab, the iPhone simulator is not. Click the home button in the simulator to return to the iPhone dock and browse around a bit. Also note the following simulator short cuts:

Command + left arrow: Rotate the device left.

Command + right arrow: Rotate the device right.

Ctrl + Command + Z: Emulates a shake gesture (this will not display visually).

Core Discussion

- **Historical approach**

iOS is a mobile operating system, developed by Apple Inc. for iPhone, iPad, and iPod Touch. Updates for iOS are released through the iTunes software, and, since iOS 5, via over-the-air software updates. The current stable release, iOS 11.3, was released on March 29, 2018.

- **Fundamental technology**

Xcode IDE

Developers could try building iOS applications using cross-platform frameworks, but Apple's Xcode IDE is the most efficient and comprehensive iOS application development tool available. Xcode is fr

Objective-C / Swift

Prior to 2014, developers building an iOS app in Xcode used Objective-C. Then Apple went to Swift, a language that's touted as being easier to learn and better designed to handle the subtleties of iOS application development.

The iOS architecture

Developers can picture iOS as four abstraction layers that define its architecture:

Cocoa Touch: supports the basic app infrastructure and delivers key application frameworks such as push notifications, multitasking and touch-based input.

Media: enables the app to deliver audio, video and graphic capabilities.

Core Services: where developers will find basic system services such as the Core Foundation and the Foundation Framework. This layer also supports features such as location and networking services.

Core OS: provides such services as the Security, Local Authentication and Core Bluetooth frameworks.

Suggested reading list: (Books & Links)

iOS Programming: The Big Nerd Ranch Guide

iOS App Development For Dummies

Swift Programming: The Big Nerd Ranch Guide

Programming in Objective-C (6th Edition) (Developer's Library) 6th Edition by Stephen G. Kochan (Author)

Objective-C Programming For Dummies 1st Edition by Neal Goldstein (Author)

Objective-C Programming: The Big Nerd Ranch Guide (2nd Edition)

Reference and bibliography:

https://developer.apple.com/

https://code.tutsplus.com/tutorials/introduction-to-iphone-sdk-development--mobile-133

https://www.apple.com/lae/ios/ios-11/

https://en.wikipedia.org/wiki/IOS

Module 2: Objective-C

Topic:

Introduction to Objective-C

Data Types and Variables

Chapter Overview:

Objective-C is a general-purpose, object-oriented programming language that adds Smalltalk-style messaging to the C programming language. This is the main programming language used by Apple for the OS X and iOS operating systems and their respective APIs, Cocoa and Cocoa Touch.

Learning Outcome:

- Introduction to Objective-C
- Learning Data Types and Variables
- A Brief History
- Knowledge about Loops
- Knowledge about Functions
- Knowledge about Arrays
- Knowledge about Foundation Framework
- Knowledge about Object-Oriented Programming

Introduction to Objective-C

A Brief History

Created primarily by Brad Cox and Tom Love, both employees of Stepstone, Objective-C can be traced back to the early 1980s. Building upon Smalltalk, one of the first object-oriented language, Cox's fascination with problems of reusability in software design and programming resulted in the creation of the language. Recognizing that compatibility with C was crucial to the success of the project, Cox began writing a pre-processor for C to add backward compatibility with C, which soon grew into an object-oriented extension to the C language.

Cox showed that the construction of interchangeable software components really only needed a few practical changes to existing conventions. Objects needed to be supported in a more flexible manner, which would work in conjunction with a usable set of libraries, allowing for code to be bundled into a single cross-platform format.

To commercialize their creation, the dynamic duo created Productivity Products International, which allowed for the sale of an Objective-C compiler with class libraries. In 1986, Cox published a book about the language entitled *Object-Oriented Programming, An Evolutionary Approach*. Although the main focus of the instructional text was to point out the issue of reusability, Objective-C has been compared feature-for-feature with the major players in the programming game ever since.

Fig: Infographic

After Steve Jobs' departure from Apple, he started a new company called NeXT. In 1988, NeXT licensed Objective-C from the owner of the trademark, releasing its own Objective-C compiler and libraries on which the NeXTstep UI and interface builder were based. The innovative nature of this graphics-based interface creation resulted in the creation of the first web browser on a NeXTstep system.

With Apple's acquisition of NeXT in 1996, Apple used OpenStep, the standard that Steve Jobs pushed forward based on the Objective-C libraries he so vehemently supported, to build Mac OS X. This included a new developer tool that was later replaced by Xcode as well as a design tool called Interface Builder. Most of Apple's present-day Cocoa API is based on NeXTstep interface objects.

At the Worldwide Developers Conference in 2006, Apple announced Objective-C 2.0, a revision of Objective-C that included syntax enhancements, performance improvements, and 64-bit support. Mac OS X officially included a 2.0-enabled compiler in October 2007. It is unclear at this time whether these language changes will be available in the GNU runtime, or if they will be implemented to be compliant with the larger Objective-C 2.0 standard

NeXT Software licensed the language in the 1988, and developed a code library called NeXTSTEP.

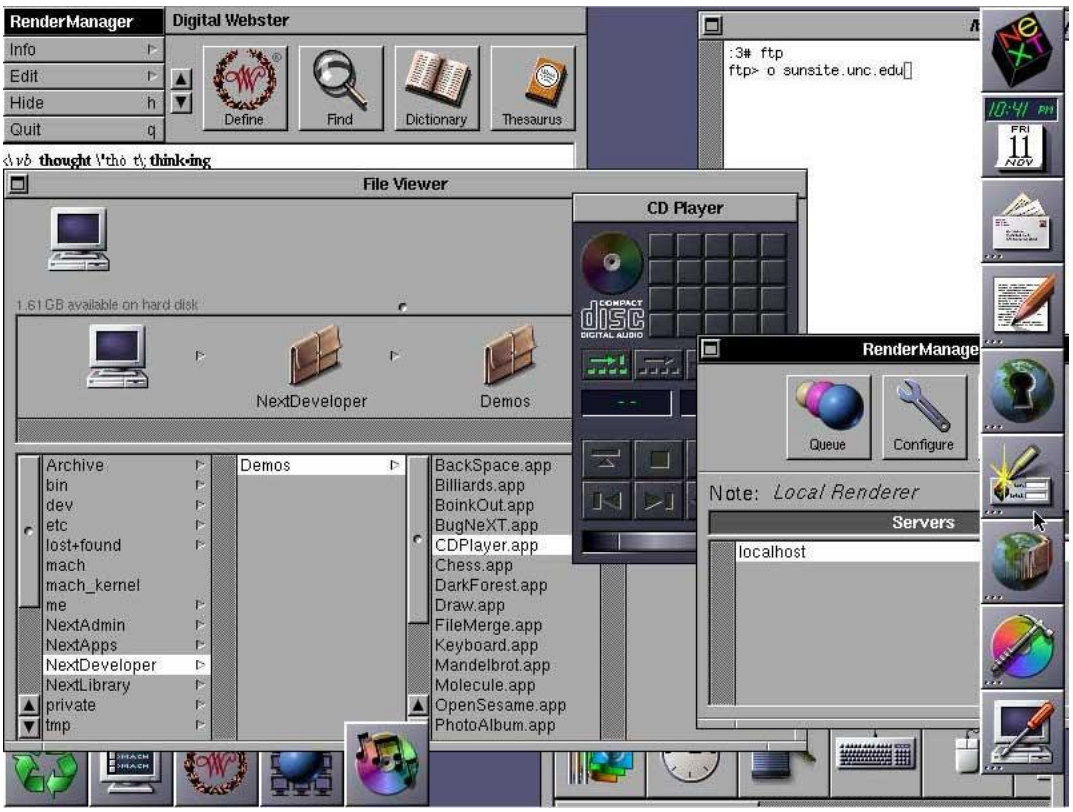


Fig: NeXTSTEP

When Apple Computer acquired NeXT in 1996, the NeXTSTEP code library was built into the core of Apple's operating system, Mac OS X. NeXTSTEP provided Apple with a modern OS foundation, which Apple could not produce on its own.

The iPhone's operating system, currently dubbed iOS, is based off of a reduced version of OS X. Therefore, iOS inherits most of the NeXTSTEP code library, along with extensive modernization and optimizations. Because NeXTSTEP was built from Objective-C, iOS mirrors the language choice. This made it easy for OS X developers to begin creating apps for the iPhone and iPod Touch.

Apple added a number of features to the Objective-C language, extending its functionality to parallel that of other languages that were beginning to arise. This major update was labeled Objective-C 2.0, and remains the language of choice for both OS X and iOS. This iteration will be covered in this tutorial.

Object-Oriented Programming

Fully supports object-oriented programming, including the four pillars of object-oriented development:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Example Code

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

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSLog (@"hello world");
    [pool drain];
    return 0;
}
```

Fig: Example code

Foundation Framework

Foundation Framework provides large set of features and they are listed below.

- It includes a list of extended datatypes like NSArray, NSDictionary, NSSet and so on.
- It consists of a rich set of functions manipulating files, strings, etc.
- It provides features for URL handling, utilities like date formatting, data handling, error handling, etc.

Primitive Data Types

The first half of this chapter looks at the native Objective-C data types and discusses how to display them using `NSLog()` format strings. The size of the data types presented in this section is system-dependent-the only way to truly know how big your data types are is to use the `sizeof()` function. For example, you can check the size of a `char` with the following:

```
1  NSLog(@"%lu", sizeof(char));
```

This should output `1`, which means that `char` takes up 1 byte of memory. The `%lu` placeholder is for unsigned long integers (discussed in more detail later), which is the return type for `sizeof()`. Upcoming sections discuss the most common sizes for Objective-C data types, but remember that this may differ from your system.

Booleans

Objective-C programs use the `BOOL` data type to store Boolean values. Objective-C also defines its own true and false keywords, which are `YES` and `NO`, respectively. To display `BOOL` values via `NSLog()`, use `%i` in the format string:

```
1  BOOL isHuman = NO;
2  NSLog(@"It's alive: %i", isHuman);
```

The `%i` specifier is used to display integers, so this should output `It's alive: 0`.

Technically, `BOOL` is a macro for the `signed char` type (discussed in the next section). This means that `BOOL` variables can store many more values than just `YES` and `NO`, which are actually macros for `1` and `0`, respectively. However, most developers will never use this extra functionality, since it can be a source of frustrating bugs in conditional statements:

```
1    BOOL isHuman = 127;
2    if (isHuman) {
3        // This will execute.
4        NSLog(@"isHuman is TRUE");
5    }
6    if (isHuman == YES) {
7        // But this *won't* execute.
8        NSLog(@"isHuman is YES");
9    }
```

Any value greater than `0` will evaluate to true, so the first condition will execute, but the second will not because `127 != 1`. Depending on how you're using your `BOOL` variables, this may or may not be a desirable distinction.

Chars

Objective-C uses the same `char` data type as ANSI C. It denotes a single-byte signed integer, and can be used to store values between `-128` and `127` or an ASCII character. To display a `char` as an integer, just use the generic `%i` specifier introduced in the previous code sample. To format it as an ASCII character, use `%c`:

```
1  char letter = 'z';
2  NSLog(@"The ASCII letter %c is actually the number %i", letter, letter);
```

As with all integer data types, it's possible to allocate an *unsigned* `char`, which can record values from `0` to `255`. Instead of the `%i` specifier, you should use `%u` as a placeholder for unsigned integers:

```
1  unsigned char tinyInt = 255;
2  NSLog(@"The unsigned char is: %u", tinyInt);
```

Short Integers

Short integers are 2-byte signed integers and should be used for values between -32768 and 32767. To display them with NSLog(), use the %hi specifier (the h is a "modifier" for the same %i used in the previous two sections). For example:

```
1  short int littleInt = 27000;
2  NSLog(@"The short int is: %hi", littleInt);
```

Unsigned shorts can be created the same way as unsigned chars and can hold up to 65535. Again, the u in %hu is the same one in %u for generic unsigned integers:

```
1  unsigned short int ulittleInt = 42000;
2  NSLog(@"The unsigned short integer is: %hu", ulittleInt);
```

"Normal" Integers

Next on the list is int, which is a 4-byte integer on most systems. Again, remember that data type size is system-dependent-the only way to know for sure how big your data types are is to use the sizeof() function:

```
1  NSLog(@"%lu", sizeof(int));
```

If your int type is indeed 4 bytes, it can hold values between -2147483648 and 2147483647.

```
1  int normalInt = 1234567890;
2  NSLog(@"The normal integer is: %i", normalInt);
```

This also means that the unsigned version can record 0 – 4294967295.

Long Integers

If int isn't big enough to meet your needs, you can move up to the long int data type, which is 8 bytes on most modern systems. This is large enough to represent values between -9223372036854775808 and 9223372036854775807. Long integers can be displayed via NSLog() by prepending the letter l to the %i or %u specifiers, as shown in the following code:

```
long int bigInt = 9223372036854775807;
NSLog(@"The big integer is: %li", bigInt);

unsigned long int uBigInt = 18446744073709551615;
NSLog(@"The even bigger integer is: %lu", uBigInt);
```

`18446744073709551615` is the maximum value for the unsigned version, which is hopefully the largest integer you'll ever need to store.

The idea behind having so many integer data types is to give developers the power to balance their program's memory footprint versus its numerical capacity.

Floats

Objective-C programs can use the `float` type for representing 4-byte floating point numbers. Literal values should be suffixed with `f` to mark the value as single precision instead of a `double` (discussed in the next section). Use the `%f` specifier to output floats with `NSLog()`:

```
float someRealNumber = 0.42f;
NSLog(@"The floating-point number is: %f", someRealNumber);
```

You can also specify the output format for the float itself by including a decimal before the `f`. For example, `%5.3f` will display 3 digits after the decimal and pad the result so there are 5 places total (useful for aligning the decimal point when listing values).

While floating-point values have a much larger range than their fixed-point counterparts, it's important to remember that they are intrinsically *not precise*. Careful consideration must be paid to comparing floating-point values, and they should never be used to record precision-sensitive data (e.g., money). For representing fixed-point values in Objective-C, please see `NSDecimalNumber` in the the Foundation Data Structures section.

Doubles

The `double` data type is a double-precision floating-point number. For the most part, you can treat it as a more accurate version of `float`. You can use the same `%f` specifier for displaying `double`s in `NSLog()`, but you don't need to append `f` to literal values:

```
double anotherRealNumber = 0.42;
NSLog(@"The floating-point number is: %5.3f", anotherRealNumber);
```

Structs

Objective-C also provides access to C structs, which can be used to define custom data structures. For example, if you're working on a graphics program and interact with many 2-dimensional points, it's convenient to wrap them in a custom type:

```
typedef struct {
    float x;
```

```
3       float y;
4   } Point2D;
```

The `typedef` keyword tells the compiler we're defining a new data type, `struct` creates the actual data structure, which comprises the variables `x` and `y`, and finally, `Point2D` is the name of the new data type. After declaring this `struct`, you can use `Point2D` just like you would use any of the built-in types. For instance, the following snippet creates the point `(10.0, 0.5)` and displays it using our existing `NSLog()` format specifiers.

```
1   Point2D p1 = {10.0f, 0.5f};
2   NSLog(@"The point is at: (%.1f, %.1f)", p1.x, p1.y);
```

The `{10.0f, 0.5f}` notation is called a compound literal, and it can be used to initialize a `struct`. After initialization, you can also assign new values to a `struct`'s properties with the `=` operator:

```
1   p1.x = -2.5f;
2   p1.y = 2.5f;
```

Structures are important for performance-intensive applications, but they sometimes prove difficult to integrate with the high-level Foundation data structures. Unless you're working with 3-D graphics or some other CPU-heavy application, you're usually better off storing custom data structures in a full-fledged class instead of a `struct`.

Arrays

While Objective-C provides its own object-oriented array data types, it still gives you access to the low-level arrays specified by ANSI C. C arrays are a contiguous block of memory allocated when they're declared, and all of their elements must be of the same type. Unlike C# arrays, this means you need to define an array's length when it's declared, and you can't assign another array to it after it's been initialized.

Because there is no way for a program to automatically determine how many elements are in an array, there is no convenient `NSLog()` format specifier for displaying native arrays. Instead, we're stuck with manually looping through each element and calling a separate `NSLog()`. For example, the following code creates and displays an array of 5 integers:

```
1   int someValues[5] = {15, 32, 49, 90, 14};
2   for (int i=0; i<5; i++) {
3       NSLog(@"The value at index %i is: %i", i, someValues[i]);
4   }
```

As you can see, C arrays look much like atomic variables, except you have to provide their length in square brackets (`[5]`). They can be initialized with the same compound literal syntax as structs, but all the values must be of the same type. Individual elements can be accessed by passing the item number in square brackets, which is common in most programming languages. In addition, you can access elements via pointers.

Pointers provide a low-level way to directly access memory addresses in a C program. And, since C arrays are just contiguous blocks of memory, pointers are a natural way to interact with items in an array. In fact, the variable holding a native array is actually a pointer to the first element in the array.

Pointers are created by prefixing the variable name with an asterisk (`*`). For example, we can create a second reference to the first element in the `someValues` array with the following code:

```
1  int someValues[5] = {15, 32, 49, 90, 14};
2  int *pointer = someValues;
```

Instead of storing an `int` value, the `*pointer` variable *points* to the memory address containing the value. This can be visualized as the following:

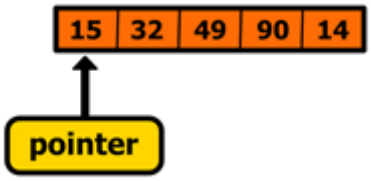


Fig: Pointer

Pointer to the first element of an array

To get the underlying value out of the memory address, we need to **dereference** the pointer using the asterisk operator, like so:

```
1  NSLog(@"The first value is: %i", *pointer);
```

This should display `15` in your output panel, since that is the value stored in the memory address pointed to by the `pointer` variable. So far, this is just a very confusing way to access a normal (non-pointer) `int` variable. However, things get much more interesting when you start *moving* pointers around with the `++` and `--` operators. For example, we can increment the pointer to the next memory address as follows:

```
1  pointer++;
2  NSLog(@"The next value is: %i", *pointer);
```

Since an array is a contiguous block of memory, the pointer will now rest at the address of the second element of the array. As a result, the `NSLog()` call should display `32` instead of `15`. This can be visualized as the following:

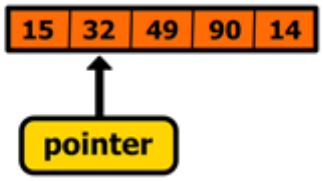


Fig: Pointer

Incrementing the pointer to the second element of an array

Pointers provide an alternative way to iterate through an array. Instead of accessing items via the square brackets (e.g., `someValues[i]`), you can simply increment the pointer and dereference it to get the next value:

```
1  for (int i=0; i<5; i++) {
2      pointer++;
3      NSLog(@"The value at index %i is: %i", i, *pointer);
4  }
```

Pointers have innumerable uses in high-performance applications, but in reality, you probably won't need to use pointers with native arrays unless you're building a data-intensive application that is seriously concerned with speed.

However, pointers are still very important to Objective-C programs because *every* object is referenced through a pointer. This is why all of the data structures in the upcoming Foundation Data Structuressection are declared as pointers (e.g., `NSNumber *someNumber`, not `NSNumber someNumber`).

Void

The `void` type represents the absence of a value. Instead of typing variables, `void` is used with functions and methods that don't return a value. For example, the `sayHello` method from the previous chapter didn't return anything, and it was thus defined with the `void` data type:

```
1  - (void)sayHello;
```

Nil and NULL

The `nil` and `NULL` keywords are both used to represent empty pointers. This is useful for explicitly stating that a variable doesn't contain anything, rather than leaving it as a pointer to its most recent memory address.

There is, however, a strict distinction between the two. The `nil` constant should only be used as an empty value for Objective-C objects-it should *not* be used to for native C-style pointers (e.g., `int *somePointer`). `NULL` can be used for either primitive pointers or Objective-C object pointers, though `nil` is the preferred choice.

Primitive Data Type Summary

The first half of this chapter introduced the primitive data types available to Objective-C programmers. We also took a brief look at pointers and the `nil` and `NULL` keywords.

It's important to remember that the value stored in a variable is completely independent from how it's interpreted. `unsigned int`s can be interpreted as `signed int`s without changing the variable in any way. That's why it's so important to make sure you're using the right format string in `NSLog()`. Otherwise, you'll be left wondering why your unsigned variables look like they're storing negative numbers. As we'll see in the next section, this isn't as much of a problem with object-oriented data types.

The remainder of this chapter focuses on the Foundation framework, which defines several object-oriented data structures that all Objective-C developers should be familiar with.

Data Types and Variables

In the Objective-C programming language, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in Objective-C can be classified as follows:

S.N.	Types and Description
1	**Basic Types:** They are arithmetic types and consist of the two types: (a) integer types and (b) floating-point types.
2	**Enumerated types:** They are again arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program.
3	**The type void:** The type specifier *void* indicates that no value is available.
4	**Derived types:** They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

The array types and structure types are referred to collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see basic types in the following section whereas other types will be covered in the upcoming chapters.

Integer Types

Following table gives you details about standard integer types with its storage sizes and value ranges:

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Floating-Point Types

Following table gives you details about standard float-point types with storage sizes and value ranges and their precision:

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The void Type

The void type specifies that no value is available. It is used in three kinds of situations:

S.N.	Types and Description
1	**Function returns as void** There are various functions in Objective-C which do not return value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status);**
2	**Function arguments as void** There are various functions in Objective-C which do not accept any parameter. A function with no parameter can accept as a void. For example, **int rand(void);**

Variable

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in Objective-C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because Objective-C is case-sensitive. Based on the basic types explained in previous chapter, there will be the following basic variable types:

Type	Description
char	Typically a single octet (one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.

Objective-C programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

Variable Definition in Objective-C:

A variable definition means to tell the compiler where and how much to create the storage for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

Here, **type** must be a valid Objective-C data type including char, w_char, int, float, double, bool or any user-defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line int i, j, k; both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Some examples are:

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

Variable Declaration in Objective-C:

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable declaration at the time of linking of the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files, which will be available at the time of linking of the program. You will use extern keyword to declare a variable at any place. Though you can declare a variable multiple times in your Objective-C program but it can be defined only once in a file, a function or a block of code.

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

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main ()
{
  /* variable definition: */
  int a, b;
  int c;
  float f;

  /* actual initialization */
  a = 10;
  b = 20;

  c = a + b;
  NSLog(@"value of c : %d \n", c);

  f = 70.0/3.0;
  NSLog(@"value of f : %f \n", f);

  return 0;
}
```

When the above code is compiled and executed, it produces the following result: 30 , 23.333334

Suggested reading list:

Programming in Objective-C (6th Edition) (Developer's Library) 6th Edition by Stephen G. Kochan (Author)

Objective-C Programming For Dummies 1st Edition by Neal Goldstein (Author)

Objective-C Programming: The Big Nerd Ranch Guide (2nd Edition)

Reference and bibliography:

https://code.tutsplus.com/tutorials/objective-c-succinctly-data-types--mobile-21986

https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html

https://www.tutorialspoint.com/objective_c/index.htm

https://github.com/github/objective-c-style-guide

http://cocoadevcentral.com/d/learn_objectivec/

Module 3: iOS App Architecture

Topic: iOS App Architecture

Chapter Overview:

Apps need to work with the iOS to ensure that they deliver a great user experience. Beyond just a good design for your app's design and user interface, a great user experience encompasses many other factors. Users expect iOS apps to be fast and responsive while expecting the app to use as little power as possible. Apps need to support all of the latest iOS devices while still appearing as if the app was tailored for the current device. Implementing all of these behaviors can seem daunting at first but iOS provides the help you need to make it happen.

In this chapter we will talk about Model-View-Controller-MVC, View Controllers, Storyboard, Outlet and Action

Learning Outcome:

- Model-View-Controller-MVC
- View Controllers
- Storyboard
- Outlet
- Action

Model-View-Controller-MVC

MVC as a Compound Design Pattern

Model-View-Controller is a design pattern that is composed of several more basic design patterns. These basic patterns work together to define the functional separation and paths of communication that are characteristic of an MVC application. However, the traditional notion of MVC assigns a set of basic patterns different from those that Cocoa assigns. The difference primarily lies in the roles given to the controller and view objects of an application.

In the original (Smalltalk) conception, MVC is made up of the Composite, Strategy, and Observer patterns.

- Composite—The view objects in an application are actually a composite of nested views that work together in a coordinated fashion (that is, the view hierarchy). These display components range from a window to compound views, such as a table view, to individual views, such as buttons. User input and display can take place at any level of the composite structure.
- Strategy—A controller object implements the strategy for one or more view objects. The view object confines itself to maintaining its visual aspects, and it delegates to the controller all decisions about the application-specific meaning of the interface behavior.
- Observer—A model object keeps interested objects in an application—usually view objects—advised of changes in its state.

The traditional way the Composite, Strategy, and Observer patterns work together is depicted by Figure 7-1: The user manipulates a view at some level of the composite structure and, as a result, an event is generated. A controller object receives the event and interprets it in an application-specific way—that is, it applies a strategy. This strategy can be to request (via message) a model object to change its state or to request a view object (at some level of the composite structure) to change its behavior or appearance. The model object, in turn, notifies

all objects who have registered as observers when its state changes; if the observer is a view object, it may update its appearance accordingly.

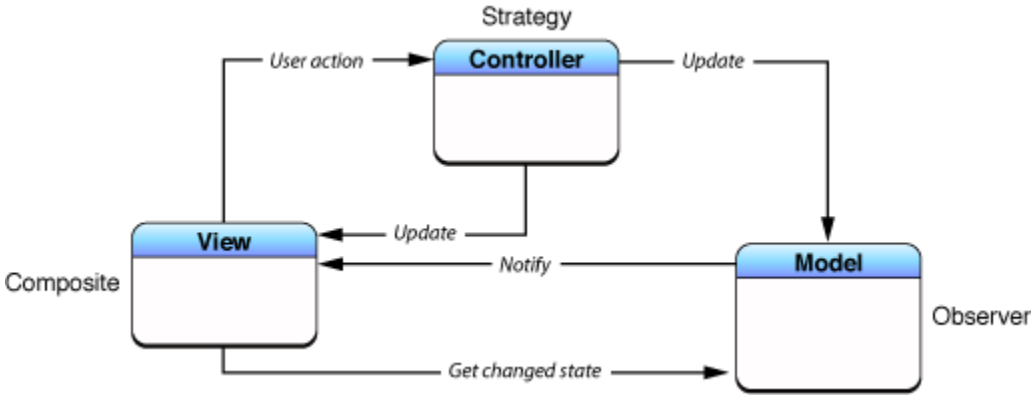


Fig: Traditional version of MVC as a compound pattern

The Cocoa version of MVC as a compound pattern has some similarities to the traditional version, and in fact it is quite possible to construct a working application based on the diagram in Figure 7-1. By using the bindings technology, you can easily create a Cocoa MVC application whose views directly observe model objects to receive notifications of state changes. However, there is a theoretical problem with this design. View objects and model objects should be the most reusable objects in an application. View objects represent the "look and feel" of an operating system and the applications that system supports; consistency in appearance and behavior is essential, and that requires highly reusable objects. Model objects by definition encapsulate the data associated with a problem domain and perform operations on that data. Design-wise, it's best to keep model and view objects separate from each other, because that enhances their reusability.

In most Cocoa applications, notifications of state changes in model objects are communicated to view objects *through* controller objects. Figure 7-2 shows this different configuration, which appears much cleaner despite the involvement of two more basic design patterns.

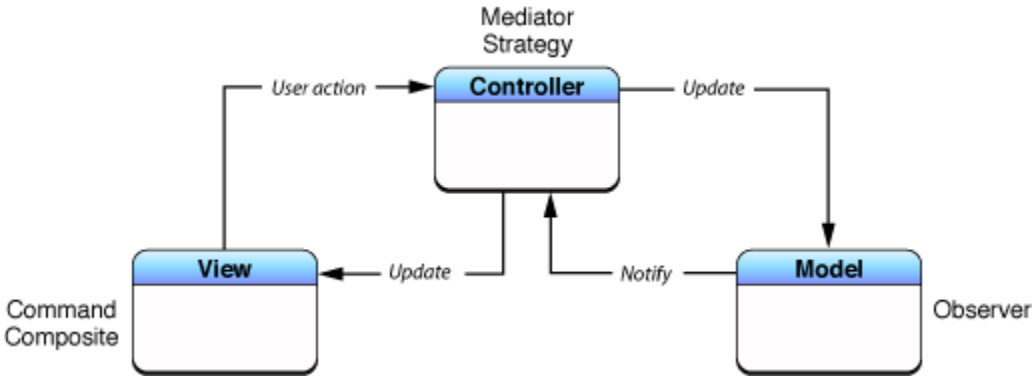


Fig: Cocoa version of MVC as a compound design pattern

The controller object in this compound design pattern incorporates the Mediator pattern as well as the Strategy pattern; it mediates the flow of data between model and view objects in both directions. Changes in model state are communicated to view objects through the controller objects of an application. In addition, view objects incorporate the Command pattern through their implementation of the target-action mechanism.

Note: The target-action mechanism, which enables view objects to communicate user input and choices, can be implemented in both coordinating and mediating controller objects. However, the design of the mechanism differs in each controller type. For coordinating

controllers, you connect the view object to its target (the controller object) in Interface Builder and specify an action selector that must conform to a certain signature. Coordinating controllers, by virtue of being delegates of windows and the global application object, can also be in the responder chain. The bindings mechanism used by mediating controllers also connects view objects to targets and allows action signatures with a variable number of parameters of arbitrary types. Mediating controllers, however, aren't in the responder chain.

There are practical reasons as well as theoretical ones for the revised compound design pattern depicted in Figure 7-2, especially when it comes to the Mediator design pattern. Mediating controllers derive from concrete subclasses of NSController, and these classes, besides implementing the Mediator pattern, offer many features that applications should take advantage of, such as the management of selections and placeholder values. And if you opt not to use the bindings technology, your view object could use a mechanism such as the Cocoa notification center to receive notifications from a model object. But this would require you to create a custom view subclass to add the knowledge of the notifications posted by the model object.

In a well-designed Cocoa MVC application, coordinating controller objects often own mediating controllers, which are archived in nib files. Figure 7-3 shows the relationships between the two types of controller objects.

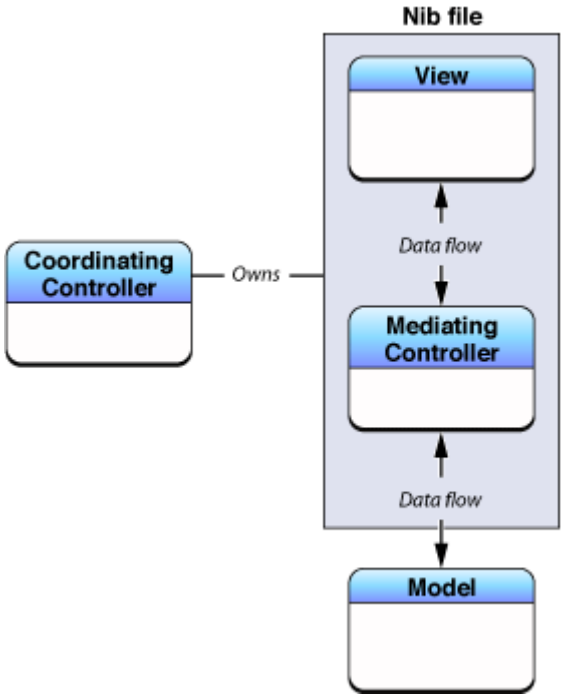


Fig: Coordinating controller as the owner of a nib file

Design Guidelines for MVC Applications

The following guidelines apply to Model-View-Controller considerations in the design of applications:

- Although you can use an instance of a custom subclass of NSObject as a mediating controller, there's no reason to go through all the work required to make it one. Use instead one of the ready-made NSController objects designed for the Cocoa bindings technology; that is, use an instance of NSObjectController,NSArrayController, NSUserDefaultsController,

or NSTreeController—or a custom subclass of one of these concrete NSController subclasses.

However, if the application is very simple and you feel more comfortable writing the glue code needed to implement mediating behavior using outlets and target-action, feel free to use an instance of a custom NSObject subclass as a mediating controller. In a custom NSObject subclass, you can also implement a mediating controller in the NSController sense, using key-value coding, key-value observing, and the editor protocols.

- Although you can combine MVC roles in an object, the best overall strategy is to keep the separation between roles. This separation enhances the reusability of objects and the extensibility of the program they're used in. If you are going to merge MVC roles in a class, pick a predominant role for that class and then (for maintenance purposes) use categories in the same implementation file to extend the class to play other roles.
- A goal of a well-designed MVC application should be to use as many objects as possible that are (theoretically, at least) reusable. In particular, view objects and model objects should be highly reusable. (The ready-made mediating controller objects, of course, are reusable.) Application-specific behavior is frequently concentrated as much as possible in controller objects.
- Although it is possible to have views directly observe models to detect changes in state, it is best not to do so. A view object should always go through a mediating controller object to learn about changes in an model object. The reason is two-fold:
    - If you use the bindings mechanism to have view objects directly observe the properties of model objects, you bypass all the advantages that NSController and its subclasses give your application: selection and placeholder management as well as the ability to commit and discard changes.
    - If you don't use the bindings mechanism, you have to subclass an existing view class to add the ability to observe change notifications posted by a model object.
- Strive to limit code dependency in the classes of your application. The greater the dependency a class has on another class, the less reusable it is. Specific recommendations vary by the MVC roles of the two classes involved:
    - A view class shouldn't depend on a model class (although this may be unavoidable with some custom views).
    - A view class shouldn't have to depend on a mediating controller class.
    - A model class shouldn't depend on anything other than other model classes.
    - A mediating controller class shouldn’t depend on a model class (although, like views, this may be necessary if it's a custom controller class).
    - A mediating controller class shouldn't depend on view classes or on coordinating controller classes.
    - A coordinating controller class depends on classes of all MVC role types.
- If Cocoa offers an architecture that solves a programming problem, and this architecture assigns MVC roles to objects of specific types, use that architecture. It will be much easier to put your project together if you do. The document architecture, for example, includes an Xcode project template that configures an NSDocumentobject (per-nib model controller) as File's Owner.

Every new iOS developer is exposed to a huge amount of information that is crucial to master: a new language, new frameworks, and Apple's recommended architectural pattern: Model-View-Controller, or MVC for short.

Getting up to speed with iOS development can be a daunting task, and more often than not, developers don't pay MVC enough attention, sometimes leading to major headaches down the road.

From a high level, MVC is as straightforward as its name. It's made up of three layers: the model, the view and the controller.

- The ***Model*** is where your data resides. Things like persistence, model objects, parsers and networking code normally live there.
- The ***View*** layer is the face of your app. Its classes are typically reusable, since there aren't any domain-specific logic in them. For example, a **UILabel** is a view that presents text on the screen, and it's easily reusable.
- The ***Controller*** mediates between the view and the model, typically via the delegation pattern. In the ideal scenario, the controller entity won't know the concrete view it's dealing with. Instead, it will communicate with an abstraction via a protocol. A classic example is the way a **UITableView** communicates with its data source via the **UITableViewDataSource** protocol.

When you put everything together, it looks like this:

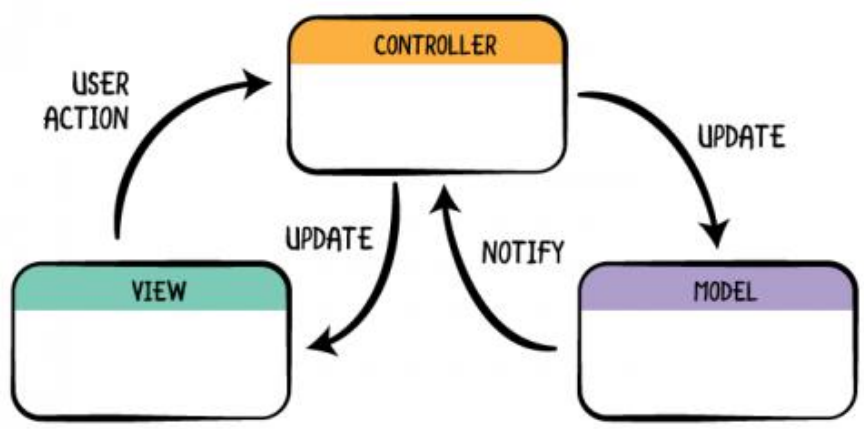


Fig: MVC Flow

As they say, the devil is in the details. When you're in the process of applying MVC, things can get tricky, as you'll see.

Apple's MVC documentation explains these layers in detail and can give you a theoretical understanding that will help you avoid potential pitfalls.

From a practical perspective, though, it leaves a lot to be desired. So instead of talking theory, let's talk shop.

The View Layer

When a user interacts with your app, they are interacting with the view layer. The view is considered the "dumb" part of your app, since it shouldn't contain any business logic. In code terms, you'll normally see:

- **UIView** subclasses. These range from a basic **UIView** to complex custom UI controls.
- A **UIViewController** (arguably). Since a **UIViewController** is strongly coupled with its own root **UIView**and its different cycles (**loadView**, **viewDidLoad**), I personally consider it to be part of this layer, but not everyone agrees.
- Animations and **UIViewController** transitions.
- Classes that are part of UIKit/AppKit, Core Animation and Core Graphics.

Typical code smells found in this layer manifest in different ways, but boil down to including anything unrelated to UI in your view layer. A classic code smell is making a network call from a **UIViewController**.

It's tempting to put a bunch of code in your **UIViewController** and be done with it, so you can meet that deadline. Don't do it! In the short term, you might save a couple of minutes, but in the long term, you could lose hours looking for a bug, or have trouble when you want to reuse code inside one view controller in another.

Use the following as a checklist when inspecting your view layer:

- Does it interact with the model layer?
- Does it contain any business logic?
- Does it try to do anything not related to UI?

If you answer "yes" to any of these questions, you probably have an opportunity to clean up and refactor.

Of course, these rules aren't written in stone and sometimes you'll need to bend them. Nonetheless, it's important to pay them respect.

Finally, if you write these classes well, you can almost always reuse them. If you don't believe me, just look at the number of UI components on GitHub!

The Controller Layer

The controller layer is the least reusable part of your app, because it involves your domain-specific rules. It should be no surprise that what makes sense in your app probably wouldn't make sense in someone else's.

Usually, you'll see classes from this layer deciding things like:

- What should be accessed first: the persistence or the network?
- How often should you refresh the app?
- What should the next screen be and in what circumstances?
- If the app goes to the background, what should be cleaned?

You should think of the controller layer as the brain of the app: It decides what happens next. Usually you'll want to heavily test these classes to make sure everything works as expected.

An Example

Now that you have a better understanding of the controller layer, let's see it in action with a simple example.

Imagine you have a **UIViewController** subclass that wants to know the list of WWDC attendees this year. To achieve this, it makes use of a controller class. Since Apple's been preaching that we should always start with a protocol.

The idea is that you'll initially set **state** to **Loading**, then update it when the list of WWDC attendees is successfully loaded (or fails).

Since you don't want the **UIViewController** to handle the response, it will use a separate object (**WWDCAttendeesUIController**) that will implement **WWDCAttendesDelegate**. This separation, allows you to easily test **WWDCAttendeesUIController** independently.

The next step is to create an abstraction for the controller, so you can inject it into your **UIViewController**:

From the point of view of the **UIViewController** subclass, the implementation would look like this:

This approach will put the fetching action on the **UIViewController** side, but leave the response handling to the **WWDCAttendeesUIController**:

You can see the **WWDCAttendeesUIController** as the UI brain, while the **WWDCAttendeesController** as the business logic brain.

Wow, that wasn't hard! But this example begs the question: Who creates the controller?

I recommend always making the controller injectable — so the owner of your **UIViewController** would provide the controller. This has two main benefits:

- ***It's easily testable.*** You can simply pass any object that complies with the **FetchNumberOfTickets** protocol.
- ***The layers are cleanly decoupled.*** This helps in defining responsibilities, which leads to an overall healthier code base.

The Model Layer

The model layer is not as self-explanatory as it may seem.

As you would expect, it will have your model objects, potentially covering most of the layer surface. In the tickets example, you would have a **Ticket** struct that would live in your model.

I find the following components to also be part of the model layer:

- ***Network Code.*** The shape should be something like this. Ideally, you'd only use a single class for network communication across your entire app.
- ***Persistence Code.*** You would implement this with Core Data or simply by saving an **NSData** blob directly to disk.
- ***Parsing Code.*** Any objects that parse network responses and the like should be included in the Model layer as well.

While the model objects and the parser are domain-specific, the network code will be highly reusable.

The controller will then use all the elements in your model layer to define the flow of information in your app.

View Controllers

View controllers present and manage a hierarchy of views. The UIKit framework includes classes for view controllers you can use to set up many of the common user interaction idioms in iOS. You use these view controllers with any custom view controllers you may need to build your app's user interface. This document describes how to use the view controllers that are provided by the UIKit framework.

Storyboard

A storyboard is a visual representation of the user interface of an iOS application, showing screens of content and the connections between those screens. A storyboard is composed of a sequence of scenes, each of which represents a view controller and its views; scenes are connected by segue objects, which represent a transition between two view controllers.

Xcode provides a visual editor for storyboards, where you can lay out and design the user interface of your application by adding views such as buttons, table views, and text views onto scenes. In addition, a storyboard enables you to connect a view to its controller object, and to manage the transfer of data between view controllers. Using storyboards is the recommended way to design the user interface of your application because they enable you to visualize the appearance and flow of your user interface on one canvas.

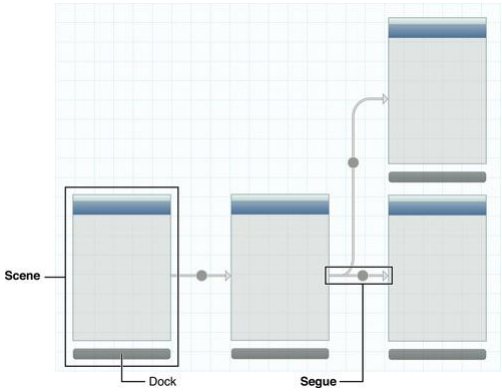


Fig: Storyboard

Outlet

An outlet is a property that is annotated with the symbol IBOutlet and whose value you can set graphically in a nib file or a storyboard. You declare an outlet in the interface of a class, and you make a connection between the outlet and another object in the nib file or storyboard. When the file is loaded, the connection is established.

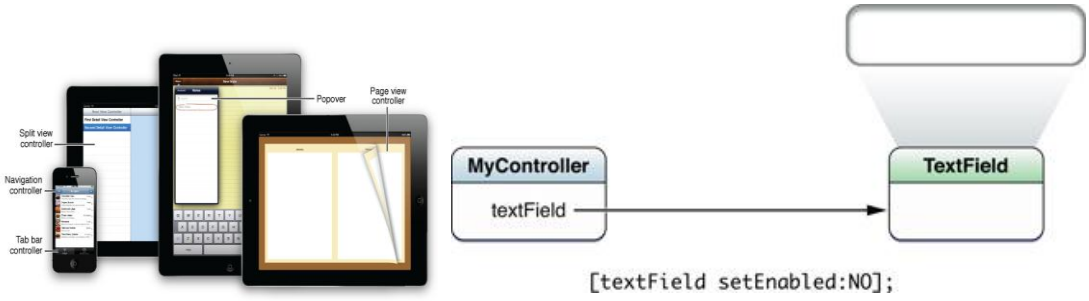


Fig: Outlet

Action

Similar to an *Outlet*, an *Action* is a connection between an element of the interface and the code with an addition of one more parameter - an **Event**.

Action is **a connection between an element of the interface and the CODE associated with an event**.

What does it mean exactly?

In context of our Acty app, we expect the user to tap on the '*ACT NeXT*' button, so that we can initiate our acting task generation. We'll use most popular and natural tapping (or touching) event for a button called **TouchUpInside**. It occurs after the user touched (*Touch*) the button and then let go (*Up*) while their finger was still within the frame of the button (*Inside*).

Suggested reading list:

iOS Programming: The Big Nerd Ranch Guide

iOS App Development For Dummies

Swift Programming: The Big Nerd Ranch Guide

Reference and bibliography:

https://developer.apple.com/library/content/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html

https://www.raywenderlich.com/132662/mvc-in-ios-a-modern-approach

https://www.raywenderlich.com/132662/mvc-in-ios-a-modern-approach

https://learnappmaking.com/model-view-controller-mvc-swift/

https://www.raywenderlich.com/160527/auto-layout-tutorial-ios-11-getting-started

https://www.appcoda.com/learnswift/auto-layout-intro.html

Module 4: Auto Layout

Topic:

Constraints, insufficient and Conflicting Constraints
Misplaced Views
Content Hugging and Compression Resistance
Size Classes

Chapter Overview:

Auto Layout is a way that lets developers create user interface by defining relationships between elements. It provides a flexible and powerful system that describes how views and the UI controls relate to each other. By using Auto Layout, you can get an incredible control over layout, with a wide range of customization, and yield the perfect interface.

Learning Outcome:

- Why Auto Layout
- Constraints, insufficient and Conflicting Constraints
- Knowledge Misplaced Views
- Content Hugging and Compression Resistance
- Discussion about Size Classes

In this chapter we will discuss briefly about Auto Layout.

Auto Layout:

Auto Layout is a fantastic tool. It does things that earlier technologies could never dream of. From the edge case handling of creation of reciprocal relationships between views, Auto Layout introduces immense power. What's more, Auto Layout is compatible with many of Apple's most exciting application programming interfaces (APIs), including animations, motion effects, and sprites.

Okay, let me give you an example and hopefully you'll have a better idea why Auto Layout is needed. In Storyboard, you place a button right in the center of the view. Run the app on both iPhone Retina (3.5-inch) and iPhone Retina (4-inch) simulators.

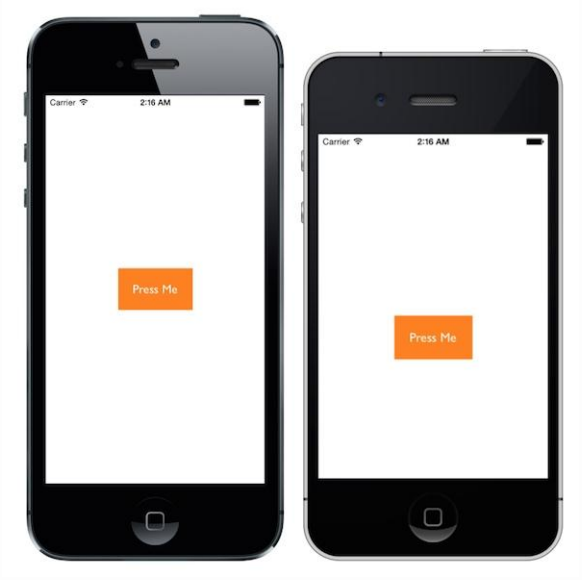


Fig: Layout

You'll end up with the above results and it turns out that the button isn't centered when running on a 3.5-inch device.

Why? What's wrong with it?

Without using Auto Layout, the UI controls (e.g. button) we layout in the Storyboard is of fixed position. In other words, we "hard-code" the frame origin of the control. For example, the "Press Me" button's frame origin is set to (104, 255). Therefore, whether you're using a 3.5-inch or 4-inch device, iOS will draw the label in the specified position. This explains why the "Press Me" button was not displayed properly on a 3.5-inch iPhone, for which the screen height is different.

Obviously, we want to the app look good on both 3.5-inch and 4-inch iPhone. And this is why we need Auto Layout.

How to Use Auto Layout in Interface Builder

Before we show you how to fix the alignment issue in the example, let's have a brief walkthrough of the Interface Builder and get to know how Auto Layout can be applied.

First, set up a new project based on the Single View Application iOS app template. In the project options, choose iPhone for the device family, save the project, then open the Storyboard. You will notice a menu at the bottom-right corner. The buttons in the menu are related to Auto Layout. You can use for alignment, sizing, spacing and resolving constraint issue.

- Align – Create alignment constraints, such as aligning the left edges of two views.
- Pin – Create spacing constraints, such as defining the width of a UI control.
- Issues – Resolve layout issues.
- Resizing – Specify how resizing affects constraints.

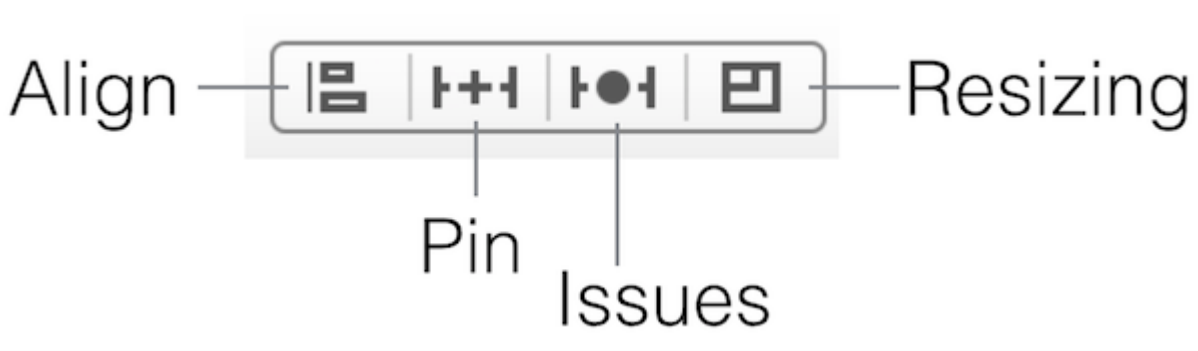


Fig: Aligning

Other than the Auto Layout menu, Apple has made it flexible for developer to setup Auto Layout by using Control+drag. You simply control-drag from any view to another view to set constraints between each other. When you release the mouse, it presents a list of possible constraints for you to select from.

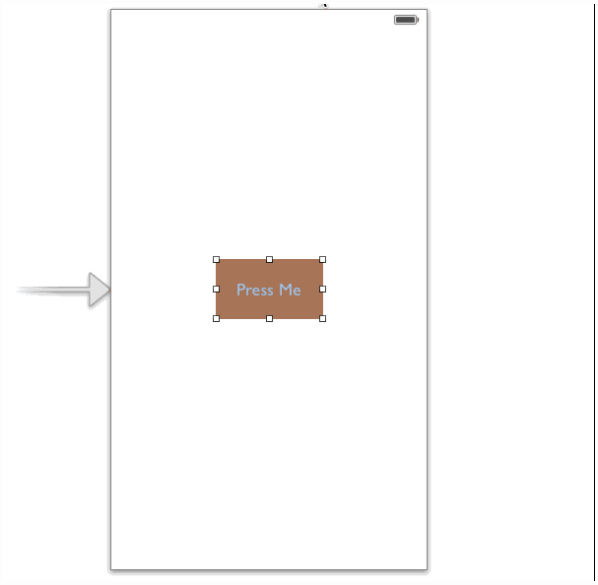


Fig: Layout

Once you setup a constraint in a view, the constraint line is displayed in either in orange or blue. The orange constraint lines indicates that there are insufficent constraints and you need to fix it.

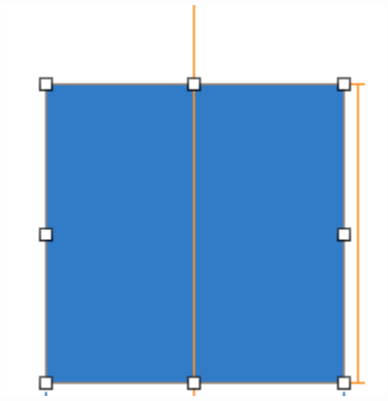


Fig: Layout

The blue constraint line indicates that your view layout is being setup correctly and there is no ambiguity.

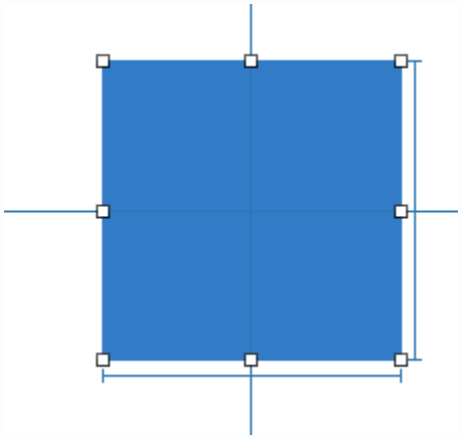


Fig: Layout

Sometimes after you create the constraint, the Interface Builder outline view shows a disclosure arrow. The red arrow also indicates that there are conflicts or ambiguities. Click the disclosure arrow, and you'll see a list of the issues. The issues are displayed on a scene-by-scene basis. Typical issues include missing constraints, conflicting constraints and misplaced views.

Practice:

Open Xcode and create a new project based on the Single View Application template. Call the app "StrutsProblem", choose iPhone and disable Storyboards:

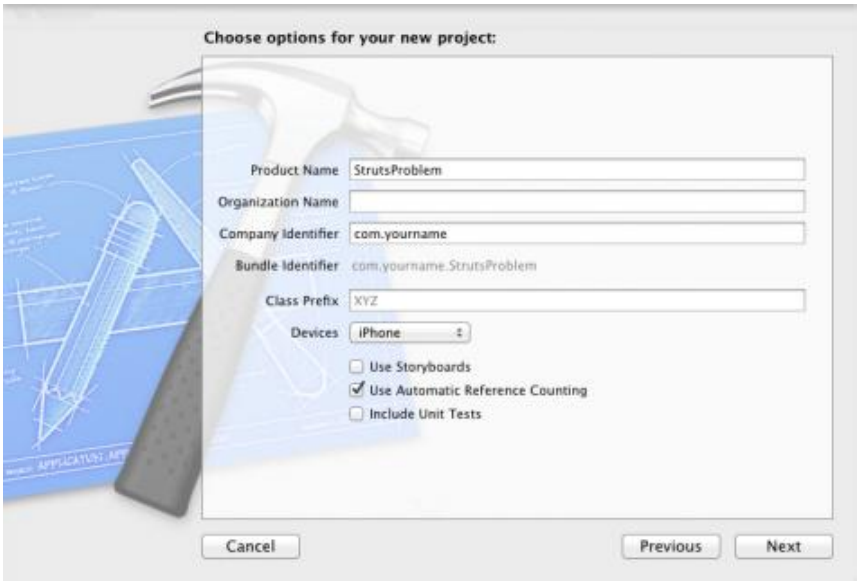


Fig: Xcode

Click on ***ViewController.xib*** to open it in Interface Builder. Before you do anything else, first disable Auto Layout for this nib. You do that in the File inspector:

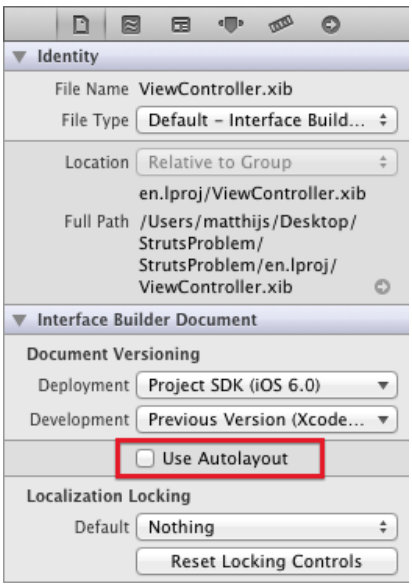


Fig: Autolayout

Uncheck the "Use Autolayout" box. Now the nib uses the old struts-and-springs model.

Note: Any new nib or storyboard files that you create with Xcode 4.5 or better will have Auto Layout activated by default. Because Auto Layout is an iOS 6 feature only, if you want to use Xcode 4.5 to make apps that are compatible with iOS 5, you need to disable Auto Layout on any new nibs or storyboard files by unchecking the "Use Autolayout" checkbox.

Drag three new views on to the main view and line them up like this:

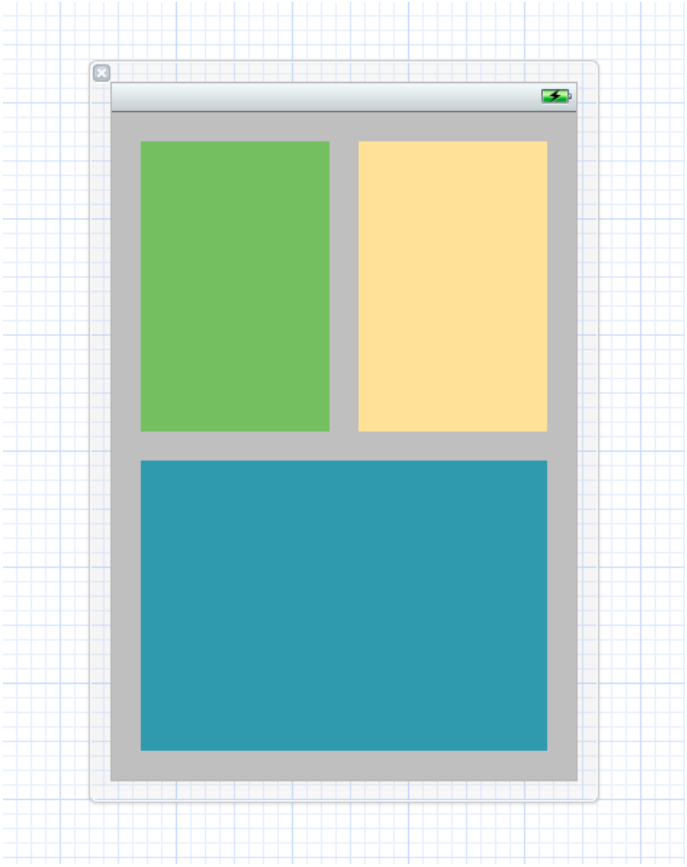


Fig: Layout

For clarity, give each view its own color so that you can see which is which.

Each view is inset 20 points from the window's borders; the padding between the views is also 20 points. The bottom view is 280 points wide and the two views on top are both 130 points wide. All views are 200 points high.

Run the app and rotate the simulator or your device to landscape. That will make the app look like this, not quite what I had in mind:

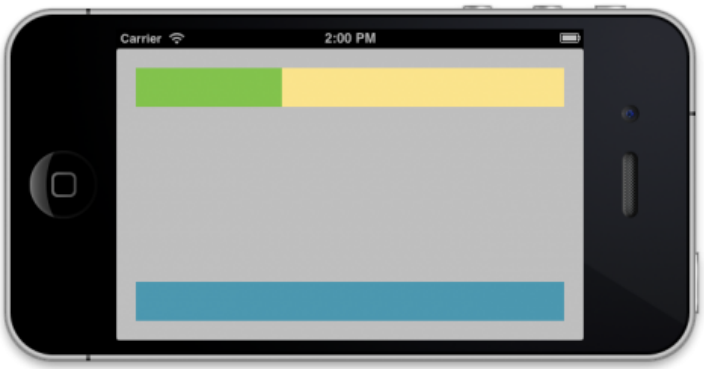


Fig: Layout

Note: You can rotate the simulator using the Hardware\Rotate Left and Rotate Right menu options, or by holding down Cmd and tapping the left or right arrow keys.

Instead, I want the app to look like this in landscape:

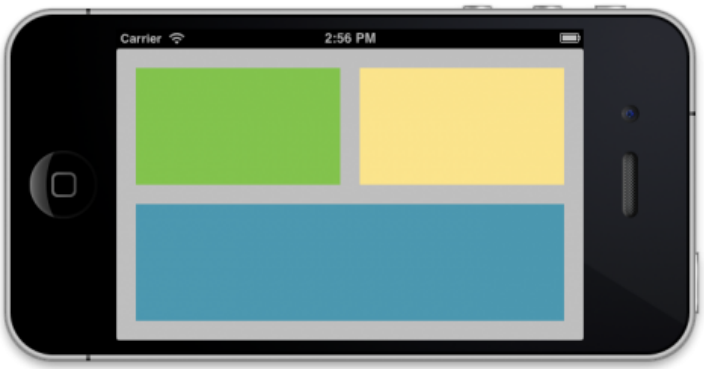


Fig: Layout

Obviously, the autosizing masks for all three views leave a little something to be desired. Change the autosizing settings for the top-left view to:

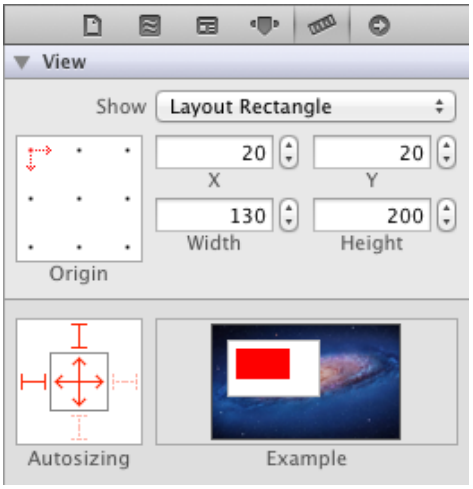


Fig: Layout

This makes the view stick to the top and left edges (but not the bottom and right edges), and resizes it both horizontally and vertically when the superview changes its size.

Similarly, change the autosizing settings for the top-right view:

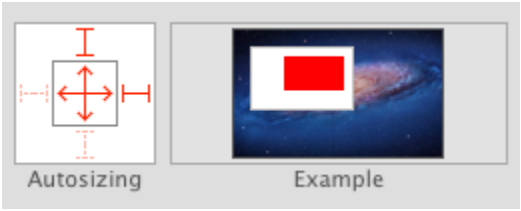


Fig: Layout

And for the bottom view:

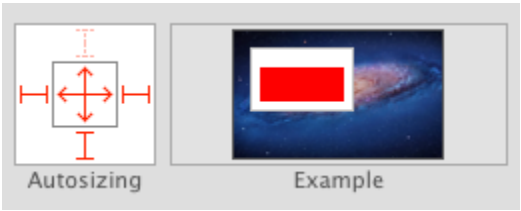


Fig: Layout

Run the app again and rotate to landscape. It should now look like this:

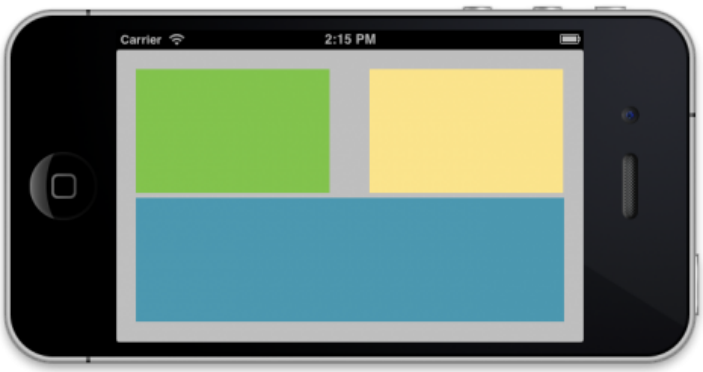


Fig: Layout

Close, but not quite. The padding between the views is not correct. Another way of looking at it is that the sizes of the views are not 100% right. The problem is that the autosizing masks

tell the views to resize when the superview resizes, but there is no way to tell them ***by how much*** they should resize.

You can play with the autosizing masks – for example, change the flexible width and height settings (the "springs") – but you won't get it to look exactly right with a 20-point gap between the three views.

Fig: Infographic

To solve this layout problem with the springs and struts method, unfortunately you will have to write some code.

UIKit sends several messages to your view controllers before, during and after rotating the user interface. You can intercept these messages to make changes to the layout of your UI. Typically you would override ***willAnimateRotationToInterfaceOrientation:duration:*** to change the frames of any views that need to be rearranged.

But before you can do that, you first have to make outlet properties to refer to the views to be arranged.

Switch to the Assistant Editor mode (middle button on the Editor toolset on the Xcode toolbar) and Ctrl-drag from each of the three views onto ***ViewController.h:***

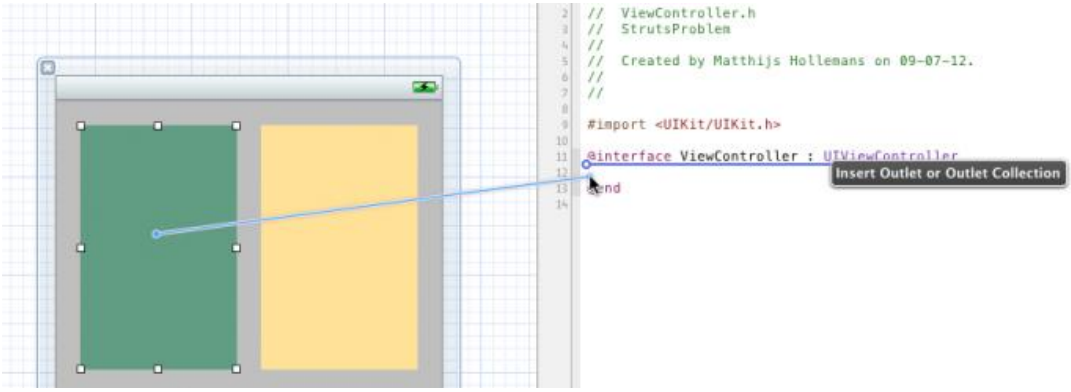


Fig: Interface

Connect the views to these three properties, respectively:

```
@property (weak, nonatomic) IBOutlet UIView *topLeftView;
@property (weak, nonatomic) IBOutlet UIView *topRightView;
```

```
@property (weak, nonatomic) IBOutlet UIView *bottomView;
```

Add the following code to ***ViewController.m***:

```
-(void)willAnimateRotationToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation
                                        duration:(NSTimeInterval)duration
{
    [super willAnimateRotationToInterfaceOrientation:toInterfaceOrientation duration:duration];

    if (toInterfaceOrientation == UIInterfaceOrientationLandscapeLeft
    ||  toInterfaceOrientation == UIInterfaceOrientationLandscapeRight)
    {
        CGRect rect = self.topLeftView.frame;
        rect.size.width = 210;
        rect.size.height = 120;
        self.topLeftView.frame = rect;

        rect = self.topRightView.frame;
        rect.origin.x = 250;
        rect.size.width = 210;
        rect.size.height = 120;
        self.topRightView.frame = rect;

        rect = self.bottomView.frame;
        rect.origin.y = 160;
        rect.size.width = 440;
        rect.size.height = 120;
        self.bottomView.frame = rect;
    }
    else
    {
        CGRect rect = self.topLeftView.frame;
        rect.size.width = 130;
        rect.size.height = 200;
        self.topLeftView.frame = rect;
```

```
        rect = self.topRightView.frame;
        rect.origin.x = 170;
        rect.size.width = 130;
        rect.size.height = 200;
        self.topRightView.frame = rect;

        rect = self.bottomView.frame;
        rect.origin.y = 240;
        rect.size.width = 280;
        rect.size.height = 200;
        self.bottomView.frame = rect;
    }
}
```

This callback occurs when the view controller is rotating to a new orientation. It looks at the orientation the view controller is rotating to and resizes the views appropriately – in this case with hardcoded offsets based on the known screen dimensions of the iPhone. This callback occurs within an animation block, so the changes in size will animate.

Don't run the app just yet. First you have to restore the autosizing masks of all three views to the following, or the autosizing mechanism will clash with the positions and sizes you set on the views in ***willAnimateRotation***:

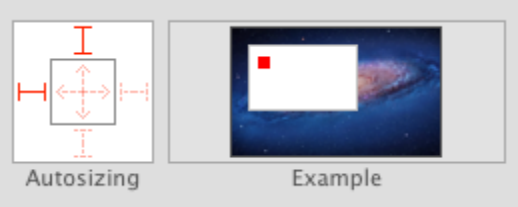


Fig: Layout

That should do it. Run the app and flip to landscape. Now the views line up nicely. Flip back to portrait and verify that everything looks good there as well.

It works, but that was a lot of code you had to write for a layout that is actually pretty simple. Imagine the effort it takes for layouts that are truly complex, especially dynamic ones where the individual views change size, or the number of subviews isn't fixed.

Fig: Infographic

Note: Another approach you can take is to make separate nibs for the portrait and landscape orientations. When the device rotates you load the views from the other nib and swap out the existing ones. But this is still a lot of work and it adds the trouble of having to maintain two nibs instead of one.

Auto Layout to the rescue!

You will now see how to accomplish this same effect with Auto Layout. First, remove ***willAnimateRotationToInterfaceOrientation:duration:*** from ***ViewController.m***, because you're now going to do this without writing any code.

Select ***ViewController.xib*** and in the File inspector panel, check the "Use Autolayout" box to enable Auto Layout for this nib file:

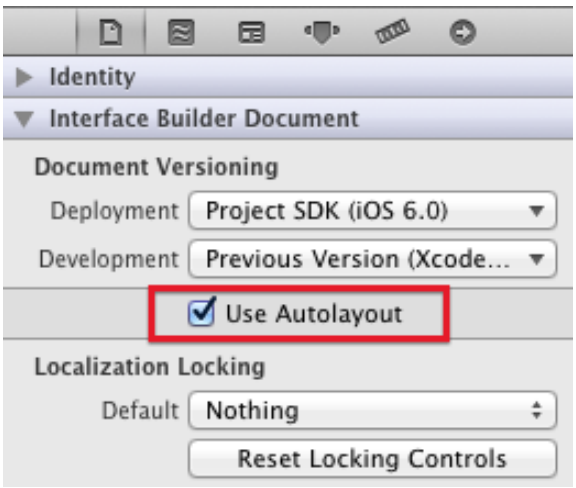


Fig: Autolayout

Note: Auto Layout is always enabled for the entire nib or storyboard file. All the views inside that nib or storyboard will use Auto Layout if you check that box.

Run the app and rotate to landscape. It should give the same messed up layout that it did earlier:

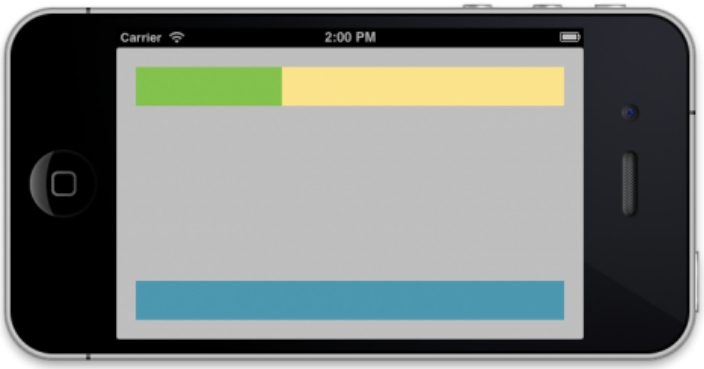


Fig: Layout

Let's put Auto Layout into action. Hold down the ***Cmd*** key while you click on the two views on the top (the green and yellow ones), so that both are selected. From Xcode's ***Editor*** menu, select ***Pin\Widths Equally***:

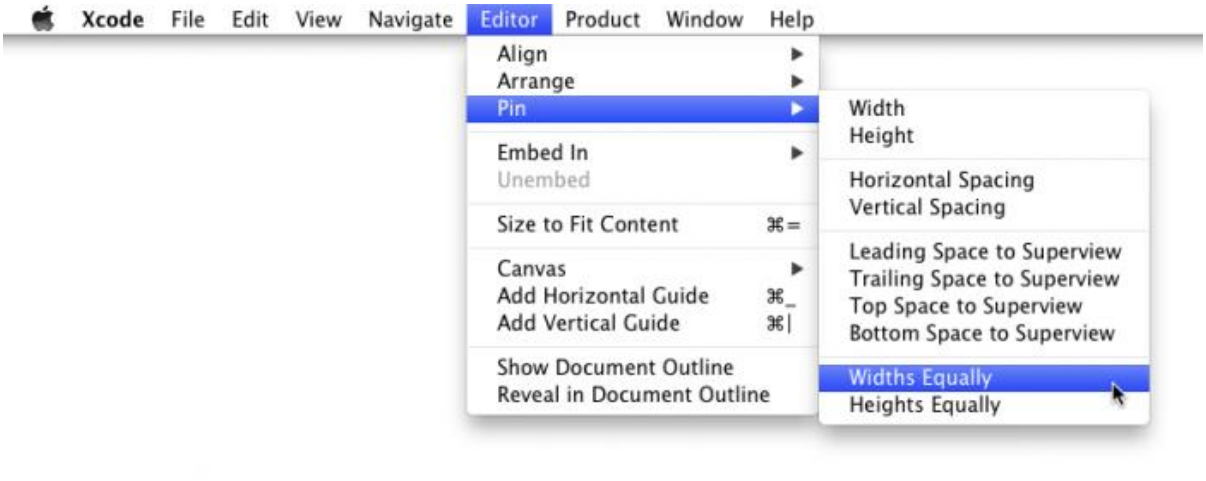


Fig: Xcode Editor

Select the same two views again and choose ***Editor\Pin\Horizontal Spacing***. (Even though the two views appear selected after you carry out the first Pin action, do note that they are in a special layout relationship display mode. So you do have to reselect the two views.)

In the Document Outline on the left, you'll notice a new section named "Constraints". This section was added when you enabled Auto Layout for the nib. You will learn all about what these constraints are and how they operate in the next section.

For now, locate the one named "Horizontal Space (170)" and delete it from the list:

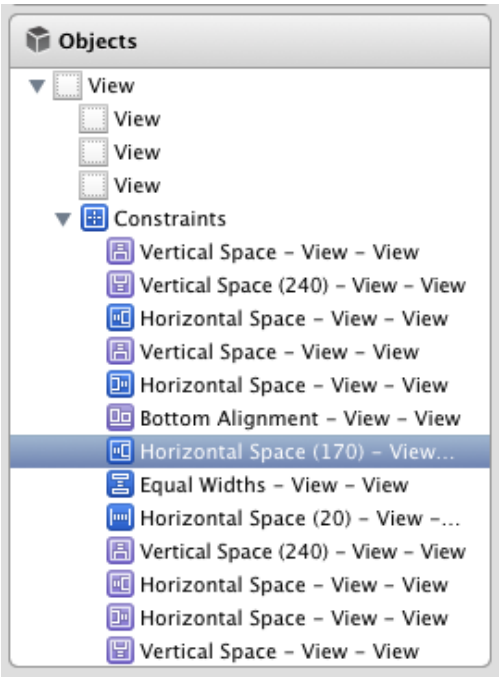


Fig: Xcode Editor

Run the app and rotate to landscape. That looks better already – the views at the top now have the proper widths and padding – but you're not quite there yet:

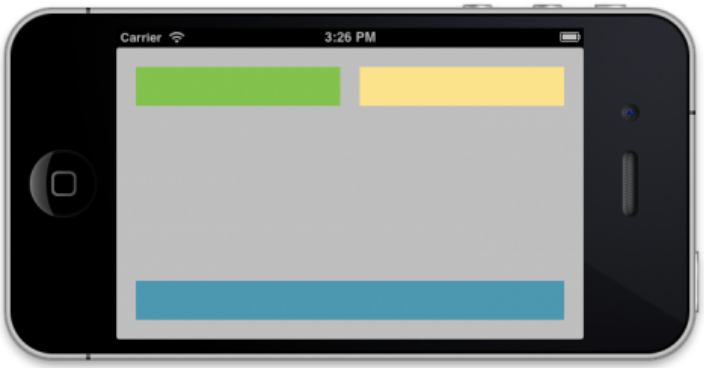


Fig: Layout

Hold down ***Cmd*** and select all three views. From the Editor menu, choose ***Pin\Heights Equally***.

Now select the top-left corner view and the bottom view (using Cmd as before), and choose ***Editor\Pin\Vertical Spacing***.

Finally, remove the "Vertical Space (240)" constraint from the list.

If you select all three views at the same time, Interface Builder should show something like this:

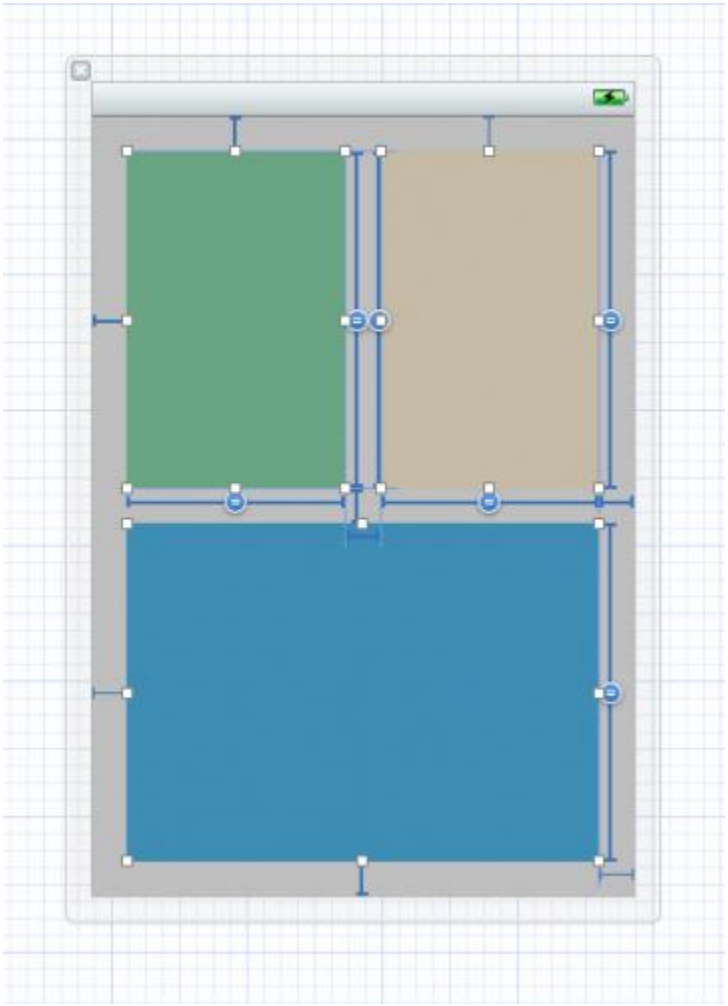


Fig: Layout

The blue “T-bar” shaped things represent the constraints between the views. It might look a bit scary, but it is actually quite straightforward once you learn what it all means.

Run the app and… voila, everything looks good again, all without writing a single line of code!

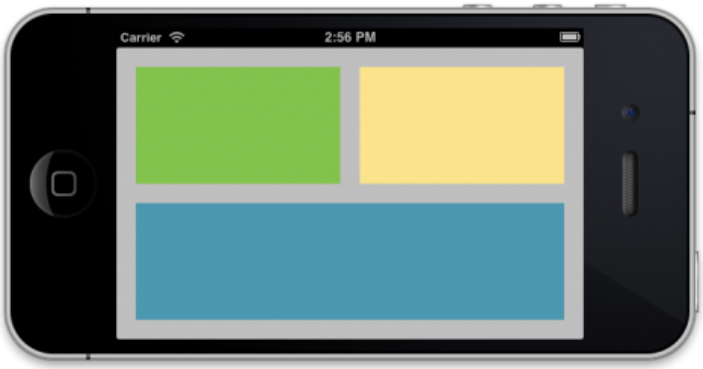


Fig: Layout

Cool, but what exactly did you do here? Rather than requiring you to hard-code how big your views are and where they are positioned, Auto Layout lets you express how the views in your layout relate to each other.

You have put the following relationships – what is known as constraints – into the layout:

- The top-left and top-right views always have the same width (that was the first pin widths equally command).
- There is a 20-point horizontal padding between the top-left and top-right views (that was the pin horizontal spacing).
- All the views always have the same height (the pin heights equally command).
- There is a 20-point vertical padding between the two views on top and the one at the bottom (the pin vertical spacing).

And that is enough to express to Auto Layout where it should place the views and how it should behave when the size of the screen changes.

Fig: Infographic

Note: There are also a few other constraints that were brought over from the springs-and-struts layout when you toggled the "Use Autolayout" checkbox. For each of the margins between the views and the edges of the screen there is now a constraint that basically says: "this view always sits at a 20-points distance from the top/bottom/left/right edge."

You can see all your constraints in the Document Outline. If you click on a constraint in the Document Outline, Interface Builder will highlight where it sits on the view by drawing a white outline around the constraint and adding a shadow to it so that it stands out:

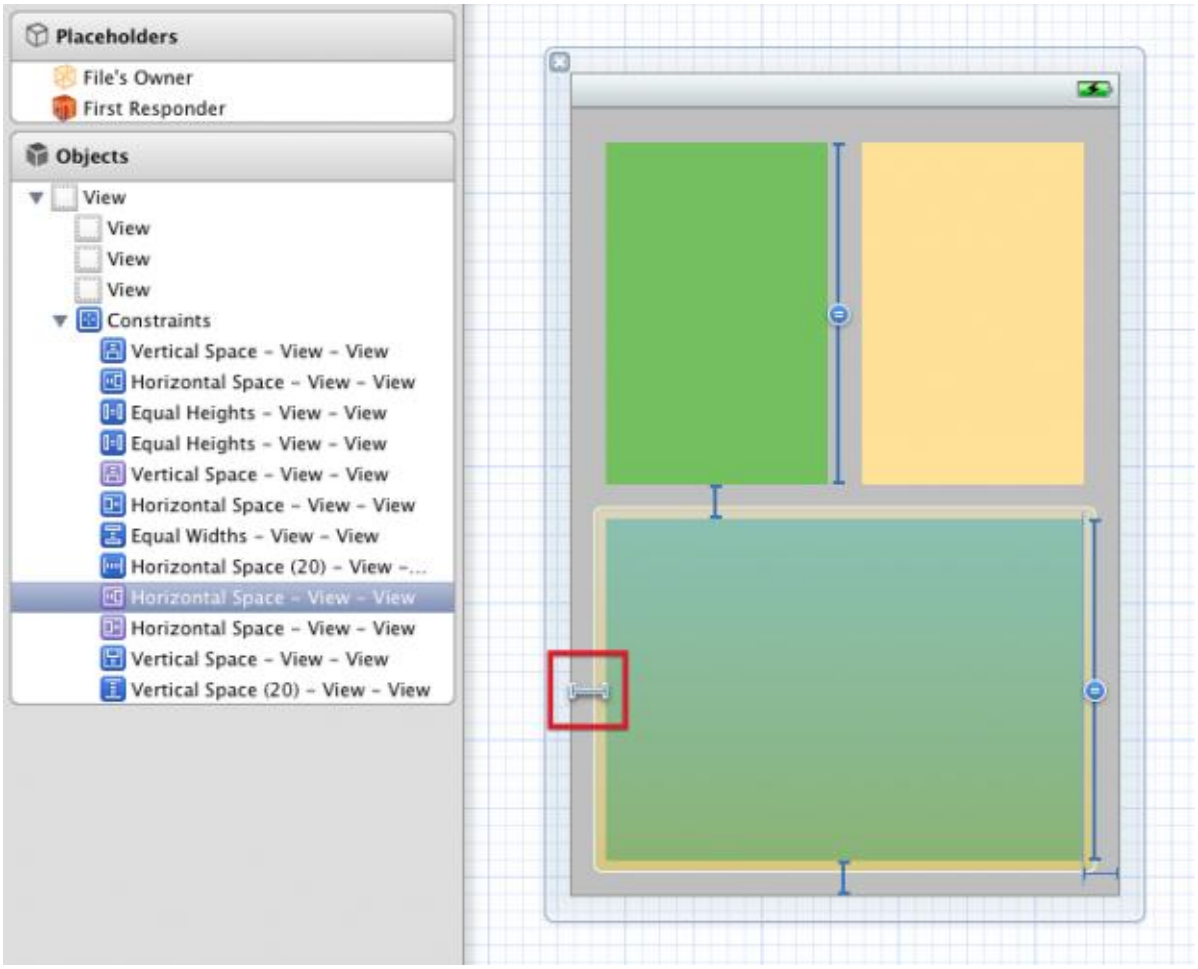


Fig: Layout

Constraints are real objects (of class ***NSLayoutConstraint***) and they also have attributes. For example, select the constraint that creates the padding between the two top views (it is named "Horizontal Space (20)") and then switch to the Attributes inspector. There you can change the size of the margin by editing the Constant field.

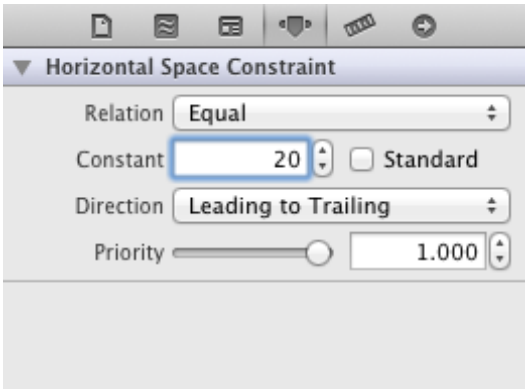


Fig: Layout

Set it to 100 and run the app again. Now the margin is a lot wider:

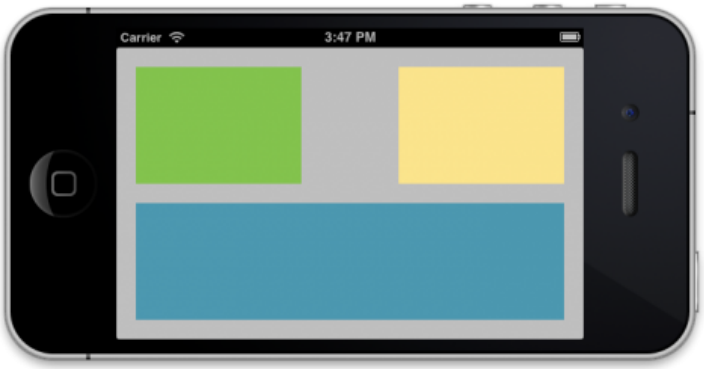


Fig: Layout

Auto Layout is a lot more expressive than springs and struts when it comes to describing the views in your apps. In the rest of this tutorial, you will learn all about constraints and how to apply them in Interface Builder to make different kinds of layouts.

How Auto Layout works

As you've seen in the test drive above, the basic tool in Auto Layout is the ***constraint***. A constraint describes a geometric relationship between two views. For example, you might have a constraint that says:

"The right edge of label A is connected to the left edge of button B with 20 points of empty space between them."

Auto Layout takes all of these constraints and does some mathematics to calculate the ideal positions and sizes of all your views. You no longer have to set the frames of your views yourself – Auto Layout does that for you, entirely based on the constraints you have set on those views.

Before Auto Layout, you always had to hard-code the frames of your views, either by placing them at specific coordinates in Interface Builder, by passing a rectangle into ***initWithFrame:***, or by setting the view's frame, bounds or center properties.

For the app that you just made, you specifically set the frames to:

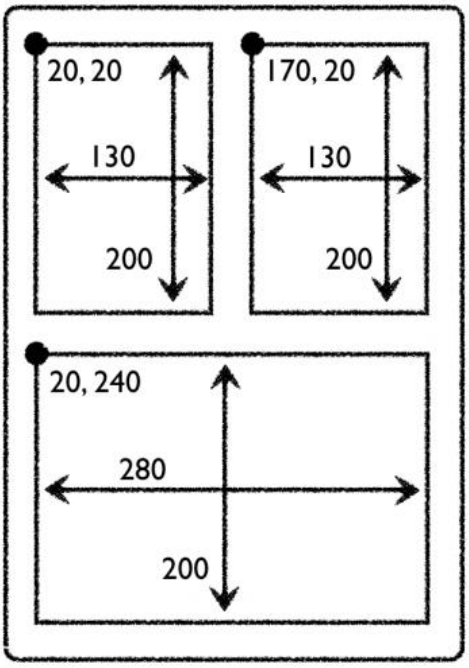


Fig: Layout

You also set auto sizing masks on each of these views:

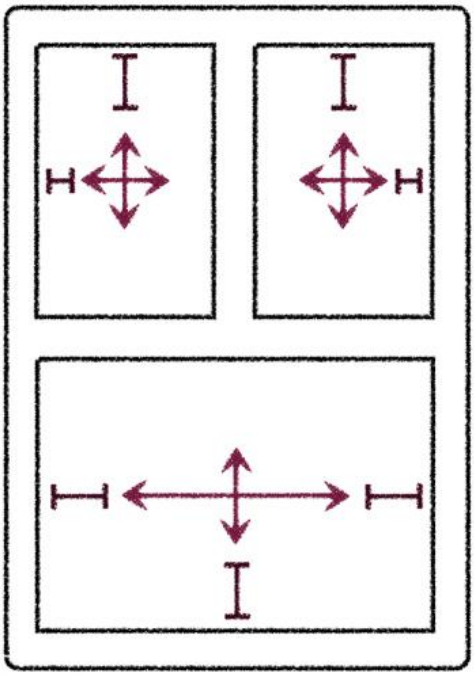


Fig: Layout

That is no longer how you should think of your screen designs. With Auto Layout, all you need to do is this:

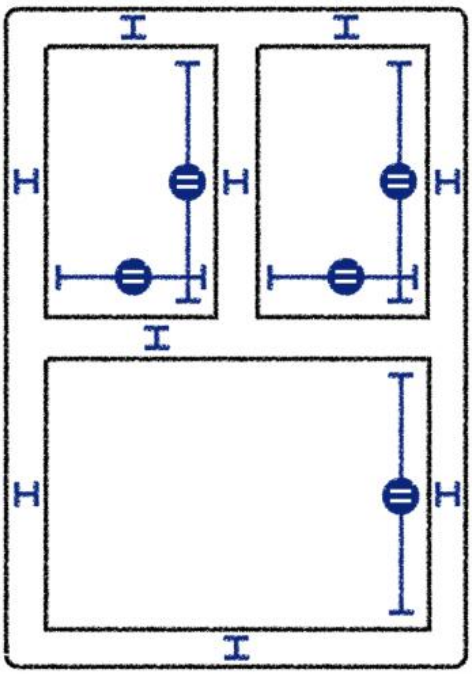


Fig: Layout

The sizes and positions of the views are no longer important; only the constraints matter. Of course, when you drag a new button or label on to the canvas it will have a certain size and you will drop it at a certain position, but that is only a design aid that you use to tell Interface Builder where to put the constraints.

Constraints, insufficient and Conflicting Constraints

Constraints

Now that you've read about the *why* of Auto Layout, this section introduces the *what*. Here's the basic vocabulary you need to start talking about this technology.

Constraints, as you learned earlier, are rules that allow you to describe view layout. They limit how things relate to each other and specify how they can be laid out. With constraints, you can say "these items are always lined up in a horizontal row" or "this item resizes itself to match the height of that item." Constraints provide a layout language that you add to views to describe geometric relationships.

The constraints you work with belong to the NSLayoutConstraint class. This Objective-C class specifies relationships between view attributes, such as heights, widths, positions, and centers. What's more, constraints are not limited to equalities. They can describe views using greater-than-or-equal and less-than-or-equal relations so that you can say that one view must be at least as big as or no bigger than another. Auto Layout development is built around creating and adjusting these relationship rules in a way that fully defines your interfaces.

Together, an interface's constraints describe the ways views can be laid out to dynamically fit any screen or window geometry. In Cocoa and Cocoa Touch, a well-defined interface layout consists of constraints that are *satisfiable* and *sufficient.*

Satisfiability

Cocoa/Cocoa Touch takes charge of meeting layout demands through its constraint satisfaction system. The rules must make sense both individually and as a whole. That is, a rule must be created in a valid manner, and it also must play a role in the greater whole. In logic systems, this is called *satisfiability*, or *validity*. A view cannot be both to the left *and* to the right of another view. So, the key challenge when working with constraints is to ensure that the rules are rigorously consistent.

Any views you lay out in IB can be guaranteed to be satisfiable, as IB offers a system that optionally checks and validates your layouts. It can even fix conflicting constraints. This is not true in code. You can easily build views and tell them to be exactly 360 points wide and 140 points wide at the same time. This can be mildly amusing if you're trying to make things fail, but it is more often utterly frustrating when you're trying to make things work, which is what most developers spend their time doing.

When rules fail, they fail loudly. At compile time, Xcode issues warnings for conflicting IB constraints and other IB-based layout issues. At runtime, the Xcode console provides verbose updates whenever the solver hits a rough patch. That output explains what might have gone wrong and offers debugging assistance.

In some cases, your code will raise exceptions. Your app terminates if you haven't implemented handlers. In other cases (such as the example that follows), Auto Layout keeps your app running by deleting conflicting constraint rules for you. This produces interfaces that can be somewhat unexpected.

Regardless of the situation, it's up to you to start debugging your code and your IB layouts to try to track down why things have broken and the source of the conflicting rules. This is never fun.

Consider the following console output, which refers to the view I mentioned that attempts to be both 360 points and 140 points wide at the same time:

The boldface in this code is mine. I've used it to highlight the sizes for each constraint, plus the reason for the error. In this example, both rules have the same priority and are inconsistent with each other.

This unsatisfiable conflict cannot be resolved except by breaking one of the constraints, which the Auto Layout system does. It arbitrarily discards one of the two size requests (in this case, the 360 size) and logs the results.

Sufficiency

Another key challenge is making sure that your rules are specific enough. An underconstrained interface (one that is *insufficient* or *ambiguous*) creates random results when faced with many possible layout solutions (see the top portion of Figure 1-3). You might request that one view lies to the right of the other, but unless you tell the system otherwise, you might end up with the left view at the top of the screen and the right view at the bottom. That one rule doesn't say anything about vertical orientation.

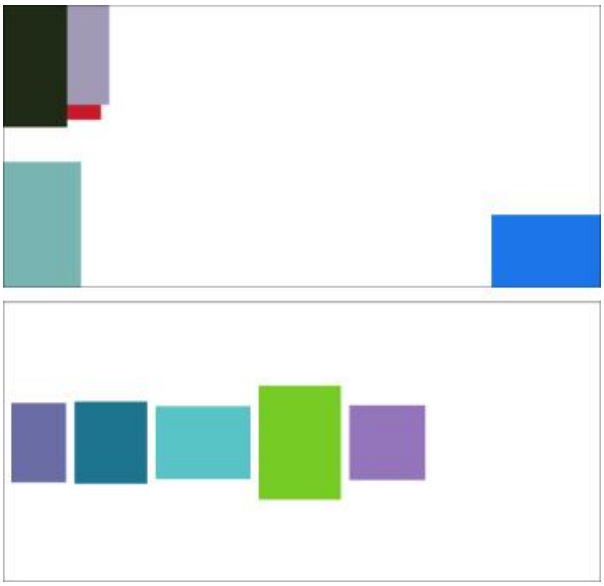


Fig: Layout

Odd layout positions (top) are the hallmark of an underconstrained layout. Although these particular views are constrained to show up onscreen, their near-random layout indicates insufficient rules describing their positions. By default, views might not show up at all, especially when they are underconstrained. Chapter 4, "Visual Formats," discusses fallback rules, which ensure that views are both visibly sized and onscreen. A sufficient layout (bottom) provides layout rules for each of its views.

A sufficient set of constraints fully expresses a view's layout, as in the bottom portion of Figure 1-3. In this case, each view has a well-defined size and position.

Sufficiency does not mean "hard coded." In the layout shown at the bottom of Figure 1-3, none of these positions are specified exactly. The Auto Layout rules say to place the views in a horizontal row, center-aligned vertically to each other. The first view is pinned off of the superview's left-center. These constraints are sufficient because every view's position can be determined from its relationships to other views.

A sufficient, or *unambiguous*, layout has at least two geometric rules per axis, or a minimum of four rules in all. For example, a view might have an origin and a size—as you would use with frames—to specify where it is and how big it is. But you can express much more with

Auto Layout. The following sufficient rule examples define a view's position and extent along one axis, as illustrated in Figure 1-4:

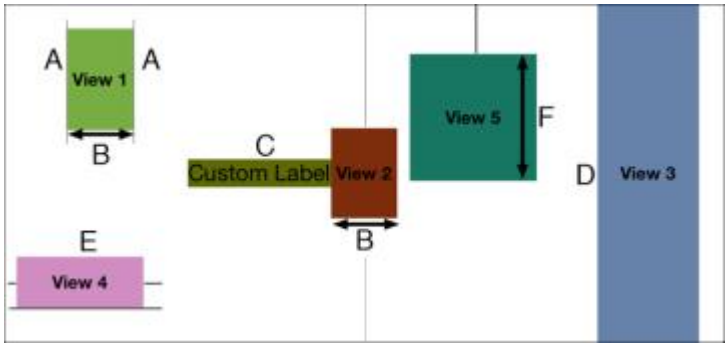


Fig: Layout

Sufficient layout requires at least two rules per axis.

- You could pin the horizontal edges (A) of a view to exact positions in its superview. (The two properties defined in this example are the view's minimum X and maximum X positions.)
- You could match the width of one view to another subview (B) and then center it horizontally to its superview (width and center X).
- You could declare a view's width to match its intrinsic content, such as the length of text drawn on it (C), and then pin its right (*trailing*) edge to the left (*leading*) edge of another view (width and maximum X).
- You could pin the top and bottom of a view to the superview (D) so that the view stretches vertically along with its superview (minimum Y and maximum Y).
- You could specify a view's vertical center and its maximum extent (E) and let Auto Layout calculate the height from that offset (center Y and maximum Y).
- You could specify a view's height and its offset from the top of the view (F) and then hang the view off the top of the superview (minimum Y and height.).

Each of these rules provides enough information along one axis to avoid ambiguity. That's because each one represents a specific declaration about how the view fits into the overall layout.

When rules fail, they lack this exactness. For example, if you supply only the width, where should the system place the item along the X-axis? At the left? Or the right? Somewhere in the middle? Or maybe entirely offscreen? Or if you only specify a Y position, how tall should the view be? 50 points? 50,000 points? 0 points? Missing information leads to ambiguous layouts.

You often encounter ambiguity when working with inequalities, as in the top image in Figure 1-3. The rules for these views say to stay within the bounds of the superview—but where? If their minimum X value is greater than or equal to their superview's minimum X value, what should that X value be? The rules are insufficient, and the layout is therefore ambiguous.

Misplaced Views

It is an error type regarding to views and related to auto layout.

Misplaced views have the correct constraints but are not in the place they will display at run time. If you select the misplaced view, it will show how far off from the actual location the

view is. The storyboard will also show a dotted rectangle giving the correct position. For example this label is a misplaced view.

Fig: Layout

To resolve a misplaced view you select the misplaced view, and in the selected view section click **Update Frames** in the resolver. You can avoid this step by using the **Update Frames** selection in the pin and align popovers. You may not want to do this immediately since there is another kind of error to get rid of first.

Content Hugging and Compression Resistance

Priorities are very much important when dealing with autolayout. Every constraint has a priority. It is just a number ranges from 0–1000 .

According to apple docs: *The layout priority is used to indicate to the constraint-based layout system which constraints are more important, allowing the system to make appropriate tradeoffs when satisfying the constraints of the system as a whole.*

The priority really come in to play only if two different constraints conflict. The system will give importance to the one with higher priority. So, Priority is the tie-breaker in the autolayout world.

Content hugging priority:

Sets the priority with which a view resists being made larger than its intrinsic size. Setting a larger value to this priority indicates that we don't want the view to grow larger than its content.

Consider the above situation where two views placed horizontally with no proper constraints for width. This will create a conflict. In this situation we need to set the content hugging priority of one view greater than that of the other.

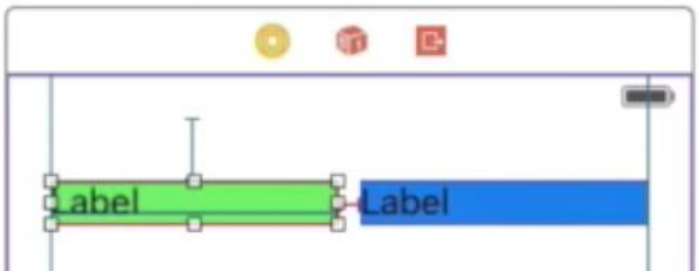


Fig: Layout

horizontal content hugging priority: green(251)- blue(251)

Consider this image. Two labels are dropped onto the view and constraints are given to the top trailing and leading sides of both the labels. Width of both these labels are not given, which

creates a conflict between these two labels. Here , both the labels are having horizontal content hugging priority equal to 251. As I have mentioned before, one view should have a higher priority constraint than the other to break the tie.

Let's set the horizontal content hugging priority of green label to 250 and let the blue labels priority remains untouched. In this case , as mentioned earlier, the one view with higher horizontal content hugging priority will not grow beyond its content size. That means the green label will grow and the blue one will stick to its intrinsic content size.

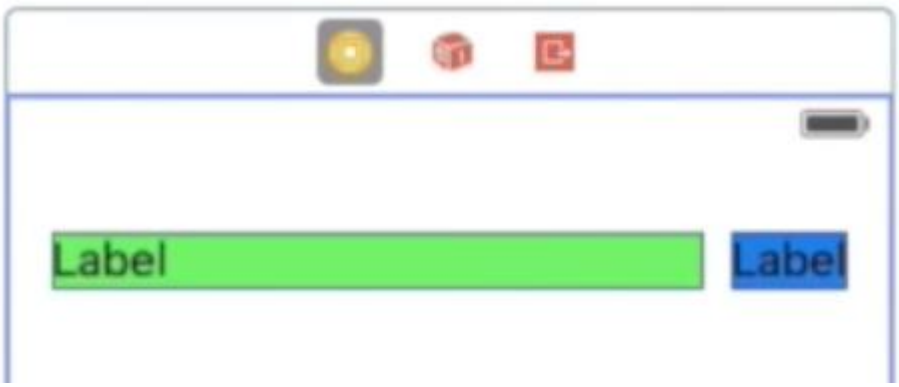


Fig: Layout

horizontal content hugging priority: green (250)- blue(251)

Similarly, If green is having a higher value means the blue label will grow beyond its intrinsic content size.

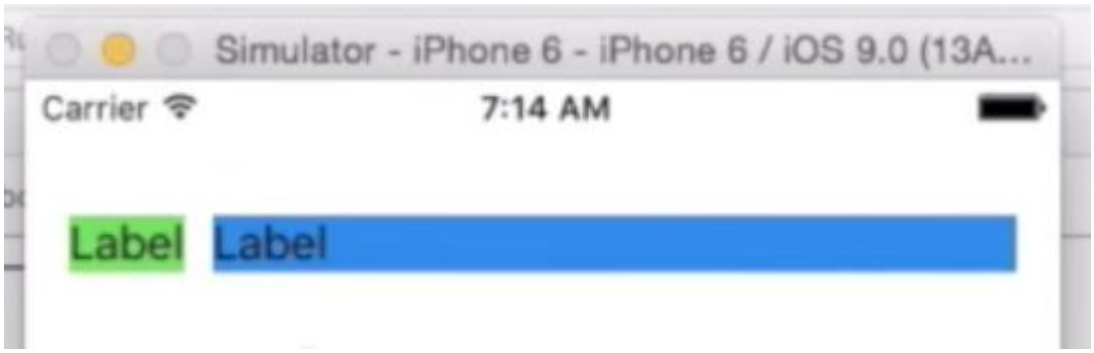


Fig: Layout

horizontal content hugging priority: green(251)- blue(250)

So, larger the content hugging priority the views bound will hug to the intrinsic content more tightly preventing the view to grow beyond its intrinsic content size.

Content compression resistance priority:

Sets the priority with which a view resists being made smaller than its intrinsic size. Setting a higher value means that we don't want the view to shrink smaller than the intrinsic content size.

Content compression resistance is pretty straight forward. There is not much complication. Higher the priority means Here is an example: Consider a button with a really long name:

Let the name be "***Button with a larger name***". We've added a simple constraint telling Auto Layout to try to keep the width of our button at 44 points. Auto Layout does as its told and collapses our button making it completely unreadable.

larger the resistance to get shrunk.

Fig: Layout

Button horizontal compression resistance priority is 750 and button width constraint priority is 1000

Don't worry, we can use Compression Resistance to stop this. Set the buttons horizontal Compression Resistance Priority to 1000. And now, change the priority of the width constraint to any value between 0 to 999. ie; less than the horizontal Compression Resistance Priority of the button. Auto Layout now allows our button's intrinsic content size to take precedent over our width contraint:

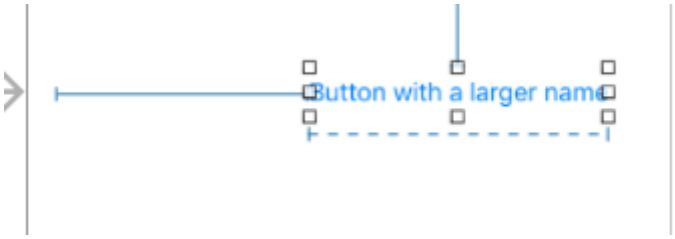


Fig: Layout

Button horizontal compression resistance priority is 1000 and button width constraint priority is 999

Size classes on iOS 10 & Xcode 8

A few years ago, were was only two devices: an iPad and an iPhone. Storyboards were separate for iPads and iPhones. For any device, there was extra code necessary to use the same app in landscape and portrait. With new phone sizes, like the iPhone 6 plus, the iPad Pro or the retro-sized iPhone SE this became a bit more of a problem. Developers could design a different storyboard for every size and orientation. That would be an excessive amount of work. Every new version of the phone or tablet would need to be different, with a separate storyboard. For all devices running iOS 10, that is four iPhones in portrait and landscape, three iPads in Portrait and landscape and ten panels for iPad multitasking. To develop a storyboard or code for each device would mean designing for twenty-four views. If you hear about *device fragmentation* this is what people are talking about: different devices need different layouts because their screens are different sizes. It's not just mobile devices that have this problem. AppleTV might run your iOS app. Televisions vary widely in size. AppleTV needs to change the view to handle those differences.

Over that last few years, there have been many solutions to *adaptive user interfaces,* which are interfaces that adapt their size and shape automatically to the device or window they happen to be in. Apple's solution to this problem is *auto layout*, which lets the system do the hard work. Using relations between views, we describe how to layout the user interface.

In iOS 8, Apple introduced *size classes*, a way to describe any device in any orientation. Size classes rely heavily on auto layout. Until iOS 8, you could escape auto layout. IN iOS8, Apple

changed several UIKit classes to depend on size classes. Modal views, popovers, split views, and image assets directly use size classes to determine how to display an image. Identical code to present a popover on an iPad causes a iPhone to present a modal view.

Size Classes

There are two sizes for size classes: **compact**, and **regular**. Sometime you'll hear about any. **Any** is the generic size that works with anything. The default Xcode layout, is **width:any height:any**. This layout is for all cases. The Horizontal and vertical dimensions are called *traits*, and can be accessed in code from an instance of UITraitCollection. The **compact** size describes most iPhone sizes in landscape and portrait. The trait of most importance is the width trait. The width is compact on all phones but the iPhone Plus models. There is one exception: the width in landscape is **regular** for an iPhone 6 Plus, which can cause some confusion. iPhone 6 Plus acts like a iPhone in portrait but an iPad in landscape. For both width and height, the full iPad and the 2/3 iPad for multitasking is the **regular** size. The 1/3 and 1/2 iPad multitasking modes are compact in width and regular in height.

There's one more variation to compact and regular. You can specify both compact and regular with **Any**. When you add **Any** to **Compact** and **Regular** there are nine size classes the developer can use.

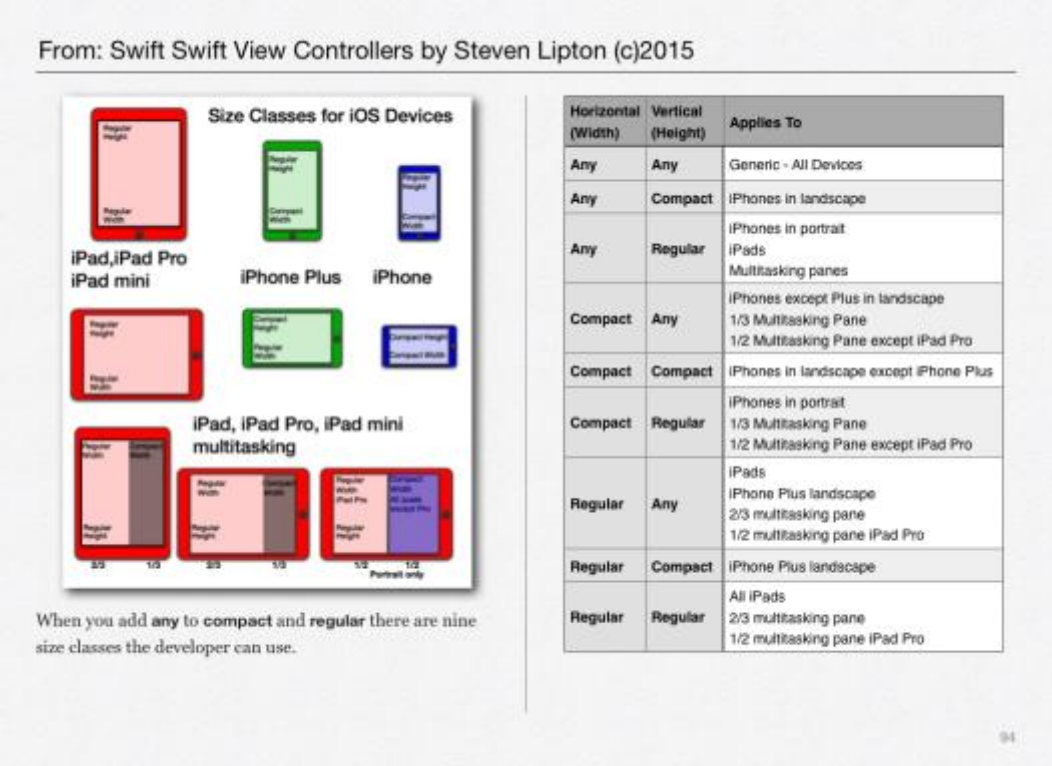


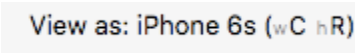
Fig: Layout

Prior to Xcode 8, developers would have to know what all these sizes are and which device each belonged. That changed in Xcode 8 with a new user interface for Interface Builder. Xcode displays as selection of devices, the developer selects the devices, then Xcode previews the layout on that device.

Viewing Size Classes

The **any** class makes working with size classes a bit more generic, saving work. Our designs start in the **width:any height:any** size class, covering all cases. If we need a special design, then we use that specific size class. Prior to Xcode 8, **any** needed to be explicit, and in some places you'll find explicit uses of **any** still. Often Xcode 8 hides **Any** from you, making it implicit instead of explicit.

Open a new single view project in XCode. Go to the storyboard. At the bottom left storyboard you'll find this:



the iPhone 6s is the default device for Interface Builder. Click this text. A new toolbar appears below, with the iPhone6s in portrait selected.

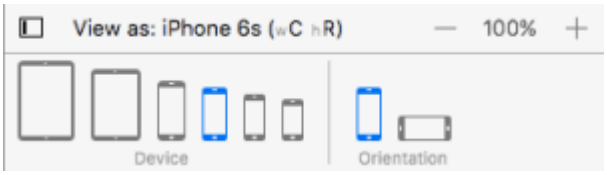


Fig: Layout

Select the iPad Pro 12.9″ on the left. The toolbar changes to include all size classes for an iPad Pro.

Fig: Layout

You'll notice the screen looks blank. The iPad is too big for the screen. In the center of the toolbar click the **100%**. In the menu that appears, click the 50%.

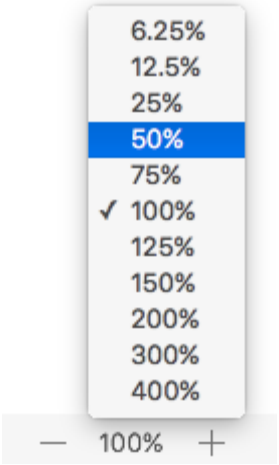


Fig: Layout

To make the device easier to see, I changed the background color attribute of the view to light gray.

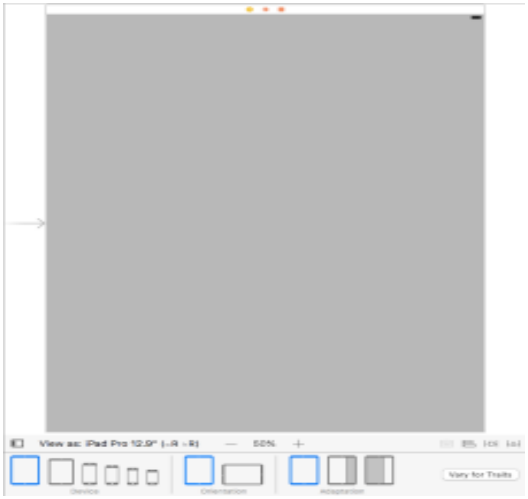


Fig: Layout

Select the **iPhone 4s** and under **Orientation** set the orientation to **Landscape**. We get a preview of a iPhone4s in landscape. Zoom in to 100% to see it better.

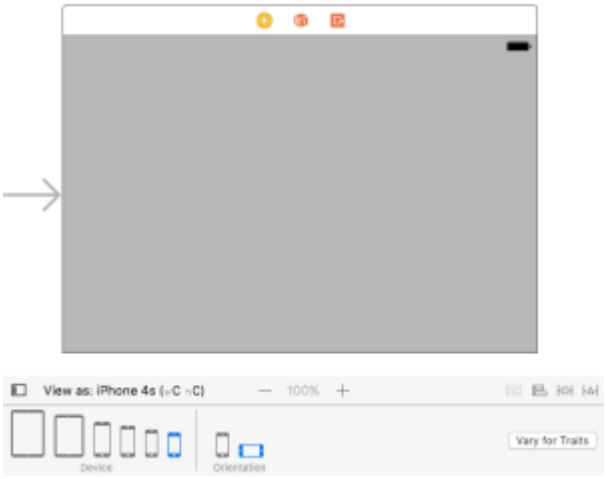


Fig: Layout

All of this is in the **Any** size. To change the size for devices, you can click the **Vary for Traits** button.

A Little Auto Layout

So far you've previewed a blank storyboard. Click the icon for a portrait phone. Add two buttons labeled **Button 1** on the top of the scene and **Button 2** towards the bottom. In the attributes inspector, Set the **Text Color** to **White** and the background of the button to **Black**.

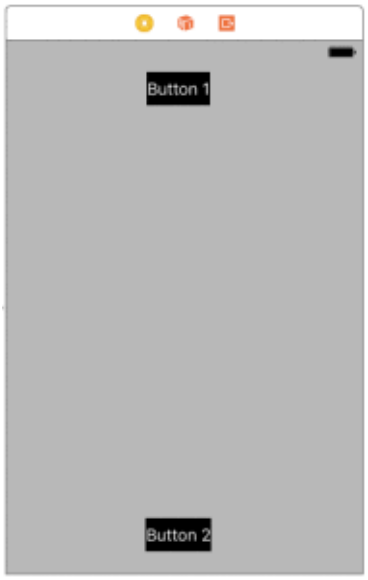


Fig: Layout

You'll need a little auto layout to take advantage of size classes. If you've never used auto layout, we'll keep it simple. Select **Button 1.**Find the pin button in the Auto Layout menu on the lower right side of Interface builder. Click to get a popup.

Fig: Layout

In the highlighted box pin to the view above by typing **10** and hitting **tab** on your keyboard. The I-Beam below turns black, and the cursor moved to the left pin. Type **10** and **tab** again to pin the button to the right margin 10 points. Type **10** and **tab** one more time to pin the right side to the right margin by 10 points. Be sure to type **tab** to make sure the I-beam shows. Toward the bottom of the popup, You'll find the **Update Frames** button. Change the value from **None** to **Items of new constraints**. Your popup should look like this:

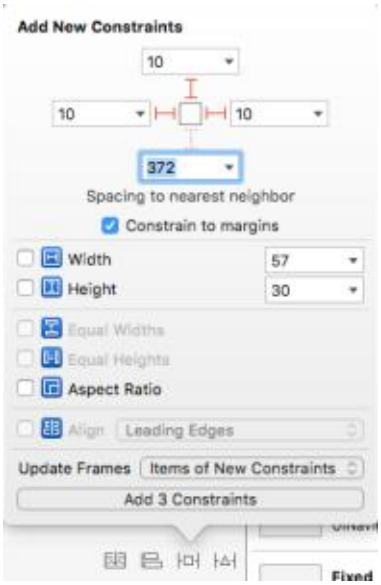


Fig: Layout

Click **Add 3 constraints** and the constraints will appear on the storyboard, stretching the button to the correct size and position.

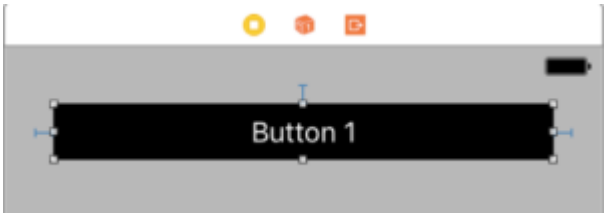


Fig: Layout

Select **Button 2** on the bottom. You'll pin this to the bottom. Click to get a popup. Tab past the first box without entering anything. try **10** then **tab** for the left pin, **10** and **tab** for the right pin, and **10** and **tab**for the bottom pin. Again set **Update frames** to **Items of new Constraints**. The popup should look like this.

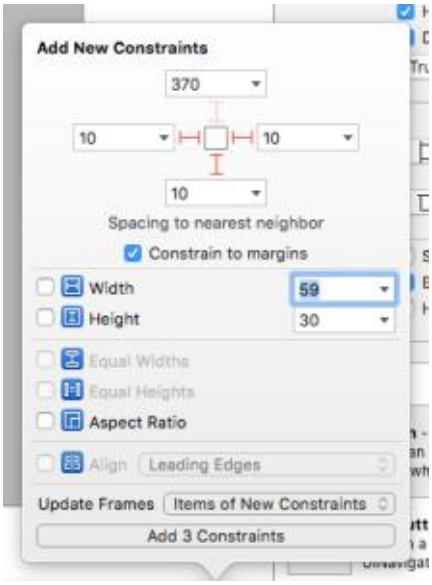


Fig: Layout

Add the three constraints. Your storyboard should look like this:

Fig: Layout

Click the portrait orientation icon  and the view resizes the buttons:

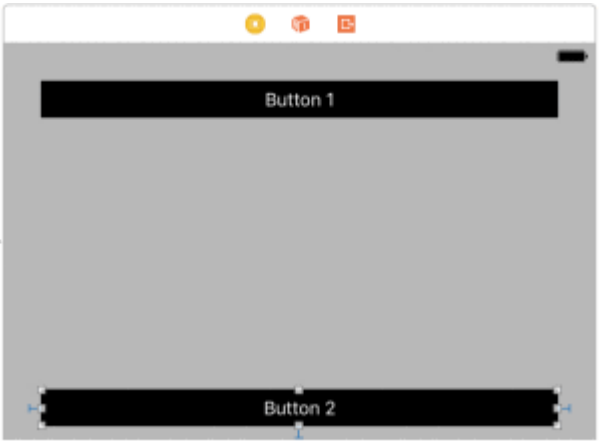


Fig: Layout

Click the iPad Pro 9.7″ icon. Close the navigation and attributes inspector panels to give yourself room. Change the scale to **75%.** You'll see this:

Fig: Layout

Varying a Trait

The horizontal and vertical dimensions we refer to as *traits*. You can change one or both traits of the displayed device. The current traits are found after the name of the device in the view as button. The current view show us a width of regular(**wR**) and height of regular(**hR**). The default iPhone6s in portrait is (**wC hR**) for compact width, Regular height.

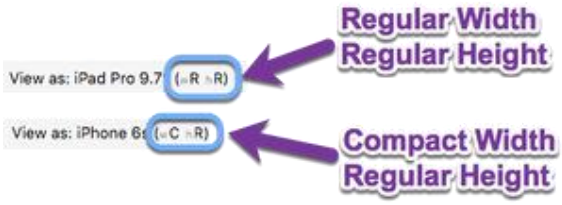


Fig: Layout

Select an **iPhone 6s Plus(wR hC)** in landscape in the toolbar. All phones in landscape are compact width and compact height except iPhone Plus. The iPhone plus models are regular width not compact. On regular width devices, usually users will hold with both hands on the sides, using their thumbs across the device for most button presses. On compact width devices, users typically will hold from the bottom for compact width devices, using the thumb up and down. You'll now change all devices with a regular width to place the buttons on the sides, which is the easier to use place for thumbs to contact them.

Click the **Vary for Traits** button on the right side of the size class toolbar. A popup appears.

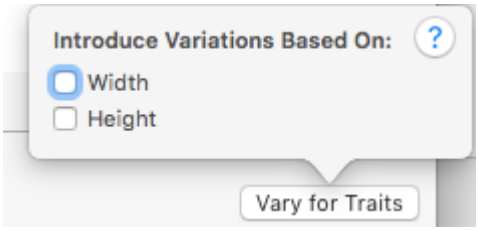


Fig: Layout

Check the **Width** checkbox. The toolbar changes color and displays all size classes affected.

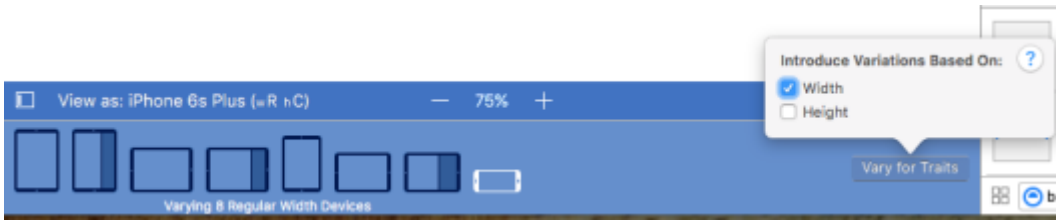


Fig: Layout

Click in the newly colored area to close the popup. You are now in the mode that changes only the devices with a regular class width. Open the attributes inspector, and change to the ruler for the size inspector.

Fig: Layout

Select the **Button 2**. Scroll down the size inspector until you find this constraint.

Fig: Layout

Select it and press **delete** on your keyboard. Select **Button 1**. Find this constraint and delete it:

Fig: Layout

The storyboard looks like this:

Fig: Layout

Select **Button 1**. Click on the pin constraint button . Select the bottom constraint and type **10** , then hit **tab**. Select **Items of new constraints**. The popup should look like this:

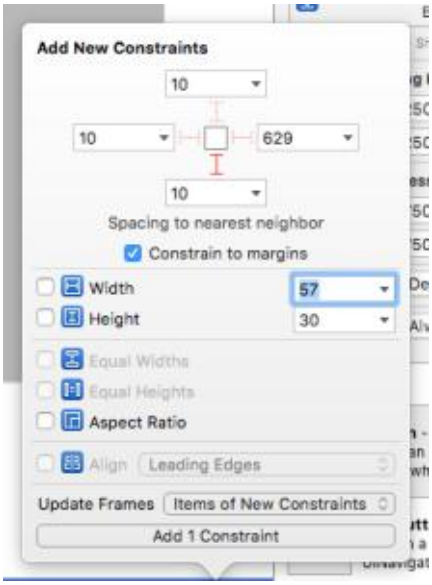


Fig: Layout

Add the constraint. Select the **Button 2.** Type **10** , then hit **tab** for the top constraint. Select Items of new constraints like this:

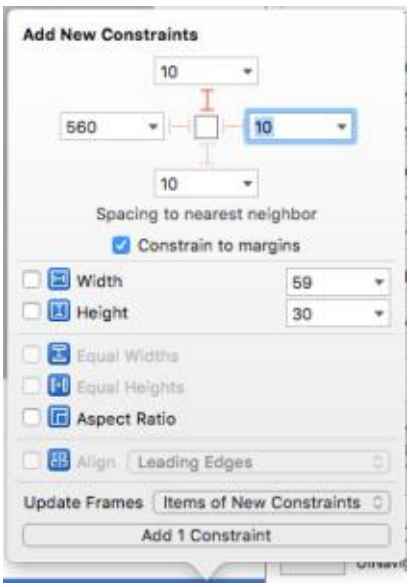


Fig: Layout

Add the new constraint. The layout now looks like this:

Fig: Layout

Press **Done varying**. The highlight color disappears. Change to **Portrait** and the buttons are on the top and bottom.

Fig: Layout

Go to an iPad Pro 9.7″ in portrait. At 75% scale it looks like this:

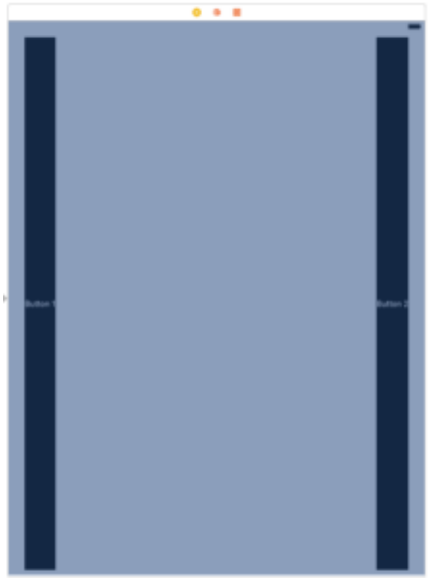


Fig: Layout

Under **Adaptations** in the size class toolbar, click the middle button. This is a multitasking view, which is compact width. The buttons are on the top and bottom again.

Fig: Layout

Class Size Variations on Attributes

You can also change attributes based on the size classes. Click the gray view. Go to the attributes inspector and click the **+** next to the background attribute.

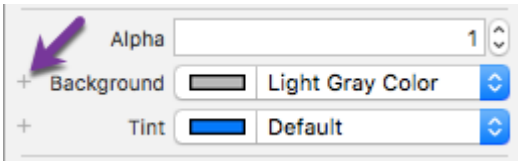


Fig: Layout

A popup appears:

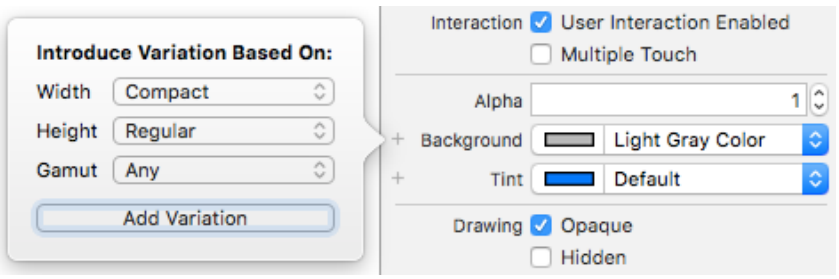


Fig: Layout

The current device's traits appear in the box. Change **Width** to **Regular** and **Height** to **Any** to make this an attribute of all regular width devices.

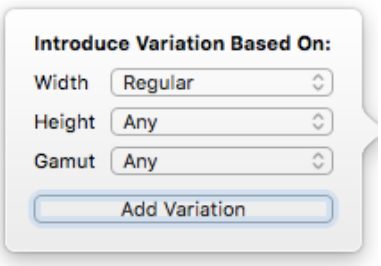


Fig: Layout

Click **Add Variation**. A new background attribute for regular width appears under the default one.

Fig: Layout

Change the wR background color to another color. I used Orange(#FF8000). Nothing changes on the storyboard. You are still on a compact width. Click the  icon in **Adaptation** to go back to regular size. The background is Orange.

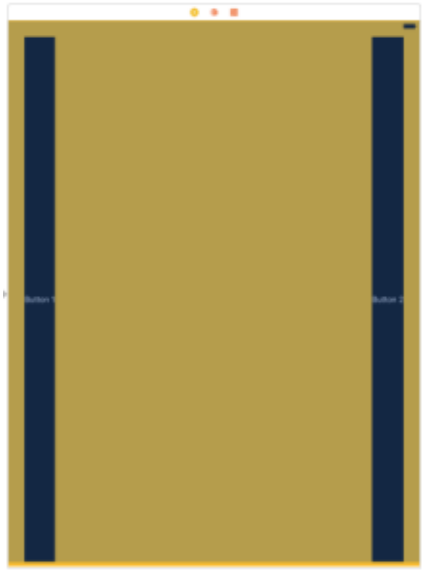


Fig: Layout

You can browse though the other devices to see which ones have orange backgrounds and which have gray ones. As you can see, Size Classes in Xcode 8 are quite powerful ways of laying out your project easily, while customizing the layout for different devices.

Suggested reading list:

iOS Programming: The Big Nerd Ranch Guide

iOS App Development For Dummies

Swift Programming: The Big Nerd Ranch Guide

Reference and bibliography:

https://makeapppie.com/2015/01/08/basic-auto-layout-a-practical-view/

https://www.raywenderlich.com/160527/auto-layout-tutorial-ios-11-getting-started

https://www.journaldev.com/10806/ios-autolayout-xcode-constraints-tutorial

https://www.raywenderlich.com/160646/uistackview-tutorial-introducing-stack-views-2

https://www.appcoda.com/learnswift/auto-layout-intro.html

Module 5: ARC and Object Initialization

Topic: Objective-C

Chapter Overview:

Instead of you having to remember when to you retain, release, and auto release, ARC evaluates the lifetime requirements of your objects and automatically insets appropriate memory management calls for you at compile time. The compiler also generates appropriate dealloc methods for you. In general, if you are only using ARC the traditional Cocoa naming conversions are important only if you need to interoperate with code the uses manual reference counting.

In this chapter we will discuss about Automatic Reference Counting (ARC) and Object Initialization

Learning Outcome:

- Automatic Reference Counting (ARC)
- Object Initialization

Automatic Reference Counting (ARC)

Automatic Reference Counting (ARC) is a compiler feature that provides automatic memory management of Objective-C objects. Rather than having to think about retain and release operations, ARC allows you to concentrate on the interesting code, the object graphs, and the relationships between objects in your application.

In Automatic Reference Counting or ARC, the system uses the same reference counting system as MRR, but it inserts the appropriate memory management method calls for us at compile-time. We are strongly encouraged to use ARC for new projects. If we use ARC, there is typically no need to understand the underlying implementation described in this document, although it may in some situations be helpful. For more about ARC, see Transitioning to ARC Release Notes.

As mentioned above, in ARC, we need not add release and retain methods since that will be taken care by the compiler. Actually, the underlying process of Objective-C is still the same. It uses the retain and release operations internally making it easier for the developer to code without worrying about these operations, which will reduce both the amount of code written and the possibility of memory leaks.

There was another principle called garbage collection, which is used in Mac OS-X along with MRR, but since its deprecation in OS-X Mountain Lion, it has not been discussed along with MRR. Also, iOS objects never had garbage collection feature. And with ARC, there is no use of garbage collection in OS-X too.

Summary

ARC works by adding code at compile time to ensure that objects live as long as necessary, but no longer. Conceptually, it follows the name memory management conventions as manual reference counting (described in *Advanced Memory Management Progrmming Guide*) by adding the appropriate memory management calls for you.

Object Initialization

Initialization sets the instance variables of an object to reasonable and useful initial values. It can also allocate and prepare other global resources needed by the object, loading them if necessary from an external source such as a file. Every object that declares instance variables should implement an initializing method—unless the default set-everything-to-zero initialization is sufficient. If an object does not implement an initializer, Cocoa invokes the initializer of the nearest ancestor instead.

init()

Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated.

Declaration

```
init()
```

Return Value

An initialized object, or nil if an object could not be created for some reason that would not result in an exception.

Discussion

An init() message is coupled with an alloc (or allocWithZone:) message in the same line of code:

```
SomeClass *object = [[SomeClass alloc] init];
```

An object isn't ready to be used until it has been initialized.

In a custom implementation of this method, you must invoke super's Initialization then initialize and return the new object. If the new object can't be initialized, the method should return nil. For example, a hypothetical BuiltInCamera class might return nil from its init method if run on a device that has no camera.

```
- (instancetype)init {
    if (self = [super init]) {
        // Initialize self
    }
    return self;
}
```

In some cases, a custom implementation of the init() method might return a substitute object. You must therefore always use the object returned by init(), and not the one returned by alloc or allocWithZone:, in subsequent code.

The init() method defined in the NSObject class does no initialization; it simply returns self. In terms of nullability, callers can assume that the NSObject implementation of init()does not return nil.

Reference and bibliography:

https://developer.apple.com/library/content/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html

https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html

https://developer.apple.com/library/content/documentation/General/Conceptual/CocoaEncyclopedia/Initialization/Initialization.html

https://developer.apple.com/documentation/objectivec/nsobject/1418641-init

Module 6: Storyboards

Topic: Storyboards

Chapter Overview:

A storyboard is a visual presentation of the user interface of an iOS application. Showing screens of content and the connections between those screens. A storyboard is composed of a sequence of scenes, each of which represents a view controller and its views; scenes are connection by segue objects. Which represent a transition between two view controllers.

In this chapter we will discuss about how to use storyboard by drag and drop using User Interface (UI) and Interface Builders (IB) Outlet.

Learning Outcome:

- Scene
- Segues
- Moving Data between Controllers
- Navigation Controller

StoryBoard

Storyboard in iOS is helps you to design the user interface of your App. Storyboards are an exciting feature first introduced in iOS 5 that save time building user interfaces for your apps. Storyboards allow you to prototype and design multiple view controller views within one file.

Before Storyboards you had to use XIB files and you could only use one XIB file per view (UITableViewCell, UITableView or other supported UIView types).

Scenes

View Controllers in storyboard are called Scenes A storyboard can contain one or more Scenes (View Controllers) and the connection or relationship between two scenes are known as Segue. This is how a typical storyboard with Scenes and Segue look

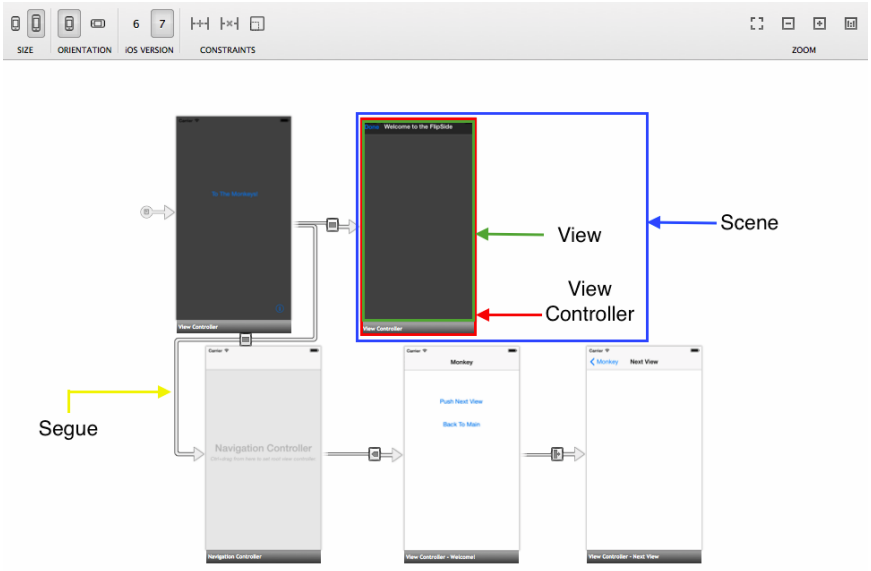


Fig: Scenes

Xcode provides a visual editor for storyboards, where you can layout and design the user interface of your application by adding views such as buttons, table views, and text views onto scenes. In addition, a storyboard enables you to connect a view to its controller object, and to manage the transfer of data between view controllers. Using storyboards in the recommended way to design the user interface of your application because they enable you to visualize the appearance and flow of your user interface on one canvas.

Segues

A segue defines a transition between two view controllers in your app's storyboard file. The starting point of a segue is the button, table row, or gesture recognizer that initiates the segue. The end point of a segue is the view controller you want to display. A segue always presents a new view controller, but you can also use an unwind segue to dismiss a view controller.

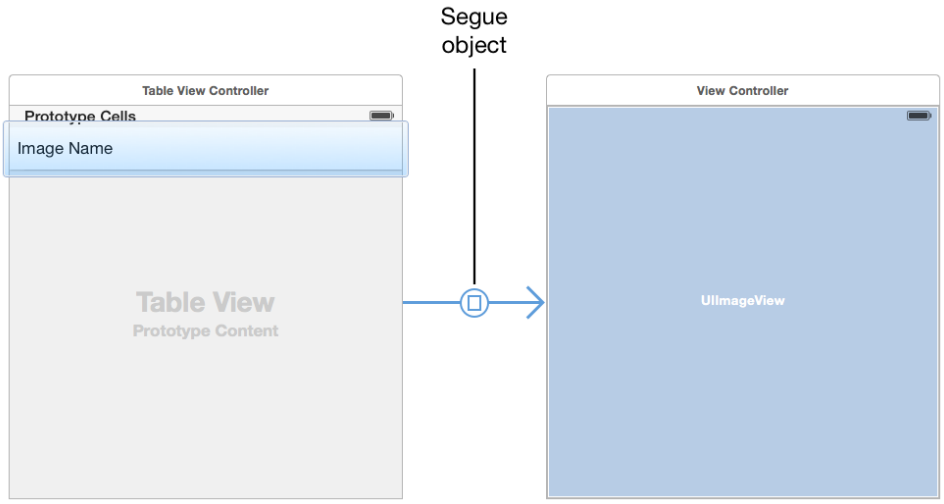


Fig: Segues

Creating a Segue Between View Controllers

Usually we use segue between two controllers for going current controller to next.

Creating an Unwind Segue

Unwind Segue use for going current controller two any other controllers (which is not necessary next to be next controller)

Types of Segues

A segue is a relationship set between two view controllers. This can be done by pressing Control then drag and drop from one View Controller to anlther. Listed below are the different types of segues

- Show
- Show Detail
- Present Modally
- PopOver Presentation
- Custom

Moving Data between Controllers

When you build an iOS app with more than one screen, you need to pass data between Your View Controllers in order for them to share contents without losing them along the way.

There are two directions in which a view controller can pass data to another view controller. For each one of these there are different techniques:

- **forward**, to the destination view controller of a transition
- **backwards**, to a view controller that was on the screen before and to which the user is will go back at some point.

Here are all the ways in which you can pass data between view controllers:

- when a segue is performed
- triggering transitions programmatically in your code
- through the state of the app
- using singletons
- using the app delegate
- assigning a delegate to a view controller
- through unwind segues
- referencing a view controller directly
- using the user defaults
- through notifications

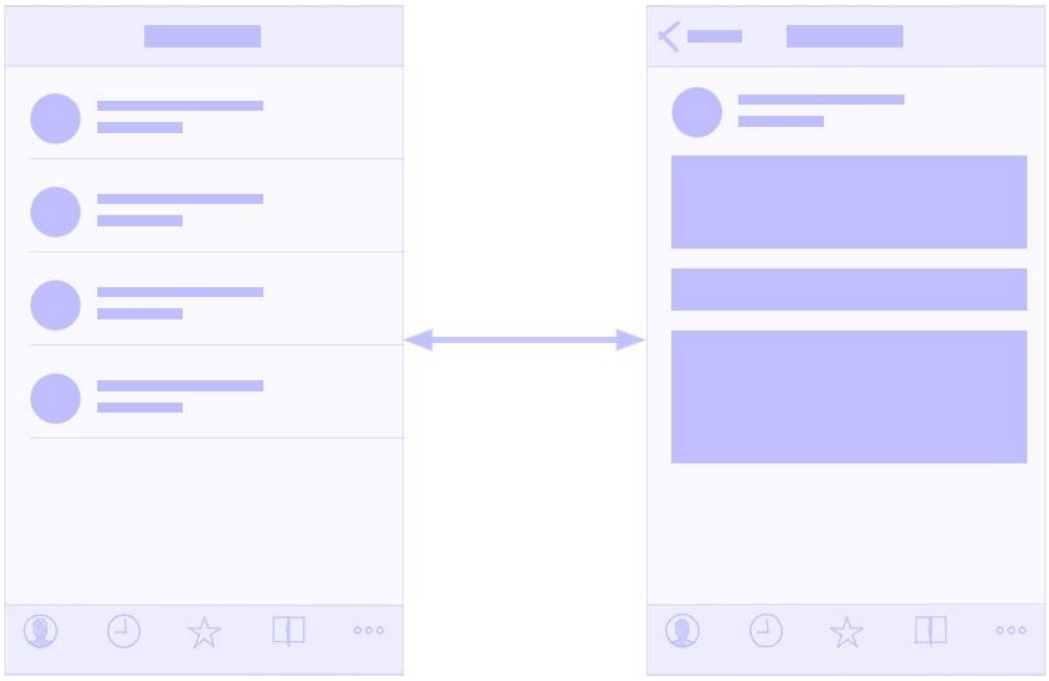


Fig: Controllers

Navigation Controller

A navigation controller manages a stack of view controllers to provide a drill-down interface for hierarchical content. The view hierarchy of a navigation controller is self-contained. It is composed of views that the navigation controller manages directly and views that are managed by content view controllers you provide. Each content view controller manages a distinct view hierarchy, and the navigation controller coordinates the navigation between these view hierarchies.

Although a navigation interface consists mostly of your custom content, there are still places where your code must interact directly with the navigation controller object. In addition to telling the navigation controller when to display a new view, you are responsible for configuring the navigation bar—the view at the top of the screen that provides context about the user's place in the navigation hierarchy. You can also provide items for a toolbar that is managed by the navigation controller.

This chapter describes how you configure and use navigation controllers in your app. For information about ways in which you can combine navigation controllers with other types of view controller objects.

Fig: Navigation

Reference and bibliography:

https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/UsingSegues.html
https://www.raywenderlich.com/160519/storyboards-tutorial-ios-10-getting-started-part-2
https://www.raywenderlich.com/160521/storyboards-tutorial-ios-11-part-1
https://matteomanferdini.com/how-ios-view-controllers-communicate-with-each-other/
https://code.tutsplus.com/tutorials/ios-sdk-passing-data-between-controllers-in-swift--cms-27151
https://developer.apple.com/library/content/documentation/WindowsViews/Conceptual/ViewControllerCatalog/Chapters/NavigationControllers.html
https://www.appcoda.com/use-storyboards-to-build-navigation-controller-and-table-view/

Module 7: Table Views

Topic:

Introduction
Data Source and Delegate
Simple app using the module taught e.g.: To Do List

Chapter Overview:

A table view is made up of zero or more sections, each with its own rows. Sections are identified by their index number within the table view, and rows are identified by their index number within a section. Any section can optionally be preceded by a section header, and optionally be followed by a section footer.

Table views can have one of two styles, UITableViewStylePlain and UITableViewStyleGrouped. When you create a UITableView instance you must specify a table style, and this style cannot be changed. In the plain style, section headers and footers float above the content if the part of a complete section is visible. A table view can have an index that appears as a bar on the right hand side of the table (for example, "A" through "Z"). You can touch a particular label to jump to the target section. The grouped style of table view provides a default background color and a default background view for all cells. The background view provides a visual grouping for all cells in a particular section. For example, one group could be a person's name and title, another group for phone numbers that the person uses, and another group for email accounts and so on. See the Settings application for examples of grouped tables. Table views in the grouped style cannot have an index.

Learning Outcome:

- Introduction
- Data Source and Delegate
- Simple app using the module taught e.g.: To Do List

Brief Overview:

Many methods of UITableView take NSIndexPath objects as parameters and return values. UITableView declares a category on NSIndexPath that enables you to get the represented row index (row property) and section index (section property), and to construct an index path from a given row index and section index (indexPathForRow:inSection: method). Especially in table views with multiple sections, you must evaluate the section index before identifying a row by its index number.

A UITableView object must have an object that acts as a data source and an object that acts as a delegate; typically these objects are either the application delegate or, more frequently, a custom UITableViewController object. The data source must adopt the UITableViewDataSource protocol and the delegate must adopt the UITableViewDelegate protocol. The data source provides information that UITableView needs to construct tables and manages the data model when rows of a table are inserted, deleted, or reordered. The delegate manages table row configuration and selection, row reordering, highlighting, accessory views, and editing operations.

When sent a setEditing:animated: message (with a first parameter of YES), the table view enters into editing mode where it shows the editing or reordering controls of each visible row, depending on the editingStyle of each associated UITableViewCell. Clicking on the insertion or deletion control causes the data source to receive tableView:commitEditingStyle:forRowAtIndexPath: message. You commit a deletion or insertion by calling deleteRowsAtIndexPaths:withRowAnimation: or insertRowsAtIndexPaths:withRowAnimation:, as appropriate. Also in editing mode, if a table-view cell has its showsReorderControl property

set to YES, the data source receives a tableView:moveRowAtIndexPath:toIndexPath: message. The data source can selectively remove the reordering control for cells by implementing tableView:canMoveRowAtIndexPath: UITableView caches table-view cells for visible rows. You can create custom UITableViewCell objects with content or behavioral characteristics that are different than the default cells; A Closer Look at Table View Cells explains how.

UITableView overrides the layoutSubviews method of UIView so that it calls reloadData only when you create a new instance of UITableView or when you assign a new data source. Reloading the table view clears current state, including the current selection. However, if you explicitly call reloadData, it clears this state and any subsequent direct or indirect call to layoutSubviews does not trigger a reload.

Introduction

Table views are versatile user interface objects frequently found in iOS apps. A table view presents data in a scrollable list of multiple rows that may be divided into sections.

Table View is one of the common UI elements in iOS apps. Most apps, in some ways, make use of Table View to display list of data. The best example is the built-in Phone app. Your contacts are displayed in a Table View. Another example is the Mail app. It uses Table View to display your mail boxes and emails. Not only designed for showing textual data, Table View allows you to present the data in the form of images. The built-in Video and YouTube app are great examples for the usage.

Table views have many purposes:

- To let users navigate through hierarchically structured data
- To present an indexed list of items
- To display detail information and controls in visually distinct groupings
- To present a selectable list of options

Fig: List view in an iPhone

Data Source and Delegate

Delegate

Delegation design pattern is a way of modifying complex objects without sub classing them. Instead of sub classing, we use the complex object as it is and put any custom code for modifying the behaviour of that object inside a separate object, which is referred to as the Delegate Object.

At predefined times, the complex object then calls the methods of the delegate object to give it a chance to run its custom code.

Delegation is a simple and powerful pattern in which one object in a program acts on behalf of, or in coordination with, another object. The delegating object keeps a reference to the other object to the delegate and at the appropriate time sends a message to it. The message informs the delegate of an event that the delegating object is about to handle or has just handled. The delegate may respond to the message by updating the appearance or state of itself or other objects in the application, and in some cases it can return a value that affects how an impending event is handled. The main value of delegation is that it allows you to easily customize the behavior of several objects in one central object.

DataSource

A data source is almost identical to a delegate. The difference is in the relationship with the delegating object. Instead of being delegated control of the user interface, a data source is delegated control of data. The delegating object, typically a view object such as a table view, holds a reference to its data source and occasionally asks it for the data it should display. A data source, like a delegate, must adopt a protocol and implement at minimum the required methods of that protocol. Data sources are responsible for managing the memory of the model objects they give to the delegating view.

Simple app using the module taught e.g.: To Do List

Make a To Do List App Using Table View and other component which is already covered.

Suggested reading list:

https://developer.apple.com/documentation/uikit/uitableview?language=objc

https://www.techotopia.com/index.php/IOS_App_Development_Essentials

Reference and bibliography:

https://developer.apple.com/documentation/uikit/uitableview?language=objc

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/TableView_iPhone/TableViewCells/TableViewCells.html

http://www.thomashanning.com/uitableview-tutorial-for-beginners/

https://developer.apple.com/library/content/documentation/General/Conceptual/CocoaEncyclopedia/DelegatesandDataSources/DelegatesandDataSources.html

Module 8: Protocols, Categories and Blocks

Topic:

Protocols
Categories
Blocks

Chapter Overview:

Objective-C allows you to define protocols, which declare the methods expected to be used for a particular situation. Protocols are implemented in the classes conforming to the protocol.

A simple example would be a network URL handling class, it will have a protocol with methods like process Completed delegate method that intimates the calling class once the network URL fetching operation is over

Learning Outcome:

- Protocols
- Categories
- Blocks

Protocols

In Objective-C, a particular class only has one parent, and its parent has one parent, and so on right up to the root object (NSObject). But what if your class needs to call methods on objects outside of its parent tree? A **protocol** is one way Objective-C solves this problem.

A protocol is a list of method declarations. If your class adopts the protocol, then you have to implement those methods in your class.

In Objective-C 2.0 and later, some protocol methods can be marked as optional. This means you don't have to implement those, but you still have to implement all of the required methods. When you do, your class is said to conform to the protocol.

Protocols are used quite a bit in iPhone development. For instance, a UITableView requires a data source and a delegate object; these must conform to the UITableViewDataSource and UITableViewDelegate protocols.

To adopt a protocol, add it to your class header file:

```
@interface FavoritesViewController : UIViewController <UITableViewDelegate, UITableViewDataSource>
```

The protocol names appear after the class declaration, inside angled brackets. When adopting more than one protocol, list them in a comma-separated list.

Then in your implementation (.m) file, implement all of the required methods for each protocol. (For Cocoa classes, consult the documentation to see which methods are required and which are optional.)

In the real world, people on official business are often required to follow strict procedures when dealing with certain situations. Law enforcement officials, for example, are required to "follow protocol" when making enquiries or collecting evidence.

In the world of object-oriented programming, it's important to be able to define a set of behavior that is expected of an object in a given situation. As an example, a table view expects to be able to communicate with a data source object in order to find out what it is required to display. This means that the data source must respond to a specific set of messages that the table view might send.

The data source could be an instance of any class, such as a view controller (a subclass of NSViewController on OS X or UIViewController on iOS) or a dedicated data source class that perhaps just inherits from NSObject. In order for the table view to know whether an object is suitable as a data source, it's important to be able to declare that the object implements the necessary methods.

Objective-C allows you to define *protocols*, which declare the methods expected to be used for a particular situation.

A simple example would be a network URL handling class, it will have a protocol with methods like processCompleted delegate method that intimates the calling class once the network URL fetching operation is over.

```
@protocol ProtocolName
@required
// list of required methods
@optional
// list of optional methods
@end
```

The methods under keyword @required must be implemented in the classes that conforms to the protocol and the methods under @optional keyword are optional to implement.

Here is the syntax for class conforming to protocol

A syntax of protocol is shown below.

```
@interface MyClass : NSObject <MyProtocol>
...
@end
```

This means that any instance of MyClass will respond not only to the methods declared specifically in the interface, but that MyClass also provides implementations for the required methods in MyProtocol. There's no need to redeclare the protocol methods in the class interface - the adoption of the protocol is sufficient.

If you need a class to adopt multiple protocols, you can specify them as a comma-separated list. We have a delegate object that holds the reference of the calling object that implements the protocol.

Categories

Sometimes, you may find that you wish to extend an existing class by adding behavior that is useful only in certain situations. In order add such extension to existing classes, Objective-C provides categories and extensions.

If you need to add a method to an existing class, perhaps, to add functionality to make it easier to do something in your own application, the easiest way is to use a category.

The syntax to declare a category uses the @interface keyword, just like a standard Objective-C class description, but does not indicate any inheritance from a subclass. Instead, it specifies the name of the category in parentheses, like this:

```
@interface ClassName (CategoryName)
@end
```

Characteristics of category

A category can be declared for any class, even if you don't have the original implementation source code.

Any methods that you declare in a category will be available to all instances of the original class, as well as any subclasses of the original class.

At runtime, there's no difference between a method added by a category and one that is implemented by the original class.

Now, let's look at a sample category implementation. Let's add a category to the Cocoa class NSString. This category will make it possible for us to add a new method getCopyRightString which helps us in returning the copyright string. It is shown below.

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

@interface NSString(MyAdditions)

+(NSString *)getCopyRightString;

@end

@implementation NSString(MyAdditions)

+(NSString *)getCopyRightString{
    return @"Copyright TutorialsPoint.com 2013";
}

@end


int main(int argc, const char * argv[])
{

   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString *copyrightString = [NSString getCopyRightString];
   NSLog(@"Accessing Category: %@",copyrightString);
   [pool drain];
    return 0;
}
```

The output will be Accessing Category: Copyright TutorialsPoint.com 2013

Even though any methods added by a category are available to all instances of the class and its subclasses, you'll need to import the category header file in any source code file where you wish to use the additional methods, otherwise you'll run into compiler warnings and errors.

In our example, since we just have a single class, we have not included any header files, in such a case we should include the header files as said above.

Blocks

An Objective-C class defines an object that combines data with related behavior. Sometimes, it makes sense just to represent a single task or unit of behavior, rather than a collection of methods.

Blocks are a language-level feature added to C, Objective-C and C++, which allow you to create distinct segments of code that can be passed around to methods or functions as if they

were values. Blocks are Objective-C objects, which means they can be added to collections like NSArray or NSDictionary. They also have the ability to capture values from the enclosing scope, making them similar to *closures* or *lambdas* in other programming languages.

Simple Block declaration syntax

```
returntype (^blockName)(argumentType);
```

Simple block implementation

```
returntype (^blockName)(argumentType)= ^{
};
```

Here is a simple example

```
void (^simpleBlock)(void) = ^{
    NSLog(@"This is a block");
};
```

We can invoke the block using

```
simpleBlock();
```

Blocks Take Arguments and Return Values

Blocks can also take arguments and return values just like methods and functions.

Here is a simple example to implement and invoke a block with arguments and return values.

```
double (^multiplyTwoValues)(double, double) =
    ^(double firstValue, double secondValue) {
                return firstValue * secondValue;
    };
double result = multiplyTwoValues(2,4);
NSLog(@"The result is %f", result);
```

Reference and bibliography:

https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithProtocols/WorkingwithProtocols.html

http://www.idev101.com/code/Objective-C/protocols.html

https://www.tutorialspoint.com/objective_c/objective_c_protocols.htm

https://www.tutorialspoint.com/objective_c/objective_c_blocks.htm

https://code.tutsplus.com/tutorials/objective-c-categories--mobile-10648

https://code.tutsplus.com/tutorials/objective-c-succinctly-categories-and-extensions--mobile-22016

Module 09: Networking

Topic: Networking

Chapter Overview:

The majority of apps require networking to connect to external services and data sources. 'Networking' means exchanging information via 'http' (Hypertext Transfer Protocol), one of the most used protocols. Every time you open your browser and retrieve or send data, you are using HTTP as the protocol. In this article you will learn how to work with networking in iOS by looking at the following options:

Learning Outcome:

- URL Loading System
- NSURLSession
- Asynchronous Downloads
- Strings and Images
- JSON
- XML
- HTTP POST Requests
- Simple app using the module taught e.g.: Weather Forecasting

Brief Discussion:

AFNetworking is the most popular networking library for iOS. Chances are high that it's the first pod you add to your Podfile. It's used as a standalone network layer and as a part of some other frameworks, like RestKit. For me it has earned it's popularity for few reasons. It's well maintained, what is very important for open source project (thought it still has long living issues). And it has well thought architecture and interface, so it is easy to use and extend to your needs.

When we perform a request using AFNetworking we can receive serialized JSON object, either dictionary or array. And we can work with it right away. But we can do better. It's much better to work not with dictionaries and arrays but our own business objects. RestKit does this job but I find it's interface quiet complex (besides that it still uses 1.x version of AFNetworking and adds it's own bugs) and never used and never will use it in my projects. So let's see how we can improve our networking code ourselves with very little effort and without using AFNetworking at all. You can download full project on GitHub.

When we make a request it has some well known signature, like it's method, set of parameters and path. As a response to the request we expect some defined type of data. Let's create helper classes which will encapsulate requests and responses.

Here is our request:

```
typedef NS_ENUM(NSUInteger, HTTPMethod){
    GET, POST, PUT, DELETE, HEAD
};


@protocol APIResponse;
```

```
@protocol APIRequest <NSObject>

- (HTTPMethod)method;
- (NSURL *)baseURL;
- (NSString *)path;
- (NSDictionary *)parameters;
- (NSDictionary *)headers;
- (Class<APIResponse>)responseClass;

@end

@protocol APIResponse <NSObject>

- (NSURLSessionDataTask *)task;
- (NSURLResponse *)response;
- (NSError *)error;
- (id)responseObject;
- (id)processedResponseObject;

- (instancetype)initWithTask:(NSURLSessionDataTask *)task
                    response:(NSHTTPURLResponse *)response
              responseObject:(id)responseObject
                       error:(NSError *)error;

- (id)processResponseObject:(NSError **)error;

@end
```

We can add now some basic implementation of these protocols, i.e. to represent request for data in JSON format:

```
@interface SimpleAPIRequest : NSObject <APIRequest>

@end

@interface JSONAPIRequest : SimpleAPIRequest
```

```
@end

@interface SimpleAPIRequest()

@property (nonatomic) HTTPMethod method;
@property (nonatomic, copy) NSString *path;
@property (nonatomic, copy) NSDictionary *parameters;
@property (nonatomic, copy) NSDictionary *headers;
@property (nonatomic) Class<APIResponse> responseClass;

@end

@implementation SimpleAPIRequest

- (instancetype)init
{
    self = [super init];
    if (self) {
        self.responseClass = [SimpleAPIResponse class];
    }
    return self;
}

@end

@implementation JSONAPIRequest

- (instancetype)init
{
    self = [super init];
    if (self) {
        self.responseClass = [JSONAPIResponse class];
        self.headers = @{@"Accept": @"application/json", @"Content-type": @"application/json"};
```

```
    }
    return self;
}

@end
@interface SimpleAPIResponse: NSObject <APIResponse>

@end

@interface JSONAPIResponse : SimpleAPIResponse

@end

@interface SimpleAPIResponse ()

@property (nonatomic, copy) NSURLSessionDataTask *task;
@property (nonatomic, copy) NSHTTPURLResponse *response;
@property (nonatomic, copy) NSError *error;
@property (nonatomic, strong) id responseObject;
@property (nonatomic, strong) id processedResponseObject;

@end

@implementation SimpleAPIResponse

- (instancetype)initWithTask:(NSURLSessionDataTask *)task response:(NSHTTPURLResponse *)response responseObject:(id)responseObject error:(NSError *)error;
{
    self = [super init];
    if (self) {
        self.task = task;
        self.response = response;
        self.error = error;
        self.responseObject = responseObject;
```

```
        if (!error) {
            NSError *serializationError;
            self.processedResponseObject = [self processResponseObject:&serializationError];
            if (serializationError) {
                self.error = serializationError;
            }
        }
    }
    return self;
}

- (id)processResponseObject:(NSError *__autoreleasing *)error
{
    return self.responseObject;
}

@end

@implementation JSONAPIResponse

- (id)processResponseObject:(NSError *__autoreleasing *)error
{
    if ([self.responseObject isKindOfClass:[NSData class]]) {
        NSError *serializationError;
        id processedResponseObject = [NSJSONSerialization
JSONObjectWithData:self.responseObject options:0 error:&serializationError];
        if (error) *error = serializationError;
        return processedResponseObject;
    }
    else {
        return nil;
    }
}

@end
```

To make requests we need some object. It will make request using `NSURLSessionTask`. Let's define it's protocol.

```
typedef void(^APIClientCompletionBlock)(id<APIResponse> response);

@protocol APIClient <NSObject>

- (NSURLSessionDataTask *)dataTaskWithAPIRequest:(id<APIRequest>)request
                                      completion:(APIClientCompletionBlock)completion;

@end
```

Foundation already has class that can create `NSURLSessionTask` - `NSURLSession`. So let's extend it and implement `APIClient` protocol in it's category:

```
@interface NSURLSession(APIClient) <APIClient>

@end

@implementation NSURLSession(APIClient)

- (NSURLSessionDataTask *)dataTaskWithAPIRequest:(id<APIRequest>)request
                                      completion:(APIClientCompletionBlock)completion;
{
    NSURL *requestUrl = [NSURL urlWithString:request.path baseURL:request.baseURL
parameters:request.parameters];
    NSURLRequest *httpRequest = [NSURLRequest requestWithMethod:request.method
url:requestUrl headers:request.headers];
    __block NSURLSessionDataTask *task;
    task = [self dataTaskWithRequest:httpRequest completionHandler:^(NSData *data,
NSURLResponse *response, NSError *error) {

        Class responseClass = [request responseClass];
        id<APIResponse> apiResponse = [[responseClass alloc] initWithTask:task
response:(NSHTTPURLResponse *)response responseObject:data error:error];

        dispatch_async(dispatch_get_main_queue(), ^{
            if (completion) { completion(apiResponse); }
```

```
        });
    }];

    [task resume];
    return task;
}

@end
```

This implementation is very generic. In callbacks we can receive instance of `APIResponse` protocol or some specific class that request instance returns from `+responseClass` method. By methods that provide specific type of response we can give clients of our code some type safety.

Now lets look how we can use that. Lets say we have some API that returns GitHub users. Lets define users request and response:

```
@interface GitHubJSONRequest : JSONAPIRequest

@end

@implementation GitHubJSONRequest

- (NSURL *)baseURL
{
    return [NSURL URLWithString:@"https://api.github.com"];
}

@end


@interface UsersRequest : GitHubJSONAPIRequest

@end

@implementation UsersRequest

- (HTTPMethod)method
```

```
{
    return GET;
}

- (NSString *)path
{
    return @"users";
}

- (Class)responseClass
{
    return [UsersResponse class];
}

@end


@interface UsersResponse : JSONAPIResponse

@property (nonatomic, strong, readonly) NSArray *users;

@end

@implementation UsersResponse

- (BOOL)processResponseObject:(NSError **)error;
{
    NSError *__error;
    id processedResponseObject = [super processResponseObject:&__error];
    if (__error || ![processedResponseObject isKindOfClass:[NSArray class]]) {
        if (error) *error = __error;
        return nil;
    }
    else {
        return [User withArray:processedResponseObject];
```

```
    }
}

- (NSArray *)users
{
    return self.processedResponseObject;
}

@end
```

Defining shorthand methods to access processed response objects (like `- (NSArray *)users`) will give our clients a straight way to access data they need and provide information about type of this data so they will not need to guess the type and cast it.

What about api client? We don't need to subclass it, we can use it's category to add behavior that we need:

```
typedef void(^UsersResponseBlock)(UsersResponse *response);

@protocol GitHubClient <APIClient>

- (NSURLSessionDataTask *)getUsers:(UsersResponseBlock)completion;

@end

@interface APIClient (GitHub) <GitHubClient>

@end

@implementation APIClient (GitHub)

- (NSURLSessionDataTask *)getUsers:(UsersResponseBlock)completion;
{
    UsersRequest *request = [[UsersRequest alloc] init];
    NSURLSessionDataTask *task = [self dataTaskWithAPIRequest:request completion:completion];
    [task resume];
    return task;
```

```
}


@end
```

First we create a request. Then we call the method of `APIClient` that actually perform request.
Adding `-getUsers:...` method will make a client of our code to be sure about what kind of response it will get - without any typecasting at all.

And that's all.

Conclusion

Let's look what we have achieved using this approach:

1. All of our requests and responses are encapsulated in small classes that are easy to read and test. When our api changes we will change only request or response class and will not need to change our api client or any other object.
2. Mapping to business objects is made at the moment when response object is created and it's done in generic way. All we need is to override template method in response class. It is also easy to test.
3. Type safety. Clients of our code should explicitly define relationships between requests and responses. Our code then guarantees that it will provide client with the right objects. Of course it's not real type safety comparing with Swift but at least we will have proper code completion and will get rid of typecasts.
4. We didn't use AFNetworking at all. So you can see how easy it is to manage networking yourself. AFNetworking of course provides much more functionality, but using described approach you can extend not just `NSURLSession`, but also `AFHTTPSessionManager` and make it more convenient to use.

URL Loading System

The URL loading system is a set of classes and protocols that allow your app to access content referenced by a URL. At the heart of this technology is the `NSURL` class, which lets your app manipulate URLs and the resources they refer to.

To support that class, the Foundation framework provides a rich collection of classes that let you load the contents of a URL, upload data to servers, manage cookie storage, control response caching, handle credential storage and authentication in app-specific ways, and write custom protocol extensions.

The URL loading system provides support for accessing resources using the following protocols:

- File Transfer Protocol (ftp://)
- Hypertext Transfer Protocol (http://)
- Hypertext Transfer Protocol with encryption (https://)
- Local file URLs (file:///)
- Data URLs (data://)

NSURLSession

The NSURLSession class and related classes provide an API for downloading content. This API provides a rich set of delegate methods for supporting authentication and gives your app the ability to perform background downloads when your app isn't running or, in iOS, while your app is suspended.

The NSURLSession class natively supports the data, file, ftp, http, and https URL schemes, with transparent support for proxy servers and SOCKS gateways, as configured in the user's system preferences.

Tasks

The basic unit of work when working with NSURLSession is the task, an instance of NSURLSessionTask. There are three types of tasks, data tasks, upload tasks, and download tasks.

- You'll most often use data tasks, which are instances of NSURLSessionDataTask. Data tasks are used for requesting data from a server, such as JSON data. The principal difference with upload and download tasks is that they return data directly to your application instead of going through the file system. The data is only stored in memory.
- As the name implies, upload tasks are used to upload data to a remote destination. The NSURLSessionUploadTask is a subclass of NSURLSessionDataTask and behaves in a similar fashion. One of the key differences with a regular data task is that upload tasks can be used in a session created with a background session configuration.
- Download tasks, instances of NSURLSessionDownloadTask, inherit directly from NSURLSessionTask. The most significant difference with data tasks is that a download task writes its response directly to a temporary file. This is quite different from a regular data task that stores the response in memory. It is possible to cancel a download task and resume it at a later point.

As you can imagine, asynchronicity is a key concept in NSURLSession. The NSURLSession API returns data by invoking a completion handler or through the session's delegate.

Asynchronous Downloads

When you execute something synchronously, you wait for it to finish before moving on to another task. When you execute something asynchronously, you can move on to another task before it finishes.

That being said, in the context of computers this translates into executing a process or task on another "thread." A thread is a series of commands (a block of code) that exists as a unit of work. The operating system can manage multiple threads and assign a thread a piece ("slice") of processor time before switching to another thread to give it a turn to do some work. At its core (pardon the pun), a processor can simply execute a command, it has no concept of doing two things at one time. The operating system simulates this by allocating slices of time to different threads.

Now, if you introduce multiple cores/processors into the mix, then things CAN actually happen at the same time. The operating system can allocate time to one thread on the first processor, then allocate the same block of time to another thread on a different processor. All of this is about allowing the operating system to manage the completion of your task while you can go on in your code and do other things.

Strings and Images

String

String objects represent character strings in Cocoa and Cocoa Touch frameworks. Representing strings as objects allows you to use strings wherever you use other objects. It also provides the benefits of encapsulation, so that string objects can use whatever encoding and storage is needed for efficiency while simply appearing as arrays of characters.

A string object is implemented as an array of Unicode characters (in other words, a text string). An immutable string is a text string that is defined when it is created and subsequently cannot be changed. To create and manage an immutable string, use the NSString class. To construct and manage a string that can be changed after it has been created, use NSMutableString.

The objects you create using NSString and NSMutableString are referred to as string objects (or, when no confusion will result, merely as strings). The term *C string*refers to the standard C char * type.

A string object presents itself as an array of Unicode characters. You can determine how many characters it contains with the length method and can retrieve a specific character with the characterAtIndex: method. These two "primitive" methods provide basic access to a string object. Most use of strings, however, is at a higher level, with the strings being treated as single entities: You compare strings against one another, search them for substrings, combine them into new strings, and so on. If you need to access string objects character-by-character, you must understand the Unicode character encoding—specifically, issues related to composed character sequences. For details see:

- *The Unicode Standard, Version 4.0*. The Unicode Consortium. Boston: Addison-Wesley, 2003. ISBN 0-321-18578-1.
- The Unicode Consortium web site: http://www.unicode.org/.

Images

You use image objects to represent image data of all kinds, and the UIImage class is capable of managing data for all image formats supported by the underlying platform. Image objects are immutable, so you always create them from existing image data, such as an image file on disk or programmatically created image data. An image object may contain a single image or a sequence of images you intend to use in an animation.

You can use image objects in several different ways:

- Assign an image to a UIImageView object to display the image in your interface.
- Use an image to customize system controls such as buttons, sliders, and segmented controls.
- Draw an image directly into a view or other graphics context.
- Pass an image to other APIs that might require image data.

Although image objects support all platform-native image formats, it is recommended that you use PNG or JPEG files for most images in your app. Image objects are optimized for reading and displaying both formats, and those formats offer better performance than most other image

formats. Because the PNG format is lossless, it is especially recommended for the images you use in your app's interface.

JSON

JSON (short for JavaScript Object Notation) is a text-based, lightweight and easy way for storing and exchanging data. It's commonly used for representing structural data and data interchange in client-server applications, serving as an alternative to XML. A lot of the services we use everyday have JSON-based APIs. Most of the iOS apps including Twitter, Facebook and Flick send data to their backend web services in JSON format.

Setup JSON

Now before we go any further we are going to create some test JSON data. Let's pretend we wanted to get the high scores for our new game. Since we don't have a database setup to actually get the data we will just create some to test with.

Copy/Paste the below into a new file and save is as high_scores.json

```
[{
  "firstName": "John",
  "score": "302"
}, {
  "firstName": "Anna",
  "score": "288"
}, {
  "firstName": "Peter",
  "score": "243"
}]
```

OK so now we have our test data, you'll want to upload it to your server so we can call it via the URL for example (www.YourWebSite.com/high_scores.json).

Retrieving and Reading the JSON via Objective-C

Now you'll want to head back into your iOS project since this is where you want to get this data.

From here Copy/Paste this code to call your URL and get the high scores

```
NSError *error;
NSString *url_string = [NSString stringWithFormat: @"http://YourWebSite.com/high_score.json"];
NSData *data = [NSData dataWithContentsOfURL: [NSURL URLWithString:url_string]];
NSMutableArray *json = [NSJSONSerialization JSONObjectWithData:data options:kNilOptions error:&error];
NSLog(@"json: %@", json);
```

The only line you need to change here is the second line with the URL, replace "http://YourWebSite.com/high_score.json" with your own website.

Now when you run your app and this code you should see the high scores in xcodes console. Just like this:

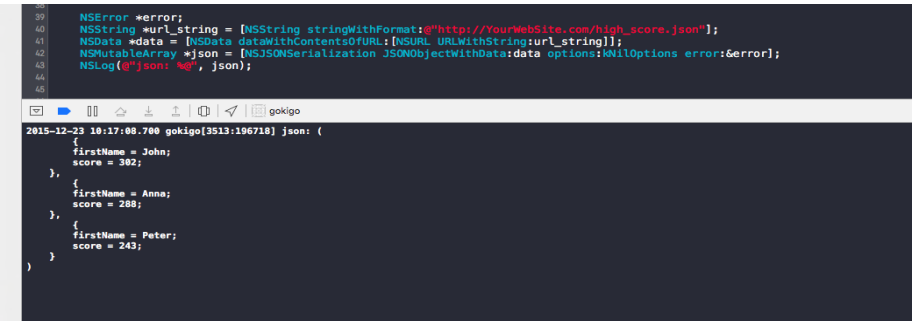


Fig: Json

XML

XML stands for extensible Markup Language. XML is a software- and hardware-independent tool for storing and transporting data.

An XMLParser notifies its delegate about the items (elements, attributes, CDATA blocks, comments, and so on) that it encounters as it processes an XML document. It does not itself do anything with those parsed items except report them. It also reports parsing errors. For convenience, an XMLParser object in the following descriptions is sometimes referred to as a parser object. Unless used in a callback, the XMLParser is a thread-safe class as long as any given instance is only used in one thread.

HTTP POST Requests

HTTP POST Request is commonly used methods for a request-response between a clients it submits data to be processed to a specified resource.

Here, I describe how one can use of POST method.

1. Set post string with actual username and password.

```
NSString *post = [NSString stringWithFormat:@"Username=%@&Password=%@",@"username",@"password"];
```

2. Encode the post string using NSASCIIStringEncoding and also the post string you need to send in NSData format.

```
NSData *postData = [post dataUsingEncoding:NSASCIIStringEncoding allowLossyConversion:YES];
```

You need to send the actual length of your data. Calculate the length of the post string.

```
NSString *postLength = [NSString stringWithFormat:@"%d",[postData length]];
```

3. Create a Urlrequest with all the properties like HTTP method, http header field with length of the post string. Create URLRequest object and initialize it.

```
NSMutableURLRequest *request = [[NSMutableURLRequest alloc] init];
```

Set the Url for which your going to send the data to that request.

```
[request setURL:[NSURL URLWithString:@"http://www.abcde.com/xyz/login.aspx"]];
```

Now, set **HTTP** method (*POST or GET*). Write this lines as it is in your code.

```
[request setHTTPMethod:@"POST"];
```

Set HTTP header field with length of the post data.

```
[request setValue:postLength forHTTPHeaderField:@"Content-Length"];
```

Also set the Encoded value for HTTP header Field.

```
[request setValue:@"application/x-www-form-urlencoded" forHTTPHeaderField:@"Content-Type"];
```

Set the HTTPBody of the urlrequest with postData.

```
[request setHTTPBody:postData];
```

4. Now, create URLConnection object. Initialize it with the URLRequest.

```
NSURLConnection *conn = [[NSURLConnection alloc] initWithRequest:request delegate:self];
```

It returns the initialized url connection and begins to load the data for the url request. You can check that whether you URL connection is done properly or not using just **if/else** statement as below.

```
if(conn) {
    NSLog(@"Connection Successful");
} else {
    NSLog(@"Connection could not be made");
}
```

5. To receive the data from the HTTP request , you can use the delegate methods provided by the URLConnection Class Reference. Delegate methods are as below.

```
// This method is used to receive the data which we get using post method.
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData*)data


// This method receives the error report in case of connection is not made to server.
- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error


// This method is used to process the data after connection has made successfully.
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
```

Simple app using the module taught e.g.: Weather Forecasting

Make a Weather Forecasting App using XML and JSON Data Parsing and other component which is already covered.

Reference and bibliography:

https://developer.apple.com/documentation/foundation/nsurlsession

https://www.raywenderlich.com/67081/cookbook-using-nsurlsession

https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/XMLParsing/Articles/UsingParser.html

https://www.appcoda.com/fetch-parse-json-ios-programming-tutorial/

https://developer.apple.com/documentation/foundation/nsjsonserialization

https://www.codementor.io/rheller/getting-reading-json-data-from-url-objective-c-du107s5mf

https://www.codementor.io/rheller/getting-reading-json-data-from-url-objective-c-du107s5mf

https://yuvarajmanickam.wordpress.com/2012/10/17/nsurlconnection-basics-for-ios-beginners/

Module 10: Local Storage

Topic: Local Storage

Chapter Overview

Data persistence is an important function of app based on any technology, be it android or iPhone. There are times when we have to work without an internet connection, which is where offline capabilities of a

mobile app come into practice.Here we will **explore local iOS data storage guidelines for iOS apps** intended to keep certain information locally. Local storage is meant for retaining web app data locally using certain frameworks, tools and methods distinctive to different platfo rms. For *iOS storage* there are different methods to choose from. The choice, however, depends upon what and how much data you want to store. Most of the times more than one method is required to implement local storage in iOS apps, as there are different persistence needs of the application viz. data gathered from web browsing, user preferences, and application settings.

Learning Outcome:

- User Defaults
- Sandboxing
- Working with Files
- Archiving
- UIDocument
- SQLite
- Core Data

Here we will discuss briefly about Local Storage.

User Defaults

The user defaults system is something that iOS inherited from OS X. Even though it was created and designed for storing user preferences, it can be used for storing any type of data as long as it's a property list type, NSString, NSNumber, NSDate, NSArray, NSDictionary, and NSData, or any of their mutable variants.

The user defaults database is nothing more than a collection of property lists, one property list per application. The property list is stored in a folder named Preferences in the application's Library folder, which hints at the property list's purpose and function.

One of the reasons that developers like the user defaults system is because it's so easy to use. Take a look at the code fragment below to see what I mean.

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

[userDefaults setBool:YES forKey:@"Key1"];
[userDefaults setInteger:123 forKey:@"Key2"];
[userDefaults setObject:@"Some Object" forKey:@"Key3"];

[userDefaults boolForKey:@"Key1"];
[userDefaults integerForKey:@"Key2"];
[userDefaults objectForKey:@"Key3"];

[userDefaults synchronize];
```

By calling the standardUserDefaults class method on NSUserDefaults, a reference to the shared defaults object is returned.

In the last line, we call synchronize on the shared defaults object to write any changes to disk. It's rarely necessary to invoke synchronize, because the user defaults system saves changes when necessary. However, if you store or update a setting using the user defaults system, it can sometimes be useful or necessary to explicitly save the changes to disk.

At first glance, the user defaults system seems to be nothing more than a key-value store located at a specific location. However, the NSUserDefaults class, defined in the Foundation framework, is more than an interface for managing a key-value store. Take a look at its class reference for more information.

Before we move on, paste the above code snippet in the application delegate's application:didFinishLaunchingWithOptions: method and run the application in the iOS Simulator. Open a new Finder window and navigate to Library > Application Support > iPhone Simulator > 7.1 > Applications (replace "7.1" with the latest version of iOS).

Find the application folder that corresponds with the application by inspecting the different, cryptically named folders in the Applications folder. The cryptically named folder is actually the application sandbox directory. In the application sandbox directory, open the Preferences folder, located in the Library folder, and inspect its contents.

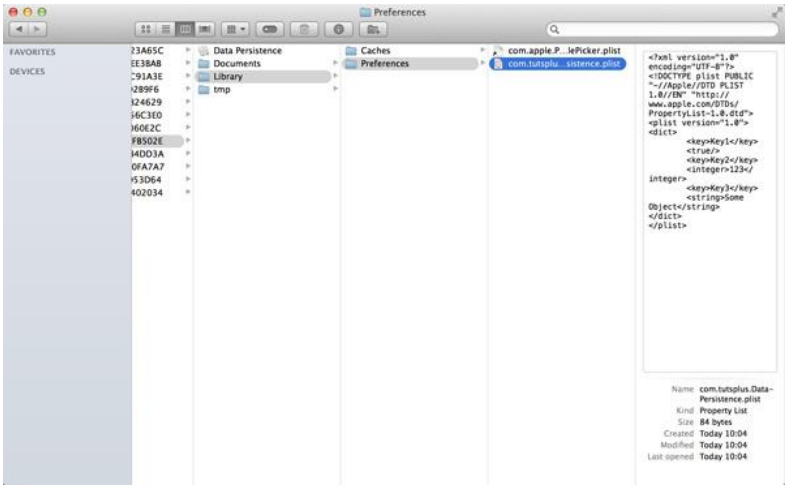


Fig: Storage View

You should see a property list with a name identical to the application's bundle identifier. This is the user defaults store for your application.

Sandboxing

Sandboxing your app is a great way to protect systems and users by limiting the privileges of an app to its intended functionality, increasing the difficulty for malicious software to compromise your users' systems.

The way this works is a virtual barrier called a "sandbox" is set up around a running program that isolates it from the rest of the system. The system cannot do this itself, so the developer voluntarily turns on sandboxing for its program. When enabled, the program will by default have no access to the system resources, including the network, user documents, the ability to open and save files, access to peripherals such as printers and cameras, and access to locations, address books, calendars, and similar central services.

The benefits of sandboxing

The benefit of sandboxing applications is that it protects programs from each other in addition to protecting user data. If someone develops a program for OS X that does not need to access the calendar, then having access to the calendar poses a potential security risk. If the program has bugs in its code, then without proper sandboxing there is a possibility that the program may access and corrupt the calendars. In the event that the program were hacked, then the calendar data would be at risk of being accessed by the attacker.

In addition to protecting user data, sandboxing will also help prevent applications from interfering with each other, and thereby increase the stability of a user's applications as a whole.

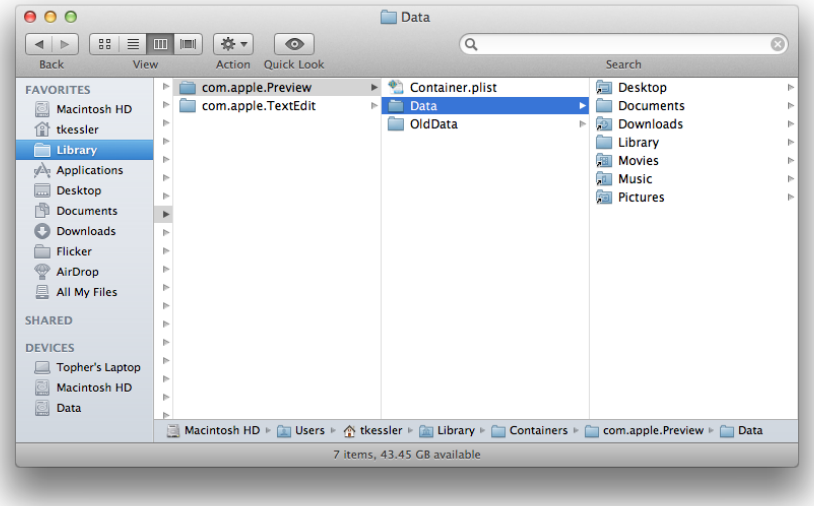


Fig: Finder window

Sandboxed applications store all their preference files, caches, and other automatically generated content in container directories, which make the program a bit easier to troubleshoot (click for larger view).

Working with Files

To work with file you have to use NSFileManager, NSFileHandle and NSData Foundation Framework classes and have to understand how the NSFileManager class in particular enables us to work with directories in Objective-C.

NSFileManager - The NSFileManager class can be used to perform basic file and directory operations such as creating, moving, reading and writing files and reading and setting file attributes. In addition, this class provides methods for, amongst other tasks, identifying the current working directory, changing to a new directory, creating directories and listing the contents of a directory.

NSFileHandle - The NSFileHandle class is provided for performing lower level operations on files, such as seeking to a specific position in a file and reading and writing a file's contents by a specified number of byte chunks and appending data to an existing file.

NSData - The NSData class provides a useful storage buffer into which the contents of a file may be read, or from which data may be written to a file.

Methods used in File Handling

The list of the methods used for **accessing** and **manipulating** files is listed below. Here, we have to replace the FilePath1, FilePath2 and FilePath strings to our required full file paths to get the desired action.

Check if file exists at a path

```
NSFileManager *fileManager = [NSFileManager defaultManager];
//Get documents directory
NSArray *directoryPaths = NSSearchPathForDirectoriesInDomains
(NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsDirectoryPath = [directoryPaths objectAtIndex:0];
if ([fileManager fileExistsAtPath:@""]==YES) {
    NSLog(@"File exists");
}
```

Comparing two file contents

```
if ([fileManager contentsEqualAtPath:@"FilePath1" andPath:@" FilePath2"]) {
  NSLog(@"Same content");
}
```

Check if writable, readable and executable

```
if ([fileManager isWritableFileAtPath:@"FilePath"]) {
  NSLog(@"isWritable");
}
if ([fileManager isReadableFileAtPath:@"FilePath"]) {
  NSLog(@"isReadable");
}
if ( [fileManager isExecutableFileAtPath:@"FilePath"]){
  NSLog(@"is Executable");
}
```

Move file

```
if([fileManager moveItemAtPath:@"FilePath1"
toPath:@"FilePath2" error:NULL]){
  NSLog(@"Moved successfully");
}
```

Copy file

```
if ([fileManager copyItemAtPath:@"FilePath1"
toPath:@"FilePath2"  error:NULL]) {
  NSLog(@"Copied successfully");
}
```

Remove file

```
if ([fileManager removeItemAtPath:@"FilePath" error:NULL]) {
   NSLog(@"Removed successfully");
}
```

Read file

```
NSData *data = [fileManager contentsAtPath:@"Path"];
```

Write file

```
[fileManager createFileAtPath:@"" contents:data attributes:nil];
```

Archiving

Archiving is the process of converting a group of related objects to a form that can be stored or transferred between applications. The end result of archiving—an archive—is a stream of bytes that records the identity of objects, their encapsulated values, and their relationships with other objects. Unarchiving, the reverse process, takes an archive and reconstitutes an identical network of objects.

The main value of archiving is that it provides a generic way to make objects persistent. Instead of writing object data out in a special file format, applications frequently store their model objects in archives that they can write out as files. An application can also transfer a network of objects—commonly known as an *object graph*—to another application using archiving. Applications often do this for pasteboard operations such as copy and paste.

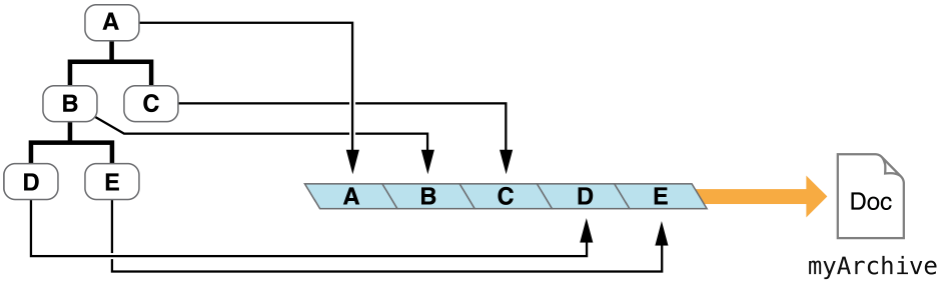


Fig: Archiving Flow

For its instances to be included in an archive, a class must adopt the NSCoding protocol and implement the required methods for encoding and decoding objects. Cocoa archives can hold Objective-C objects, scalar values, C arrays, structures, and strings. Archives store the types of objects along with the encapsulated data, so an object decoded from a stream of bytes is of the same class as the object that was originally encoded into the stream.

UIDocument

An abstract base class for managing discrete portions of your app's data.

Overview

Applications that make use of UIDocument and its underlying architecture get many benefits for their documents:

- Asynchronous reading and writing of data on a background queue. Your application's responsiveness to users is thus unaffected while reading and writing operations are taking place.
- Coordinated reading and writing of document files that is automatically integrated with cloud services.
- Support for discovering conflicts between different versions of a document (if that occurs).
- Safe-saving of document data by writing data first to a temporary file and then replacing the current document file with it.
- Automatic saving of document data at opportune moments; this mechanism includes support for dealing with suspend behaviors.

In the Model-View-Controller design pattern, a UIDocument object is a model object or model-controller object—it manages the data of a document or the aggregate model objects that together constitute the document's data. You typically pair it with a view controller that manages the view presenting the document’s contents. UIDocument provides no support for managing document views.

Document-based applications include those that can generate multiple documents, each with its own file-system location. A document-based application must create a subclass of UIDocument for its documents.

SQLite

If your application is data driven and works with large amounts of data, then you may want to look into SQLite. What is SQLite? The tagline on the SQLite website reads "Small. Fast. Reliable. Choose any three." which sums it up nicely.

SQLite is a library that implements a lightweight embedded relational database. As its name implies, it's based on the SQL standard (Structured Query Language) just like MySQL and PostgreSQL.

The main difference with other SQL databases is that SQLite is portable, very lightweight, and that it's serverless instead of a separate process accessed from the client application. In other words, it's embedded in the application and therefore very fast.

The SQLite website claims that it's the most widely deployed SQL database. I don't know if that's still the case, but it's certainly a popular choice for client-side data storage. Aperture and iPhoto, for example, rely on SQLite for some of their data storage.

The advantage SQLite has over working directly with objects is that SQLite is orders of magnitude faster, which is largely due to how relational databases and object oriented programming languages fundamentally differ.

To bridge the gap between SQLite and Objective-C, a number of Object Relational Mapping (ORM) solutions have been created over time. The ORM that Apple has created for iOS and OS X is named Core Data, which we'll take a look at later in this lesson.

Practice

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

@interface FailedBankInfo : NSObject {
    int _uniqueId;
    NSString *_name;
    NSString *_city;
    NSString *_state;
}

@property (nonatomic, assign) int uniqueId;
@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *city;
@property (nonatomic, copy) NSString *state;

- (id)initWithUniqueId:(int)uniqueId name:(NSString *)name city:(NSString *)city
    state:(NSString *)state;

@end
```

And replace FailedBankInfo.m with the following:

```
#import "FailedBankInfo.h"

@implementation FailedBankInfo

@synthesize uniqueId = _uniqueId;
@synthesize name = _name;
@synthesize city = _city;
@synthesize state = _state;

- (id)initWithUniqueId:(int)uniqueId name:(NSString *)name city:(NSString *)city
    state:(NSString *)state {
    if ((self = [super init])) {
        self.uniqueId = uniqueId;
        self.name = name;
```

```
        self.city = city;
        self.state = state;
    }
    return self;
}

- (void) dealloc {
    self.name = nil;
    self.city = nil;
    self.state = nil;
    [super dealloc];
}

@end
```

This is pretty standard Objective-C – there should be no surprises here. We're just creating a class to store the few pieces of data we'll be displaying in our table view, and make a convenience constructor.

Next we're going to create a helper class to handle all of the interaction with our sqlite3 database. This is good practice because by keeping everything abstracted, it makes it easier to switch to another storage method if we wanted to in the future.

So make another subclass of NSObject like you did above, but name it FailedBankDatabase.h. Replace FailedBankDatabase.h with the following:

```
#import <Foundation/Foundation.h>
#import <sqlite3.h>

@interface FailedBankDatabase : NSObject {
    sqlite3 *_database;
}

+ (FailedBankDatabase*)database;
- (NSArray *)failedBankInfos;

@end
```

Here we include the header file for sqlite3 at the top, and keep a member variable to store the pointer to our SQLite database. We also declare a static function to return the singleton instance of our FailedBankDatabase object, and declare a method to return an array of all of the FailedBankInfos from our database.

Erase everything in FailedBankDatabase.m and add the following to the top:

```
#import "FailedBankDatabase.h"
#import "FailedBankInfo.h"

@implementation FailedBankDatabase

static FailedBankDatabase *_database;

+ (FailedBankDatabase*)database {
    if (_database == nil) {
        _database = [[FailedBankDatabase alloc] init];
    }
    return _database;
}
```

First we import our header files, then we add the standard code to create a singleton instance of FailedBankDatabase for ease of access.

Add the following next:

```
- (id)init {
    if ((self = [super init])) {
        NSString *sqLiteDb = [[NSBundle mainBundle] pathForResource:@"banklist"
            ofType:@"sqlite3"];

        if (sqlite3_open([sqLiteDb UTF8String], &_database) != SQLITE_OK) {
            NSLog(@"Failed to open database!");
        }
    }
    return self;
}

- (void)dealloc {
    sqlite3_close(_database);
    [super dealloc];
}
```

When we initialize our object, we construct a path to our database file. We're storing the database in our application's bundle, so we use the pathForResource method to obtain the path.

Note that since the database is stored in our main bundle, that means we couldn't write to the database. This is fine for this app, but if you have an app that you need to both read and write to the database, check out my GDataXML tutorial for an example of how to save your bundled data to the documents directory for editing.

Once we have the path to the database, we open it up with the sqlite3_open API call. It will return an error if anything goes wrong – otherwise we're good to go! Note that when we're done we should close the database handle with sqlite3_close – I put that in the dealloc method.

Now for the fun part – retrieving the data from the database!

```
- (NSArray *)failedBankInfos {

    NSMutableArray *retval = [[[NSMutableArray alloc] init] autorelease];
    NSString *query = @"SELECT id, name, city, state FROM failed_banks
        ORDER BY close_date DESC";
    sqlite3_stmt *statement;
    if (sqlite3_prepare_v2(_database, [query UTF8String], -1, &statement, nil)
        == SQLITE_OK) {
        while (sqlite3_step(statement) == SQLITE_ROW) {
            int uniqueId = sqlite3_column_int(statement, 0);
            char *nameChars = (char *) sqlite3_column_text(statement, 1);
            char *cityChars = (char *) sqlite3_column_text(statement, 2);
            char *stateChars = (char *) sqlite3_column_text(statement, 3);
            NSString *name = [[NSString alloc] initWithUTF8String:nameChars];
            NSString *city = [[NSString alloc] initWithUTF8String:cityChars];
            NSString *state = [[NSString alloc] initWithUTF8String:stateChars];
            FailedBankInfo *info = [[FailedBankInfo alloc]
                initWithUniqueId:uniqueId name:name city:city state:state];
            [retval addObject:info];
            [name release];
            [city release];
            [state release];
            [info release];
        }
        sqlite3_finalize(statement);
    }
    return retval;

}
@end
```

Here we construct our SQL string, and execute it with the sqlite3_prepare_v2 API call. We then step through each row, and pull out the return values one by one. We have to do a little conversion here to get the data from UTF8 strings into NSStrings, then we construct FailedBankInfo objects based on the data and add it to our array.

We have to call sqlite3_finalize to clean up the memory used for the statement, then we return the data.

So let's see if this works. Open up FailedBanksAppDelegate.m and add the following imports to the top of the file:

```
#import "FailedBankDatabase.h"
#import "FailedBankInfo.h"
```

Then add the following inside applicationDidFinishLaunching:

```
NSArray *failedBankInfos = [FailedBankDatabase database].failedBankInfos;
for (FailedBankInfo *info in failedBankInfos) {
    NSLog(@"%d: %@, %@, %@", info.uniqueId, info.name, info.city, info.state);
}
```

If all goes well, you should see lines like the following in your deubug log:

```
1: Desert Hills Bank, Phoenix, AZ
2: Unity National Bank, Cartersville, GA
3: Key West Bank, Key West, FL
```

Among the numerous applications existing on the App Store today, it would be hard for someone to find more than a few of them that do not deal with data. Most of the apps handle some sort of data, no matter in what format they are, and always perform some actions upon it. There are various solutions offered to developers for storing and managing data, and usually each one of them is suitable for different kind of applications. However, when working with large amount of data, the preferred method it seems like a one-way path: That is the use of a database.

Indeed, making use of a database can solve various kind of problems that should be solved programmatically in other cases. For programmers who love working with databases and SQL, this is the favorite data-managing method at 90% of the cases, and the first think that crosses their minds when talking about data. Also, if you were used to working with other databases or database management systems (DBMSs), then you'll really appreciate the fact that you can keep applying your SQL knowledge in the iOS platform as well.

The database that can be used by apps in iOS (and also used by iOS) is called **SQLite**, and it's a *relational database*. It is contained in a C-library that is embedded to the app that is about to use it. Note that it does not consist of a separate service or daemon running on the background and attached to the app. On the contrary, the app runs it as an integral part of it. Nowadays, SQLite lives its third version, so it's also commonly referred as *SQLite 3*.

Fig: Infographic

SQLite is not as powerful as other DMBSs, such as MySQL or SQL Server, as it does not include all of their features. However, its greatness lies mostly to these factors:

- It's lightweight.
- It contains an embedded SQL engine, so almost all of your SQL knowledge can be applied.
- It works as part of the app itself, and it doesn't require extra active services.
- It's very reliable.
- It's fast.
- It's fully supported by Apple, as it's used in both iOS and Mac OS.
- It has continuous support by developers in the whole world and new features are always added to it.

Focusing now on our tutorial, let me start by stating that my goal is not to show you how to become a SQLite expert. Instead, my plan is to implement a database class step by step, which will utilize the most important features of the SQLite library, but it will also become a reusable tool for your own applications. Unfortunately, even though SQLite is supported by Apple, a mechanism or a pre-made database management library does not exist. Going into more details, the database class that we will implement will be capable of executing all the standard SQL queries (*select*, *insert*, *update*, *delete*). The most important is that we'll create it in such way, so it accepts clear SQL statements, and if you had worked with SQL in the past, you'll definitely know what to do here too.

Besides the database class that we'll develop, we'll also create a sample application to test it and to see it in action. More details about that you'll find in the next section though. Note that the queries we'll write in the demo app will be simple enough for the sake of the tutorial, however be sure that more complex queries can be executed as well.

As a final word before we proceed, I would recommend to make a web search and read some more stuff about the SQLite itself. Of course, the first one should be the the official website.

Core Data

Core Data is a framework that you use to manage the model layer objects in your application. It provides generalized and automated solutions to common tasks associated with object life cycle and object graph management, including persistence.

Core Data typically decreases by 50 to 70 percent the amount of code you write to support the model layer. This is primarily due to the following built-in features that you do not have to implement, test, or optimize:

- Change tracking and built-in management of undo and redo beyond basic text editing.
- Maintenance of change propagation, including maintaining the consistency of relationships among objects.
- Lazy loading of objects, partially materialized futures (faulting), and copy-on-write data sharing to reduce overhead.
- Automatic validation of property values. Managed objects extend the standard key-value coding validation methods to ensure that individual values lie within acceptable ranges, so that combinations of values make sense.
- Schema migration tools that simplify schema changes and allow you to perform efficient in-place schema migration.
- Optional integration with the application's controller layer to support user interface synchronization.
- Grouping, filtering, and organizing data in memory and in the user interface.
- Automatic support for storing objects in external data repositories.
- Sophisticated query compilation. Instead of writing SQL, you can create complex queries by associating an NSPredicate object with a fetch request.
- Version tracking and optimistic locking to support automatic multiwriter conflict resolution.
- Effective integration with the macOS and iOS tool chains.

Core Data provides a relational object oriented model that can be serialized into an XML, binary, or SQLite store. Core Data even supports an in-memory store.

Why should you use Core Data instead of SQLite? By asking this question, you wrongly assume that Core Data is a database. The advantage of using Core Data is that you work with objects instead of raw data, such as rows in a SQLite database or data stored in an XML file. Even though Core Data had some difficult years when it was first released, it has grown into a robust framework with a lot of features, such as automatic migrations, change tracking, faulting, and integrated validation.

Another great feature that many developers appreciate is the Core Data model editor built into Xcode that lets developers model their data model through a graphical interface.

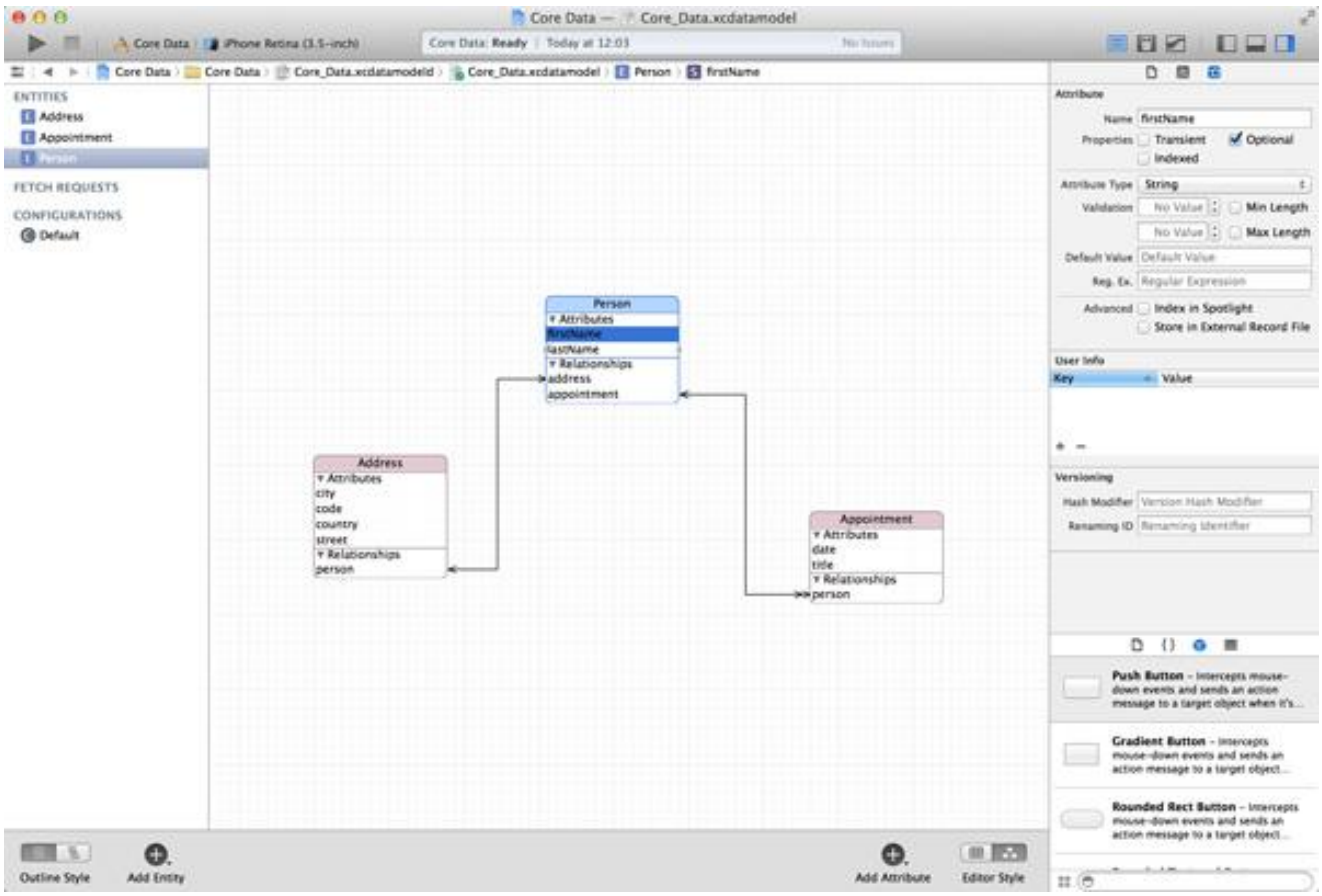


Fig: Database model view

Simple app using the module taught e.g.: Quiz App, Notepad

Make a Quiz App or Notepad App using SQLite, core data, files and other component which is already covered

Reference and bibliography:

https://developer.apple.com/documentation/foundation/userdefaults

https://www.hackingwithswift.com/read/12/2/reading-and-writing-basics-userdefaults

https://developer.apple.com/library/content/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html

https://code.tutsplus.com/tutorials/data-persistence-and-sandboxing-on-ios--mobile-14078

https://www.techotopia.com/index.php/Working_with_Files_in_Objective-C

https://www.techotopia.com/index.php/Working_with_Directories_in_Objective-C

Module 11: Multitouch, Taps, and Gestures

Topic: Multitouch, Taps, and Gestures

Chapter Overview:

When the user interacts with the touch screen of an iPhone or iPad the hardware detects the physical contact and notifies the operating system. The operating system subsequently creates an event associated with the interaction and passes it into the currently active application's event queue where it is subsequently picked up by the event loop and passed to the current first responder object; the first responder being the object with which the user was interacting when this event was triggered (for example a UIButton or UIView object). If the first responder has been programmed to handle the type of event received it does so (for example a button may have an action defined to call a particular method when it receives a touch event). Having handled the event, the responder then has the option of discarding that event, or passing it up to the next responder in the response chain (defined by the object's next property) for further processing, and so on up the chain. If the first responder is not able to handle the event it will also pass it to the next responder in the chain and so on until it either reaches a responder that handles the event or it reaches the end of the chain (the UIApplication object) where it will either be handled or discarded. In This Chapter we will discuss about:

Learning Outcome:

- Uses of Touch notification method
- Touch Notification Methods
- Gesture Recognizers
- Sensor & Inputs

Introduction

Touch Notification Methods

Gesture Recognizers

Sensor & Inputs

Simple app using the module taught

Introduction

In terms of physical points of interaction between the device and the user, the iPhone and iPad provide four buttons, a switch and a touch screen. Without question, the user will spend far more time using the touch screen than any other aspect of the device. It is essential, therefore, that any application be able to handle gestures (touches, multitouches, taps, swipes and pinches etc) performed by the user's fingers on the touch screen.

When set to YES, the view receives all touches associated with a multi-touch sequence and starting within the view's bounds. When set to NO, the view receives only the first touch event in a multi-touch sequence that start within the view's bounds. The default value of this property is NO.

Note

This property does not affect the gesture recognizers attached to the view. Gesture recognizers receive all touches that occur in the view.

Other views in the same window can still receive touch events when this property is NO. If you want this view to handle multi-touch events exclusively, set the values of both this property and the exclusiveTouch property to YES. This property does not prevent a view from being asked to handle multiple touches. For example, two subviews may both forward their touches to a common parent, such as a window or the root view of a view controller. This property determines how many touches initially targeting the view are delivered to that view.

Touch Notification Methods

Touch screen events cause one of four methods on the first responder object to be called. The method that gets called for a specific event will depend on the nature of the interaction. In order to handle events, therefore, it is important to ensure that the appropriate methods from those outlined below are implemented within your responder chain.

touchesBegan method

The touchesBegan method is called when the user first touches the screen. Passed to this method are an argument called touches of type NSSet and the corresponding UIEvent object. The touches object contains a UITouch event for each finger in contact with the screen. The tapCount method of any of the UITouch events within the touches set can be called to identify the number of taps, if any, performed by the user. Similarly, the coordinates of an individual touch can be identified from the UITouch event either relative to the entire screen or within the local view itself.

touchesMoved method

The touchesMoved method is called when one or more fingers move across the screen. As fingers move across the screen this method gets called multiple times allowing the application to track the new coordinates and touch count at regular intervals. As with the touchesBegan method, this method is provided with an event object and an NSSet object containing UITouch events for each finger on the screen.

touchesEnded method

This method is called when the user lifts one or more fingers from the screen. As with the previous methods, touchesEnded is provided with the event and NSSet objects.

touchesCancelled method

When a gesture is interrupted due to a high level interrupt, such as the phone detecting an incoming call, the touchesCancelled method is called.

Gesture Recognizers

A gesture-recognizer object—or, simply, a gesture recognizer—decouples the logic for recognizing a sequence of touches (or other input) and acting on that recognition. When one of these objects recognizes a common gesture or, in some cases, a change in the gesture, it sends an action message to each designated target object.

The UIGestureRecognizer class

• **UITapGestureRecognizer** – This class is designed to detect when a user taps on the screen of the device. Both single and multiple taps may be detected based on the configuration of the class instance.

• **UIPinchGestureRecognizer** – Detects when a pinching motion is made by the user on the screen. This motion is typically used to zoom in or out of a view or to change the size of a visual component.

• **UIPanGestureRecognizer** – Detects when a dragging or panning gesture is made by the user.

• **UIScreenEdgePanGestureRecognizer** – Detects when a dragging or panning gesture is performed starting near the edge of the display screen.

• **UISwipeGestureRecognizer** – Used to detect when the user makes a swiping gesture across the screen. Instances of this class may be configured to detect motion only in specific directions (left, right, up or down).

• **UIRotationGestureRecognizer –** Identifies when the user makes a rotation gesture (essentially two fingers in contact with the screen located opposite each other and moving in a circular motion).

• **UILongPressGestureRecognizer** – Used to identify when the user touches the screen with one or more fingers for a specified period of time (also referred to as "touch and hold").

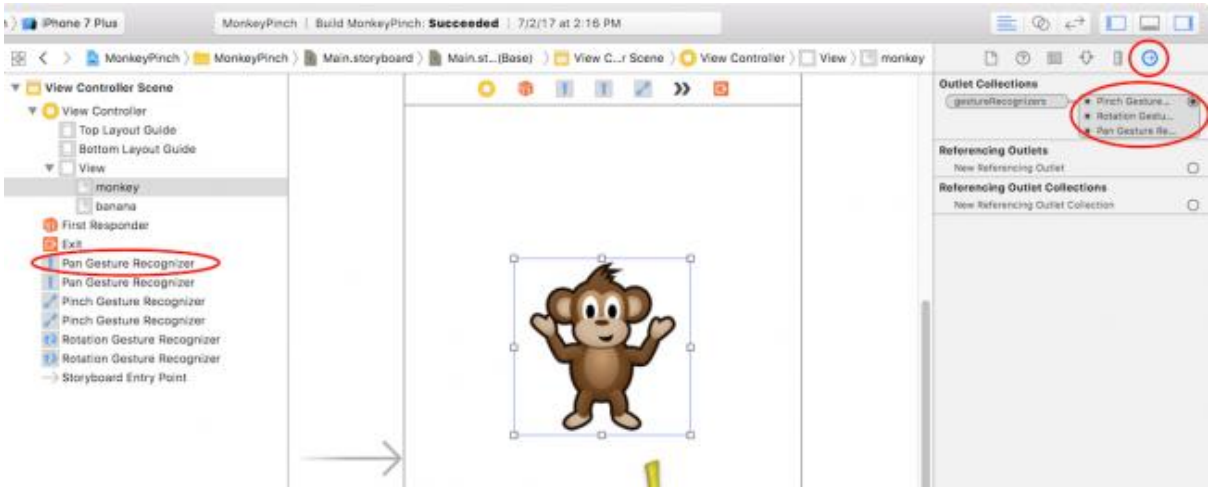


Fig: UILongPressGestureRecognizer

These gesture recognizers must be attached to the view on which the gesture will be performed via a call to the view object's addGestureRecognizer method. Recognizers must also be assigned an action method that is to be called when the specified gesture is detected. Gesture recognizers may subsequently be removed from a view via a call to the view's removeGestureRecognizer method, passing through as an argument the recognizer to be removed.

A window delivers touch events to a gesture recognizer before it delivers them to the hit-tested view attached to the gesture recognizer. Generally, if a gesture recognizer analyzes the stream of touches in a multi-touch sequence and doesn't recognize its gesture, the view receives the full complement of touches. If a gesture recognizer recognizes its gesture, the remaining touches for the view are cancelled. The usual sequence of actions in gesture recognition follows a path determined by default values of the cancelsTouchesInView, delaysTouchesBegan, delaysTouchesEnded properties:

- cancelsTouchesInView—If a gesture recognizer recognizes its gesture, it unbinds the remaining touches of that gesture from their view (so the window won't deliver them). The window cancels the previously delivered touches with a (touchesCancelled(_:with:)) message. If a gesture recognizer doesn't recognize its gesture, the view receives all touches in the multi-touch sequence.
- delaysTouchesBegan—As long as a gesture recognizer, when analyzing touch events, has not failed recognition of its gesture, the window withholds delivery

of touch objects in the began phase to the attached view. If the gesture recognizer subsequently recognizes its gesture, the view doesn't receive these touch objects. If the gesture recognizer doesn't recognize its gesture, the window delivers these objects in an invocation of the view's touchesBegan(_:with:) method (and possibly a follow-up touchesMoved(_:with:) invocation to inform it of the touches current location).

- delaysTouchesEnded—As long as a gesture recognizer, when analyzing touch events, has not failed recognition of its gesture, the window withholds delivery of touch objects in the ended phase to the attached view. If the gesture recognizer subsequently recognizes its gesture, the touches are cancelled (in a touchesCancelled(_:with:) message). If the gesture recognizer doesn't recognize its gesture, the window delivers these objects in an invocation of the view's touchesEnded(_:with:) method.

Sensor & Inputs

Microphone

You can record audio by using a class AVAudioRecorder that provides audio recording capability in your application.

Using an audio recorder, you can:

- Record until the user stops the recording
- Record for a specified duration
- Pause and resume a recording
- Obtain input audio-level data that you can use to provide level metering

In iOS, the audio being recorded comes from the device connected by the user—built-in microphone or headset microphone, for example. In macOS, the audio comes from the system's default audio input device as set by a user in System Preferences.

You can implement a delegate object for an audio recorder to respond to audio interruptions and audio decoding errors, and to the completion of a recording.

To configure a recording, including options such as bit depth, bit rate, and sample rate conversion quality, configure the audio recorder's settings dictionary. Use the settings keys described in *AV Foundation Audio Settings Constants*.

To configure an appropriate audio session for recording, refer to AVAudioSession and AVAudioSessionDelegate.

The AVAudioRecorder class is intended to allow you to make audio recordings with very little programming overhead

Camera

The iOS library provides the class UIImagePickerController which is the user interface for managing the user interaction with the camera or with the photo library. As usual, the UIImagePickerController requires the use of a delegate to respond to interactions.

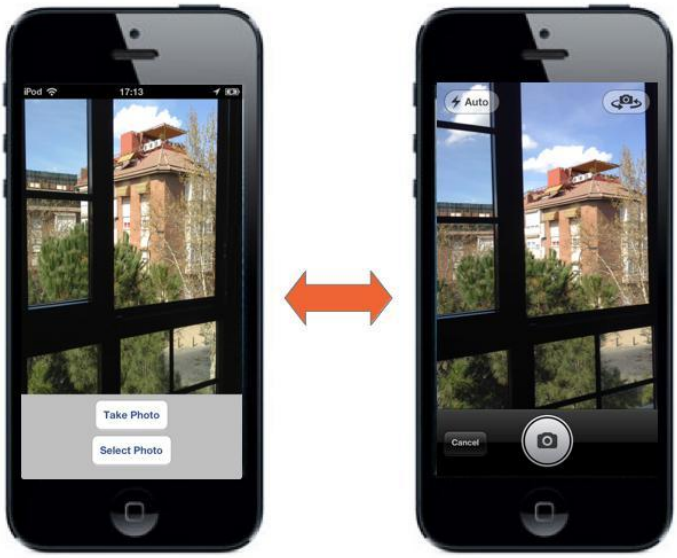


Fig: iPhone Camera

To help you understand the usage of UIImagePickerController, we'll build a simple camera app. The example application is very simple: we will have a main window with a big UIImageView to show the selected photo, and two buttons: one to take a new photo, and the other one to select a photo from the photo library.

Simple app using the module taught

Now this is the time to go make another app using module we learned.

Reference and bibliography:

https://www.techotopia.com

https://developer.apple.com/documentation/uikit/uigesturerecognizer

https://www.techotopia.com/index.php/Identifying_Gestures_using_iOS_8_Gesture_Recognizers

https://www.raywenderlich.com/162745/uigesturerecognizer-tutorial-getting-started

Module 12: Drawing

Topics: Drawing

Chapter Overview:

High-quality graphics are an important part of your app's user interface. Providing high-quality graphics not only makes your app look good, but it also makes your app look like a natural extension to the rest of the system. iOS provides two primary paths for creating high-quality graphics in your system: OpenGL or native rendering using Quartz, Core Animation, and UIKit. This document describes native rendering. (To learn about OpenGL drawing, see *OpenGL ES Programming Guide*.)

Learning Outcome:

- Some of the basic principles behind the drawing of two dimensional graphics.
- Using the Quartz 2D API.
- Obtaining the graphics context.
- Implementing the *drawRect* method
- Handling of colors and transparency.

In this chapter we will discuss about:

Core Graphics and Quartz 2D

Points, Coordinates and Pixels

Graphics Context

Core Graphics and Quartz 2D

Core Graphics, also known as Quartz 2D, is an advanced, two-dimensional drawing engine available for iOS, tvOS and macOS application development. Quartz 2D provides low-level, lightweight 2D rendering with unmatched output fidelity regardless of display or printing device. Quartz 2D is resolution- and device-independent.

Quartz 2D is a two-dimensional drawing engine accessible in the iOS environment and from all Mac OS X application environments outside of the kernel.

You can use the Quartz 2D application programming interface (API) to gain access to features such as path-based drawing, painting with transparency, shading, drawing shadows, transparency layers, color management, anti-aliased rendering, PDF document generation, and PDF metadata access. Whenever possible.

In Mac OS X, Quartz 2D can work with all other graphics and imaging technologies—Core Image, Core Video, OpenGL, and QuickTime. It's possible to create an image in Quartz from a QuickTime graphics importer, using the QuickTime function GraphicsImportCreateCGImage.

Similarly, in iOS, Quartz 2D works with all available graphics and animation technologies, such as Core Animation, OpenGL ES, and the UIKit classes.

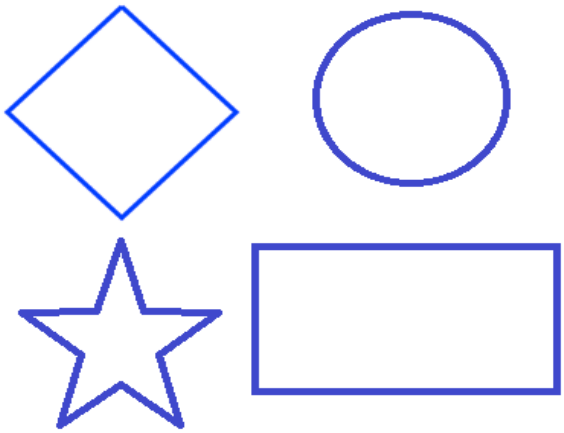
Draw Method:

The first time a view is displayed, and each time part of that view needs to be redrawn as a result of another event, the draw method of the view is called. Drawing is achieved, therefore,

by subclassing the UIView class, implementing the draw method and placing within that method the Quartz 2D API calls to draw the graphics.

In instances where the draw method is not automatically called, a redraw may be forced via a call to the setNeedsDisplay or setNeedsDisplayInRect methods.

Core Graphics and Quartz 2D will facilitate us to draw Graphics context, Paths, Color and Color Spaces, Patterns, Shadows, Gradient and many more. For Example:



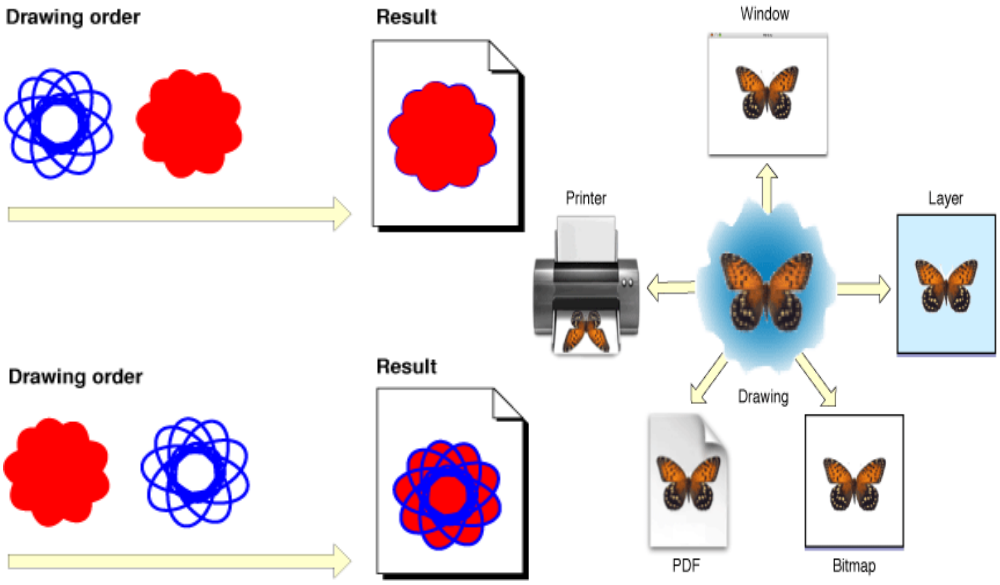


Fig: 2D Graphics content drawing

Points, Coordinates and Pixels

The Quartz 2D API functions work on the basis of points. These are essentially the x and y coordinates of a two dimensional coordinate system on the device screen with 0, 0 representing the top left-hand corner of the display. These coordinates are stored in the form of CGFloat variables.

An additional C structure named CGPoint is used to contain both the x and y coordinates to specify a point on the display. Similarly, the CGSize structure stores two CGFloat values designating the width and height of an element on the screen.

Further, the position and dimension of a rectangle can be defined using the CGRect structure which contains a CGPoint (the location) and CGSize (the dimension) of a rectangular area.

Of key importance when working with points and dimensions is that these values do not correspond directly to screen pixels. In other words there is not a one to one correlation between pixels and points. Instead the underlying framework decides, based on a scale factor, where a point should appear and at what size, relative to the resolution of the display on which the drawing is taking place. This enables the same code to work on both higher and lower resolution screens (for example an iPhone 3GS screen and an iPhone 6s retina display) without the programmer having to worry about it.

For more precise drawing requirements, iOS version 4 and later allows the scale factor for the current screen to be obtained from UIScreen, UIView, UIImage, and CALayer classes allowing the correlation between pixels and points to be calculated for greater drawing precision. For iOS 3 or older the scale factor is always returned as 1.0.

Graphics Context

A graphics context represents a drawing destination. It contains drawing parameters and all device-specific information that the drawing system needs to perform any subsequent drawing commands. A graphics context defines basic drawing attributes such as the colors to use when drawing, the clipping area, line width and style information, font information, compositing options, and several others.

You can obtain a graphics context by using Quartz context creation functions or by using higher-level functions provided by one of the Mac OS X frameworks or the UIKit framework in iOS. Quartz provides functions for various flavors of Quartz graphics contexts including bitmap and PDF, which you can use to create custom content.

This chapter shows you how to create a graphics context for a variety of drawing destinations. A graphics context is represented in your code by the data type CGContextRef, which is an opaque data type. After you obtain a graphics context, you can use Quartz 2D functions to draw to the context, perform operations (such as translations) on the context, and change graphics state parameters, such as line width and fill color.

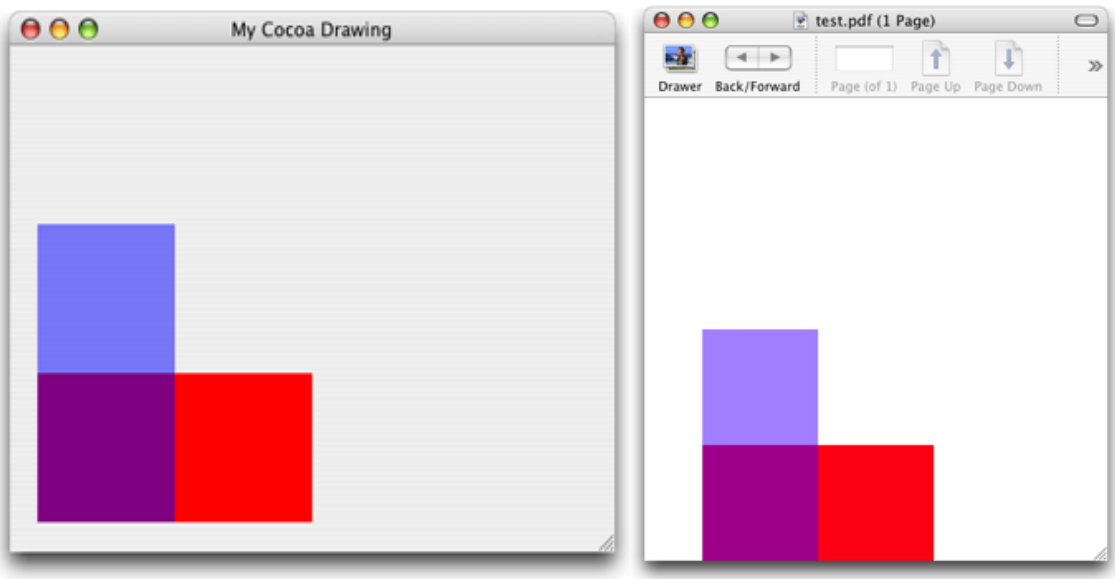


Fig: Graphics Context

Reference and bibliography:

https://developer.apple.com/documentation/coregraphics

https://developer.apple.com/library/content/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/Introduction/Introduction.html

https://www.techotopia.com/index.php/Drawing_iOS_8_2D_Graphics_in_Swift_with_Core_Graphics

Module 13: Animation

Topics: Animation

Chapter Overview:

Animation is a critical part of your iOS user interfaces. Animation draws the user's attention toward things that change, and adds a ton of fun and polish to your apps UI.

Even more importantly, in an era of "flat design", animation is one of the key ways to make your app stand apart from others.

Learning Outcome:

- Basics of Core Animation before working step-by-step
- Through an example to demonstrate the implementation of motion, rotation and scaling animation.

In this section, you'll learn how to use UIView animation to do the following:

Core Animation

UIView Core Animation Blocks

Animation Curves

Transformations

Core Discussion:

The majority of the visual effects used throughout the iOS user interface are performed using *Core Animation*. Core Animation provides a simple mechanism for implementing basic animation within an iOS application. If you need a user interface element to gently fade in or out of view, slide smoothly across the screen or gracefully resize or rotate before the user's eyes, these effects can be achieved using Core Animation in just a few lines of code.

In animation part we will cover Core Animation, UIView Core Animation Block, Animation curve and transformation.

Core Animation

Core Animation is a graphics rendering and animation infrastructure available on both iOS and OS X that you use to animate the views and other visual elements of your app.

With Core Animation, most of the work required to draw each frame of an animation is done for you. All you have to do is configure a few animations

Parameters (such as the start and end points) and tell Core Animation to start. Core Animation does the rest, handing most of the actual drawing work off to the

Onboard graphics hardware to accelerate the rendering. This automatic graphics acceleration results in high frame rates and smooth animations without burdening the CPU and slowing down app.

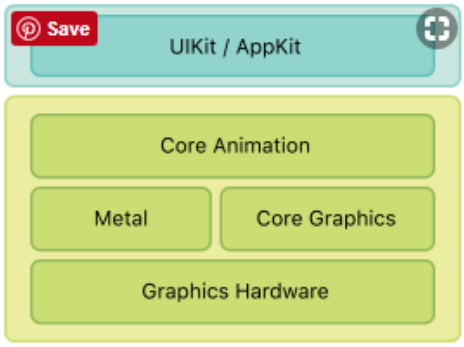


Fig: Graphics Architecture

UIView Core Animation Blocks

The concept of Core Animation involves the use of so-called *animation block* methods. Animation block methods are used to mark the beginning and end of a sequence of changes to the appearance of a UIView and its corresponding subviews. Once the end of the block is reached, the animation is performed over a specified duration. For the sake of example, consider a UIView object that contains a UIButton connected to an outlet named *the Button.* The application requires that the button gradually fade from view over a period of 3 seconds. This can be achieved by making the button transparent through the use of the *alpha* property:

```
theButton.alpha = 0;
```

Simply setting the alpha property to 0, however, causes the button to immediately become transparent. In order to make it fade out of sight gradually we need to place this line of code in a call to the *animateWithDuration:* animation block method as follows:

```
[UIView animateWithDuration:3.0 animations:^{
    _theButton.alpha = 0.0;
}];
```

A variety of properties may also be defined within the animation block. For example, the start of the animation can be delayed using the delay argument of the *animateWithDuration:delay:options:animations:completion* method call. The following example delays the start of the 3 second fade out animation sequence by 5 seconds:

```
[UIView animateWithDuration:3.0
        delay: 5.0
        options: UIViewAnimationOptionCurveLinear
        animations:^{
          _theButton.alpha = 0.0;
        }
        completion:nil];
```

Animation Curves

In addition to specifying the duration of an animation sequence, the linearity of the animation timeline may also be defined by specifying an *animation curve* setting for the *options* argument of the *animateWithDuration* class method. This setting controls whether the animation is performed at a constant speed, whether it starts out slow and speeds up and so on. There are currently four possible animation curve settings:

- **UIViewAnimationCurveLinear** – The animation is performed at constant speed for the specified duration and is the option declared in the above code example.
- **UIViewAnimationCurveEaseOut** – The animation starts out fast and slows as the end of the sequence approaches
- **UIViewAnimationCurveEaseIn** – The animation sequence starts out slow and speeds up as the end approaches.
- **UIViewAnimationCurveEaseInOut** – The animation starts slow, speeds up and then slows down again.

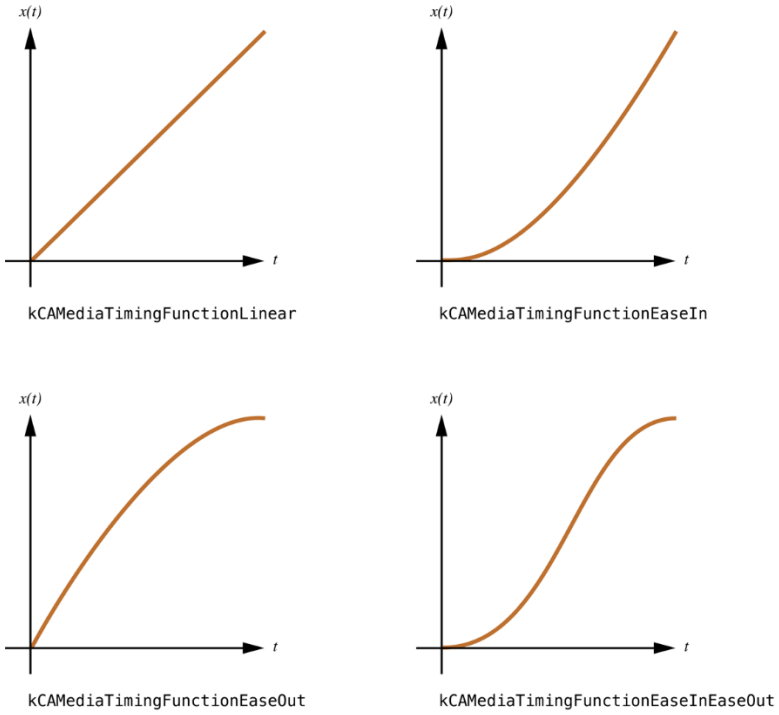


Fig: Animation Curves

Transformations

Transformations allow changes to be made to the coordinate system of a screen area. This essentially allows the programmer to rotate, resize and translate a UIView object. A call is made to one of a number of transformation functions and the result assigned to the *transform* property of the UIView object.

For example, to change the scale of a UIView object named *myView* by a factor of 2 in both height and width:

```
myView.transform = CGAffineTransformMakeScale(2, 2);
```

Similarly, the UIView object may be rotated using the *CGAffineTransformMakeRotation()* function which takes as an argument the angle (in radians) by which the view is to be rotated. The following code, for example, rotates a view by 90 degrees:

```
myView.transform = CGAffineTransformMakeRotation( 90 * M_PI  / 180);
```

The key point to keep in mind with transformations is that they become animated effects when performed within an animation block. The transformations evolve over the duration of the animation and follow the specified animation curve in terms of timing.

Reference and bibliography:

https://developer.apple.com/library/content/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/AnimatingViews/AnimatingViews.html

https://www.raywenderlich.com/173544/ios-animation-tutorial-getting-started-3

https://medium.com/@joncardasis/better-ios-animations-with-catransaction-72a7425673a6

Module 14: App States

Tropics: App States

Chapter Overview:

Apps developed for early iOS versions (before iOS 4.0) supported three states: non-running, inactive, and active. An application delegate for pre-iOS 4.0 apps received two important method calls: application DidFinishLaunching and application Will Terminate. When an app received an application DidFinishLaunching message, it was an opportunity for information to be retrieved from the previous launch to restore the app to its last used state. The status, application Will Terminate, was used to notify the app when the app was preparing to shut down. This gave the developer an opportunity to save any unsaved data or specific state information.

Learning Outcome:

- Deferent State of App
- Lifecycle of iOS app
- Working in background
- Uses of app memory
- Execute in background

In this chapter we will discuss about:

App State

App Lifecycle

Moving to the Background

Memory Usage

Background Execution

App State

Currently, there are five possible application states that would be cause for the app to prepare for a transition - such as a shutdown or moving to the background. In certain cases, an app might need to continue processing in the background. However, there is certainly no reason for the app to process any graphics, animations, or display-specific routines. The five states of an iOS app - as listed in the iOS App Programming Guide - include the following:

1. ***Non-running*** - The app is not running.
2. ***Inactive*** - The app is running in the foreground, but not receiving events. An iOS app can be placed into an inactive state, for example, when a call or SMS message is received.
3. ***Active*** - The app is running in the foreground, and receiving events.
4. ***Background*** - The app is running in the background, and executing code.
5. ***Suspended*** - The app is in the background, but no code is being executed.

App Lifecycle

Apps are a sophisticated interplay between your custom code and the system frameworks. The system frameworks provide the basic infrastructure that all apps need to run, and you provide the code required to customize that infrastructure and give the app the look and feel you want. To do that effectively, it helps to understand a little bit about the iOS infrastructure and how it works.

iOS frameworks rely on design patterns such as model-view-controller and delegation in their implementation. Understanding those design patterns is crucial to the successful creation of an app. It also helps to be familiar with the Objective-C language and its features.

Moving to the Background

When moving from foreground to background execution, use the applicationDidEnterBackground: method of your app delegate to do the following:

- **Prepare to have your app's picture taken.** When your applicationDidEnterBackground: method returns, the system takes a picture of your app's user interface and uses the resulting image for transition animations. If any views in your interface contain sensitive information, you should hide or modify those views before theapplicationDidEnterBackground: method returns. If you add new views to your view hierarchy as part of this process, you must force those views to draw themselves, as described in Prepare for the App Snapshot.
- **Save any relevant app state information.** Prior to entering the background, your app should already have saved all critical user data. Use the transition to the background to save any last minute changes to your app's state.
- **Free up memory as needed.** Release any cached data that you do not need and do any simple cleanup that might reduce your app's memory footprint. Apps with large memory footprints are the first to be terminated by the system, so release image resources, data caches, and any other objects that you no longer need. For more information, see Reduce Your Memory Footprint.

Your app delegate's applicationDidEnterBackground: method has approximately 5 seconds to finish any tasks and return. In practice, this method should return as quickly as possible. If the method does not return before time runs out, your app is killed and purged from memory. If you still need more time to perform tasks, call the beginBackgroundTaskWithExpirationHandler: method to request background execution time and then start any long-running tasks in a secondary thread. Regardless of whether you start any background tasks, the applicationDidEnterBackground: method must still exit within 5 seconds.

Note: The system sends the UIApplicationDidEnterBackgroundNotification notification in addition to calling the applicationDidEnterBackground:method. You can use that notification to distribute cleanup tasks to other objects of your app.

Depending on the features of your app, there are other things your app should do when moving to the background. For example, any active Bonjour services should be suspended and the app should stop calling OpenGL ES functions. For a list of things your app should do when moving to the background, see Being a Responsible Background App.

The Background Transition Cycle

When the user presses the Home button, presses the Sleep/Wake button, or the system launches another app, the foreground app transitions to the inactive state and then to the

background state. These transitions result in calls to the app delegate's applicationWillResignActive: and applicationDidEnterBackground:methods, as shown in Figure 4-5. After returning from the applicationDidEnterBackground: method, most apps move to the suspended state shortly afterward. Apps that request specific background tasks (such as playing music) or that request a little extra execution time from the system may continue to run for a while longer.

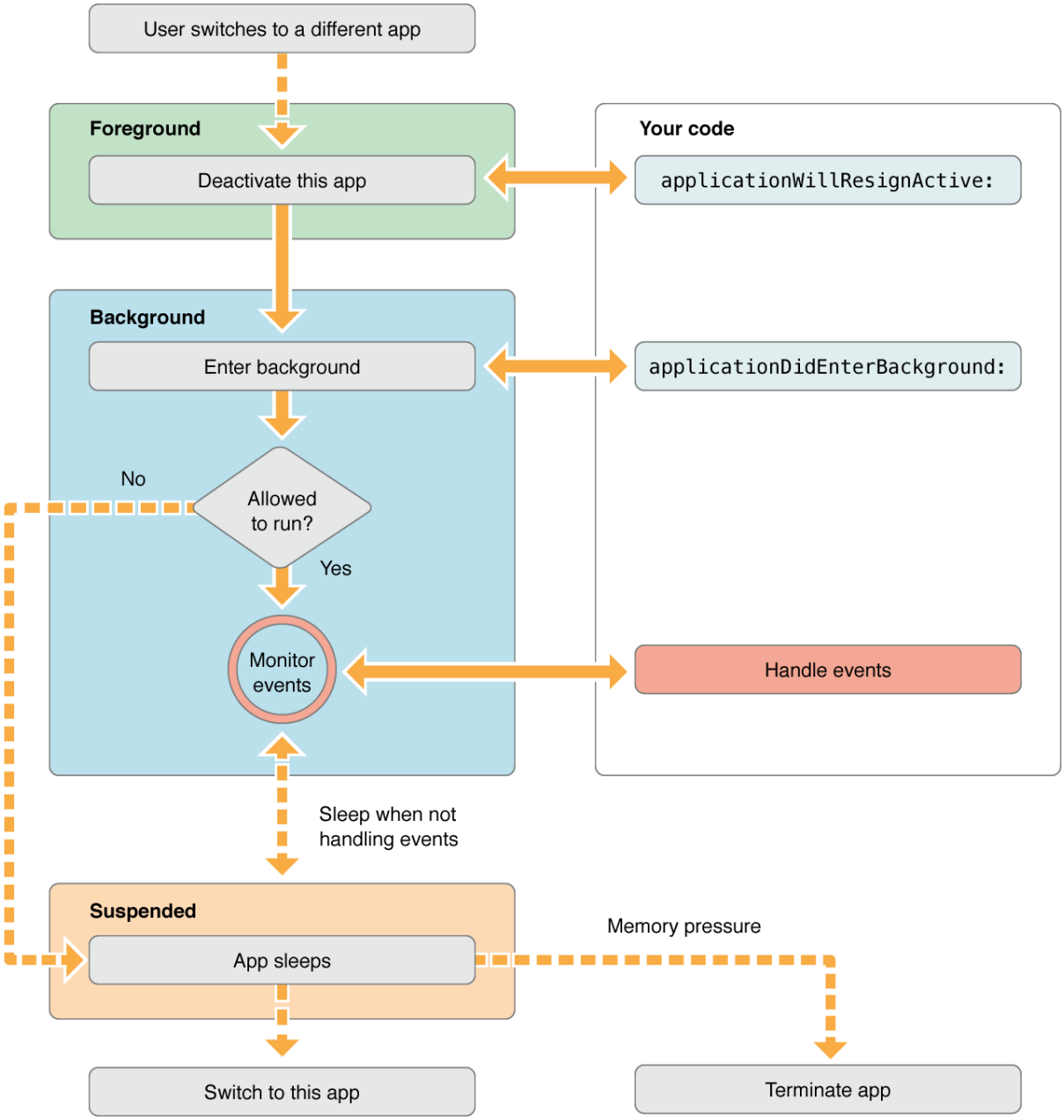


Fig: Moving from the foreground to the background

Shortly after an app delegate's applicationDidEnterBackground: method returns, the system takes a snapshot of the app's windows. Similarly, when an app is woken up to perform background tasks, the system may take a new snapshot to reflect any relevant changes. For example, when an app is woken to process downloaded items, the system takes a new snapshot so that can reflect any changes caused by the incorporation of the items. The system uses these snapshot images in the multitasking UI to show the state of your app.

If you make changes to your views upon entering the background, you can call the snapshotViewAfterScreenUpdates: method of your main view to force those changes to be rendered. Calling the setNeedsDisplay method on a view is ineffective for snapshots because the snapshot is taken before the next drawing cycle, thus preventing any changes from being rendered. Calling the snapshotViewAfterScreenUpdates: method with a value of YES forces an immediate update to the underlying buffers that the snapshot machinery uses.

Memory Usage

Apps are encouraged to use as little memory as possible so that the system may keep more apps in memory or dedicate more memory to foreground apps that truly need it. There is a direct correlation between the amount of free memory available to the system and the relative performance of your app. Less free memory means that the system is more likely to have trouble fulfilling future memory requests.

To ensure there is always enough free memory available, you should minimize your app's memory usage and be responsive when the system asks you to free up memory.

Observe Low-Memory Warnings

When the system dispatches a low-memory warning to your app, *respond immediately*. Low-memory warnings are your opportunity to remove references to objects that you do not need. Responding to these warnings is crucial because apps that fail to do so are more likely to be terminated. The system delivers memory warnings to your app using the following APIs:

- The applicationDidReceiveMemoryWarning: method of your app delegate.
- The didReceiveMemoryWarning method of your UIViewController classes.
- The UIApplicationDidReceiveMemoryWarningNotification notification.
- Dispatch sources of type DISPATCH_SOURCE_TYPE_MEMORYPRESSURE. This technique is the only one that you can use to distinguish the severity of the memory pressure.

Upon receiving any of these warnings, your handler method should respond by immediately freeing up any unneeded memory. Use the warnings to clear out caches and release images. If you have large data structures that are not being used, write those structures to disk and release the in-memory copies of the data.

If your data model includes known purgeable resources, you can have a corresponding manager object register for the UIApplicationDidReceiveMemoryWarningNotification notification and remove strong references to its purgeable resources directly. Handling this notification directly avoids the need to route all memory warning calls through the app delegate.

Note: You can test your app's behavior under low-memory conditions using the Simulate Memory Warning command in iOS Simulator.

Reduce Your App's Memory Footprint

Starting off with a low footprint gives you more room for expanding your app later. Table 7-1 lists some tips on how to reduce your app's overall memory footprint.

Table 7-1 Tips for reducing your app's memory footprint

Tip	**Actions to take**

Eliminate memory leaks.	Because memory is a critical resource in iOS, your app should never have memory leaks. Use the Instruments app to track down leaks in your code, both in Simulator and on actual devices. For more information on using Instruments, see *Instruments User Guide*.
Make resource files as small as possible.	Files reside on disk but must be loaded into memory before they can be used. Compress all image files to make them as small as possible. (To compress PNG images—the preferred image format for iOS apps—use the pngcrush tool.) You can make property list files smaller by writing them out in a binary format using the NSPropertyListSerialization class.
Use Core Data or SQLite for large data sets.	If your app manipulates large amounts of structured data, store it in a Core Data persistent store or in a SQLite database instead of in a flat file. Both Core Data and SQLite provides efficient ways to manage large data sets without requiring the entire set to be in memory all at once.
Load resources lazily.	You should never load a resource file until it is actually needed. Prefetching resource files may seem like a way to save time, but this practice actually slows down your app right away. In addition, if you end up not using the resource, loading it wastes memory for no good purpose.

Allocate Memory Wisely

Table 7-2 lists tips for improving memory usage in your app.

Table 7-2 Tips for allocating memory

Tip	Actions to take
Impose size limits on resources.	Avoid loading a large resource file when a smaller one will do. Instead of using a high-resolution image, use one that is appropriately sized for iOS-based devices. If you must use large resource files, find ways to load only the portion of the file that you need at any given time. For example, rather than load the entire file into memory, use the mmap and munmap functions to map portions of the file into and out of memory. For more information about mapping files into memory, see *File-System Performance Guidelines*.
Avoid unbounded problem sets.	Unbounded problem sets might require an arbitrarily large amount of data to compute. If the set requires more memory than is available, your app may be unable to complete the calculations. Your apps should avoid such sets whenever possible and work on problems with known memory limits.

Background Execution

When the user is not actively using your app, the system moves it to the background state. For many apps, the background state is just a brief stop on the way to the app being suspended. Suspending apps is a way of improving battery life it also allows the system to devote important system resources to the new foreground app that has drawn the user's attention.

Most apps can move to the suspended state easily enough but there are also legitimate reasons for apps to continue running in the background. A hiking app might want to track the user's position over time so that it can display that course overlaid on top of a hiking map. An audio app might need to continue playing music over the lock screen. Other apps might want to download content in the background so that it can minimize the delay in presenting that content to the user. When you find it necessary to keep your app running in the background, iOS helps you do so efficiently and without draining system resources or the user's battery. The techniques offered by iOS fall into three categories:

- Apps that start a short task in the foreground can ask for time to finish that task when the app moves to the background.
- Apps that initiate downloads in the foreground can hand off management of those downloads to the system, thereby allowing the app to be suspended or terminated while the download continues.
- Apps that need to run in the background to support specific types of tasks can declare their support for one or more background execution modes.

Always try to avoid doing any background work unless doing so improves the overall user experience. An app might move to the background because the user launched a different app or because the user locked the device and is not using it right now. In both situations, the user is signaling that your app does not need to be doing any meaningful work right now. Continuing to run in such conditions will only drain the device's battery and might lead the user to force quit your app altogether. So be mindful about the work you do in the background and avoid it when you can.

Additional Discussion:

Aps are a sophisticated interplay between your custom code and the system frameworks. The system frameworks provide the basic infrastructure that all apps need to run, and you provide the code required to customize that infrastructure and give the app the look and feel you want. To do that effectively, it helps to understand a little bit about the iOS infrastructure and how it works.

iOS frameworks rely on design patterns such as mode-view-controller and delegation in their implementation. Understanding those design patterns is crucial to the successful creation of an app. It also helps to be familiar with the Objective-C language and its features. If you are new to iOS programming, read Start Developing iOS Apps (Swift) for an introduction to iOS apps and the Objective-C language.

The Main Function

The entry point for every C-based app is the main function and iOS apps are no different. What is different is that for iOS apps you do not write the main function yourself. Instead,

Xcode creates this function as part of your basic project. Listing 2-1 shows an example of this function.

Reference and bibliography:

https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html

https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/StrategiesforHandlingAppStateTransitions/StrategiesforHandlingAppStateTransitions.html

https://www.imore.com/how-to-manage-background-app-refresh-iphone-ipad

https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html

Module 15: Notifications

Topics:

Local Notifications

Push Notifications

Chapter Overview

Local notifications and remote notifications are ways to inform users when new data becomes available for your app, even when your app is not running in the foreground. For example, a messaging app might let the user know when a new message has arrived, and a calendar app might inform the user of an upcoming appointment. The difference between local and remote notifications is straightforward:

- With *local notifications*, your app configures the notification details locally and passes those details to the system, which then handles the delivery of the notification when your app is not in the foreground. Local notifications are supported on iOS, tvOS, and watchOS.
- With *remote notifications*, you use one of your company's servers to push data to user devices via the Apple Push Notification service. Remote notifications are supported on iOS, tvOS, watchOS, and macOS.

Both local and remote notifications require you to add code to support the scheduling and handling of notifications in your app. For remote notifications, you must also provide a server environment that is capable of receiving data from user devices and sending notification-related data to the *Apple Push Notification service (APNs)*, which is an Apple-provided service that handles the delivery of remote notifications to user devices.

Learning Outcome:

This chapter has looked at

- Some of the tasks that can be performed when an application is in the background.
- How to deal with push up notification.
- How to deal with local Notification.

Local Notification

With *local notifications*, your app configures the notification details locally and passes those details to the system, which then handles the delivery of the notification when your app is not in the foreground. Local notifications are supported on iOS, tvOS, and watchOS

Push Notification

Apple Push Notification Service (commonly referred to as Apple Notification Service or APNs) is a platform notification service created by Apple Inc. that enables third party application developers to send notification data to applications installed on Apple devices. The notification information sent can include badges, sounds, newsstand updates, or custom text alerts. It was first launched with iOS 3.0 on June 17, 2009. APNs support for local applications was later added to the Mac OS X API beginning with the release of Mac OS X 10.7 ("Lion"). Support for website notifications was later added with the release of Mac OS X 10.9 ("Mavericks").

Notification in iOS 10

Apple has been running and upgrading their OS to make notifications more dynamic and actionable. In iOS 9, you could mark reminders as done and reply to a text message directly inside the notification. With iOS 10, notifications are even more interactive and rich.

A lot third-party developers are experimenting with rich notification interactions now, and saw a hint of what's to come during Apple's WWDC 2016 keynote several month ago with a sample Uber notification, that shows you the map of your driver's route and allows you to call your driver or cancel the request without ever jumping into the Uber app.

Apple WWDC 2016 HIghlight

Now, iPhone users can view **photos**, watch **videos**, and listen to **audio** right from inside a notification. The notification can also display a live data and information, so they can see a location point moving in a real time while friend typing a response to a text. Third-party developers can use all of these capabilities to create rich, interactive, and powerful notification experiences for their own apps.

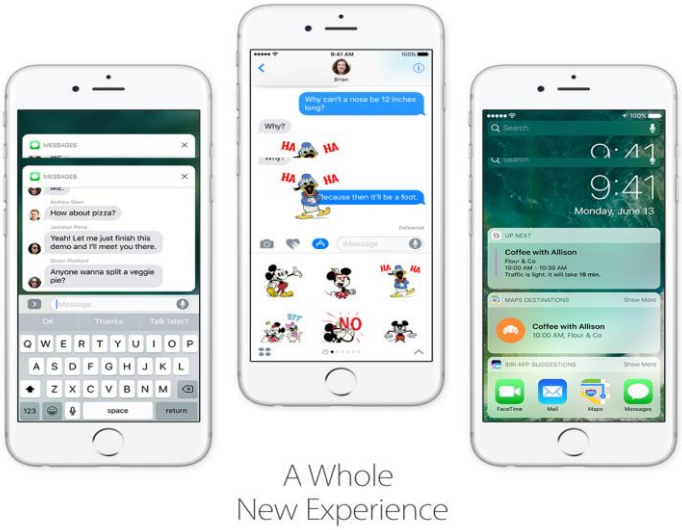


Fig: iPhone Interface

Push Notification on Apple Website

Fig: iPhone Interface

Message Push Notification from Wired.com

There are two new types of content in iOS 10: rich media and custom app interfaces. Rich media means mobile marketers can now add **images**, **GIFs**, and **audio** and **video** files to push notifications (look here for the detail). These push notifications will appear with a thumbnail preview—a visual divergence from iOS 9, which was index-oriented. An added feature is that push notifications can now be **edited**, **deleted**, or **collapsed** via message retraction, which will probably be the savior of multiple mobile marketers this year.

I was also expecting there will be a lot of cat and dog GIFs when marketers implemented this push notification, we know that smart brands will be testing a wide range of rich media types like GIF, tone, and targeting. Marketers especially retailer should consider about pushing products personalized to individual customers and using this feature to show their product. A publisher like media and news might also use rich notification to give a short summary of Breaking News. Whatever the industry, I think the C-level people need to hold a meeting with their engineering team now to make sure that every cute cat GIF finds a mobile home.

When to Use Local and Remote Notifications

Because apps on iOS, tvOS, and watchOS are not always running, local notifications provide a way to alert the user when your app has new information to present. For example, an app that pulls data from a server in the background can schedule a local notification when some interesting piece of information is received. Local notifications are also well suited for apps such as calendar and to-do list apps that need to alert the user at a specific time or when a specific geographic location is reached.

Remote notifications are appropriate when some or all of the app's data is managed by your company's servers. With remote notifications, you decide when you want to push a notification to the user's device. For example, a messaging app would use remote notifications to let users know when new messages arrive. Because they are sent from your server, you can send remote notifications at any time, including when the app is not running on the user's device

Additional Discussion:

Remote Notifications allow applications to receive notifications from a remote server and process that notification in the background. And Unlike the *Remote Notifications* functionality, local notifications allow alerts to be triggered from within the local application without the need to rely on a remote server. In this Module we will discuss about these two types of notification.

Reference:

https://developer.apple.com/documentation/uikit/uilocalnotification
https://developer.apple.com/documentation/pushkit/pkpushregistry
https://developer.apple.com/documentation/uikit/uipushbehavior

Module 16: Core Location Framework

Topics:

Basics

MapKit Framework

Simple App using the module taught

Chapter Overview:

Core Location provides services for determining a device's geographic location, altitude, orientation, or position relative to a nearby iBeacon. The framework uses all available onboard hardware, including Wi-Fi, GPS, Bluetooth, magnetometer, barometer, and cellular hardware to gather data.

Learning Outcome:

This chapter has provided an overview of the use of the iOS Core Location Framework to obtain location information within an iOS application and implementation of Mapkit in application.

The first time that your app requests authorization, its authorization status is indeterminate and the system prompts the user to grant or deny the request (as shown in Figure 1). The system records the user's response and does not display this panel upon subsequent requests.

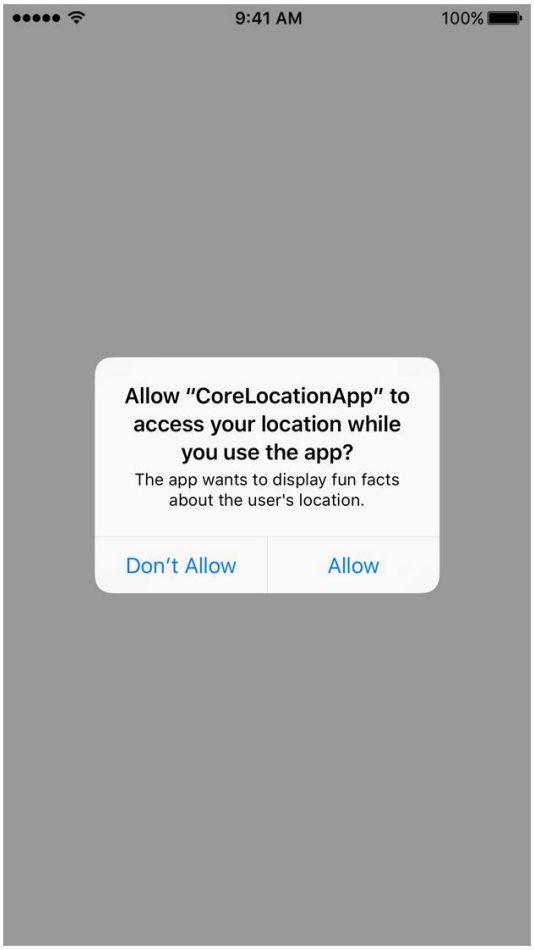


Fig: Requesting authorization to use location services

After requesting permission and determining whether services are available, you start most services using the CLLocationManager object and receive the results in your associated delegate object.

Basics

Core Location provides services for determining a device's geographic location, altitude, orientation, or position relative to a nearby iBeacon. The framework uses all available onboard hardware, including Wi-Fi, GPS, Bluetooth, magnetometer, barometer, and cellular hardware to gather data. The first time that your app requests authorization, its authorization status is indeterminate and the system prompts the user to grant or deny the request (as shown in Figure 1). The system records the user's response and does not display this panel upon subsequent requests.

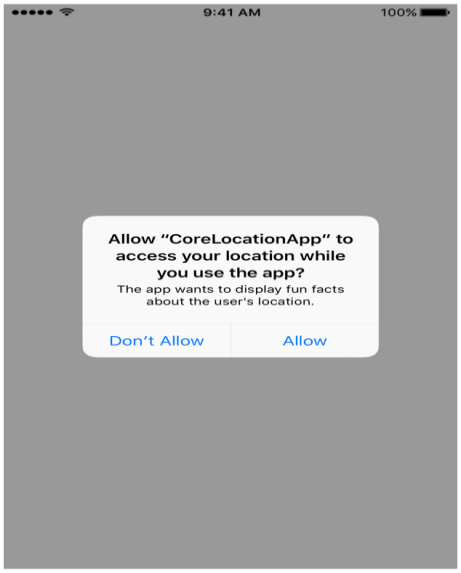


Fig: Requesting authorization to use location services

After requesting permission and determining whether services are available, you start most services using the CLLocationManager object and receive the results in your associated delegate object.

MapKit

Display map or satellite imagery directly from your app's interface, call out points of interest, and determine placemark information for map coordinates.

Core Discussion:

Use the MapKit framework to embed maps directly into your own windows and views. You can add annotations and overlays to the map to call out points of interest or user destinations. You can also provide text completion for users typing in the name of a point of interest.

If your app offers transit directions, you can make your directions available to Maps. You can also use Maps to supplement the directions that you provide in your app. For example, if your app only provides directions for subway travel, you can use Maps to provide walking directions to and from subway stations.

MapKit Framework

The Map Kit framework lets you embed a fully functional map interface into your app. The map support provided by this framework includes many features of the Maps app in both iOS and OS X. You can display standard street-level map information, satellite imagery, or a combination of the two. You can zoom, pan, and pitch the map programmatically, display 3D buildings, and annotate the map with custom information. The Map Kit framework also provides automatic support for the touch events that let users zoom and pan the map.

To use the features of the Map Kit framework, turn on the Maps capability in your Xcode project (doing so also adds the appropriate entitlement to your App ID). Note that the only way to distribute a maps-based app is through the iOS App Store or Mac App Store. If you're unfamiliar with entitlements, code signing, and provisioning, start learning about them in *App Distribution Quick Start*.

Simple App using the module taught

Now this is the time to go make another app using module we learned.

Additional Discussion:

Location services provide a way to improve your app by enhancing user experience. If you're developing a travel app, you can base on the users' current location to search for nearby restaurants or hotels. You can also find the location feature in most of the Photo apps that saves where the pictures are taken. The Core Location framework provides the necessary Objective-C interfaces for obtaining information about the user's location. With the GPS coordinate obtained, you can make use of the API to decode the actual street or utilize the Map framework to further display the location on Map.

References:

https://developer.apple.com/documentation/mapkit

https://www.raywenderlich.com/160517/mapkit-tutorial-getting-started

Module 17: Unit Testing

Topics:

Introduction

XCTest Framework

Xcode Services

Overview:

Writing tests isn't glamorous, but since tests can keep your sparking app from turning into a bug-ridden piece of junk, it sure is necessary. If you're reading this iOS Unit Testing and UI Testing tutorial, you already know you should write tests for your code and UI, but you're not Maybe you already have a "working" app but no tests set up for it, and you want to be able to test any changes when you extend the app. Maybe you have some tests written, but aren't sure whether they're the right tests.

Learning Outcome:

- Unit testing with xCode workflow
- Use of XCTest Framework for Unit testing

Introduction

Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use

XCTest Framework

Create and run unit tests, performance tests, and UI tests for your Xcode project.

Use the XCTest framework to write unit tests for your Xcode projects that integrate seamlessly with Xcode's testing workflow.

Tests assert that certain conditions are satisfied during code execution, and record test failures (with optional messages) if those conditions are not satisfied.

Tests can also measure the performance of blocks of code to check for performance regressions, and can interact with an application's UI to validate user interaction flows.

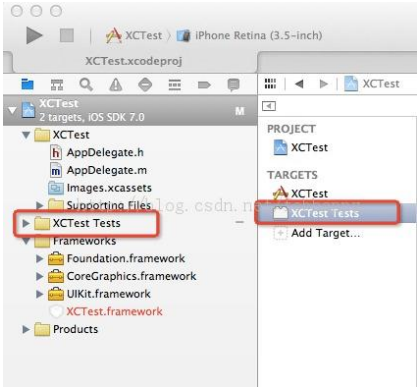


Fig: XCTest

Xcode Services

The Xcode Test Navigator provides the easiest way to work with tests; you'll use it to create test targets and run tests on your app

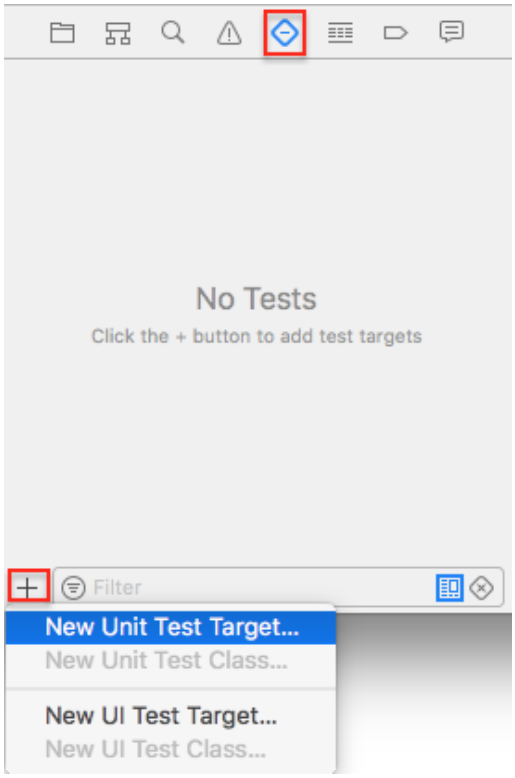


Fig: Services

Additional Discussion:

Use the XCTest framework to write unit tests for your Xcode projects that integrate seamlessly with Xcode's testing workflow.

Tests assert that certain conditions are satisfied during code execution, and record test failures (with optional messages) if those conditions are not satisfied. Tests can also measure the performance of blocks of code to check for performance regressions, and can interact with an application's UI to validate user interaction flows.

Xcode provides you with capabilities for extensive software testing. Testing your projects enhances robustness, reduces bugs, and speeds the acceptance of your products for distribution and sale. Well-tested apps that perform as expected improve user satisfaction. Testing can also help you develop your apps faster and further, with less wasted effort, and can be used to help multiperson development efforts stay coordinated.

Reference:

https://developer.apple.com/library/content/samplecode/UnitTests/Introduction/Intro.html
https://www.raywenderlich.com/150073/ios-unit-testing-and-ui-testing-tutorial
https://medium.com/ios-seminar/the-magic-of-ios-unit-testing-with-xctest-and-swift-3-8889c838b911

Module 18: SpriteKit

Topics:

Introduction to SpriteKit
Simple game using SpriteKit

Chapter Overview:

SpriteKit is a graphics rendering and animation infrastructure that you can use to animate arbitrary textured images, otherwise known as sprites. SpriteKit provides a traditional rendering loop that alternates between determining the contents of and rendering frames. You determine the contents of the frame and how those contents change. SpriteKit does the work to render that frame efficiently using graphics hardware. SpriteKit is optimized for applying arbitrary animations or changes to your content. This design makes SpriteKit more suitable for games and apps that require flexibility in how animations are handled.

Learning Outcome:

- Introduce to SpriteKit
- 2D gaming System
- Made Simple Game using SpriteKit

Introduction to SpriteKit

Sprite Kit is a powerful graphics framework ready-made for developing 2D action games, platformers, puzzle games, and much more. Get introduced to the Sprite Kit API and learn key details about controlling and rendering sprites. Discover how to leverage built-in physics support to make animations look real, and learn about using particle systems to create essential game effects such as fire, snow, explosions, and smoke. This is the first of two must-attend sessions for all developers creating games for iOS or OS X.

The SpriteKit World in Theory

Before you go any further, do you have a moment to talk… about physics?

In SpriteKit you work in two environments. The graphical world that you see on the screen and the physics world, which determines how objects move and interact.

The first thing you need to do when using SpriteKit physics is to change the *world* according to the needs of your game. The world object is the main object in SpriteKit that manages all of the objects and the physics simulation. It also sets up the *gravity* that works on physics bodies added to it. The default gravity is -9.81 thus similar to that of the earth. So, as soon as you add a body it would “fall down”.

Once you have created the world object, you can add things to it that interact according to the principles of physics. For this the most usual way is to create a sprite (graphics) and set its *physics body*. The properties of the body and the world determine how it moves.

Bodies can be *dynamic objects* (balls, ninja stars, birds, …) that move and are influenced by physical forces, or they can be *static objects* (platforms, walls, …) that are not influenced by those forces. When creating a body you can set a ton of different properties like shape, density, friction and many more. Those properties heavily influence how the body behaves within the world.

When defining a body, you might wonder about the *units* of their size and density. Internally SpriteKit uses the metric system (SI units). However within your game you usually do not need to worry about actual forces and mass, as long as you use consistent values.

Once you've added all of the bodies you like to your world, SpriteKit can take over and do the simulation. Now that you have a basic understanding of how things should work, let's see it in code! Time for some Breakout!

The SpriteKit framework adds new features to make it easier to create high-performance, battery-efficient 2D games. With support for custom OpenGL ES shaders and lighting, integration with SceneKit, and advanced new physics effects and animations, you can add force fields, detect collisions, and generate new lighting effects in your games.

Simple game using SpriteKit

Getting Started

Start by creating a new project. For this start up XCode, go to *File\New\Project* and choose the *iOS\Application\SpriteKit Game* template. Set the product name to *BreakoutSpriteKitTutorial*, select *Devices>iPhone* and then click *Next*. Select the location on your hard drive to save your project and then click *Create*.

Your game will need some graphics. You probably want to have at least a graphic for the ball, the paddle, the bricks, and a background image. You can download them from here. Drag and drop the files from Finder on to your XCode project. Make sure that the checkmark for *Copy items into destination group's folder (if needed)*is ticked and press the *Finish* button.

Open *MyScene.m*. This class creates your game scene. The template includes some extra code that you will not need. So, replace the contents of the file with the following:

```
#import "MyScene.h"

static NSString* ballCategoryName = @"ball";
static NSString* paddleCategoryName = @"paddle";
static NSString* blockCategoryName = @"block";
static NSString* blockNodeCategoryName = @"blockNode";

@implementation MyScene

-(id)initWithSize:(CGSize)size {
    if (self = [super initWithSize:size]) {
        SKSpriteNode* background = [SKSpriteNode spriteNodeWithImageNamed:@"bg.png"];
        background.position = CGPointMake(self.frame.size.width/2, self.frame.size.height/2);
        [self addChild:background];
    }
    return self;
}

@end
```

The code here is very basic. First, you define a few constants that will help you to identify the game objects. Then, **initWithSize:** initializes an empty scene with a specified size, creates a sprite using an image, positions it in the middle of the screen, and adds it to the scene.

```
import UIKit
import SpriteKit

class GameViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        if let scene = GameScene(fileNamed:"GameScene.sks") {
            // Configure the view.
            let skView = self.view as! SKView
            skView.showsFPS = true
            skView.showsNodeCount = true

            /* Sprite Kit applies additional optimizations to improve rendering performance */
            skView.ignoresSiblingOrder = false

            /* Set the scale mode to scale to fit the window */
            scene.scaleMode = .AspectFill
            skView.presentScene(scene)
        }
    }
}
```

References:

https://developer.apple.com/spritekit/

https://www.raywenderlich.com/187645/spritekit-tutorial-for-beginners-2

https://www.appcoda.com/spritekit-introduction/

https://code.tutsplus.com/tutorials/spritekit-from-scratch-fundamentals--cms-26326

Module 19: Revenue

Topic:

AdMob Integration

In App Purchase

iAD Integration

Chapter Overview:

This year the iTunes App Store, the first mobile app store launched that set off mobile app development as a business, turns 9. For many indie developers, mid and big size companies it's been a bumpy road with many ups and downs. There-is-an-app-for-that moment has long gone, the focus from app novelty has shifted to innovation and long-term profitability. The app business is matured and has become part and parcel of the world economy. Over the years, several monetization models emerged and lead to establishing numerous mobile advertising networks, affiliate networks as off-app-store channels for app developers to generate apps revenue.

Learning Outcome:

- iOS app revenue system
- How to integrate adMob to your application
- How to integrate iAD
- In-App-purchase system

In this chapter we will talk about:

AdMob Integration

In App Purchase

iAD Integration

AdMob Integration

Among all the mobile ad networks, it is undeniable that Google's AdMob is the most popular one. Similar to iAd, Google provides SDK for developers to embed ads in their iOS app. Google sells the advertising space (e.g. banner) within your app to a bunch of advertisers. You earn ad revenue when a user views or clicks your ads.

To use AdMob in your apps, you will need to use the Google Mobile Ads SDK. The integration is not difficult. To display a simple ad banner, it just takes a few lines of code and you're ready to start making a profit from your app.

There is no better way to learn the AdMob integration than by trying it out. As usual, we'll work on a sample project and then add a banner ad. You can download the Xcode project template from **here**.

Apply a Google AdMob Account

Before you can integrate your apps with Google AdMob, you'll need to first enroll into the AdMob service. Now open the link below using Safari or your favorite browser:

https://www.google.com/admob/

As AdMob is now part of Google, you can simply sign in with your Google account or register a new one. AdMob requires you to have a valid AdSense account and AdWords account. If you don't have one or both of these accounts, follow the sign-up process and connect them to your Google Account.

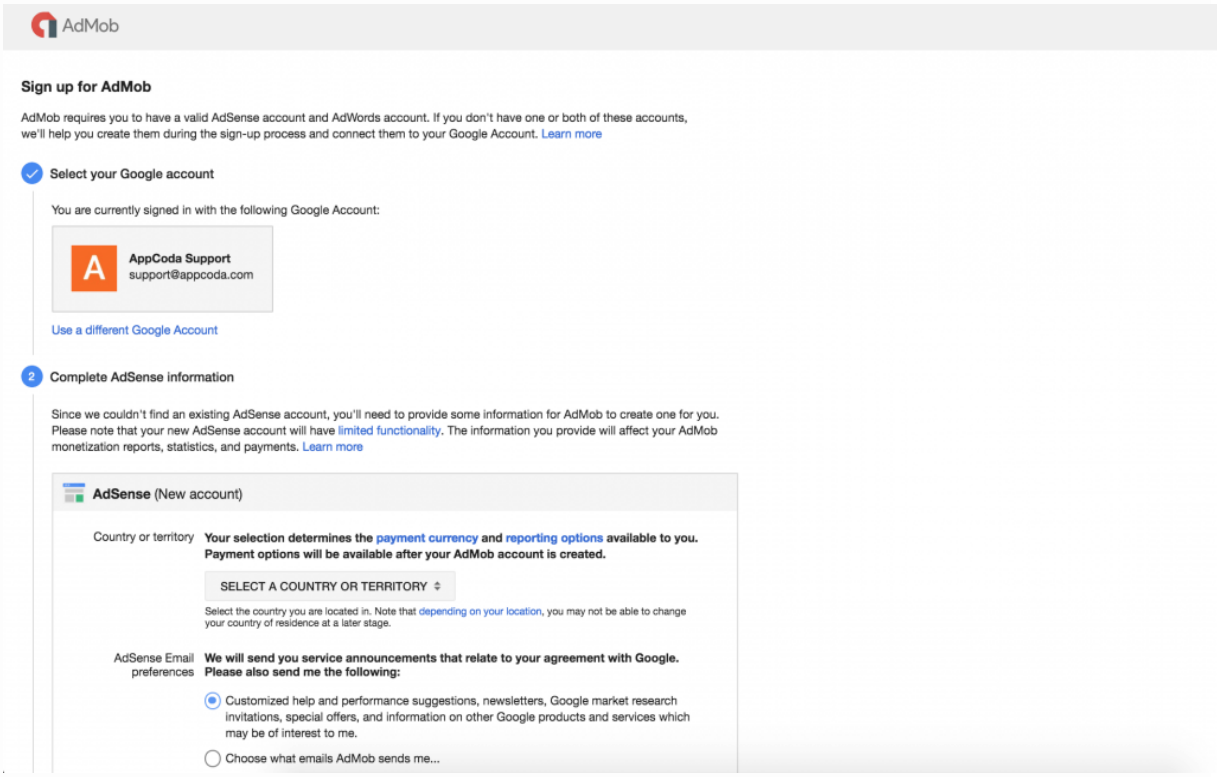


Fig: Google AdMob

Once you finish the registration, you will be brought to the AdMob dashboard. Here you can click the `Monetize New App` button to create a new app in your AdMob account.

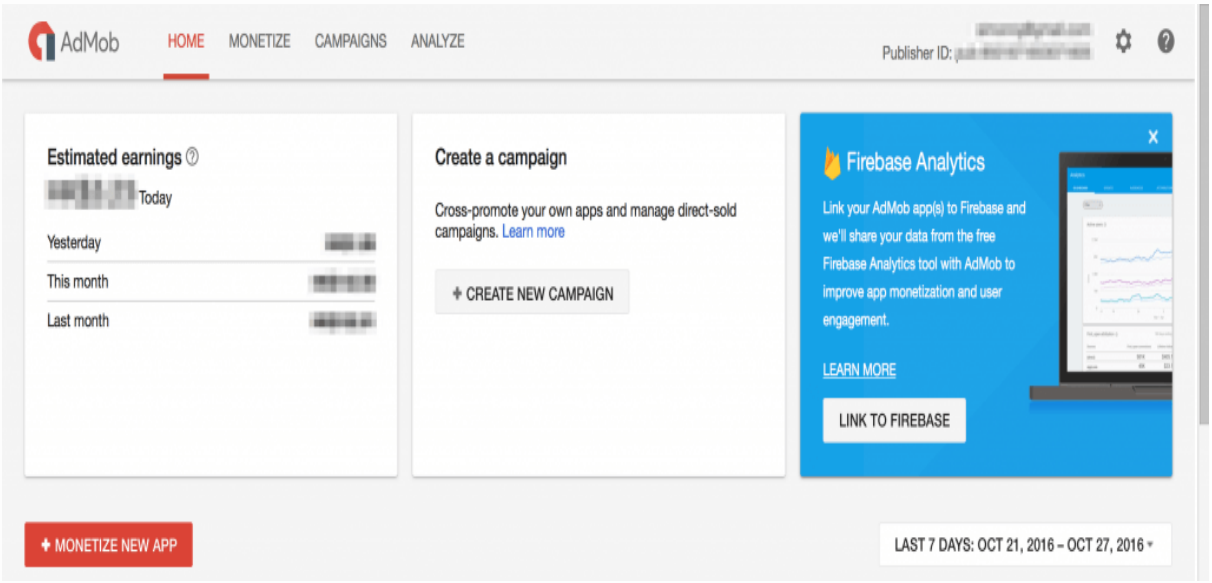


Fig: Google AdMob

In the next screen, choose the *Add Your App Manually* option. We will register the app by filling in the form manually. In future, if you already have an app on the App Store, you can use the search option to retrieve your app.

Set the app name to *GoogleAdMobDemo* and choose iOS for the platform option. Click `Add App` to proceed to the next step. AdMob will then generate an App ID for the app and ask you to choose the supported ad format.

For this demo, we use the banner ad format. So select *Banner* and accept the default options. For the Ad unit name, set it to *AdBanner*.

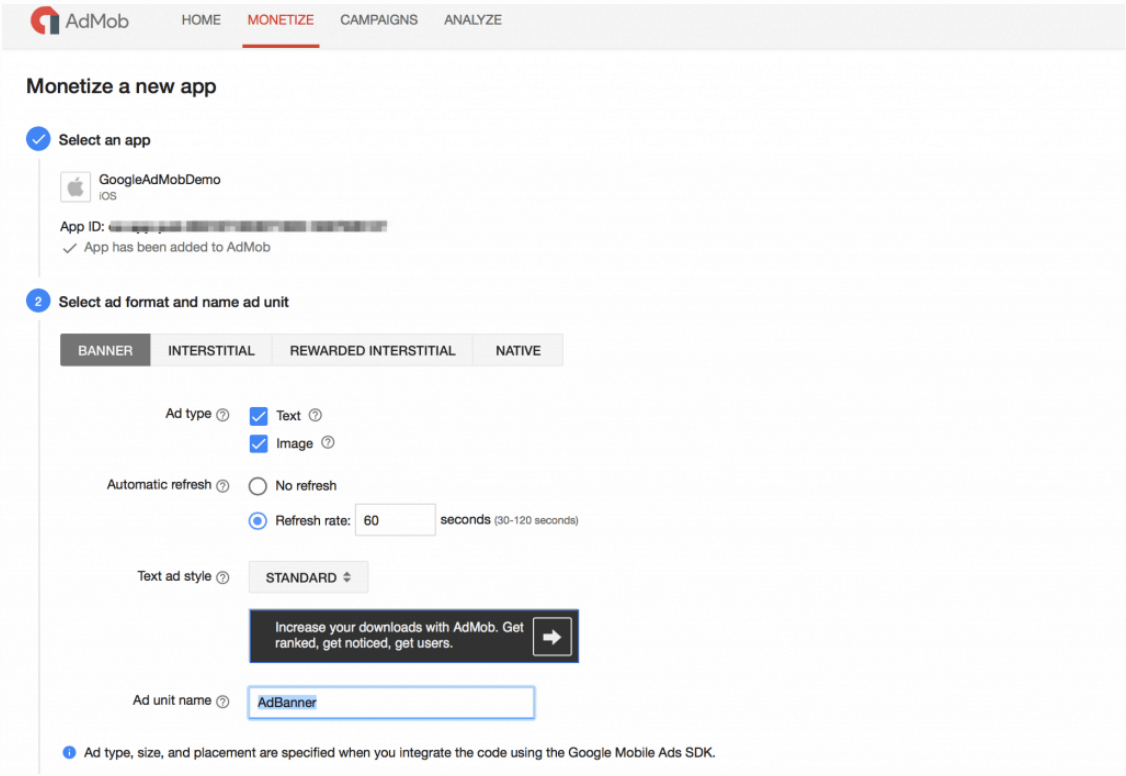


Fig: Google AdMob

Click `Save` to generate the ad unit ID. In the next step, you can just click `Skip` to skip the configuration of Firebase Analytics.

This completes the configuration of your new app. You will find the App ID and Ad unit ID in the implementation instructions. Please save these information. We will need them in the later section when we integrate AdMob with our Xcode project.

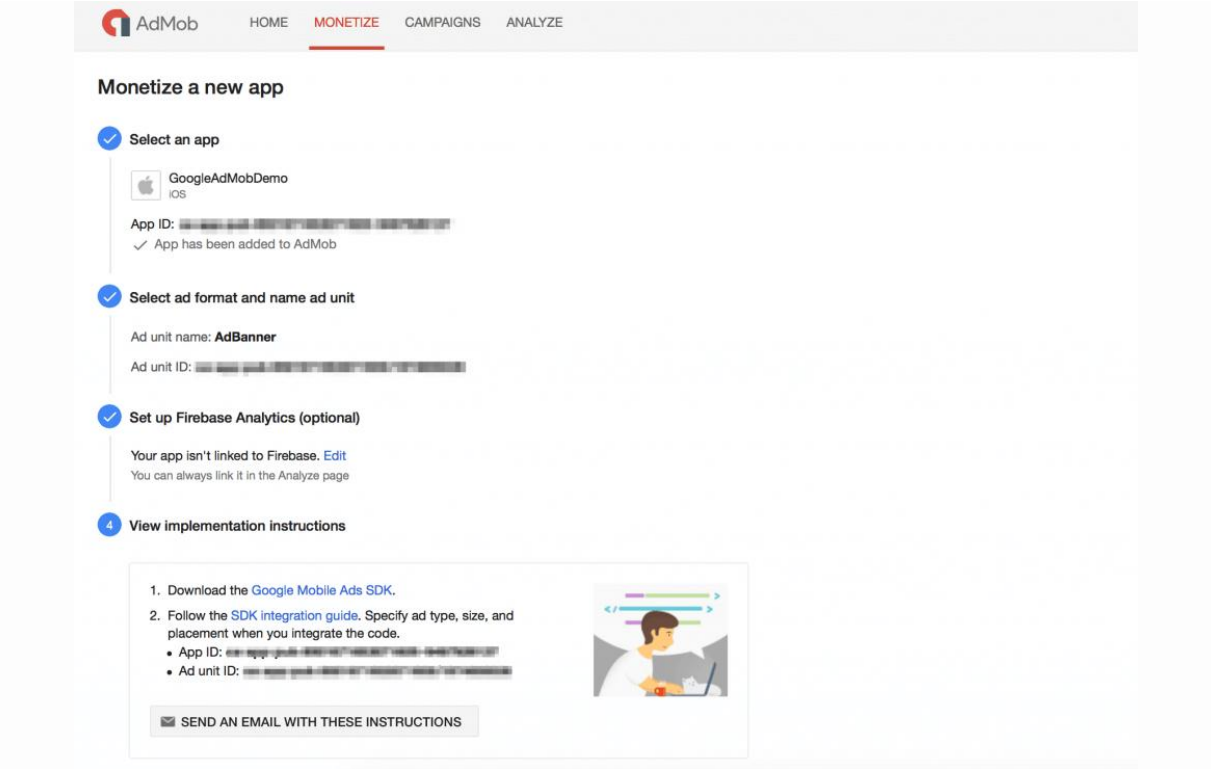


Fig: Google AdMob

Using Google Mobile Ads Framework

Now that you have completed the configuration in AdMob, let's move to the actual implementation. Fire up Xcode and open `GoogleAdDemo.xcworkspace` of the starter project. Please note that it is `GoogleAdDemo.xcworkspace` instead of `GoogleAdDemo.xcodeproj` I suggest you compile and run the project template so that you have a basic idea of the demo app; it's a simple table-based app that displays a list of articles. We will tweak it to show an advertisement to earn some extra revenue.

AdMob, not to be confused with Adsense, is a mobile app advertising platform designed specifically for app developers. AdMob helps app developers monetize their mobile apps by showing ads in their mobile apps.

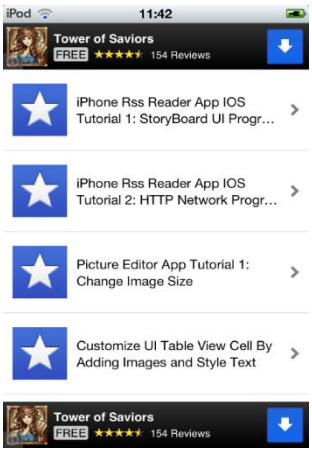


Fig: Google A

In App Purchase

Overview

You can use in-app purchases to sell a variety of content, including subscriptions, new features, and services. There are four in-app purchase types you can offer.

Consumable

Users can purchase different types of consumables, such as lives or gems in a game, to further their progress through an app. Consumable in-app purchases are used once, are depleted, and can be purchased again.

Non-Consumable

Users can purchase non-consumable, premium features within an app. Non-consumables are purchased once and do not expire, such as additional filters in a photo app. Apple can host content associated with your non-consumable in-app purchases.

Auto-Renewable Subscriptions

Users can purchase access to services or periodically updated content, such as monthly access to cloud storage or a weekly subscription to a magazine. Users are charged on a recurring basis until they decide to cancel.

Offering Subscriptions

Non-Renewing Subscriptions

Users can purchase access to services or content for a limited duration, such as a season pass to streaming content. This type of subscription does not renew automatically, so users need to renew each time.

See What's New

Watch WWDC 2017 session videos to see the latest on in-app purchases, including promoting in-app purchases and new StoreKit APIs.

- What's New in StoreKit
- Advanced StoreKit

Freemium Business Model

Learn how developers across a range of categories approach using the freemium model in their apps.

- Using the Freemium Model

Customer Payment Methods

The App Store makes it easy to offer your content to users around the world. Users can pay for your apps and in-app purchases with credit or debit cards, carrier billing, digital wallets, and App Store and iTunes gift cards, depending on their region. Learn more

Preparing

Before you can offer in-app purchases, you'll need to sign the Paid Applications Agreement and set up your banking and tax information.

iTunes Connect Developer Help: Agreements, tax, and banking overview

Set Up Xcode Configurations

Use Xcode to enable the in-app purchase service for your app.

Xcode Help: Add a capability to a target

Create Your In-App Purchases in iTunes Connect

Configure your in-app purchases in iTunes Connect, and include details such as name, pricing, and description that highlights the features and functionality of your in-app purchase. You can also create and maintain your in-app purchases using XML.

iTunes Connect Developer Help: Create an in-app purchaseMetadata Specification for XML

Designing and Building

Design Your In-App Purchase Experience

The user interface for your in-app purchase should fit well with the rest of your app and effectively showcase your products.

Human Interface GuidelinesDesigning a Great In-App Purchase ExperienceOptimizing In-App Purchases

Implement Your In-App Purchases

Use the StoreKit Framework to embed the in-app purchase into your app and securely process purchases of content and services. Make sure to complete the steps in the implementation checklist.

In-App Purchase Programming GuideStoreKit API ReferenceUsing StoreKit for In-App Purchases with Swift 3What's New in StoreKitAdvanced StoreKit

Validate Receipts

Receipts provide a valuable record of the sale. Consider using receipt validation code to protect your content and prevent unauthorized purchases.

Receipt Validation Programming GuidePreventing Unauthorized Purchases with Receipts

Testing

Test Transactions

Use the Apple sandbox testing environment to test your in-app purchases without incurring charges.

iTunes Connect Developer Help: Create a Sandbox Tester AccountIn-App Purchase Programming Guide: Suggested Testing Steps

Test the Full User Experience

Use TestFlight to gain valuable feedback on your app and in-app purchases from a wider audience before releasing your app on the App Store. Invite users on your team in iTunes Connect, and up to external 10,000 testers using just their email address. All in-app purchases are free during the beta testing period, and will not carry over when the testing period is over.

TestFlight Beta Testing

Publishing

Submit Your In-App Purchases for Review

Once you've completed testing, verify that you've followed the App Review Guidelines and implementation checklist, then submit using iTunes Connect.

App Store Review GuidelinesImplementation ChecklistiTunes Connect Developer Help: Submit an in-app purchase

Promote Your In-App Purchases on the App Store

With iOS 11, you can choose to promote up to 20 in-app purchases at a time on your product page, increasing discoverability for content previously only found inside your app. Users can browse in-app purchases directly on the App Store and start a purchase even before downloading your app.

Promoting Your In-App Purchases

Distribute Promo Codes

Give press and influencers early access to your app's in-app purchases with promo codes from iTunes Connect. You can give away up to 100 promo codes for each in-app purchase item, up to a maximum of 1,000 codes per app.

You can use in-app purchases to sell a variety of content, including subscriptions, new features, and services.

There are four in-app purchase types you can offer.

Consumable

Users can purchase different types of consumables, such as lives or gems in a game, to further their progress through an app.

Consumable in-app purchases are used once, are depleted, and can be purchased again.

Non-Consumable

Users can purchase non-consumable, premium features within an app. Non-consumables are purchased once and do not expire, such as additional filters in a photo app.

Apple can host content associated with your non-consumable in-app purchases.

Auto-Renewable Subscriptions

Users can purchase access to services or periodically updated content, such as monthly access to cloud storage or a weekly subscription to a magazine.

Users are charged on a recurring basis until they decide to cancel.

Non-Renewing Subscriptions

Users can purchase access to services or content for a limited duration, such as a season pass to streaming content. This type of subscription does not renew

Automatically, so users need to renew each time.

iAD Integration

About the iAd App Network Shutdown

As of December 31, 2016, the iAd App Network is no longer available.

Promoting Your App

If you'd like to promote your apps, you can advertise using Search Ads, Apple News, or third party networks and advertising sellers.

Updating Your App

Apps that have implemented the iAd.Framework classes should not crash solely because of the deprecation. You can remove the iAd framework and connection on your next regular app update or submission.

The following classes and their APIs in the iAd SDK are deprecated:

- ADBannerView
- ADInterstitialAd
- UIViewController (iAdAdditions)
- AVPlayerViewController (iAdPreroll)
- MPMoviePlayerController (iAdPreroll)
- ADBannerViewDelegate
- ADInterstitialAdDelegate

Apps that are not updated will experience the following issues:

- No ads will be returned in your app.
- A deprecation warning will appear in Xcode.
- If your app implements ADBannerView and the ADBannerViewDelegate, the bannerView:didFailToReceiveAdWithError: delegate method, it will be called with the ADError enum.
- If your app implements ADInterstitialAd and the ADInterstitialAdDelegate, the interstitialAd:didFailWithError: delegate method will be called with an ADError enum.
- If your app implements the iAdAdditions category on UIViewController, setting the ADInterstitialPresentationPolicy will have no effect and calling requestInterstitialAdPresentation: or canDisplayBannerAds: will always return NO.
- If your app implements the iAdPreroll category on AVPlayerViewController or the iAdPreroll category on MPMoviePlayerController, calling playPrerollAdWithCompletionHandler: will return an ADError enum.
- When your ADBannerView delegate receives this error, your app should look for other ads (from other ad-networks).
- Apps that implement the iAdAdditions category on UIViewController and use automatic presentation, should not notice any impact. Apps that manually manage the ADBannerView may see a blank space.
- If you have implemented a mediation layer, contact the mediation provider to understand impact to your app.
- ADClient APIs (requestAttributionDetailsWithBlock and addClientToSegments) will continue to work.

iAd is used to display ads, served by the apple server. iAd helps us in earning revenue from an iOS application.

References:

https://developers.google.com/admob/android/quick-start
https://developer.apple.com/documentation/iad
https://developer.apple.com/in-app-purchase/

Module 20: Advanced iOS Concepts

Topic:

iBeacon

Localization

Chapter Overview: iBeacon is Apple's version of the Bluetooth-based beacon concept, which allows Bluetooth devices to broadcast or receive tiny and static pieces of data within short distances. In simplistic words, it consists of two parts: a broadcaster (beacon device) and a receiver (smartphone app). The broadcaster is always advertising "I am here, and my ID is…", while the receiver detects these Bluetooth radio packets and does whatever it needs to do based on how close or far it is from them.

Learning Outcome:

- Apple Advance devices
- iBeacone and how iBeacon can make easy out life
- Introduce with Localization Process

iBeacon

Think of beacons as "buttons or links to the physical world around you". In the same way that web pages rely on buttons as a primary way of user interaction, beacons are used by apps to trigger events and call actions, allowing users to interact with digital or physical things, such as door locks, discounts, automation systems or simple notifications.

From a technical point of view, you can think of iBeacons as small digital lighthouses, just like those used to indicate where a port of shoreline is. Normally, the observer/receiver is an iOS app, while the broadcaster/transmitter can be a battery-powered sensor, an USB Bluetooth dongle, an Arduino kit, a Mac computer or an iOS device. The broadcaster side only sends data. The standard beacon advertisement consists of an UUID, a major and a minor value only. For example:

UUID: B9407F30-F5F8-466E-AFF9-25556B57FE6D
Major ID: 1
Minor ID: 2

The broadcaster (iBeacon) doesn't do anything else besides sending this piece of information every fraction of a second or so. The UUID is an unique identifier. For example, if Starbucks decides to deploy beacon sensors inside its store and make an app that can tell the user once they arrive at a specific store, they would define a UUID that is unique to their app and the beacons inside their stores. Inside the stores, they would place beacon devices and configure each of them to use a different "minor" value. For example, at the store A, they would place all beacon devices broadcasting the Starbucks UUID, major value 1, minor 1 near the door, minor 2 near the mugs display and minor value 3 near the cashier. At store B, they would use the same UUID, but major 2 and minor values according to the location inside the store.

With the information broadcasted by each beacon, the app can detect them and tell how close (or far) the phone is from each of them and then perform actions, display alerts to the user, offer discounts, turn lights on and off, open doors and so on.

Apps can use region monitoring to be notified when a user crosses geographic boundaries or when a user enters or exits the vicinity of a beacon. While a beacon is in range of an iOS device, apps can also monitor for the relative distance to the beacon. You can use these capabilities to develop many types of innovative location-based apps. Because geographical

regions and beacon regions differ, the type of region monitoring you decide to use will likely depend on the use case of your app.

In iOS, regions associated with your app are tracked at all times, including when the app isn't running. If a region boundary is crossed while an app isn't running, that app is relaunched into the background to handle the event. Similarly, if the app is suspended when the event occurs, it's woken up and given a short amount of time (around 10 seconds) to handle the event. When necessary, an app can request more background execution time using the beginBackgroundTaskWithExpirationHandler: method of the UIApplication class.

In OS X, region monitoring works only while the app is running (either in the foreground or background) and the user's system is awake. As a result, the system doesn't launch apps to deliver region-related notifications.

Localization:

Localization Overview

We are going to create a small application called "Book Store", which will displays the details of a book with a cover image, title, author and description. The original app is in English. We'll localize the app together and make it available in French. So at the end of the tutorial, you'll have an app that supports two localizations: English and French.

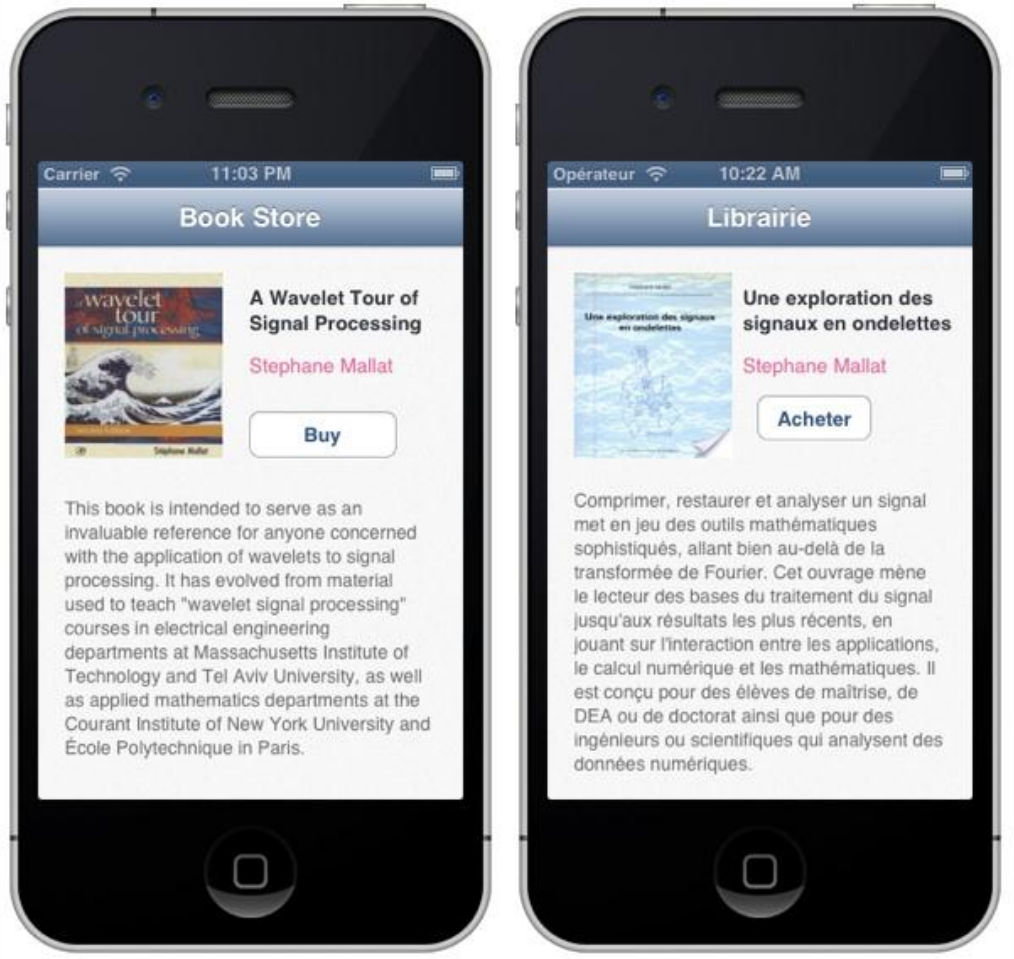


Fig: Localization overview

To make the app available in French, we'll demonstrate how to localize:

- the text in the storyboards
- the cover image
- the dynamic message displayed by code
- the name of the app as displayed in home screen

For those of you who don't know how an app selects a localization, it refers to the language settings of iOS (General > International > Language). An app uses this preference as the key for accessing the localized resources for the requested language.

Note: To make the tutorial as simple as possible, the app do not have any functionalities and the book data is static. Also, please note you'll need to use Xcode v4.6 or up.

Getting Started

It's time to set up the project by creating a new project based on the Single View Application iOS app template. In the project options, you'll need to check the storyboard and Automatic Reference Counting Boxes. This app will provide only iPhone Views, so choose iPhone for the device family and save the project. Open the Storyboard, design the user interface as shown below:

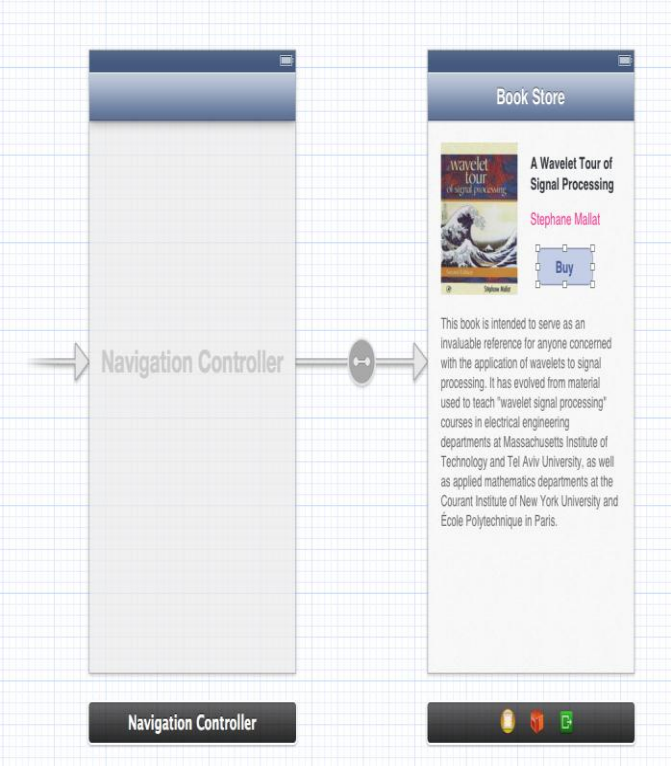


Fig: Storyboards of BookStore

The focus of this tutorial is on Localization. So to save you time from setting up the project,

Localizing the Storyboard

Now, let's start to localize the project. In Xcode, click on Book Store in the project navigator and select the Info tab of the project. Look for the localizations section in the Info tab. Apparently, it shows a single localization: English. Check the "Use Base Internalization" checkbox.

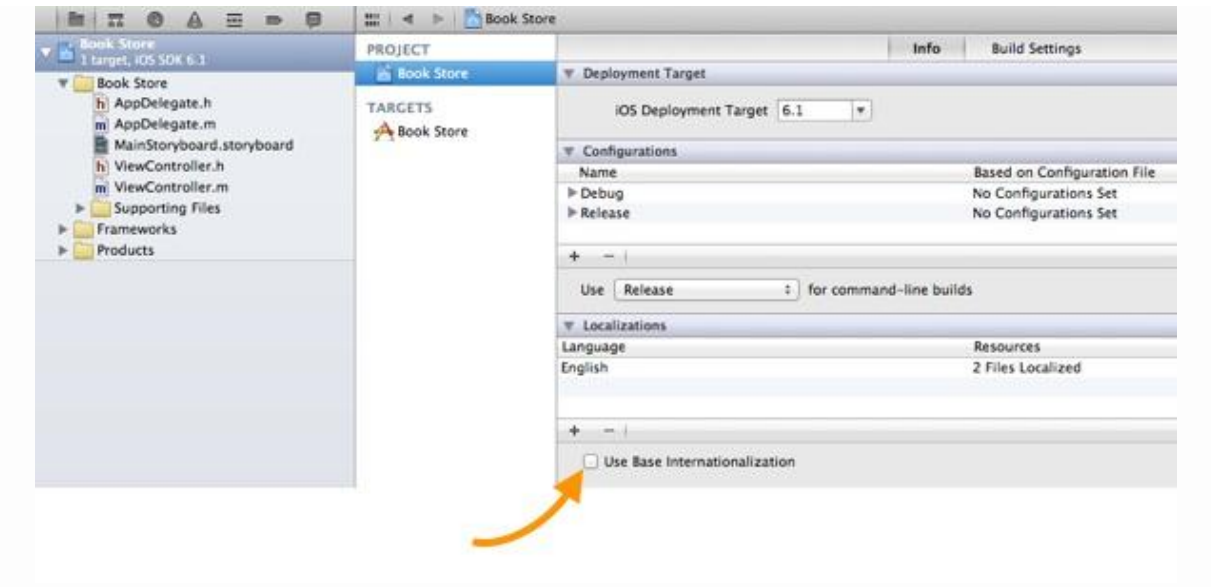


Fig: Storyboards

You'll then be prompted to select the storyboard to use for the base internationalization.

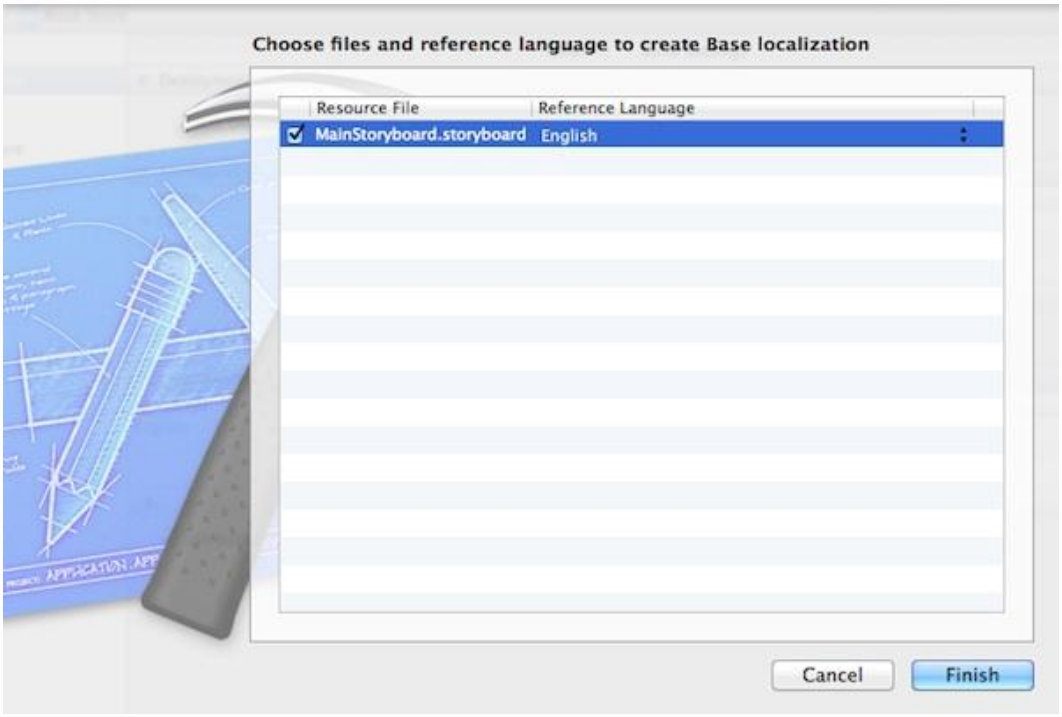


Fig: Storyboards

Make sure that MainStoryboard.storyboard is selected and that the reference language is English. Click the Finish button.

What we just did is to create a base storyboard. The concept of base internationalization was first introduced in Xcode 4.5. An app has just one set of storyboards that is localized to the default language. These storyboards is known as the base internationalization. Later when we create a localization, Xcode automatically generates a strings file containing all the text in the base storyboard.

Why Base Internationalization?

Before Xcode 4.5, there is no such concept of base internationalization. Xcode replicates the whole set of storyboards for each localization. Say, your app is localized into 5 languages. Xcode generates 5 sets of storyboards for localization purpose. There is a major drawback for this localization process. When you need to add a new UI element in the storyboards, you'll need to add the same element for each localization. That's a tedious process. This is the reason why Apple introduced base internationalization in Xcode 4.5.

With the base storyboard configured, it's now ready to add a localization for French. First, select the view controller in Storyboard. Then switch back to the project info. Click the + button at the bottom of the Localizations section and choose French from the popup list.

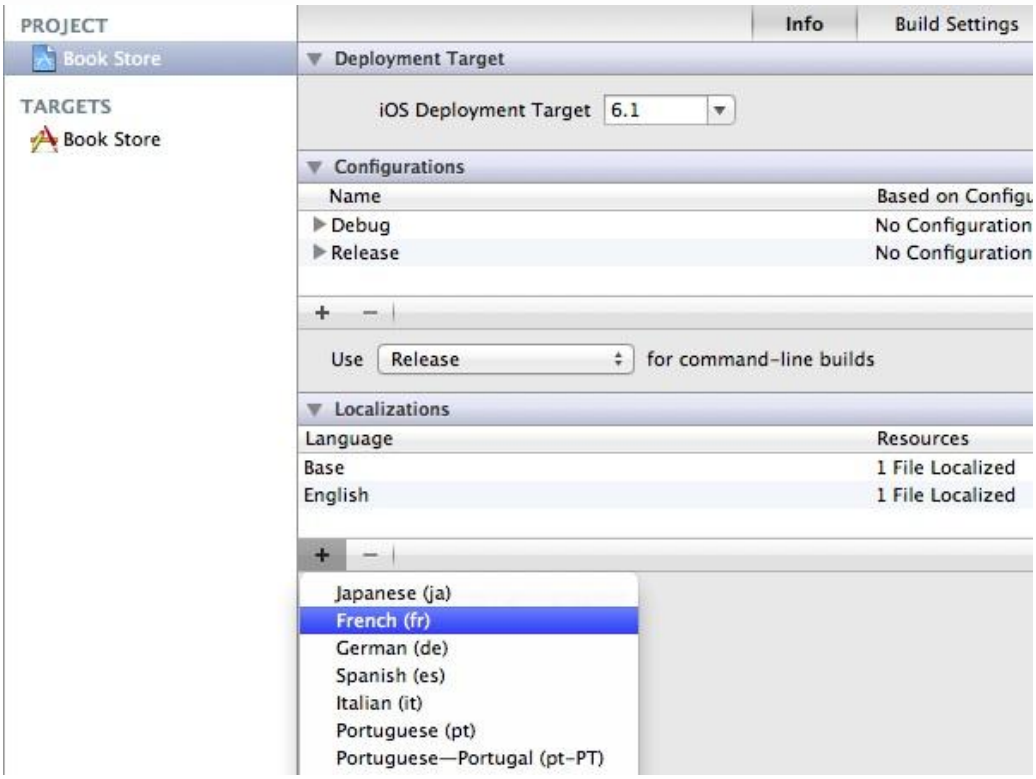


Fig: Storyboards

Adding a new localization

Next, you'll be asked to choose the resource files and language for creating the French localization. You can refer to the below screenshot for the selected options. Leave all the files checked and click Finish.

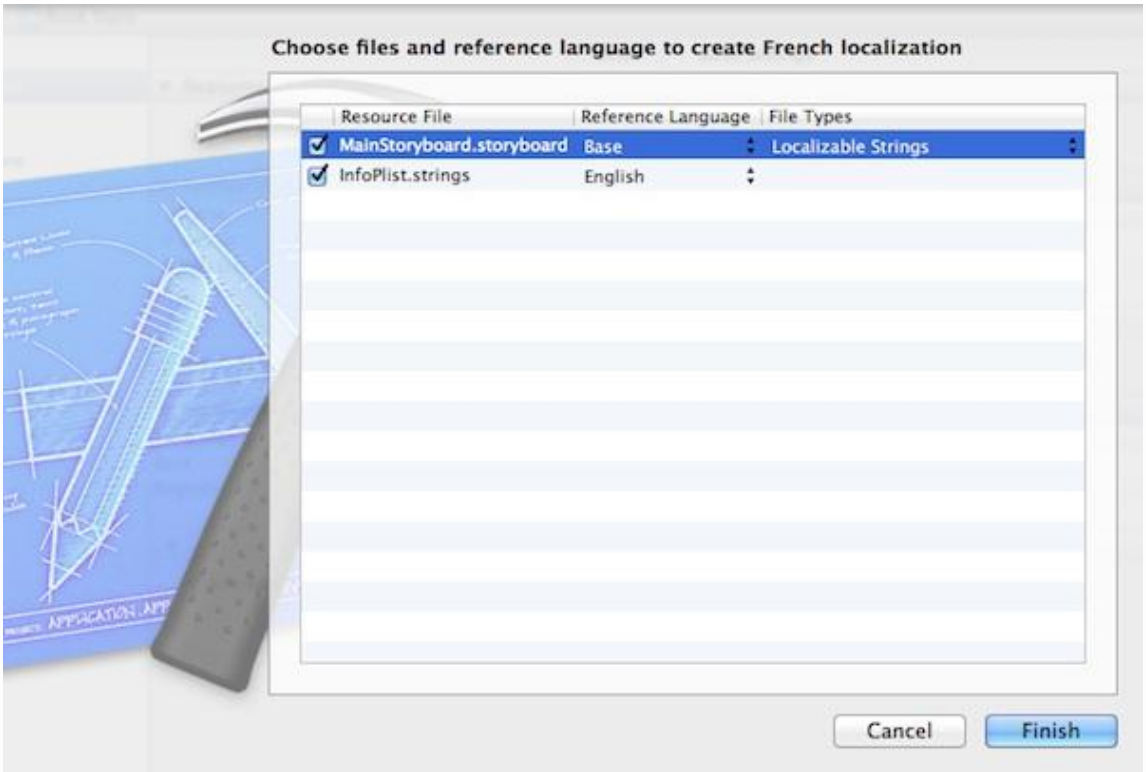


Fig: Storyboards

Once done, you should notice in the project navigator that there is now a triangle symbol next to the MainStoryboard.storyboard. Expand it and you'll find both the base storyboard and a strings file of the storyboard in French.

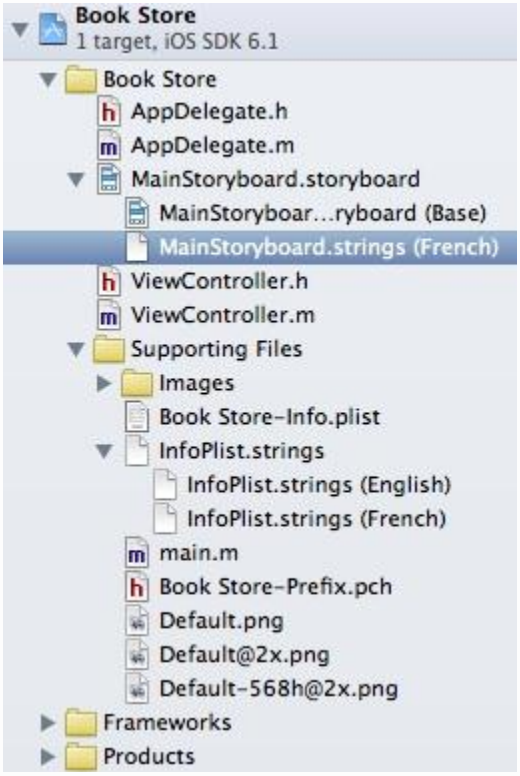


Fig: Storyboards

Select the french strings file (i.e. MainStoryboard.strings (French)) and you should find a list of strings in key/value pairs:

```
/* Class = "IBUILabel"; text = "Stephane Mallat"; ObjectID = "0N3-up-Ts6"; */
"0N3-up-Ts6.text" = "Stephane Mallat";


/* Class = "IBUINavigationItem"; title = "Book Store"; ObjectID = "9EY-vY-nyh"; */
"9EY-vY-nyh.title" = "Book Store";

/* Class = "IBUILabel"; text = "This book is intended to serve as an invaluable reference for anyone concerned with the application of wavelets to signal processing. It has evolved from material used to teach \"wavelet signal processing\" courses in electrical engineering departments at Massachusetts Institute of Technology and Tel Aviv University, as well as applied mathematics departments at the Courant Institute of New York University and École Polytechnique in Paris. "; ObjectID = "gkh-KY-W1e"; */
"gkh-KY-W1e.text" = "This book is intended to serve as an invaluable reference for anyone concerned with the application of wavelets to signal processing. It has evolved from material used to teach \"wavelet signal processing\" courses in electrical engineering departments at Massachusetts Institute of Technology and Tel Aviv University, as well as applied mathematics departments at the Courant Institute of New York University and École Polytechnique in Paris. ";

/* Class = "IBUIButton"; normalTitle = "Buy"; ObjectID = "jul-3l-6x7"; */
"jul-3l-6x7.normalTitle" = "Buy";


/* Class = "IBUILabel"; text = "A Wavelet Tour of Signal Processing"; ObjectID = "xi7-zh-v4N"; */
"xi7-zh-v4N.text" = "A Wavelet Tour of Signal Processing";
```

When we add a new localization, Xcode scans through the base storyboard, extracts those textual items to be localized and put them into the strings file. As you can see above, the visible strings of storyboards such as label, title of the navigation bar and button title are put into the strings file. All entries are in key/value pairs. The first part of key is the object ID of the UI item. You can find the object ID of the UI object under Identity Inspector. For instance, the object ID of the Buy button is shown here:

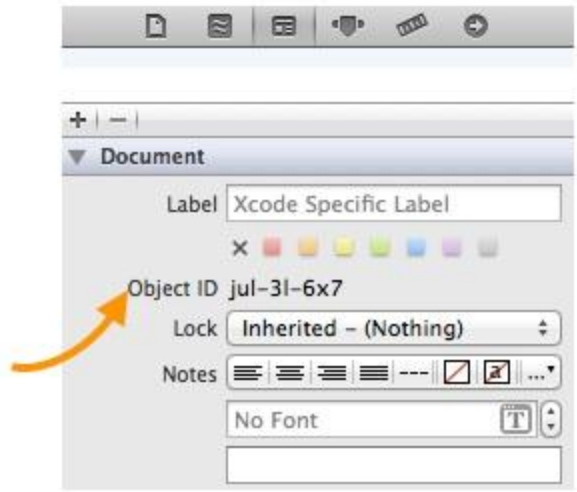


Fig: Storyboards

With the strings file generated, your job is to translate all the values into French. The contents of the MainStoryboard.strings file for the French localization will look like this:

```
/* Class = "IBUILabel"; text = "Stephane Mallat"; ObjectID = "0N3-up-Ts6"; */
"0N3-up-Ts6.text" = "Stephane Mallat";

/* Class = "IBUINavigationItem"; title = "Book Store"; ObjectID = "9EY-vY-nyh"; */
"9EY-vY-nyh.title" = "Librairie";


/* Class = "IBUILabel"; text = "This book is intended to serve as an invaluable reference for anyone concerned with the application of wavelets to signal processing. It has evolved from material used to teach \"wavelet signal processing\" courses in electrical engineering departments at Massachusetts Institute of Technology and Tel Aviv University, as well as applied mathematics departments at the Courant Institute of New York University and École Polytechnique in Paris. "; ObjectID = "gkh-KY-W1e"; */
"gkh-KY-W1e.text" = "Comprimer, restaurer et analyser un signal met en jeu des outils mathématiques sophistiqués, allant bien au-delà de la transformée de Fourier. Cet ouvrage mène le lecteur des bases du traitement du signal jusqu'aux résultats les plus récents, en jouant sur l'interaction entre les applications, le calcul numérique et les mathématiques. Il est conçu pour des élèves de maîtrise, de DEA ou de doctorat ainsi que pour des ingénieurs ou scientifiques qui analysent des données numériques.";

/* Class = "IBUIButton"; normalTitle = "Buy"; ObjectID = "jul-3l-6x7"; */
"jul-3l-6x7.normalTitle" = "Acheter";

/* Class = "IBUILabel"; text = "A Wavelet Tour of Signal Processing"; ObjectID = "xi7-zh-v4N"; */
"xi7-zh-v4N.text" = "Une exploration des signaux en ondelettes";
```

Localizing Image

It's very easy to localize image using Xcode. As a demo, we will localize the cover image. Select the "cover.jpeg" in the project navigator. Next, bring up the file inspector, and locate the

localization section, in which you'll see a button named "Localize…". Click the button and you'll be prompted for confirmation. Choose English and click the "Localize" button to confirm.

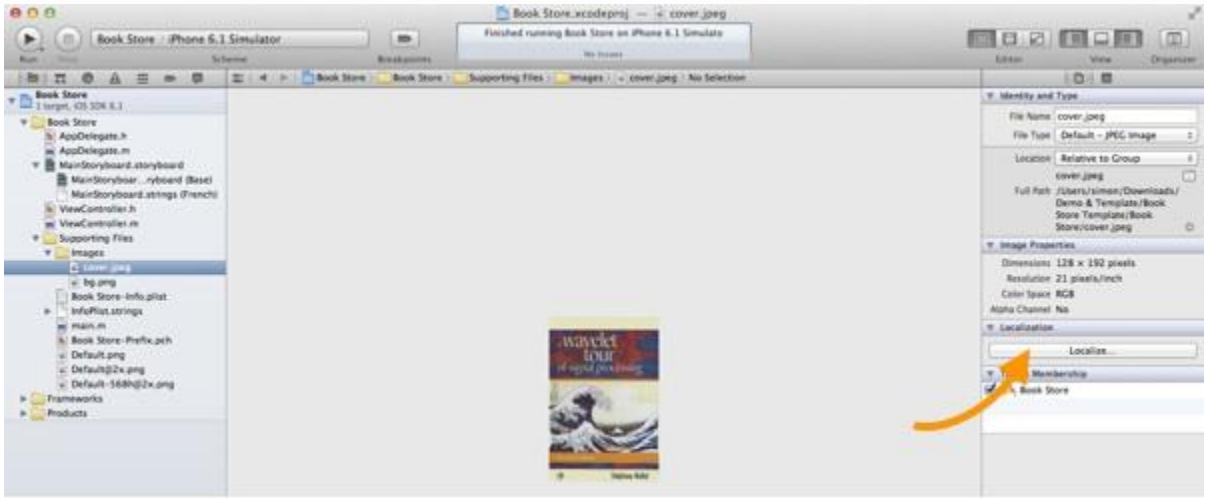


Fig: Storyboards

Localizing Cover Image

After that, you'll see the French option in the localization section. Select the "French" localization and you'll find two versions of cover.jpeg.

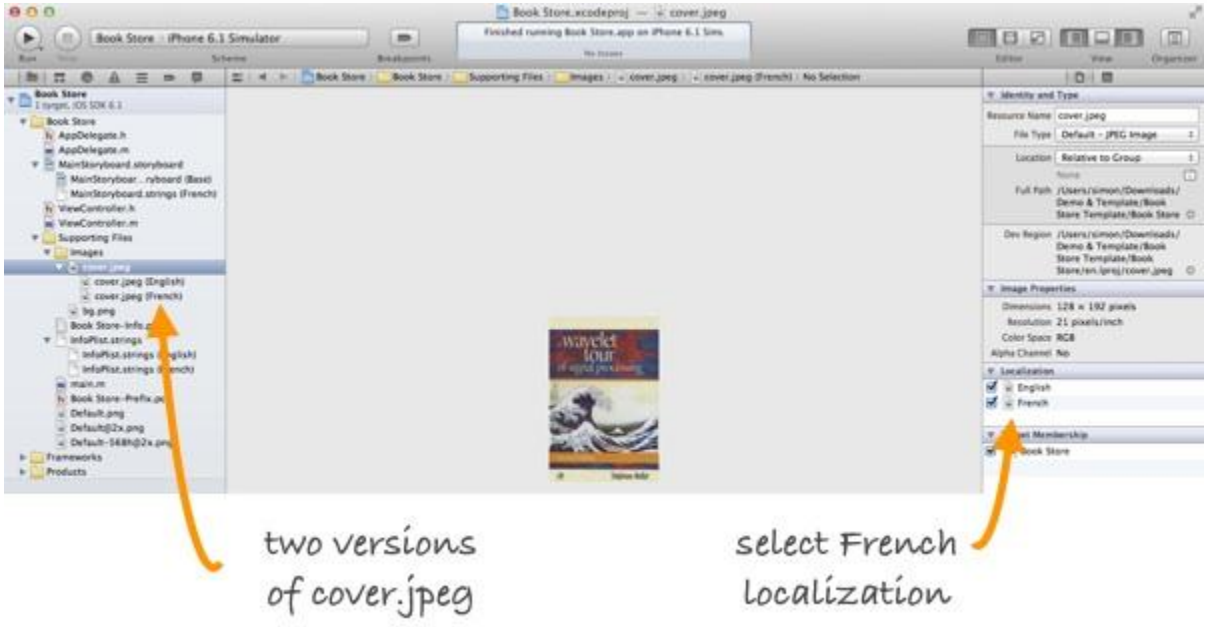


Fig: Storyboards

Adding French Localization for Image

Switch back to the finder and local the project directory. You'll find two folders: en.lproj and fr.lproj. Both folders are automatically generated by Xcode for localization. The en.lproj folder stores resource files for English localization, while fr.lproj folder is for French localization. If you look into both folders, each one contains the cover.jpeg file. Download the French version of the cover image from **here** (or use whatever image you like). Copy the cover image just downloaded and replace the one in fr.lproj folder.

Now it's time to test out the app. If you've got everything right, here are what you'll see in English and French.

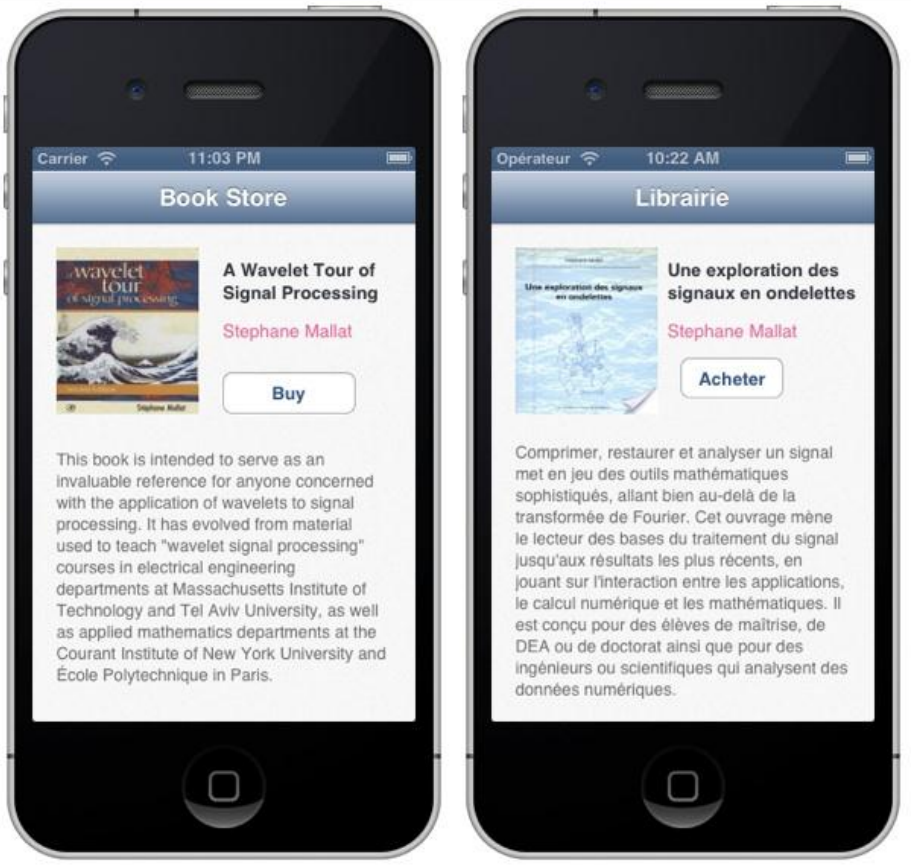


Fig: Storyboards

Tip: To change the language in iPhone Simulator, click home button. Select Settings > General > Language and select French.

Localizing Dynamic Strings

So far we just demonstrated the localization of static items. But in many places in applications, you create NSString instances dynamically or display string literals to the user. These strings will also need to be localized in a localized application.

To demo how to localize text displayed by code, I've added a few lines of code for the "Buy" action:

```
1 - (IBAction)buy:(id)sender
2 {
3     [[[UIAlertView alloc] initWithTitle:@"Confirmation"
4                                 message:NSLocalizedString(@"BOOK_PURCHASE", @"Message")
5                                delegate:nil
6                       cancelButtonTitle:@"OK"
7                       otherButtonTitles:nil] show];
8 }
```

The code is simple. It just shows up an alert displaying the "Thanks for Your Purchase!" message. But instead of using a hardcoded message, we use the NSLocalizedString macro to get the string. The NSLocalizedString macro fetches a localized string from the

Localizable.strings file for the current localization. It takes two arguments: a key and a comment. At runtime, the macro returns the value for the key in the localization that corresponds to the user's preferred language. The comment is for your own reference only.

If you run the app and tap the Buy button, you'll only see an alert view with "BOOK_PURCHASE" message as we haven't created the Localizable.strings file. This is the default file for storing localized strings that are used by code.

Now let's create the Localizable.strings file. In the project navigator, create a new file under the Supporting Files folder. Select the Strings file template and click the Next button.

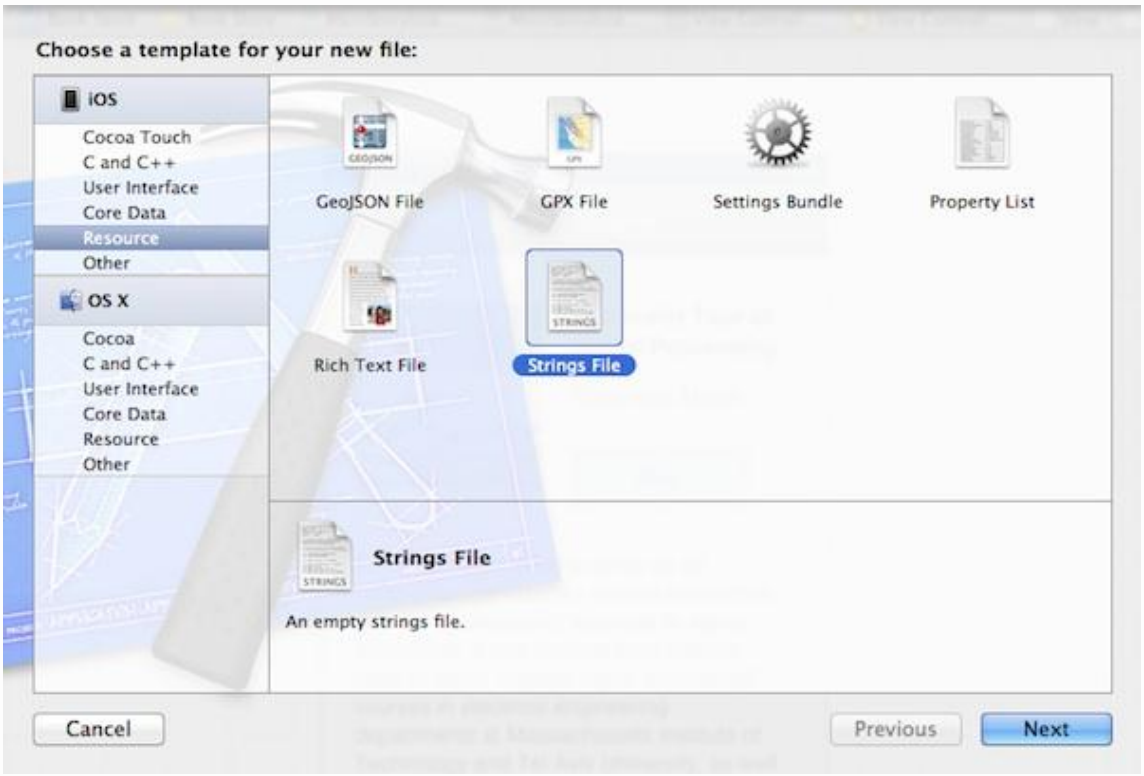


Fig: Storyboards

When prompted, save the file as Localizable.strings. Next, we'll use the similar procedures to localize the file. Select the Localizable.strings and click the "Localize…" button in the File Inspector.

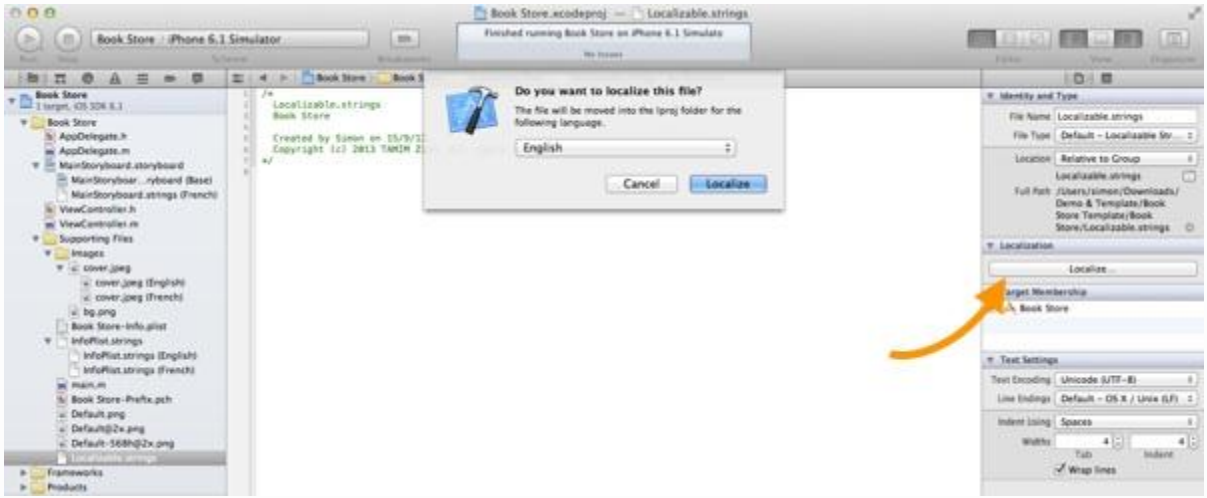


Fig: Storyboards

When the confirmation dialog appears, make sure that English is selected and then click the Localize button. Next, select the French checkbox to add a Localizable.strings file to the French localization. If you've got everything correct, you should have two version of Localizable.Strings file.

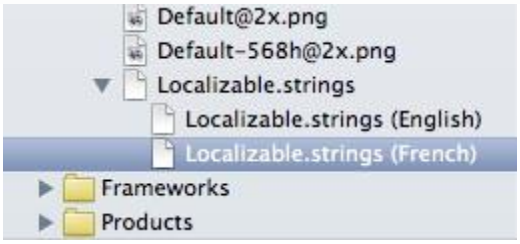


Fig: Storyboards

Localizable.strings File in English and French

Put the following message in the English version of Localizable.strings file:

```
1 "BOOK_PURCHASE" = "Thanks for the Purchase!";
```

Translate the message and put it in the French version of Localizable.strings file:

```
1 "BOOK_PURCHASE" = "merci pour l'achat!";
```

You're done. Compile and run the app again. Depending on the language setting, you should get different alert message when tapping the Buy button.

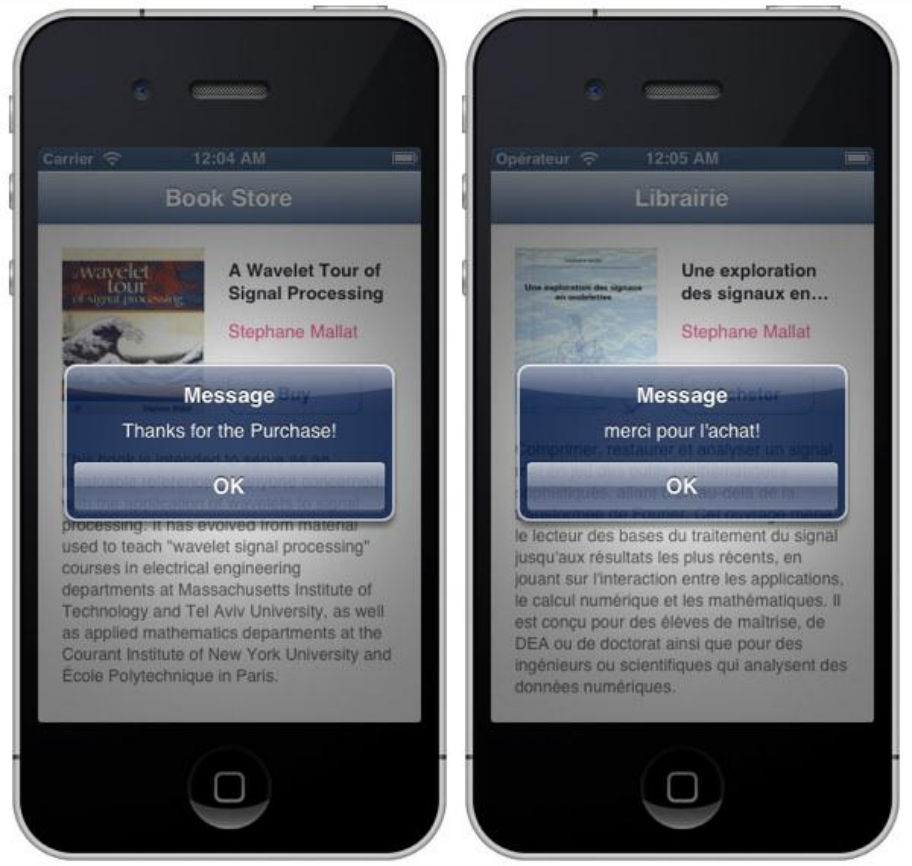


Fig: Storyboards

Localizing the App Name

In some cases, you may also want to localize the app name. When we create the French localization, Xcode already generates the French version of InfoPlist.strings. You can put key/value pairs in the InfoPlist.string file to override the values stored in Info.plist file.

So to display a different app name in the French version, simply add the following string in the InfoPlist.string (French) file:

```
1 "CFBundleDisplayName" = "Librairie";
```

This will change the name of app as appeared on the springboard. Compile and run the app again to test it out.

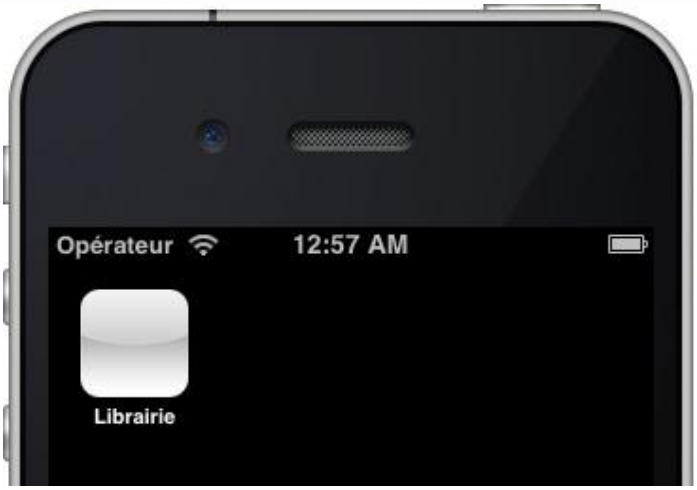


Fig: App Name in French

Tip: Try to clean the build and reset the iPhone Simulator if the app name doesn't change.

Summary

In this tutorial, we covered the localization process in iOS programming. Now you should have a basic idea of localizing an iPhone app.

For your complete reference, you can **download the source code of the Xcode project**. Please remember that the project should be run on Xcode 4.6 or up.

As always, please leave us comment and share your thought about the tutorial.

Localization is the process of making your app support other languages. In many cases, you make your app with English user interface first and then localize the app to other languages such as Japanese.
The process of localization is tedious, and steps of it change little by little as XCode gets updated. In this blog post I am going to explain the whole steps based on the latest XCode which is XCode 7. 3.1.

Reference:

https://developer.apple.com/ibeacon/

http://www.ibeacon.com/what-is-ibeacon-a-guide-to-beacons/

Module 21: Deployment

Topics:

Uploading to AppStore
Details of iTunes Connect, App ID, Provisioning, Certificate
Generating IPA and Testing over TestFlight

Chapter Overview:

Overview iPad and iPhone devices can transform your business and how your employees work. They can significantly boost productivity and give your employees the freedom and flexibility to work in new ways, whether in the office or on the go. Embracing this modern way of working leads to benefits across the entire organization. Users have better access to information, so they feel empowered and are able to creatively solve problems. By supporting iOS, IT departments are viewed as shaping the business strategy and solving real-world problems, rather than just fixing technology and cutting costs. Ultimately everyone benefits, with a reinvigorated workforce and new business opportunities everywhere. Setting up and deploying iPad and iPhone throughout your business has never been easier. With key programs from Apple and a third-party mobile device management solution, your organization can easily deploy iOS devices and content at scale. • Mobile device management (MDM) allows you to configure and manage your devices, and wirelessly distribute and manage your apps. • The Device Enrollment Program (DEP) automates enrollment of Apple devices into your MDM solution to streamline deployment. • The Volume Purchase Program (VPP) lets you purchase apps and books in bulk and distribute them to users. This document offers guidance on deploying iOS devices in your organization and helps you create a deployment plan that best suits your environment. These programs and tools, described in the Deployment Steps section of this overview, are covered in greater detail in the online iOS Deployment Reference.

Learning Outcome:

- Define the role of Mobile Device Management (MDM), Volume Purchase Program (VPP), and Device Enrollment Program (DEP) in the successful deployment of iOS devices
- Identify and respond to challenges when deploying and managing devices without MDM
- Explain what an Apple ID is and identify when it's needed
- Discuss how security is relevant in the context of a given deployment scenario
- Design and implement a deployment solution for a given scenario
- Configure DEP registered devices for over-the-air enrollment
- Configure and manage settings, apps, and content on user-owned and company-owned devices

Uploading to AppStore

App store (or app marketplace) is a type of **digital distribution** platform for iOS apps. We need to upload our app here if we want to someone use our app. And there is a process for uploading app.

Details of iTunes Connect, App ID, Provisioning, Certificate

You have to create a record in iTunes Connect before you can upload an app for distribution on the App Store. This record includes all the information that needed to manage the app through the distribution process and appears on the store for the app.

An app ID is a string used to specify an app, or set of apps. An app ID's primary use is to specify which apps are authorized to be signed and launched. An app ID has two parts: the team ID followed by the bundle ID search string. The team ID is a 10-character string generated by Apple. Each development team is assigned a unique team ID used to identify all your apps. A bundle ID search string is traditionally a reverse-domain-name style string.

A provisioning profile is a collection of digital entities that uniquely ties developers and devices to an authorized iPhone Development Team and enables a device to be used for testing. A Development Provisioning Profile must be installed on each device on which you wish to run your application code.

Certificates Apple Developer Program membership is required to request, download, and use signing certificates issued by Apple. You must also be the Team Agent or an admin of your development team to request distribution certificates used for submitting apps to the App Store or Mac App Store.

Generating IPA and Testing over TestFlight

You create an archive of your app regardless of the type of distribution method you select. Xcode archives allow you to build your app and store it, along with critical debugging information, in a bundle that’s managed by Xcode. Choose Product > Archive to create archive. Xcode will create and store archive in Archives organizer.

Deployment Steps

This section provides a more detailed look at each of the four steps for deploying devices and content: preparing the environment, setting up devices, deploying them, and managing them. Again, the steps you use will depend on whether the organization or the user owns the devices.

Prepare after identifying the right deployment method for your organization, follow these steps to lay the groundwork for deployment; you can take these actions even before you have your devices in hand. Evaluate your infrastructure iPhone and iPad integrate seamlessly into most standard enterprise IT environments. It’s important to assess your existing network infrastructure to make sure your organization takes full advantage of everything that iOS offers. Wi-Fi and networking Consistent and dependable access to a wireless network is critical to setting up and configuring iOS devices. Confirm that your company’s Wi-Fi network can support multiple devices with simultaneous connections from all your users. You might need to configure your web proxy or firewall ports if devices are unable to access Apple’s activation servers, iCloud, or the iTunes Store. Apple and Cisco have also optimized how iPhone and iPad communicate with a Cisco wireless network, paving the way for other advanced networking features such as fast roaming and Quality of Service (QoS). Administrator User Prepare Evaluate your infrastructure Select an MDM solution Enroll in Apple Deployment Programs Use Apple ID, iTunes Store, and iCloud accounts, if applicable Set up Configure devices Distribute apps and books Opt in to company’s MDM solution Download and install apps and books Deploy Accept invitation to VPP No administrator action necessary No user action necessary Manage Administer devices Deploy and manage additional content

Discover additional apps to use iOS Deployment Overview for Business

Evaluate your VPN infrastructure to make sure users are able to securely access company resources remotely via their iOS devices. Consider using the VPN On Demand feature of iOS so that a VPN connection is initiated only when needed. If you plan to use Per-App VPN, make sure your VPN gateways support these capabilities and you purchase sufficient licenses to cover the appropriate number of users and connections. You should also make sure that your network infrastructure is set up to work correctly with Bonjour, Apple's standards-based, zero-configuration network protocol. Bonjour enables devices to find services on a network automatically. iOS devices use Bonjour to connect to AirPrint-compatible printers and AirPlay-compatible devices, such as Apple TV. Some apps also use Bonjour to discover other devices for collaboration and sharing.

For more detail on Wi-Fi and networking for enterprise deployments, see the iOS Deployment Reference: help.apple.com/deployment/ios

Learn more about Bonjour: www.apple.com/support/bonjour

Mail, contacts, and calendars If you use Microsoft Exchange, verify that the ActiveSync service is up to date and configured to support all users on the network. If you're using the cloud-based Office 365, ensure that you have sufficient licenses to support the anticipated number of iOS devices that will be connected. If you don't use Exchange, iOS also works with standards-based servers, including IMAP, POP, SMTP, CalDAV, CardDAV, and LDAP. Caching Server An integrated feature of macOS Server, Caching Server stores a local copy of frequently requested content from Apple servers, helping to minimize the amount of bandwidth needed to download content on your network. Caching Server speeds up the download and delivery of software through the App Store, the Mac App Store, the iTunes Store, and the iBooks Store. It can also cache software updates for faster downloading to iOS devices.

Learn more about Caching Server: www.apple.com/macos/server/features/#caching-server

iTunes support iTunes isn't required for devices using iOS 5 or later, but you might want to support it so users can activate devices, sync media, or back up their devices to a computer. iTunes supports several deployment configuration options that are appropriate for enterprise use, including disabling access to explicit content, defining which network services users can access within iTunes, and determining whether new software updates are available for users to install. Select an MDM solution The Apple management framework for iOS gives organizations the ability to securely enroll devices in the corporate environment, wirelessly configure and update settings, monitor policy compliance, deploy apps and books, and remotely wipe or lock managed devices. These management features are enabled by third-party MDM solutions. A variety of third-party MDM solutions are available to support different server platforms. Each solution offers different management consoles, features, and pricing. Before choosing a solution, review the resources listed below to evaluate which management features are most relevant to your organization. In addition to third-party MDM solutions, a solution from Apple is available called Profile Manager, a feature of macOS Server.

Learn more about MDM: www.apple.com/ipad/business/it/management.html

Learn more about Profile Manager: www.apple.com/macos/server/features/#profile-manager

iOS Deployment Overview for Business

Enroll in Apple Deployment Programs Apple Deployment Programs are a suite of programs that make it easy to manage your devices and content. The program agent is the highest-level administrator for these programs and has full administrative control of the Apple Deployment Programs portal for your organization. If you are new to Apple Deployment Programs, the account created during enrollment will be your program agent account. The same program agent account can be used to enroll in each program. Device Enrollment Program DEP provides a fast, streamlined way to deploy organization-owned iOS and macOS devices that are purchased directly from Apple or participating Apple Authorized Resellers or carriers. You can simplify initial setup by automating MDM enrollment and supervising devices without having to physically touch or prepare them before users get them. And you can further simplify the setup process for users by removing specific steps in Setup Assistant, so users are up and running quickly. You can also control whether or not the user may remove the MDM profile from the device.

Learn more about the Device Enrollment Program: www.apple.com/business/dep

Volume Purchase Program VPP allows businesses to purchase iOS apps and books in volume and distribute them to employees. You can pay with a corp Configuring devices with MDM To enable management, securely enroll your devices with an MDM server using a configuration profile—an XML file that allows you to distribute configuration information to an iOS device. These profiles automate the configuration of settings, accounts, restrictions, and credentials; they can be delivered through MDM if you need to configure many devices and prefer a low-touch, over-the-air deployment.

Profiles can also be sent as an email attachment, downloaded from a web page, or installed on devices through Apple Configurator 2 on Organization-owned devices. Use DEP to enable automatic MDM enrollment of your users' devices upon activation. You can also end the MDM relationship with a device by using the MDM console to remove the configuration profile that contains the MDM server information.

User-owned devices. Employees can decide whether or not to enroll their device in MDM. And to disassociate from MDM at any time, they simply remove the configuration profile from their device. But you should consider incentives for users to remain managed. For example, you might require users to enroll in MDM to get Wi-Fi network access—using your MDM solution to automatically provide the wireless credentials. Once a device is enrolled, an administrator can initiate an MDM policy, option, or command. Then the iOS device receives notification of the administrator's action via the Apple Push Notification service (APNs), so it can communicate directly with its MDM server over a secure connection. With a network connection, devices can receive APNs commands anywhere in the world. However, no confidential or proprietary information is transmitted via APNs.

Configuring devices with Apple Configurator 2 (optional)

Accelerate your initial deployments with the completely redesigned Apple Configurator 2. This free macOS application allows you to update iOS devices to the latest version of iOS, configure device settings and restrictions, and install apps and other content. After initial setup, you can continue to manage everything over the air using MDM. Apple Configurator 2 has an entirely new user interface focused on your devices and the discrete tasks you want to perform on them. The application integrates seamlessly with DEP, enabling devices to automatically enroll in MDM using DEP settings. Custom workflows can be created within Apple Configurator 2 using Blueprints to combine discrete tasks.

Learn more about Apple Configurator 2: help.apple.com/configurator/mac/2.0/

Supervised devices Supervision provides additional management capabilities for iOS devices owned by your organization, allowing restrictions such as disabling AirDrop or placing the device in Single App Mode. It also provides the ability to enable a web filter via a global proxy to ensure that the users' web traffic stays within the organization's guideline, prevent users from resetting their device to factory defaults, and many more. By default, all iOS devices are nonsupervised. You can use DEP to enable supervised mode, or you can manually enable supervision using Apple Configurator 2. Even if you don't plan to use any supervised-only features now, consider supervising your devices when you set them up, so you can take advantage of supervised-only features in the future. Otherwise, you'll need to wipe devices that have already been deployed. Supervision isn't about locking down a device; rather, it enhances company-owned devices by extending management capabilities. In the long run, supervision provides even more options for your enterprise. For a complete list of supervised settings, see the iOS Deployment Reference.

Distribute apps and books Apple offers extensive programs to help your organization take advantage of the great apps and content available for iOS. With these capabilities, you can distribute apps and books purchased through VPP or apps you've developed in-house to devices and users, so your users have everything they need to be productive. At the time of purchase, you'll need to determine your distribution method: managed distribution or redemption codes. Managed distribution With managed distribution, use your MDM solution or Apple Configurator 2 to manage apps and books purchased from the VPP store in any country where the app is available. To enable managed distribution, you must first link your MDM solution to your VPP account using a secure token. Once you're connected to your MDM server, you can assign VPP apps and books, even if the App Store is disabled.

Assign VPP apps to devices. Using your MDM solution or Apple Configurator 2, assign apps directly to devices. This method saves several steps in the initial rollout, making your deployment significantly easier and faster, while giving you full control over managed devices and content. After an app is assigned to a device, the app is pushed to that device via MDM and no invitation is required. Anyone using that device has access to the app.

Assign VPP apps and books to users. Use your MDM solution to invite users through email or a push notification message. To accept the invitation, users sign in on their devices with a personal Apple ID. The Apple ID is registered with the VPP service, but remains completely private and not visible to the administrator. Once users agree to the invitation, they're connected to your MDM server so they can start receiving assigned apps and books. Apps are automatically available for download on all of a user's devices, with no additional effort or cost to you. When apps you've assigned are no longer needed by a device or a user, they can be revoked and reassigned to different devices and users, so your organization retains full ownership and control of purchased apps. However, once distributed, books remain the property of the recipient and cannot be revoked or reassigned. Redemption codes you can also distribute content using redemption codes. This method permanently transfers an app or a book to the user who redeems the code. Redemption codes are delivered in a spreadsheet format. A unique code is provided for each app or book in the quantity purchased. Each time a code is redeemed, the spreadsheet is updated in the VPP store, allowing you to view the number of redeemed codes at any time. Distribute the codes using MDM, Apple Configurator 2, email, or an internal website.

In addition to basic setup and configuration, Apple Configurator 2 can be used to install apps and content. For personally enabled deployments, you can preinstall all your apps, saving time and network bandwidth. And for nonpersonalized deployments, you can fully set up your devices all the way to the Home screen. When you configure devices with Apple

Configurator 2, you can install App Store apps, in-house apps, and documents. App Store apps require VPP. Documents are available for apps that support iTunes file sharing. To review or retrieve documents from iOS devices, connect them to a Mac running Apple Configurator 2.

Deploy iOS makes it simple for employees to start using their devices right out of the box, without requiring help from IT. Distribute your devices Once devices have been prepared and set up in the first two steps, they are ready for distribution. For personally enabled deployments, give devices to users who can use the streamlined Setup Assistant for further personalization and finalize setup. For nonpersonalized deployments, distribute devices to your shift employees or kiosks designed to charge and secure the devices. iOS Deployment Overview for Business

Setup Assistant Out of the box, users can activate their devices, configure basic settings, and start working right away with Setup Assistant in iOS. Beyond choosing basic settings, users can also customize their personal preferences, such as language, location, Siri, iCloud, and Find My iPhone. Devices that are enrolled in DEP can be automatically enrolled in MDM right within the Setup Assistant. Allow users to personalize for personally enabled and BYOD deployments, allowing users to personalize their devices with their own Apple IDs increases productivity, because users choose which apps and content will allow them to best accomplish their tasks and goals. Apple ID An Apple ID is an identity that's used to log in to various Apple services such as FaceTime, iMessage, the iTunes Store, the App Store, the iBooks Store, and iCloud. These services give users access to a wide range of content for streamlining business tasks, increasing productivity, and supporting collaboration. To get the most out of these services, users should use their own Apple IDs. Users who don't have an Apple ID can create one even before they receive a device. Setup Assistant also enables users to create a personal Apple ID if they don't have one. Users do not need a credit card to create an Apple ID.

Learn how to create an Apple ID without a credit card: support.apple.com/en-us/HT204034

Sign up for an Apple ID: appleid.apple.com iCloud iCloud allows users to automatically sync documents and personal content—such as contacts, calendars, documents, and photos—and keep them up to date between multiple devices.* Users can also back up an iOS device automatically when connected to Wi-Fi and use Find My iPhone to locate a lost or stolen iPhone, iPad, iPod touch, or Mac. Some services—such as Photo Stream, iCloud Keychain, iCloud Drive, and iCloud Backup—can be disabled through the use of restrictions, either entered manually on the device or set via configuration profiles. In addition, an MDM solution can prevent managed apps from being backed up to iCloud. This capability gives users the benefits of using iCloud for personal data while preventing corporate information from being stored in iCloud. Data from corporate accounts, such as Exchange, or data stored within enterprise in-house apps is also not backed up to iCloud. Note: iCloud is not available in all areas, and iCloud features may vary by area.

Learn more about iCloud: www.apple.com/icloud

Manage once your users are up and running, a wide range of administrative capabilities is available for managing and maintaining your devices and content over time. Administer your devices a managed device can be administered by the MDM server through a set of specific tasks. These tasks include querying devices for information, as well as initiating management tasks that allow you to manage devices that are out of policy, lost, or stolen. Queries an MDM

server can query devices for a variety of information, including hardware information such as serial number, device UDID, or Wi-Fi MAC address, as well as software information such as the iOS version and a detailed list of all apps installed on the device. This information helps to ensure that users maintain the appropriate set of apps. iOS Deployment Overview Management tasks When a device is managed, an MDM server may perform a wide variety of administrative tasks, including changing configuration settings automatically without user interaction, locking or wiping a device remotely, or clearing the passcode lock so users can reset forgotten passwords. An MDM server may request an iOS device to begin AirPlay mirroring to a specific destination or end a current AirPlay session. Lost Mode With iOS 9.3 or later, your MDM solution can place a supervised device in Lost Mode remotely. This action locks the device and allows a message with a phone number to be displayed on the Lock screen. With Lost Mode, supervised devices that are lost or stolen can be located because MDM remotely queries for their location. Lost Mode doesn't require find my iPhone to be enabled. Activation Lock With iOS 7.1 or later, you can use MDM to enable Activation Lock when a user turns on Find My iPhone on a supervised device. This allows your organization to benefit from the theft-deterrent functionality of Activation Lock, while still allowing you to bypass the feature if a user is unable to authenticate with their Apple ID. Deploy and manage additional content Organizations often need to distribute apps so their users are productive. At the same time, organizations need to control how apps connect to internal resources or how data security is handled when a user transitions out of the organization—all while coexisting alongside the user's personal apps and data. Internal app portals you have the option of creating an internal app portal for your employees, where they can easily find apps for their iOS devices. In-house apps, App Store app URLs or VPP codes, or custom B2B VPP codes can be linked from this portal, making it a single destination for users. You can manage and secure this site centrally. In addition, it's easy to build a portal internally or explore third-party MDM solutions to manage app distribution. Managed content Managed content involves the installation, configuration, management, and removal of App Store and custom in-house apps, accounts, books, and documents. • Managed apps. In iOS, managed apps allow an organization to distribute free, paid, and enterprise apps over the air using MDM, while also providing the right balance of protecting corporate data and respecting user privacy. Managed apps can be removed remotely by an MDM server or when users remove their own devices from MDM. Removing the app also removes the data associated with the app. If an app remains assigned to a user through VPP, or if the user redeemed an app code using a personal Apple ID, the app can be downloaded again from the App Store, but it will not be managed by MDM. Managed accounts. MDM can help your users get up and running quickly by setting up their mail and other accounts automatically. Depending on the MDM solution provider and integration with your internal systems, account payloads can also be pre-populated with a user's name, mail address, and, where applicable, certificate identities for authentication and signing. • Managed books and documents. Using MDM tools, books, ePub books, and PDF documents can be automatically pushed to user devices, so employees always have what they need. At the same time, managed books can be shared only with other managed apps or mailed using managed accounts. When the materials are no longer necessary, they can be removed remotely. Books purchased through VPP can be distributed through managed book distribution, but cannot be revoked and reassigned. A book already purchased by the user cannot be managed unless the book is explicitly assigned to the user by VPP. iOS Deployment Overview for Business.

Managed app configuration App developers can identify app settings and capabilities that can be enabled when installed as a managed app. Install these configuration settings before or after the managed app is installed. For example, IT establishes a set of default preferences for a Sharepoint app, so the user doesn't need to manually configure server settings. Leading MDM solution providers have established the AppConfig Community and a standard schema

that all app developers can use to support managed app configuration. The AppConfig Community is focused on providing tools and best practices around native capabilities in mobile operating systems. The community helps enable a more consistent, open, and simple way to configure and secure mobile apps to increase mobile adoption in business. Learn more about the AppConfig Community: www.appconfig.org Managed data flow MDM solutions provide specific features that enable corporate data to be managed at a granular level so that it does not leak out to users' personal apps and cloud services. • Managed Open In. This restriction protects corporate data by controlling which apps and accounts are used to open documents and attachments. IT organizations can configure a list of apps available in the sharing panel to keep work documents in corporate apps and prevent personal documents from being opened in managed apps. This policy also applies to third-party document providers and third-party keyboard apps.

Single App Mode. This setting helps the user stay focused on a task while using an iOS device by limiting the device to a single app. Developers can also enable this functionality within their apps to allow apps to enter and exit Single App Mode independently. • Prevent backup. This restriction prevents managed apps from backing up data to iCloud or iTunes. Disallowing backup prevents managed app data from being recovered if the app is removed via MDM but later reinstalled by the user. Support Options Apple provides a variety of programs and support options for iOS users and IT administrators. AppleCare for Enterprise For companies looking for complete coverage, AppleCare for Enterprise can help reduce the load on your internal help desk by providing technical support for employees over the phone, 24/7, with one-hour response times for top-priority issues. The program provides IT department-level support for all Apple hardware and software, as well as support for complex deployment and integration scenarios, including MDM and Active Directory. AppleCare OS Support AppleCare OS Support provides your IT department with enterprise-level phone and email support for iOS, macOS, and macOS Server deployments. It offers up to 24/7 support and an assigned technical account manager, depending on the level of support you purchase. With direct access to technicians for questions on integration, migration, and advanced server operation issues, AppleCare OS Support can increase your IT staff's efficiency in deploying and managing devices and resolving issues. iOS Deployment Overview for Business. 10 AppleCare Help Desk Support AppleCare Help Desk Support provides priority telephone access to Apple's senior technical support staff. It also includes a suite of tools to diagnose and troubleshoot Apple hardware, which can help large organizations manage their resources more efficiently, improve response time, and reduce training costs. AppleCare Help Desk Support covers an unlimited number of support incidents for hardware and software diagnosis, as well as troubleshooting and issue isolation for iOS devices. AppleCare for iOS device users every iOS device comes with a one-year limited warranty and complimentary telephone technical support for 90 days after the purchase date. This service coverage can be extended to two years from the original purchase date with AppleCare+ for iPhone, AppleCare+ for iPad, or the AppleCare Protection Plan (APP) for iPod touch. You can call Apple's technical support experts as often as you like with questions. Apple also provides convenient service options when devices need to be repaired. In addition, AppleCare+ for iPhone and AppleCare+ for iPad offer up to two incidents of accidental damage coverage, each subject to a service fee. iOS Direct Service Program As a benefit of AppleCare+ and the AppleCare Protection Plan, the iOS Direct Service Program enables your help desk to screen devices for issues without calling AppleCare or visiting an Apple Store. If necessary, your organization can directly order a replacement iPhone, iPad, iPod touch, or in-box accessory. Learn more about AppleCare programs: www.apple.com/support/professional Summary whether your company deploys iOS devices to a group of users or across the entire organization, you have many options for easily deploying and managing devices. Choosing

the right strategies for your organization can help your employees be more productive and accomplish their work in entirely new ways.

Learn more about integrating iOS into enterprise IT environments: www.apple.com/ipad/business/it

Reference:

help.apple.com/deployment/ios

Module 22: Project Work

2 Completed projects have to be completed and delivered for every student (First one is a group project and the last one is the individual project) (25 hrs.)

In this section student have to implement their knowledge, learned during this course. And must make an individual project using those technologies:

IDE:	Xcode
Programming Language: Objective C (Object oriented)	
View:	Storyboard
Layout:	Auto Layout
Notification:	Local and push Notification
Networking:	Json Parsing, XML
Scroll:	Table view &Collection view
Location:	Map Kit
Gaming Library:	Sprite kit
Database:	SQLite
Design Pattern:	MVC

www.ictd.gov.bd

www.gameapp.gov.bd

www.facebook.com/MobileSkill

www.wis-sdmga.com

WIS Consortium

Workspace
IBF
Softcell

INSIGHT
BANGLADESH
FOUNDATION
EMPOWERING YOU

workspace
Infotech Ltd.

SOFTCELL
SOLUTION LIMITED