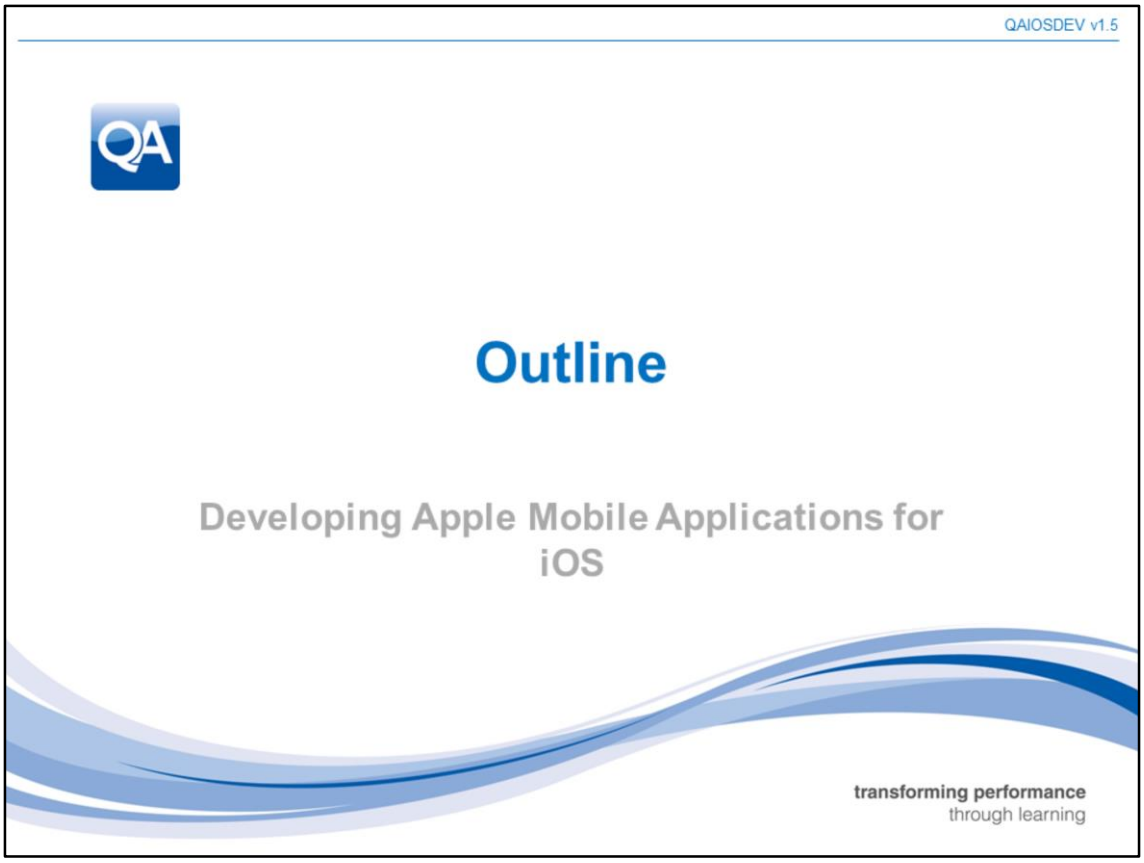
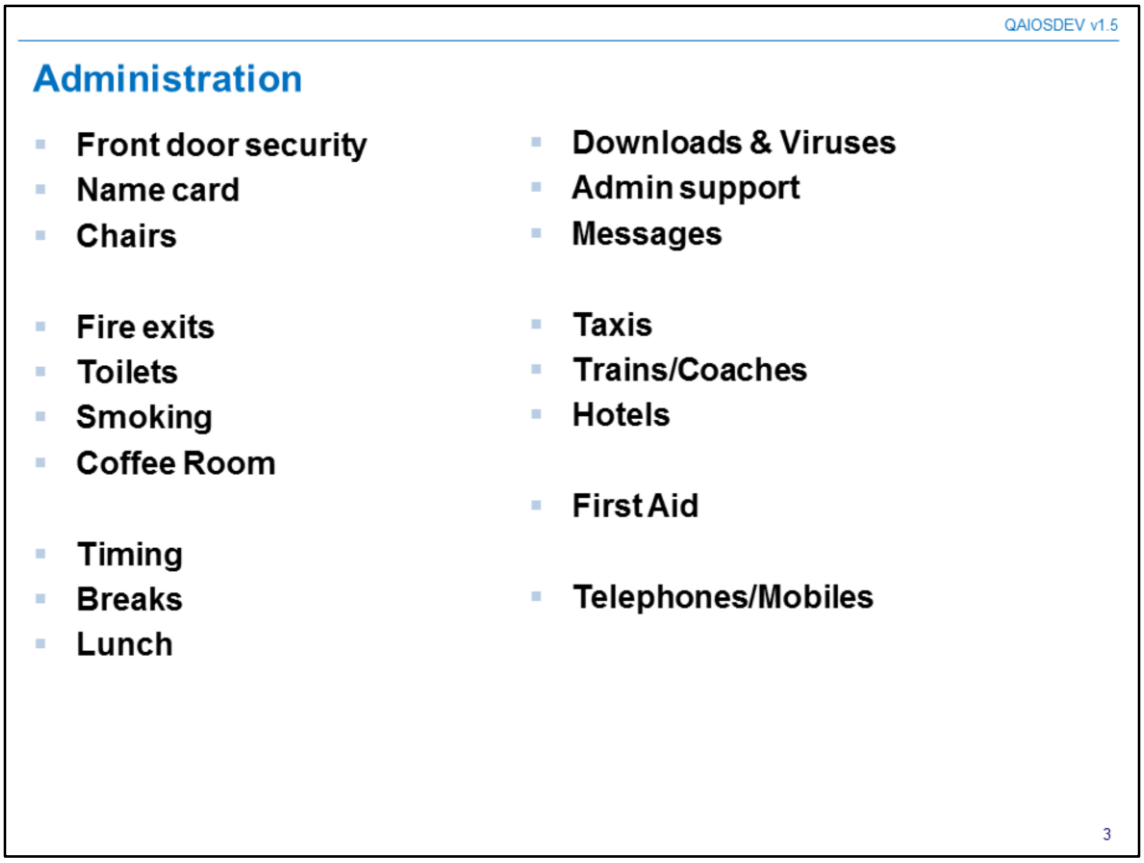

QAIOSDEV v1.5
Outline
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Overview

- **Objectives**
  - To explain the aims and objectives of the course
- **Contents**
  - Course administration
  - Course objectives and assumptions
  - Introductions
  - Any questions?
- **Exercise**
  - Locate the exercises
  - Locate the help files

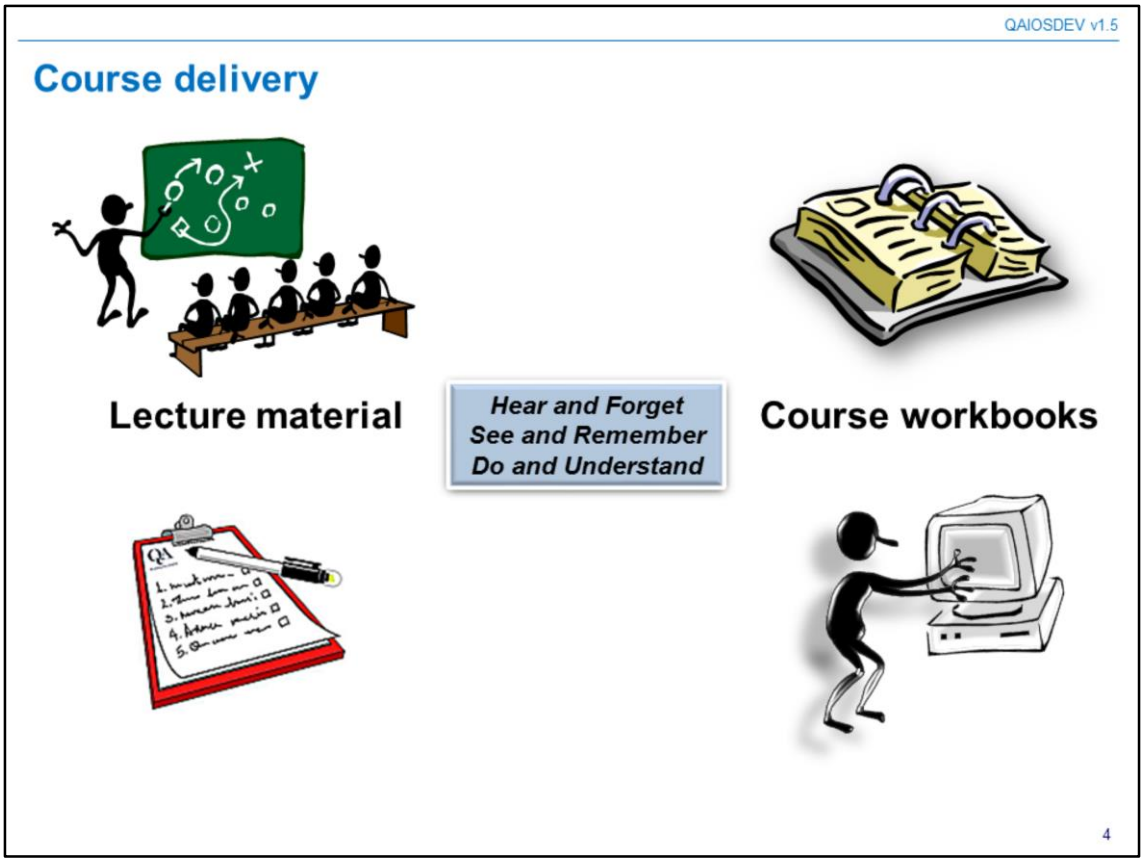
2




We need to deal with practical matters right at the beginning.

Above all, please ask if you have any problems regarding the course or practical arrangements. If we know early on that something is wrong, we have the chance to fix it. If you tell us after the course, it's too late! We ask you to fill in an evaluation form at the end of the course. If you alert us a problem for the first time on the feedback form at the end of the course the we have not had the opportunity to put it right.

If this course is being held at your company's site, much of this will not apply or will be outside our control.




The course will be made up of lecture material coupled with the course workbook, informal questions and exercises, and structured practical sessions. Together, these different teaching techniques will help you to absorb and understand the material in the most effective way.

The course notebooks contain all the overhead foils that will be shown, so you do not need to copy them. In addition, there are extra textual comments (like these) below the foils, which are there to amplify the foils, provide further information. Hopefully, these notes mean you will not need to write too much and can listen and observe during the lectures. There is, however, space to make your own annotations too.

The appendices cover material that is beyond the scope of the course, together with some help and guidelines. There are also appendices on Bibliography and Internet resources to help you find more information after the course.

In the practical exercise sessions, you will be given the opportunity to experiment and consolidate what has been taught during the lecture sessions. Please, please tell the instructor if you are having difficulty in these sessions. It is sometimes difficult to see that someone is struggling, so please be direct.

## A training experience

- **A course should be**
  - A two-way process
  - A group process
  - An individual experience



The best courses are not those in which the instructor spends all his or her time pontificating at the front of the class. Things get more interesting if there is dialogue, so please feel free to make comments or ask questions. At the same time, the instructor has to think of the whole group, so if you have many queries, he or she may ask to deal with them off-line.

Work with other people during practical exercise sessions. The person next to you may have the answer, or you may know the remedy for them. Obviously, do not simply 'copy from' or 'jump-in on' your neighbour but group collaboration can help with the enjoyment of a course.

We are also individuals. We work at different paces and may have special interests in particular topics. The aim of the course is to provide a broad picture for all. Do not be dismayed if you do not appear to complete exercises as fast as the next person. The practical exercises are there to give plenty of practical opportunities; they do not have to be finished and you may even choose to focus for a long period on the topic that most interests you. Indeed, there will be parts labelled 'if time allows' that you may wish to save until later to give yourself time to read and absorb the course notes. If you have finished early, there is a great deal to investigate. Such "hacking" time is valuable. You may not get the opportunity to do it back in the office!

## Course aims and objectives

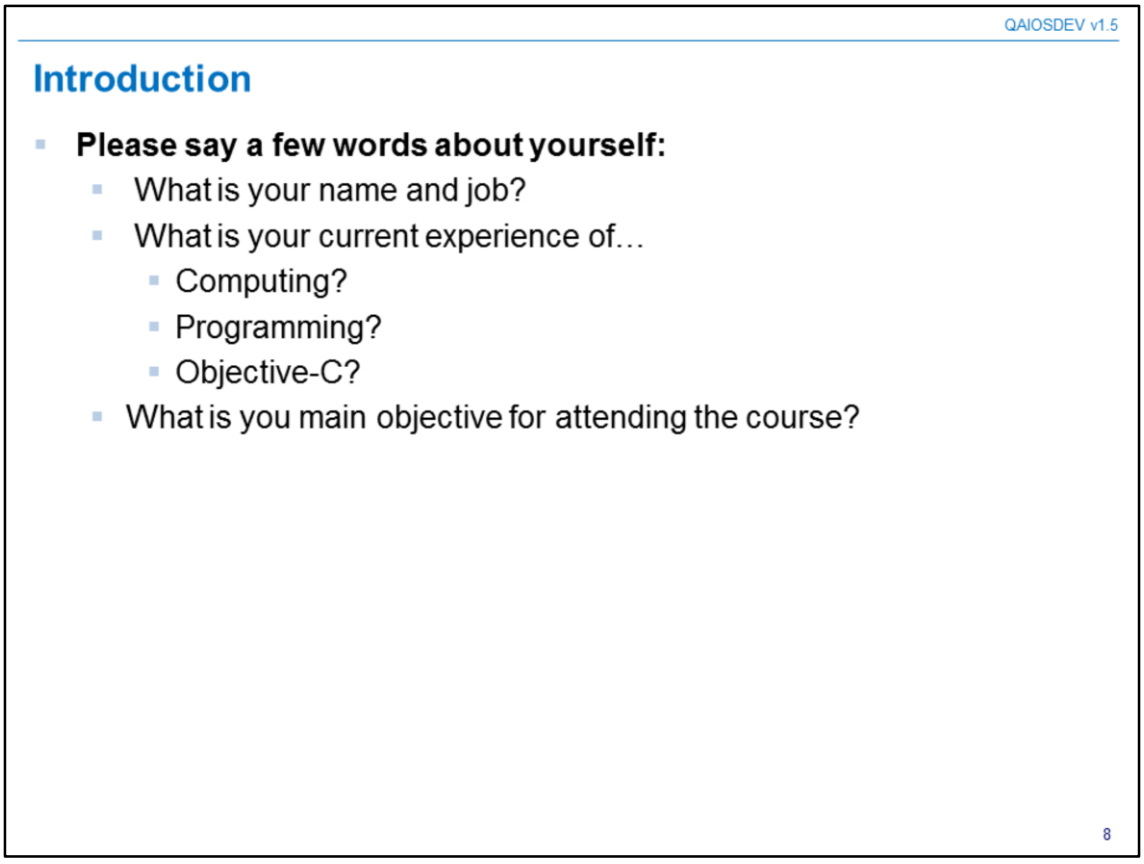
- **By the end of the course you will be able to do the following:**
  - Write an iPhone, iPad and universal app
  - Create a UI using just code
  - Use Interface Builder to create a UI
  - Work with the main user interface controls
  - Make effective use of View Controllers and Table Views
  - Create and use Data Stores
  - Work with Core Data and iCloud
  - Use multithreading techniques (indirectly)
  - Apply Animation
  - Access Cloud Data
  - Handle Notifications
  - Handle multitasking and background tasks
  - Analyse apps for performance and stability

## Assumptions

- **This course assumes the following prerequisites**
  - Either
    - You already have good experience of programming with Objective-C
    - Have attended QA Course "The Objective-C Programming Language" – QAOBJC
  - This course contains an Objective-C refresher
    - Beginners to the language are likely to find the course challenging
- **If there are any issues please tell your instructor now**



**If you are not sure of any of these**
**please inform the instructor**
**as soon as you can**
**and they will**
**do their best to help you.**



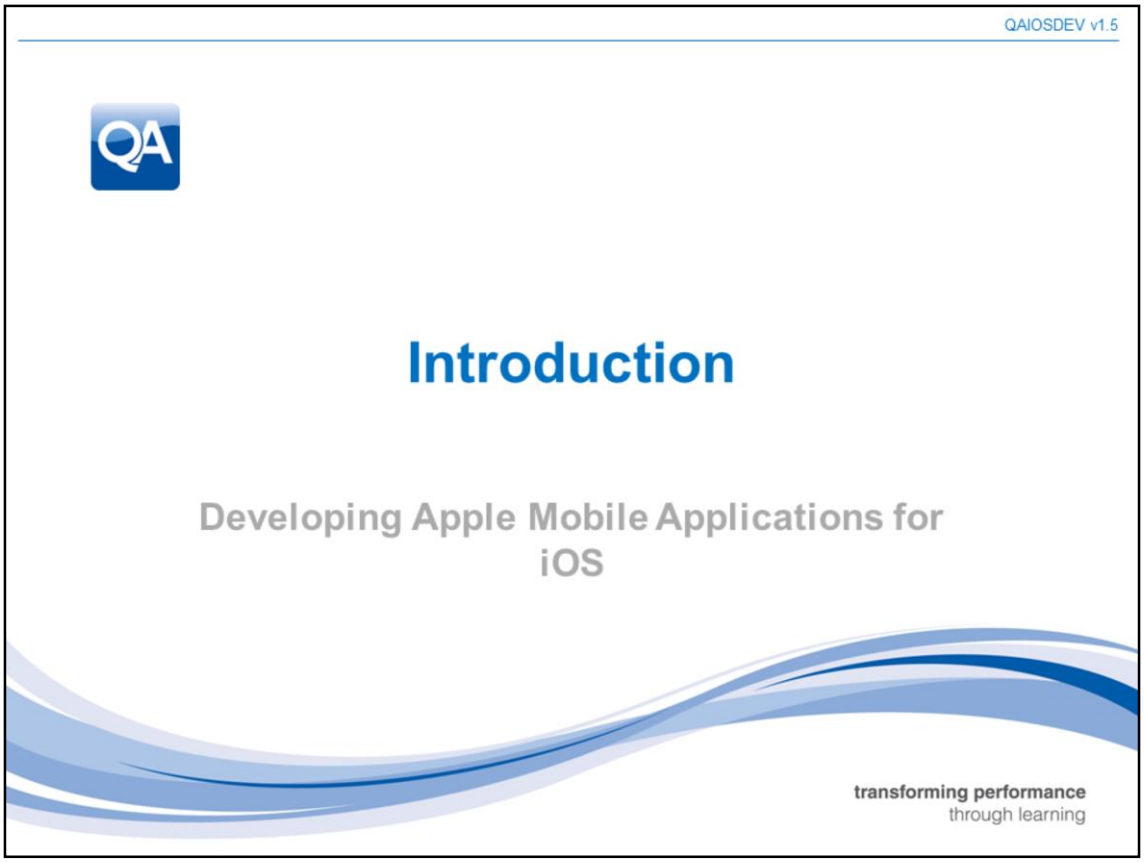

One of the great benefits of courses is meeting other people. They may have similar interests, have encountered similar problems and may even have found the solution to yours. The contacts made on the course can be very useful.
It is useful for us all to be aware of levels of experience. It will help the instructor judge the level of depth to go into and the analogies to make to help you understand a topic. People in the group may have specialised experience that will be helpful to others.
It is worth highlighting particular interests, as we may be able to address them during the course. However, it is a general course that aims to cover a broad range of topics, so the instructor may have to deal with some areas during a coffee break or over lunch.

## Any questions

- **Golden Rule**
  - "There is no such thing as a stupid question"
- **First amendment to the Golden Rule**
  - "... even when asked by an instructor"
- **Corollary to the Golden Rule**
  - "A question never resides in a single mind"



Please feel free to ask questions.
Teaching is a much more enjoyable and productive process if it is interactive.
You will no doubt think of questions during the course; if so, ask them!


QAIOSDEV v1.5
Introduction
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - To introduce iOS and iOS development
- **Contents**
  - iOS Overview
  - Cocoa Touch Framework
  - Developing for iOS
  - Xcode and Objective-C
  - Overview of an iOS app
  - Multitasking

12

## iOS History

- **Jan 9th 2007 Steve Jobs introduced iPhone**
- **Derived from Mac OS X**
  - Unix OS based on the "Darwin" foundation framework
- **October 17th 2007**
  - Beta SDK released
- **July 11 2008**
  - iOS 2 released
  - App Store Launched
- **June 17 2009**
  - iOS 3 released (Copy and Paste, MMS)
- **June 2010**
  - iOS 4 released (Multitasking)
- **October 2011**
  - iOS 5 released (iCloud)
- **September 2012**
  - iOS 6 – New Maps app, Passbook, Facebook integration plus SDK enhancements
- **October 2013**
  - iOS 7 released – New GUI plus SDK enhancements



iOS as it was (iPhone OS) was released in January 2007 with the announcement of the first iPhone.

It's a Unix based OS derived from Mac OSX built using the Darwin OS foundation frameworks. You will still see references to Darwin in some of the debug logs.

The big release was the SDK in 2007 because it allowed developers to write apps for the iPhone and with the launch of the app store in 2008 the platform has grown in strength ever since.

An new version of iOS has been released subsequently pretty much every year.

## iOS Framework Libraries

- **Framework libraries form part of each of the iOS layers**
  - Gives the developer access to the OS
  - High and low level libraries
- **Choose frameworks from the highest layer if possible**
  - More sophisticated and complex features
  - Do more with less code
  - Low level libraries provide lower level features
- **Framework Libraries written in:**
  - C (lower level)
  - Objective C



The framework libraries form part of each of the the iOS software services layers. This means they can be accessed by the developer from C, C++ and Objective-C applications.

The highest level library is Cocoa Touch which uses the services of the layers below and provides higher level grouped services. Typically, these are the services we as developers will use most often as they abstract a lot of complexity into relatively simple method calls.

You can go deeper into the other libraries and often you will to achieve more complex and sophisticated outcomes or to gain more control.


You can develop Objective-C on other platforms.

To develop for Apple apps you need an Intel based Mac. 4GB memory.

The development environment XCode can be downloaded from the Mac app store or it's free if you join the Apple developer program.

## Xcode

- **Development Environment**
  - Mac
  - iPhone/iPad
- **Consists of**
  - Code Editor
  - Build Engine
  - Build Configuration Manager
  - iPhone/iPad Simulators
  - Performance Tools
  - Provisioning tools
  - App submission utility
  - Version Management
- **Xcode is huge!**
  - We won't cover all of it's features in detail

16

Xcode is a complete development environment and provides you everything you need to develop iOS applications. Not bad for a free tool.

## Xcode's Windows

- **File Explorer**
  - Files can be organised into file groups
  - File groups are logical not physical
- **Code Window**
  - Intellisense
  - Error and warning highlighting
  - Breakpoints & Debugging
- **Output Window**
  - Logging
  - GDB
- **Debug Window**
  - Variables viewer (locals, globals etc)
  - Breakpoint and execution

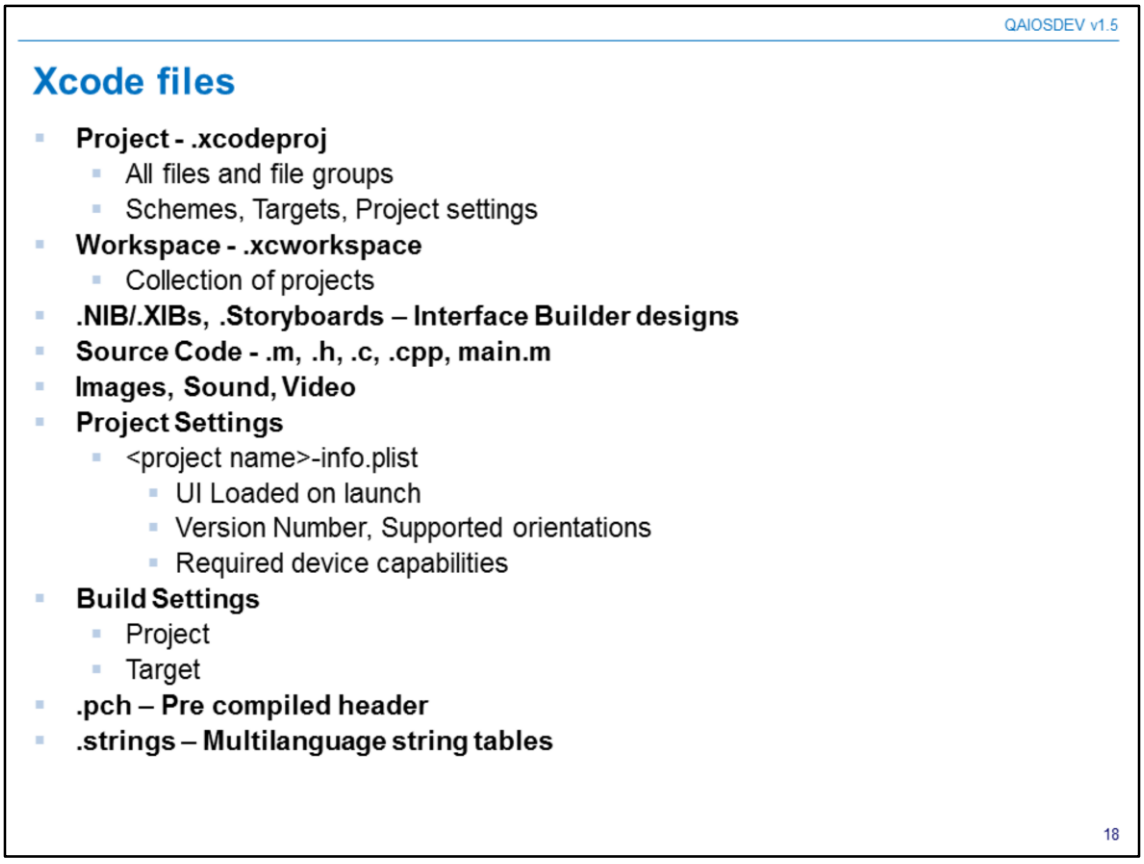


The file explorer is where you manage all of the files in your project. You can organize files into logical groups. File groups don't actually create an underlying directory structure on disk they are just for logical organization within your project.

The code window has colour coding and intellisense plus you can set breakpoints and carry out advanced debugging tasks.

The output windows displays everything that is being logged by the application, You can add your own logging code and view your output in this window. You also have access to a powerful command line processor called GDB that gives you access to debug information about the executing app.

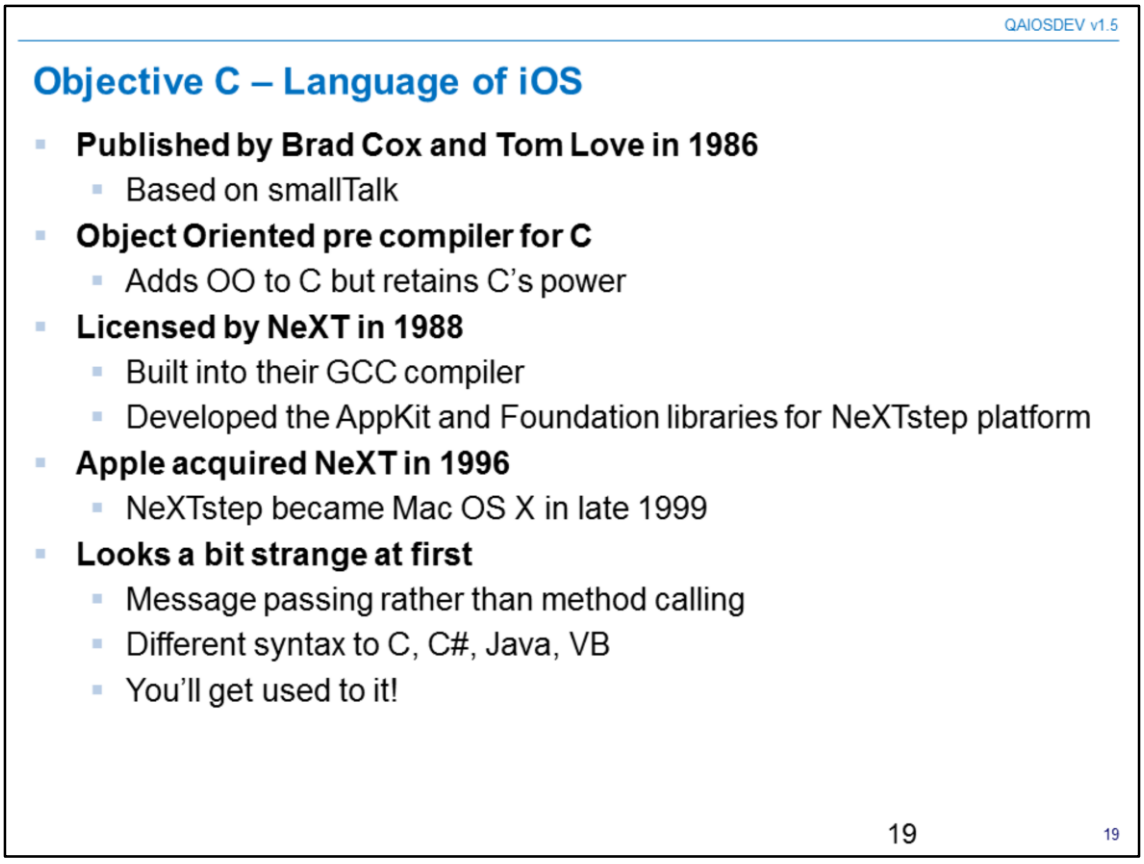
The debug window displays the current state of variables in scope at the current execution point.




Xcode uses many types of files. Project files (.xcodeproj) reference all the files that make up a project and include source files, images, NIB/Xibs, storyboards etc.

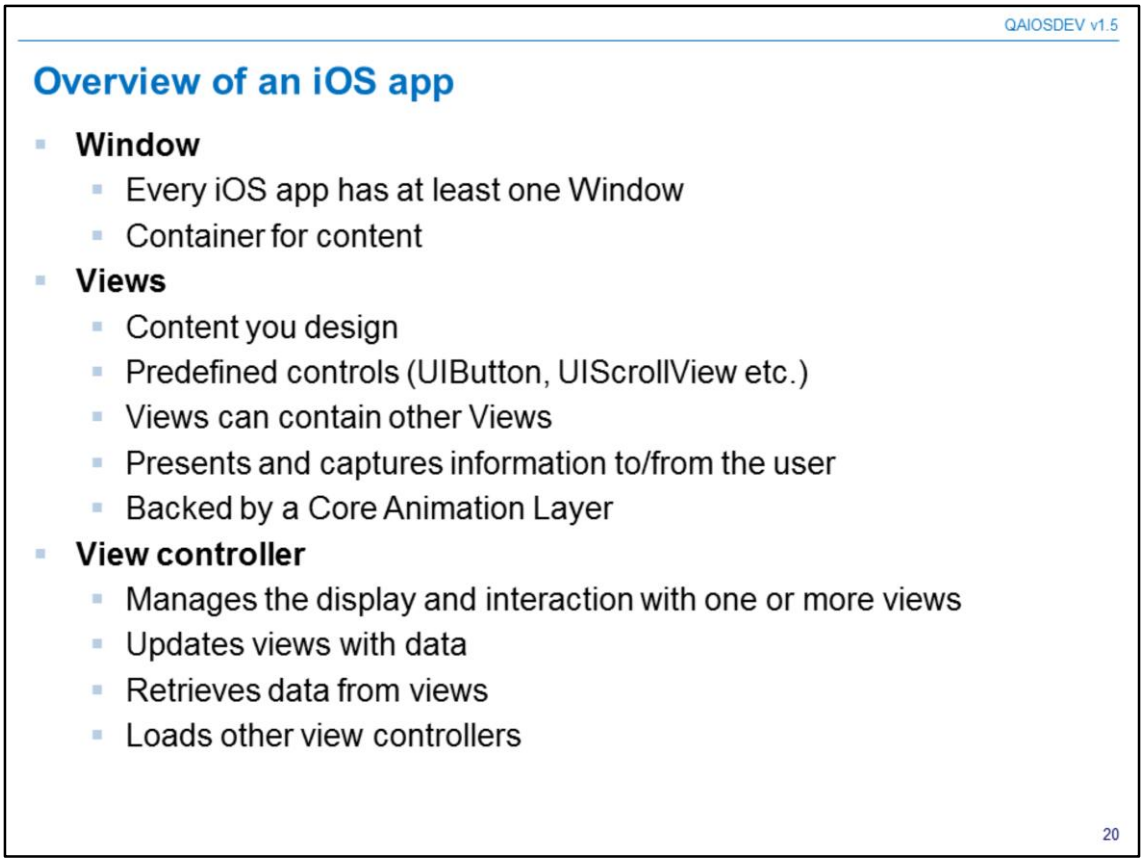
A scheme sets up an entire build configuration for a target app or unit test build. It specifies whether the output is a debug or release build, the debugger to use together with runtime options and diagnostics to use.

Workspaces allow multiple projects to be grouped together.




Objective-C is the language of iOS published in 1986 and adopted by NeXT in 1988 eventually making it into Mac OSX by 1999.

Based on smallTalk and aims to create an truly object oriented language that was still 100% compatible with the C language.

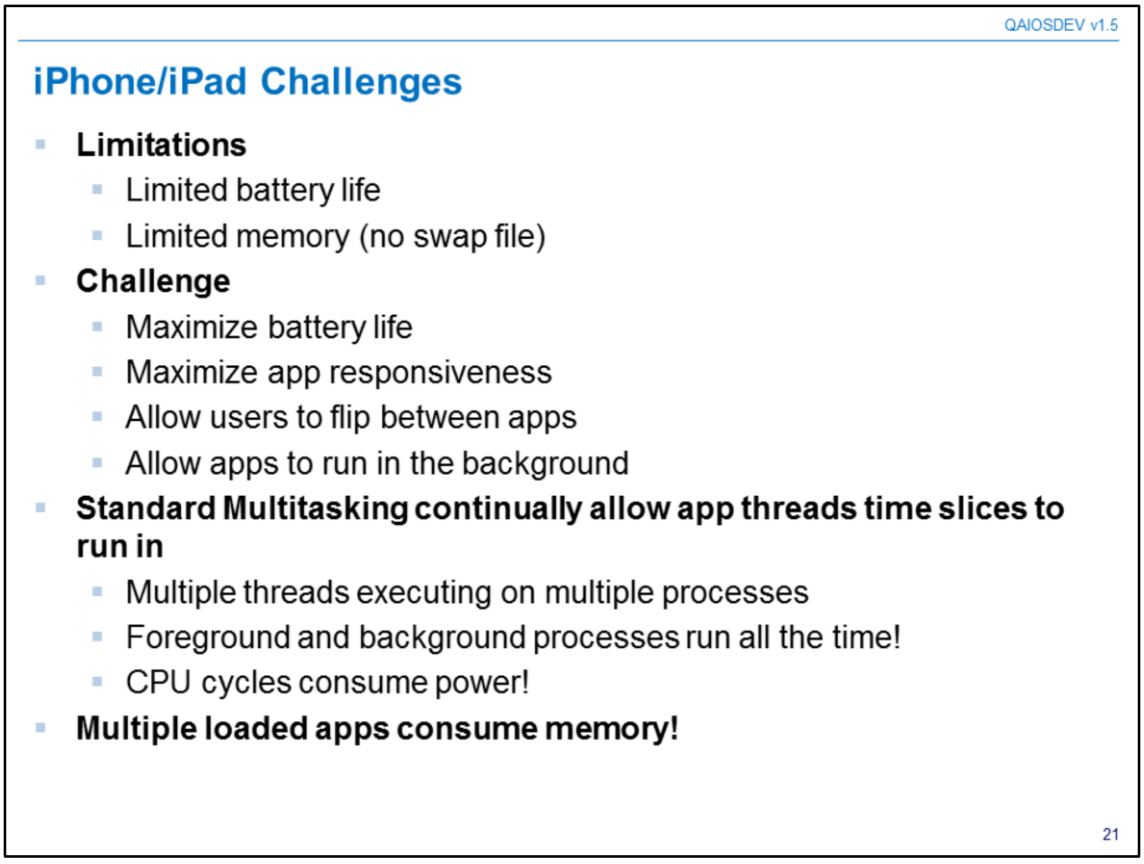



Every iOS app has at least one window. A window is used to present content on the screen.

Views are used to present content onto a window and a windows can contain multiple views. Views themselves can also contain sub views.

View controllers acts a a managing interface between the view and your model (data storage). The view controller will update a view with data the view will allow the user to modify that data, the view controller can then receive user events such as button clicks and other gestures. Some of these events may trigger save action to the model.

View controllers also manage system events like phone orientation changes and low memory conditions.



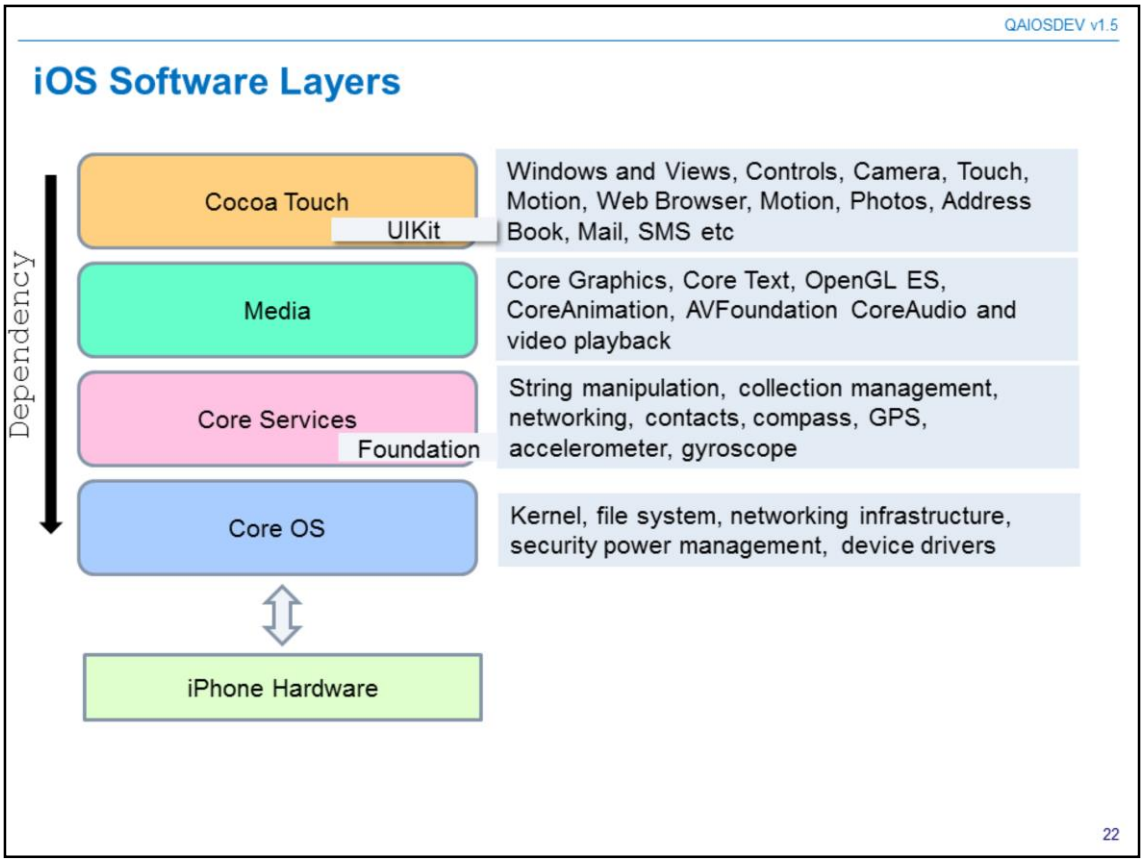

By their nature, mobile devices have limited amounts of available battery power available to them. Their need for portability and hence small size has meant their maximum memory has also been limited.

The challenge for a mobile OS producer therefore is to produce an OS that can maximize battery life and use memory as efficiently as possible.

We have all become used to multitasking operating systems that allow apps to run simultaneously by routinely scheduling the CPU to run threads for each of the running apps in turn. It works well and with multiple CPUs and cores even the most CPU intense apps run side by side quite happily.

On the mobile device scheduling a CPU in this way would consume battery power really quickly so alternate strategies for multitasking have had to be found.

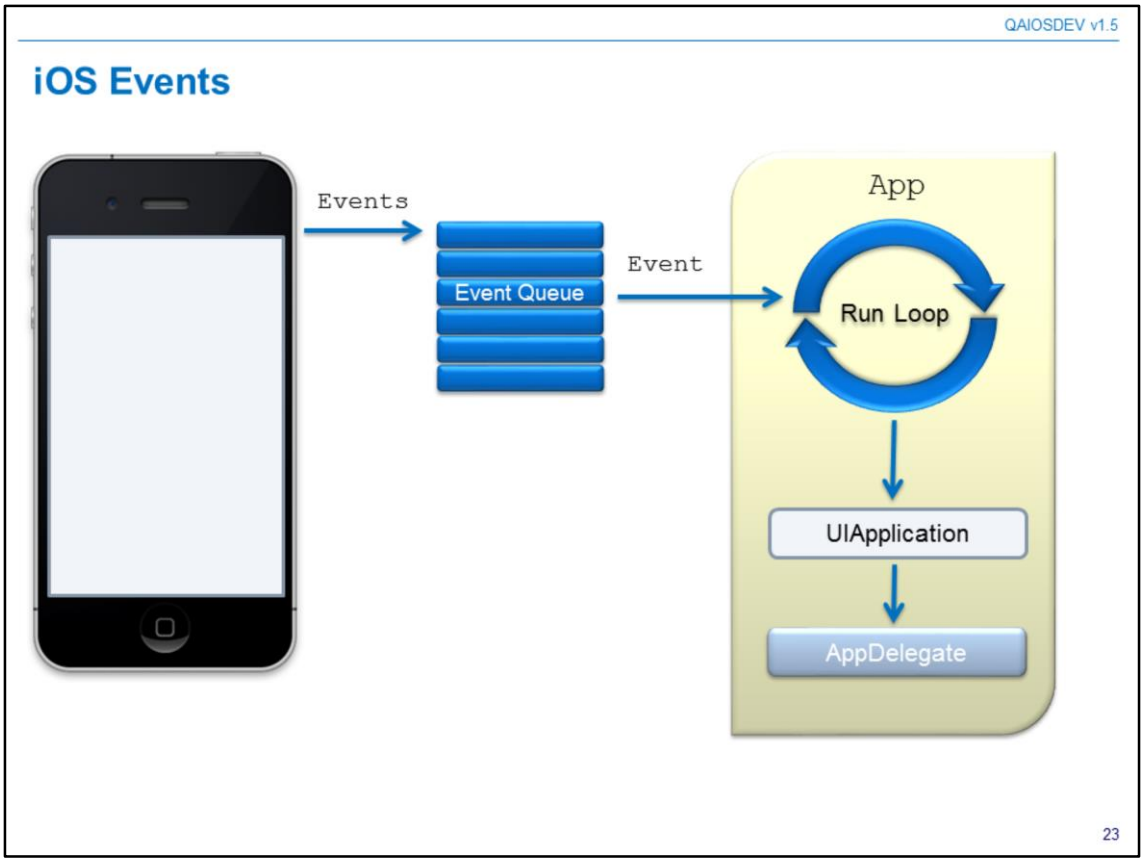
Prior to iOS 4 apps ran individually and as one app loaded, the previous one unloaded. iOS 4 onwards allowed multiple apps to be loaded but only one could be loaded in the foreground with the others either suspended or temporarily running in a background mode.




iOS is built up from a number of logical software service layers. Starting at the hardware interface the services of each layer are used by the ones above to build increasingly rich and complex services.

The service layers are each made up of a set of framework libraries that can be accessed by the developer using either C, C++ or Objective-C.

The two key frameworks used most for iOS development are UIKit and Foundation. Together they give the core infrastructure classes for developing iOS applications.




When the user interacts with an app they generate various kinds of events. Those events are directed to an event queue and eventually pass to the apps main execution loop that handles event messages. Events are then notified in the the main application class of all iOS applications, UIApplication, from where they are dispatched to the interested parties in the application. We will see throughout this course how the various events generated by iOS itself should be handled to create a well behaved and responsive app experience.

## 32 bit and 64 bit apps

- Apple A7 process supports 64 bit instruction set
- iOS running on a 64 bit device provides
  - 32 & 64 bit versions of systems frameworks
- XCode 5.0.1 can build a combined 32/64 bit app bundle
- 64 bit version runs on 64 bit devices and vice versa
- 64 bit version is preferred and is faster
- Apps need to be converted to run safely in 64 bit mode

## Deploying an App to an iOS Device

- **You can't just install and app to an iOS device**
- **Apple needs to know everything!**
  - Developer of the app is a valid Apple Developer
  - App is valid (signed and registered with a valid certificate)
  - App has been created for the specific device
- **You must:**
  - Be a registered developer with Apple Developer Program
  - Have registered your devices with your Apple account
  - Have registered your App with your account
  - Create a provisioning profile for the app
  - Deploy the app with it's provisioning profile to the device

## Provisioning Process

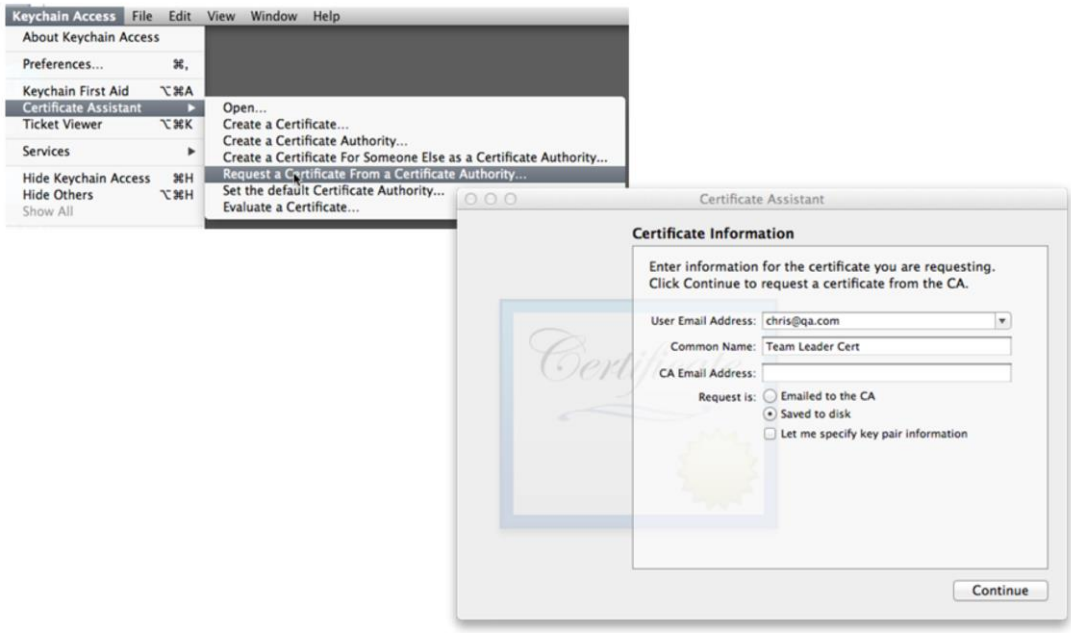
- **Sign up to the Apple Developer Program**
  - Individual
  - Company
  - Enterprise
  - University
- **Create a Code Signing Certificate**
  - Generate Request on Mac
  - Submit to Apple Developer Center
  - Download and install generated certificate
- **Create a Provisioning Profile**
  - Contains details of
    - Code Signing Certificate
    - List Devices this profile can sign for
    - App Id to be signed (or wildcard)
  - Download and Install Provisioning profile

## Summary

- **iOS is a great platform to develop for**
- **Constant innovation by Apple**
  - iOS capability
  - Hardware
- **Four Layer Services model**
- **Objective-C and Xcode are great tools for iOS development**
- **Windows, Views, View Controllers**
- **iOS and Multitasking challenges**

# Create a Signing Request

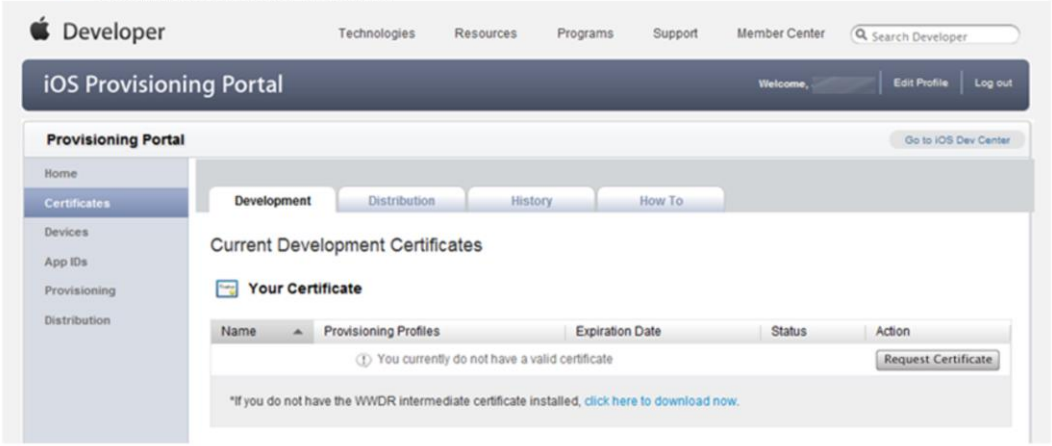
- **Utilities/KeyChain Access**




28

# Submit Certificate to Developer Center

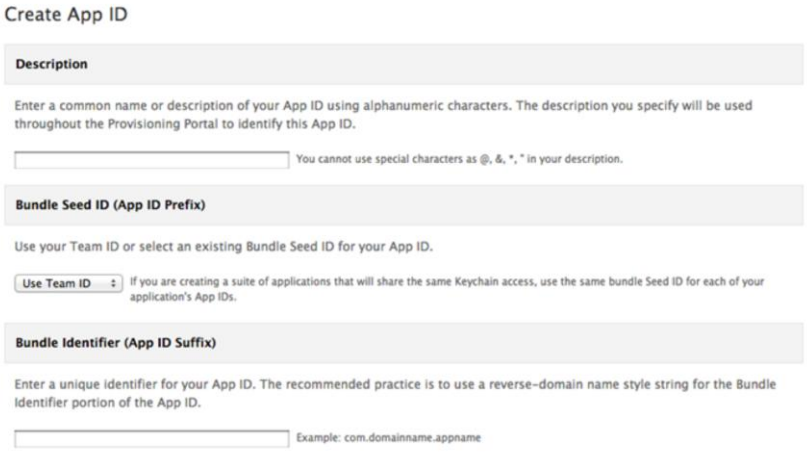
- **Sign in to Apple Developer Center**
  - Using Signing Request
  - Request Certificate
  - Download and Install




29

# App ID

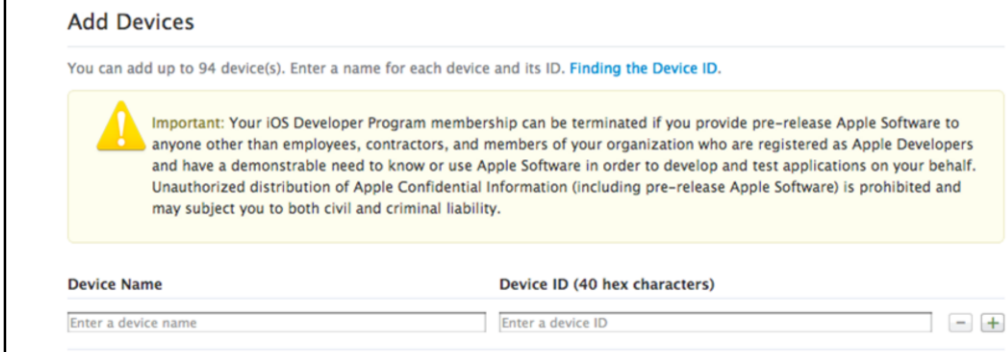
- **Consists of**
  - Familiar Name – My App
  - App ID Prefix (generated 10 character string)
  - Bundle Identifier – **com.qa.MyApp**




30

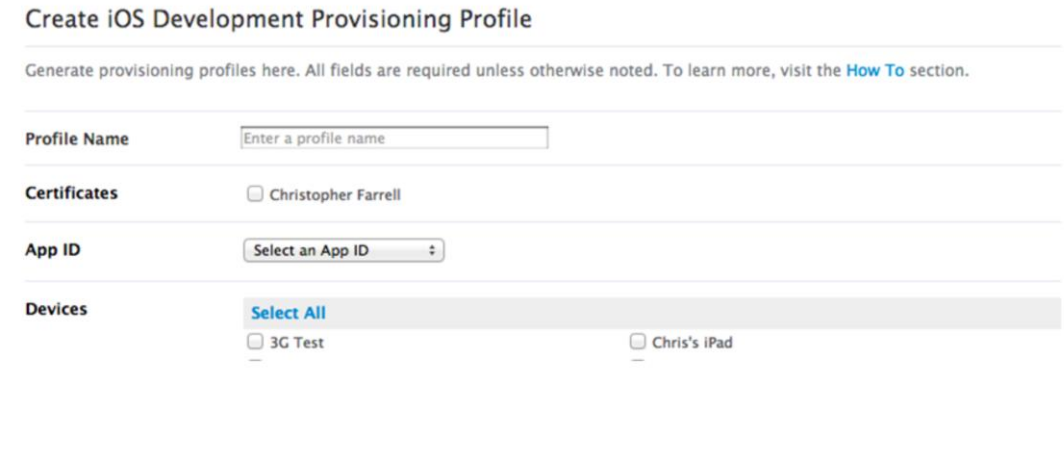
# Devices

- **Consist of**
  - Device Name
  - Device Id – Obtained from Xcode Organizer or iTunes



# Provisioning Profile

- **Consists of**
  - Familiar Name
  - Signing Certificate
  - App Id ( or wildcard)
  - Devices



## Development Provisioning

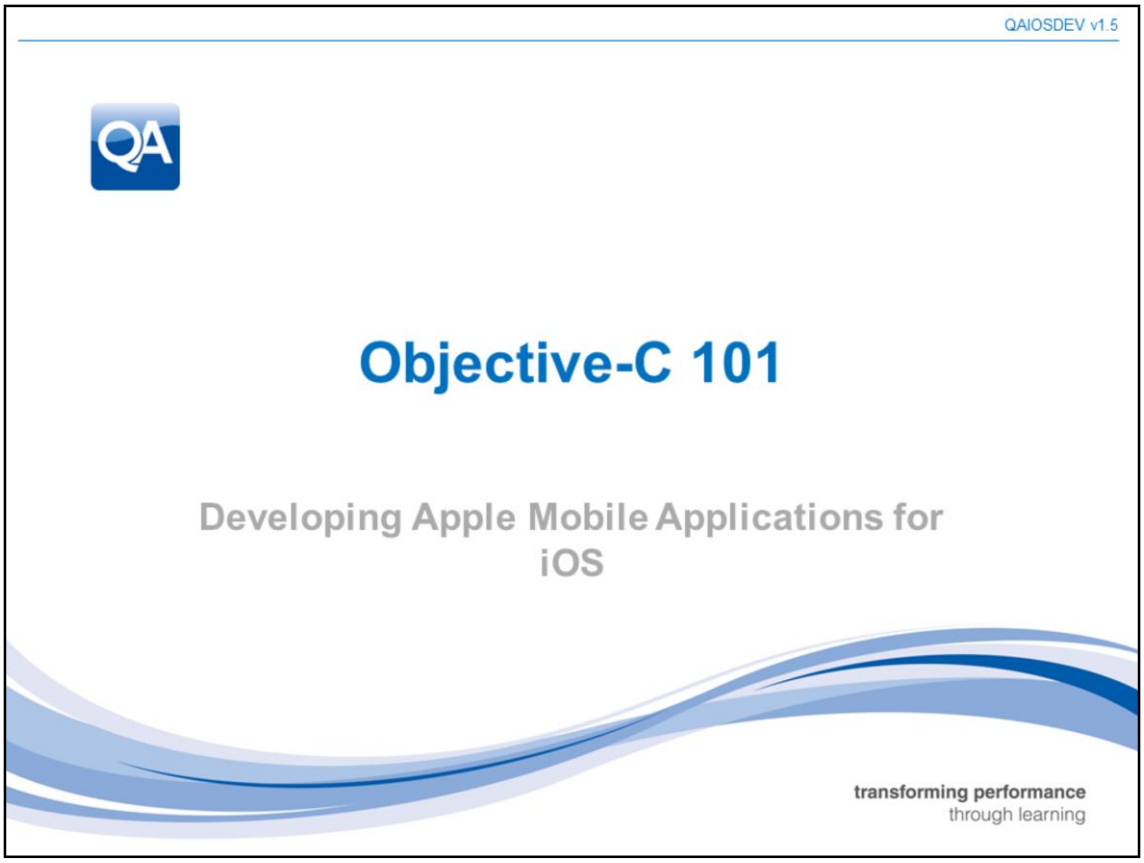
- **To install an app to a device**
  - App must be signed with a valid Code Signing Certificate
  - Have a valid Provisioning Profile installed containing
    - App Id (or wildcard app id)
    - List of Devices (iPhones, iPads) that can be installed to
    - Reference to the Code Signing Certificate to be used
- **Developer needs to install a valid**
  - Code Signing Certificate
  - Provisioning profile – Lists valid devices for an app (iPhones, iPads)
- **Developer needs to Build app with Code Signing Identity set as the provisioning profile – Build Settings**


33

## Team Provisioning Profile

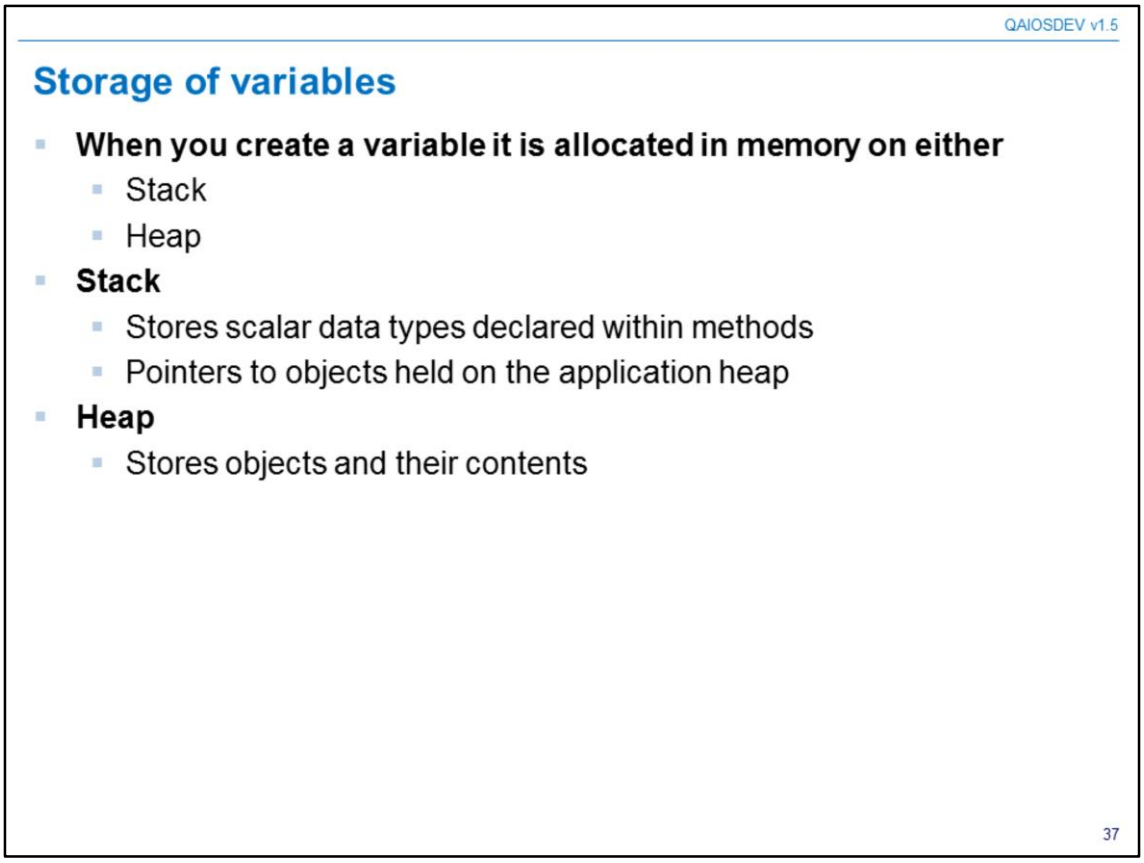
- **Created automatically**
  - Developer Signing Certificate
  - Wildcard App Id
  - Lists all Devices
- **Can be used by all developers to deploy**
  - Any app
  - To All registered devices

34


QAIOSDEV v1.5
Objective-C 101
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - To refresh your Objective-C knowledge
- **Contents**
  - Overview
  - Data Types
  - Creating Classes
    - Properties
    - Methods
- **Hands on Lab**




Creating a variable inevitably means allocating storage for it somewhere in memory. The two main storage areas for an application are on either a thread's stack or on the application heap.

The stack is a temporary storage area used for the storage of scalar variables and object pointers defined within the context of a method call.

The heap is the long term memory storage area that stores object instances and their contents.

## Declaring variables

- **Standard C syntax**

```
int val1=20;
```

  - Type name
  - Variable Name
  - Assignment (optional but good practice)
- **BOOL**

```
BOOL isMarried=YES; // or TRUE, FALSE, NO
```

- **char**

```
char initial= 'a';
char buffer[]="Hello";
```

- **double**

```
double testVal=100.78;
```



Declaring variables in Objective-C will be familiar to any C, C++, C# or Java developer.

The basic syntax is a data type followed by a variable name with an optional assignment clause where the variable can be initialised.

QAIOSDEV v1.5

## Objective-C Data Types (Mac OSX, iOS)

- **Scalar types**

| Type (iOS) | Size (bits) | Range (signed) | Range (unsigned) |
|---|---|---|---|
| int | 32 | ± 2147483648 | 0 to 4294967295 |
| short | 16 | ± 32768 | 0 to 65535 |
| long | 32(iOS) 64(OSX) | iOS(as int) OSX(long long) | iOS(as int) OSX(long long) |
| long long | 64 | ± 9.2233 E18 | 0 to ~ 1.8446 E19 |
| BOOL | 8 | NA | NA |
| char | 8 | -128 to 127 | 0 to 255 |

| Type (iOS) | Size (bits) | Min | Max |
|---|---|---|---|
| float | 32 | 1.175494E-38 | 3.402823E+38 |
| double | 64 | 2.225074E-308 | 1.797693E+308 |

39

As with any advanced programming languages Objective-C has a basic set of fixed size primitive (or scalar) data types directly inherited from C. These basic data types are used to represent the simplest kinds of data such as text, integers, floating point numbers and boolean values.

On creation, scalar data types are usually stored on an applications stack which is a structure used to keep track of state during and calls to the numerous function calls that occur during a programs execution.

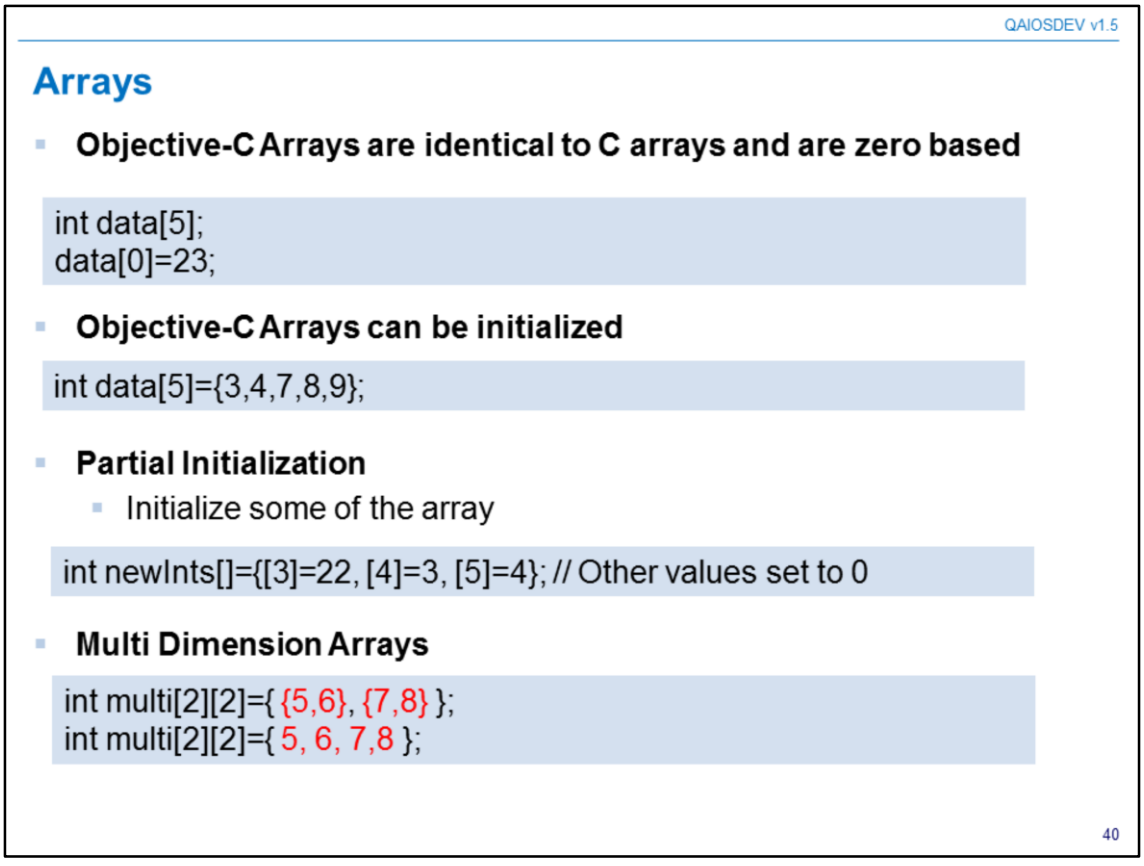
int is the basic whole number data type, which depending on platform, can occupy 32 bits giving it an unsigned range (positive values only) of 0 to $2^{32}$ -1 or signed ± $2^{16}$ -1.

short and long are used as type modifiers where short reduces the storage size of an int to 16 bits and long increases it to 64.

The size of int, short and long are OS dependent. See table above. To get a long int the variable has to be declared as long long.

float is a single precision floating point number and double is double precision number. By default the compiler will store all floating point numbers as doubles regardless of your declaration. To force a number to be treated as a float you

have to place f at the end of the number e.g. float num=12.6f;



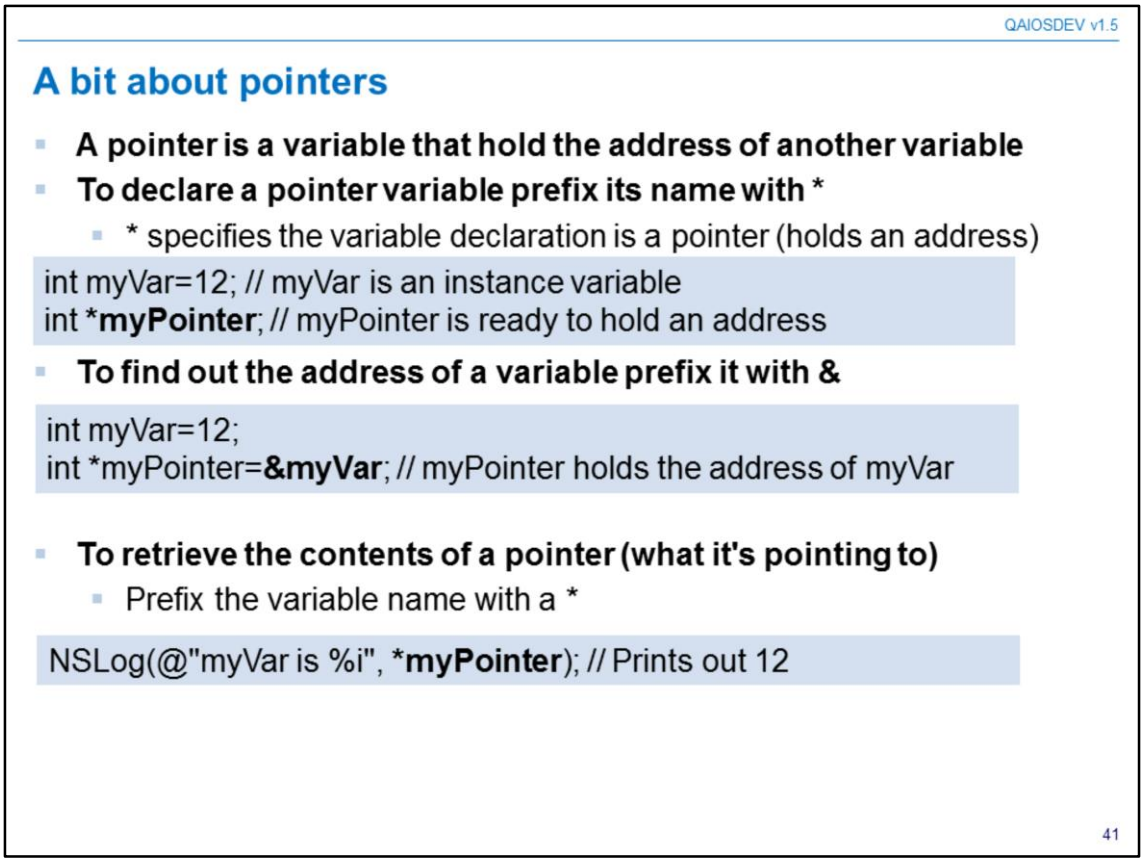

Arrays can be defined as in C and you can initialize an array as part of the declaration.

Using curly braces you can provide a comma separated list of initialization values. It's also possible to partially initialize the array by providing index specifiers in the initialization list.

It's even possible to initialize multi dimensional arrays. Initialization braces can only be used for initialization. You can't use them for assignment to variables later on.

The main downside to arrays is there lack of extended support. You can't easily find the length of an array for example.

In a later chapter, we will see how the framework classes NSArray, NSMutable array and NSDictionary give rich collection processing support. But for pure performance, it's hard to beat the simple array.




Pointers are simply variables that hold the memory addresses of other variables. They are useful because they allow us to share variables directly with other parts of our application.

C is a language that relies heavily on the use of pointers. Objective-C uses pointers too but their use is light.

There are basically two kinds of variables. Instances variables which hold the actual data and pointer variables, which hold the address of instance variables. They are easy to identify because pointer variables are always declared with an * character in front of them.

```
int myVar=23;  //Instance
int *myPointer; //Pointer
```

Pointers are always specific to the data type they are pointing to so int * is for pointers to integers.

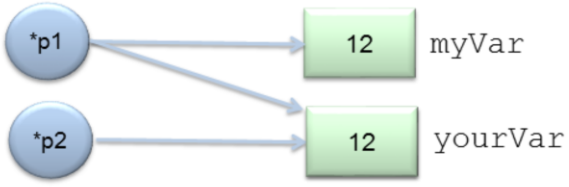
QAIOSDEV v1.5

## A bit more about pointers

- **Take care when assigning pointers**

```
int myVar=12, yourVar=24;

int *p1=&myVar;
int  *p2=&yourVar;
```

```
*p2=*p1; // By Value assignment
```




43

You have to be careful when manipulating pointers. Specifically when assigning.

From the above slide, assigning the contents of one pointer to the contents of another is straightforward enough and looks like this:

*p2=*p1;

Both pointers are still pointing to their own variables it's just that now they contain the same value.

This is called assignment by value where you are just manipulating the contents of one of the variables via its pointer.

The problem comes when you instead assign the pointer variables directly as in:

p2=p1;

This is entirely different because now you have assigned the address of p1 to the p2 pointer which means p2 is now pointing to the same variable as p1.

This is a common mistake and often not what was intended. Good to remember.

QAIOSDEV v1.5

## NSLog

- **Useful logging command**
- **Writes to XCode output window**

```
int myValue=5;
NSLog(@"myValue=%i", myValue);
float newValue=20.45;
NSLog(@"myValue=%i, newValue=%f", newValue);
```

- Useful format strings

| Format String | Use |
|---|---|
| %i | Integer (signed) |
| %@ | Objective-C Object (NSString) |
| %s | C String |
| %f | float, double |
| %c | character |

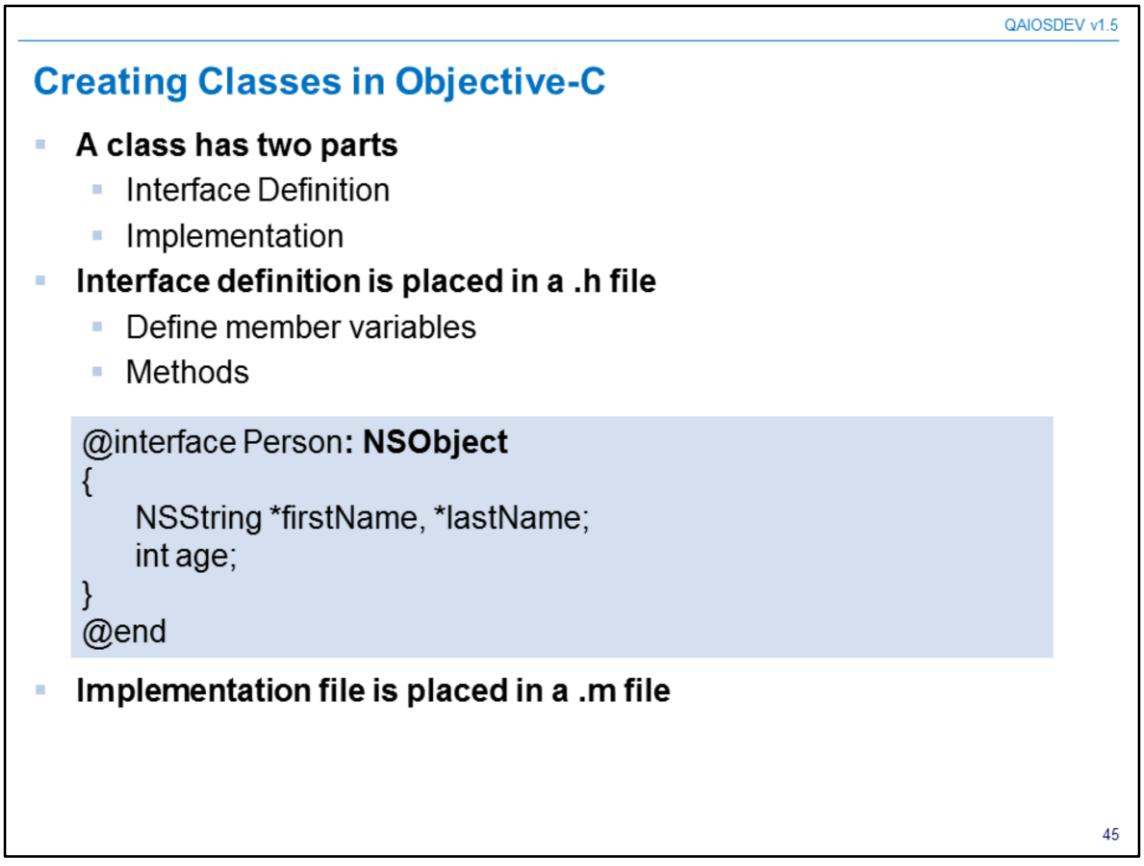
44

Objective-C has a very useful utility function called NSLog. It is widely used to write debug info to XCode's console window.

Its syntax is NSLog(@"FORMAT STRING", format string parameters);

Although XCode has good debugging tools, you will use this a lot.

| | |
|---|---|
| %@ | Objective-C Object |
| %% | '%' character |
| %d, %D,%i | Signed 32-bit integer |
| %u, %U | Unsigned 32-bit integer |
| %hi | Signed 16-bit integer |
| %hu | Unsigned 16-bit integer |
| %qi | Signed 64-bit integer |
| %qu | Unsigned 64-bit integer |
| %x | Unsigned 32-bit integer (hex) |
| %X | Unsigned 32-bit integer (HEX) |
| %qx | Unsigned 64-bit integer |
| %qX | Unsigned 64-bit integer |
| %o, %O | Unsigned 32-bit integer (Octal) |




As an object oriented language Objective-C allows us to define classes.

This actually happens in two parts because Objective-C separates a class's definition (properties and methods) from its actual implementation (the code). Usually, the interface definition and the implementation are stored in separate files with the interface stored in a file with a .h extension and the implementation with a .m extension.

The interface of a class is usually public and allows other developers to see its properties and methods. The implementation can be compiled and linked into a library binary that hides the actual implementation of the class from the developer. To reuse a library, all a developer has to do is add the library to their project and then include the .h file for the relevant class in their own code wherever they need to use it.

A class definition always starts with @interface and ends with @end. Classes can hold their own instance variables, commonly know as ivars. These are placed inside curly braces {} and follow the variable declaration rules discussed in the previous chapter.

NSObject is the base type of all Objective-C objects and allows your class to behave as a dynamic runtime object.

## Controlling Access to class data

- **Class instance variables (ivars) can be**
  - public (accessible everywhere)
  - private (accessible from within class only)
  - protected (default) (accessible within class or within derived classes)
  - package – public to all classes in executable, private externally

```
@interface Person
{
    @public
    NSString *firstName, *lastName;
    @private
    BOOL enabled;
   @protected
    int age;
}
@end
```

46

You can control access to ivars by adding access modifiers in front of their declaration.

As you can see above @public has been added in front of firstName and lastName meaning they are publicly accessible from outside of the class. This means when an object is created of type Person the code can get direct access to firstName and lastName. This isn't always a good idea because one of the benefits of encapsulation is data protection.

To prevent outside access you can instead mark your ivars @private which, as the name suggest means they are inaccessible externally but can be accessed by all methods within the class.

@protected is the default access and is like @private but it also allows ivar access to any classes that inherit from it. We will look at class inheritance later on.

## Creating objects

- **Referencing a class**
  - Before you can use a class you must import its definition

```
#import "Person.h"
```

- **Two stage process to create an object**
  - Allocate (alloc)
  - Initialize (init)

```
Person *person2=[[Person alloc] init]; // Shorter syntax, v. common
```

- **new method simplifies this further**

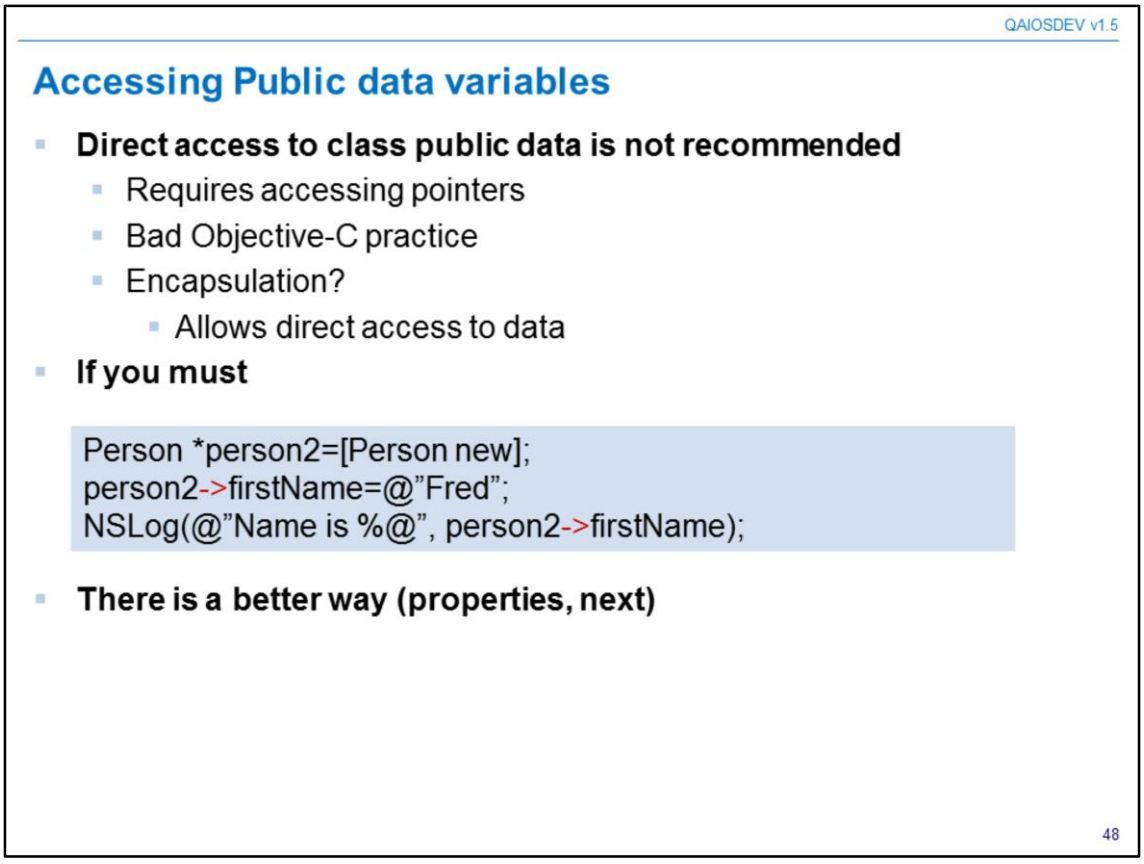
```
Person *person1=[Person new]; // Calls alloc/init
```



Creating an instance of an object is a two stage process. You first have to allocate it using the alloc statement. Next you need to initialize it using an init statement.

You can provide your own custom versions of init that take parameters and act as a kind of constructor.

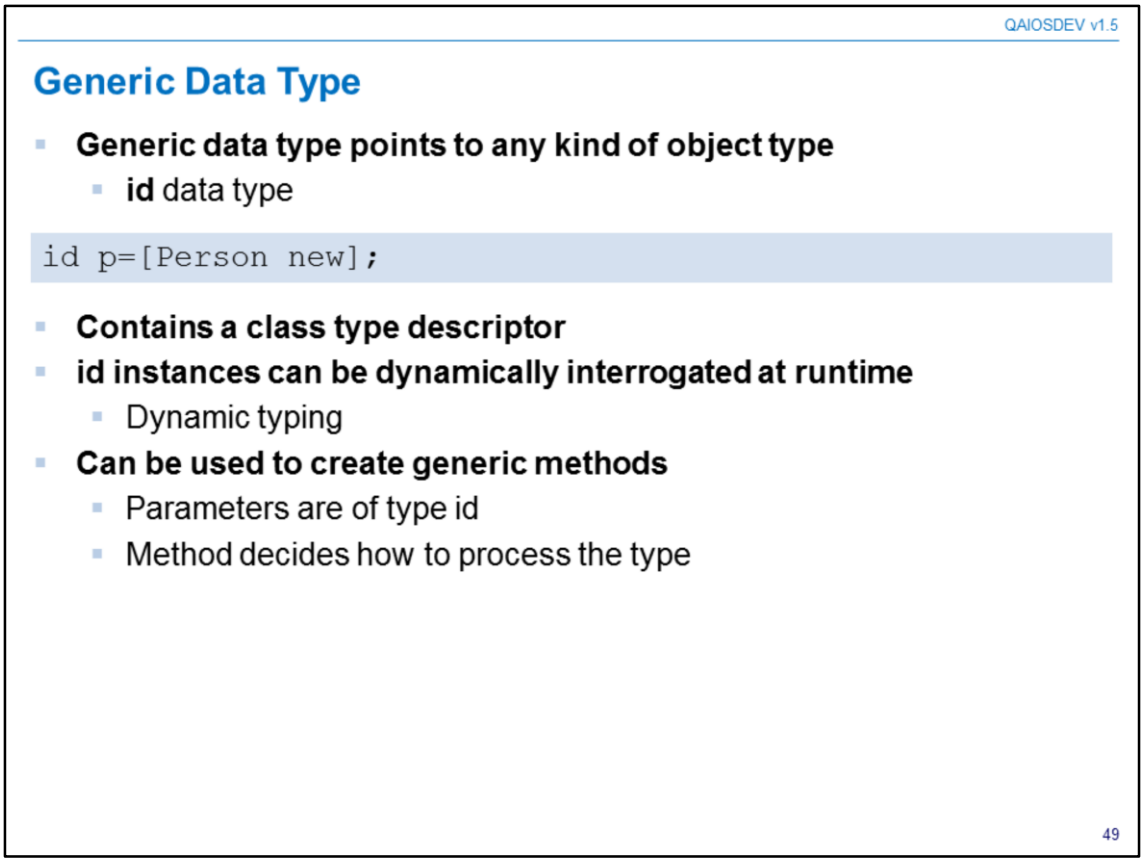
In Objective-C 2.0 the new keyword simplifies things by calling alloc/init for you.




To access a public variable you have to directly dereference its pointer using the -> operator.

This goes against the principles of encapsulation and allows direct unchallenged access to an object's data. As a result, it's not recommended but if you need to, it's here.

We will see later on how you can define properties to control access to your classes data.

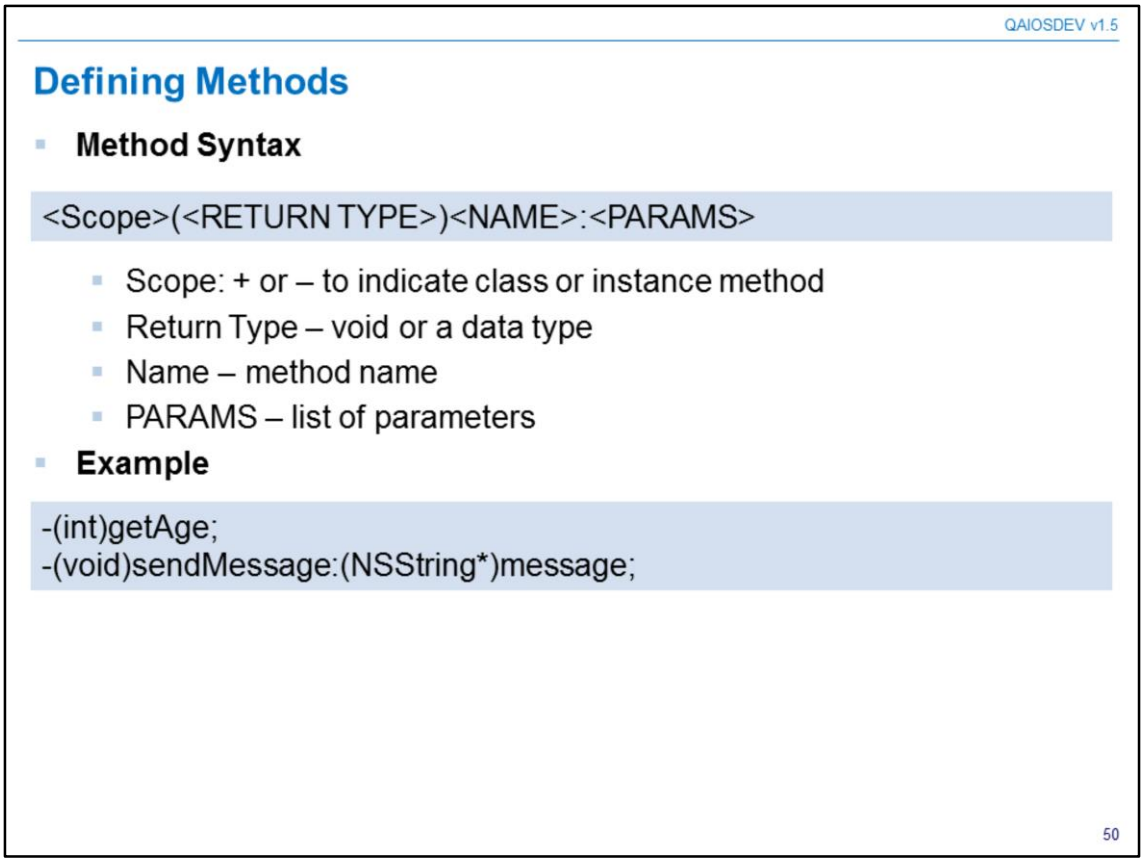



The id data type is a scalar type that can be used as a pointer to any kind of object type. The Objective-C compiler treats it as a special case and is a bit more complex than just a simple pointer.

You can use id to make your code more generic. With it you could write generic methods that take id parameters and allow the methods themselves to decide how to handle the objects passed in.

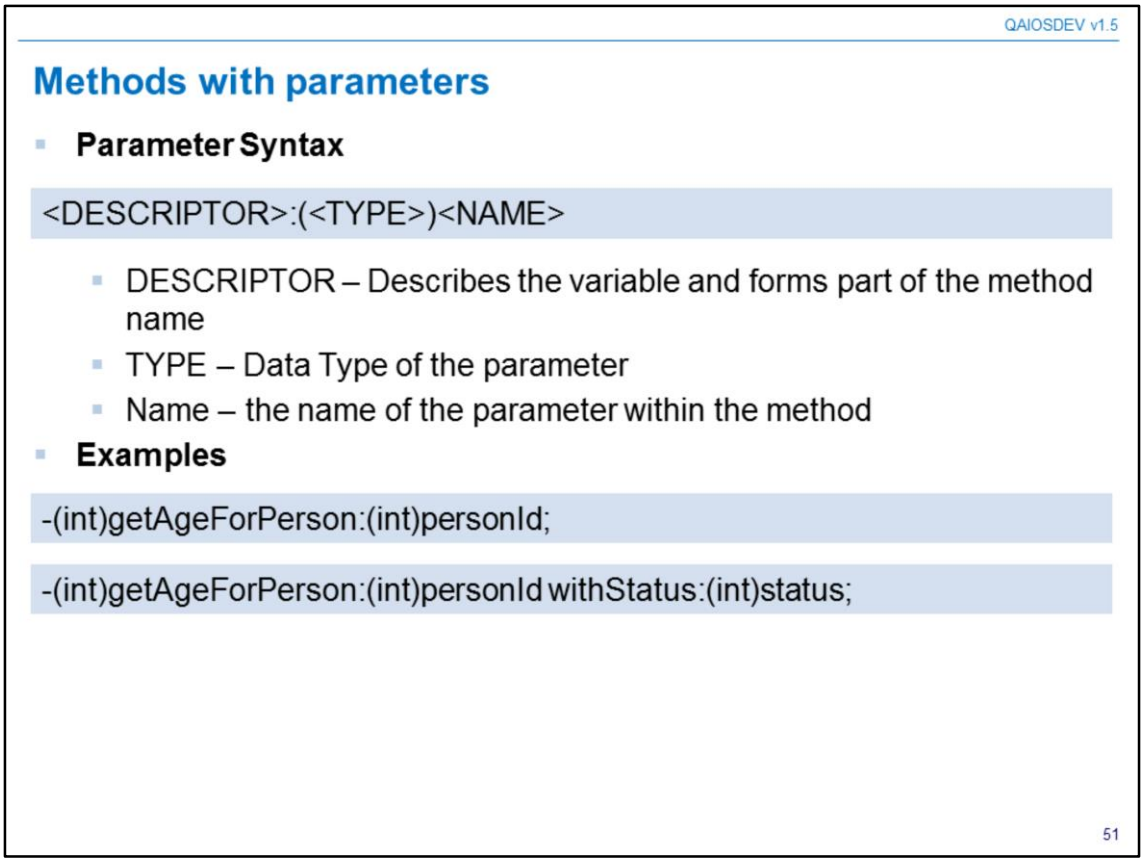
One of the great things about the id data type is that it is self-describing. This means, at runtime, it's actual data type can be determined and code can respond accordingly.

This means you can write generic code that makes decisions at runtime on how to handle objects based on the type of objects it's dealing with at the time.

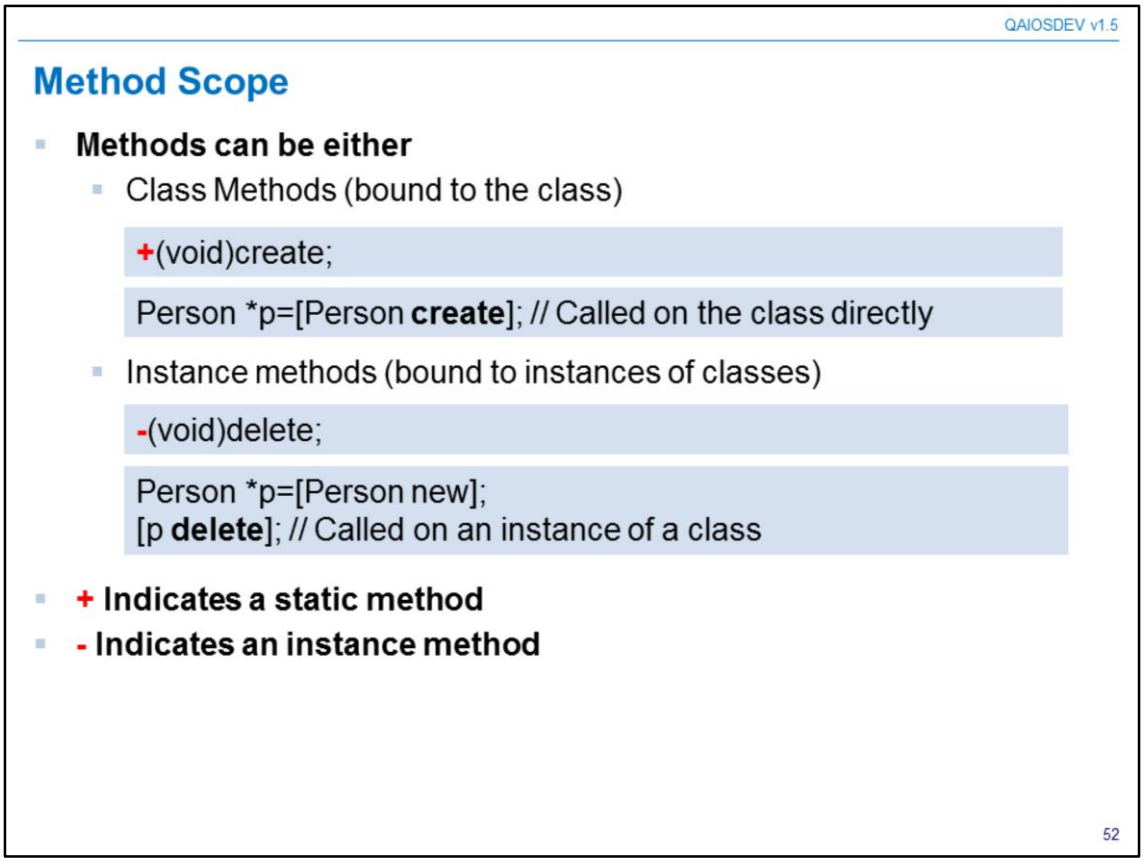



You can also add method definitions to the interface. Here you describe all of your methods, return types and parameters.

You also indicate if a method has class scope or instance scope both of which are described in the next slide.




Parameters have a descriptor that you should use to describe the parameter. A method's descriptors actually form part of the method name.
Following the descriptor is a colon, the parameter type within brackets and the parameter name.




A class can have two kinds of methods Class methods and Instance methods.

A class method can be accessed directly from the class itself. You don't have to declare an instance you; can just use the method directly. These methods are useful for class wide behaviour.

Instance methods are specific to instances of classes and so you can only use them after having declared an instance of a class first.

+ in front of the class name indicates a class method. – indicates an instance method.

## Bringing it all together

- **Person.h**

```
@interface Person: NSObject
{
    @public
    NSString *firstName, *lastName;
    @private
    BOOL enabled;
   @protected
    int age;
}
+(void)create;
-(NSString*)getFirstName;
-(void)setFirstName:(NSString*)name;


@end
```



Bringing it all together you can see the method definitions are placed within the interface outside of the curly braces.

## Implementing methods

- @interface defines the class
- @implementation is where it's implemented (.m file)

```
#import "Person.h"

@implementation Person

-(NSString*)getFirstName
{
        return firstName;
}

-(void)setFirstName:(NSString*)name
{
        firstName=name;
}
@end
```

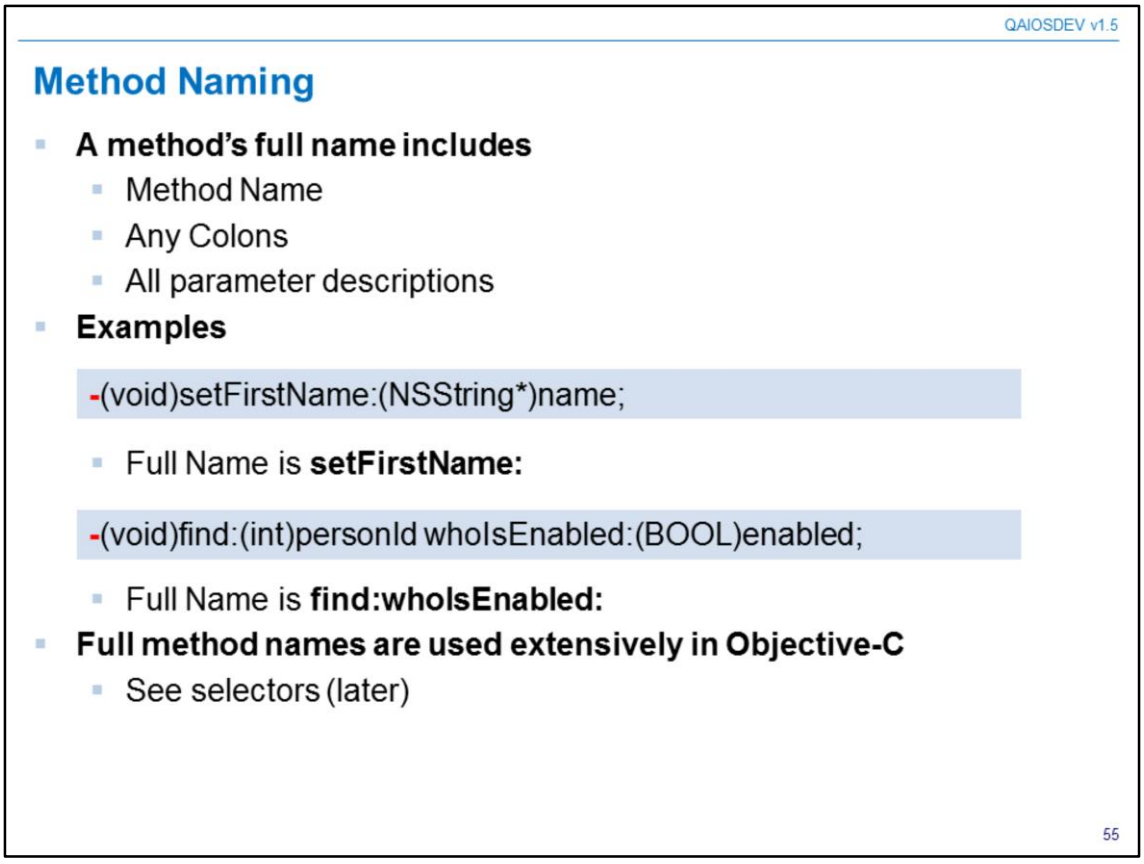


After defining the class the next task is to provide the actual implementation for it.

Implementation code is added to files with a .m extension. For classes you need to import the class definition defined earlier in the .h file.

import will include a definition file in the implementation and guarantees that it will only ever be loaded once. It's common to add multiple import statements to your classes when reusing classes.

To import your own files you surround the file name in quotes. For framework classes you use < >.




All but the first method parameter have a parameter descriptor that actually form part of the method name.

To determine a full method name you have to include any colons (:) and any parameter descriptors.

So

-(void)getPerson:(int)personId forAge:(int)age; has a full method name of getPerson:forAge:

Understanding full method names is important because they are used extensively in Objective-C.

## Calling Methods

- **Objective-C doesn't call methods directly**
  - Sends messages to objects (hence odd syntax)
  - Runtime binds message, object and method together
- **Instance methods**

```
[person1 setFirstName:@"Fred"];

NSString *name=[person1 getFirstName];

NSLog(@"Person is %@", name);
```

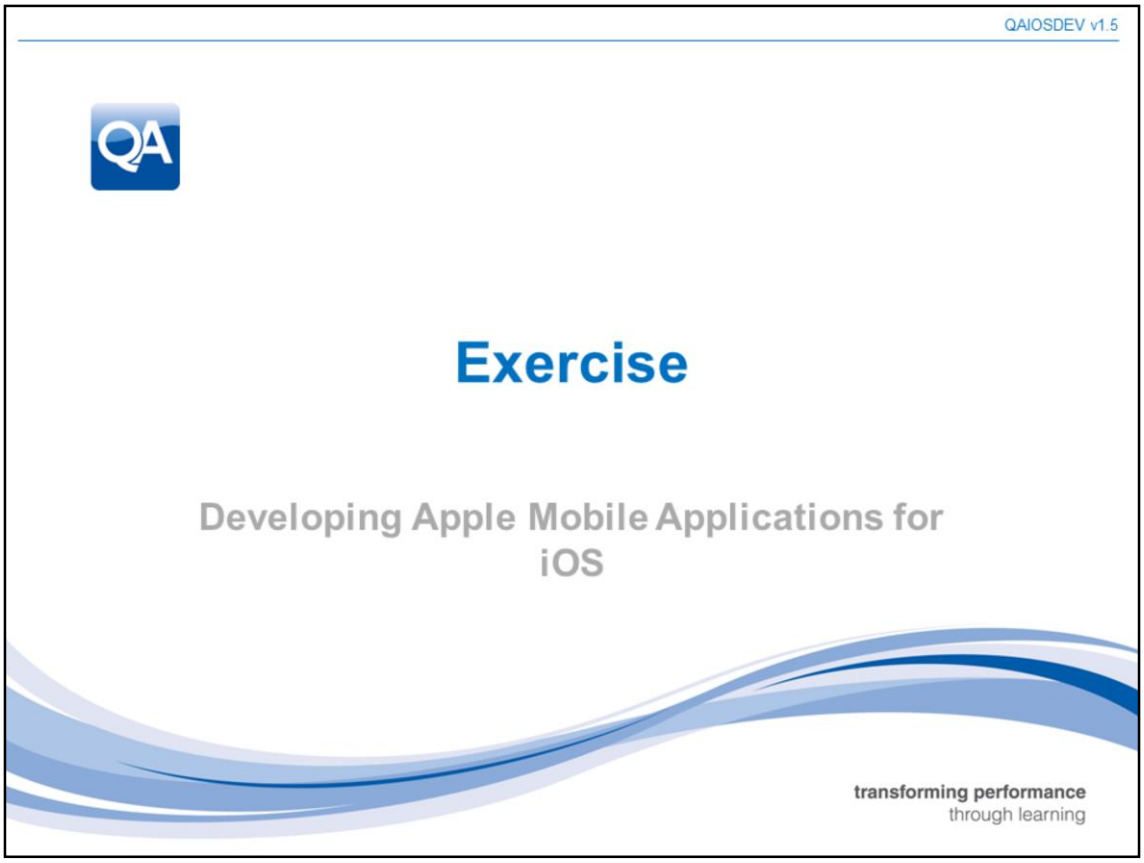
- **Class Methods**

```
Person *person1=[Person create];
```



To call a method on an object in Objective-C you actually have to send the object a message instead. This message contains all of the necessary parameters and information necessary to make a method call but it's up to the dynamic runtime to bind the message to an object's actual method implementation. Most other programming languages directly bind method calls to implementations at compile time. Objective-C's decoupling of the process allows developers to create dynamic algorithms not possible otherwise. Objective-C classes, for example, can be written to redirect attempts to bind messages to methods they don't support a technique known a message forwarding.

Instance methods can only be called on a declared instances of a class. Class methods can be called directly on the class itself.


QAIOSDEV v1.5
Exercise
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Summary

- **Objective-C Object Oriented C**
- **Objects send messages to each other**
- **Objective-C Runtime manages messaging**
- **Basic set of data types**
- **Classes allow**
  - Local instance variables
  - Methods
  - Properties

## Pre Compiling Methods

- **Methods are compiled to C Functions**

```
@implementation Person
- (void)saySomething:(NSString*)message
{

}
@end
```

- **Becomes C function**

```
void -[Person saySomething:](id self, SEL _cmd, NSString*
message)
{


}
```

59

Just as classes are precompiled into instances of structs, methods are precompiled into C Functions. The class name and methods name are combined to produce a C function name where the caller and selector are passed in as parameters along with the original parameters in the methods.

## Class Metadata

- **Class info stored as instances of a C struct**

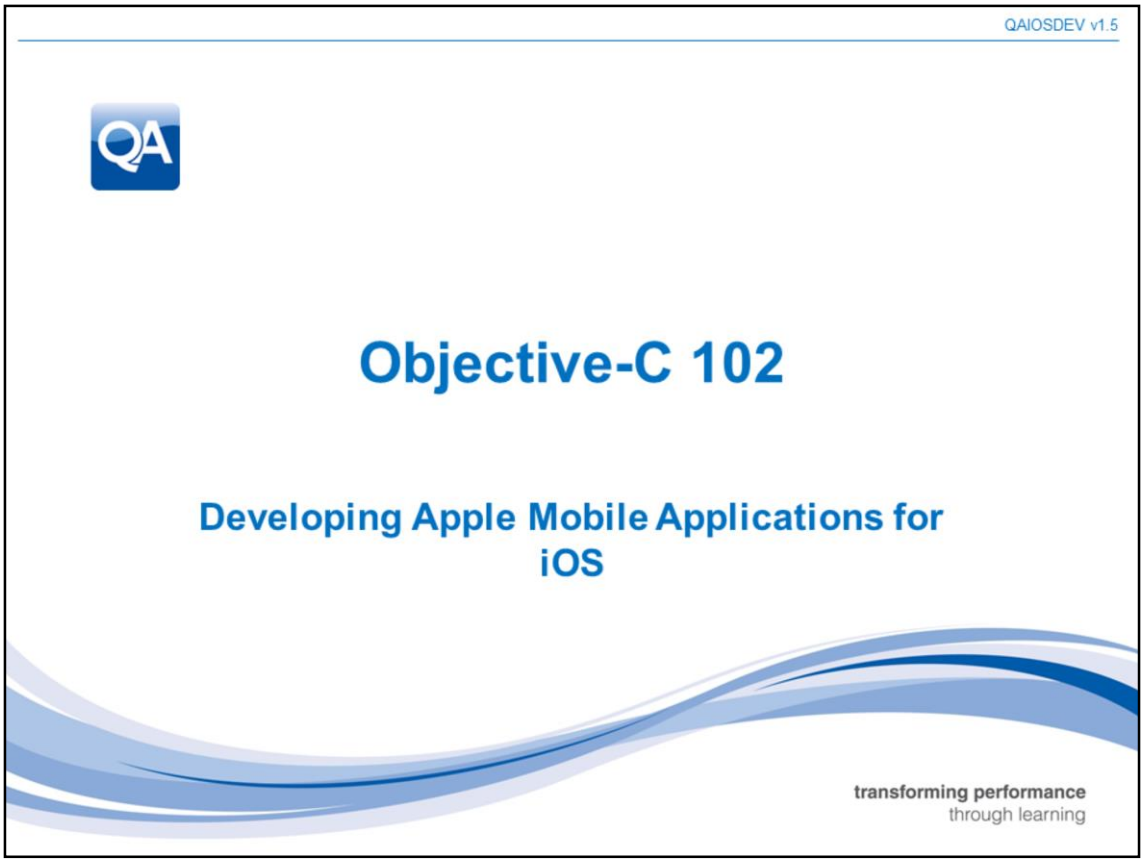
```
struct objc_class {
    Class isa;
    Class super_class;
    const char *name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
}

typedef struct objc_class *Class;
```



obc_class is a C struct that is used to hold the metadata for classes.

It includes the name, super class, size and lists of ivars and a pointer list of a it's functions.

Notice the class has an isa pointer that points to the classes root base class. This is often referred to as the meta class and in Mac and iOS development will usually be NSObject.


QAIOSDEV v1.5
Objective-C 102
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - To refresh your Objective-C knowledge
- **Contents**
  - Overview
  - Data Types
  - Creating Classes
    - Properties
    - Methods
    - Inheritance
  - Protocols
  - Delegation
  - Selectors
  - Categories
- **Hands on Lab**

## Properties

- **@property defines the property in the interface – creates**
  - getter method
  - setter method
  - instance variable

```
@interface person: NSObject

@property (nonatomic, strong) NSString  *firstName;

@end
```



Adding a property to a class couldn't be simpler as most of the work is done for you. In the interface file add the @property definition including the name and data type. You also include some attributes that define how the attribute will behave.

Next add an @synthesize statement in your implementation file and the compiler will do the rest of the work creating the property for you.

## Using Properties

- **Properties allow the familiar dot syntax**

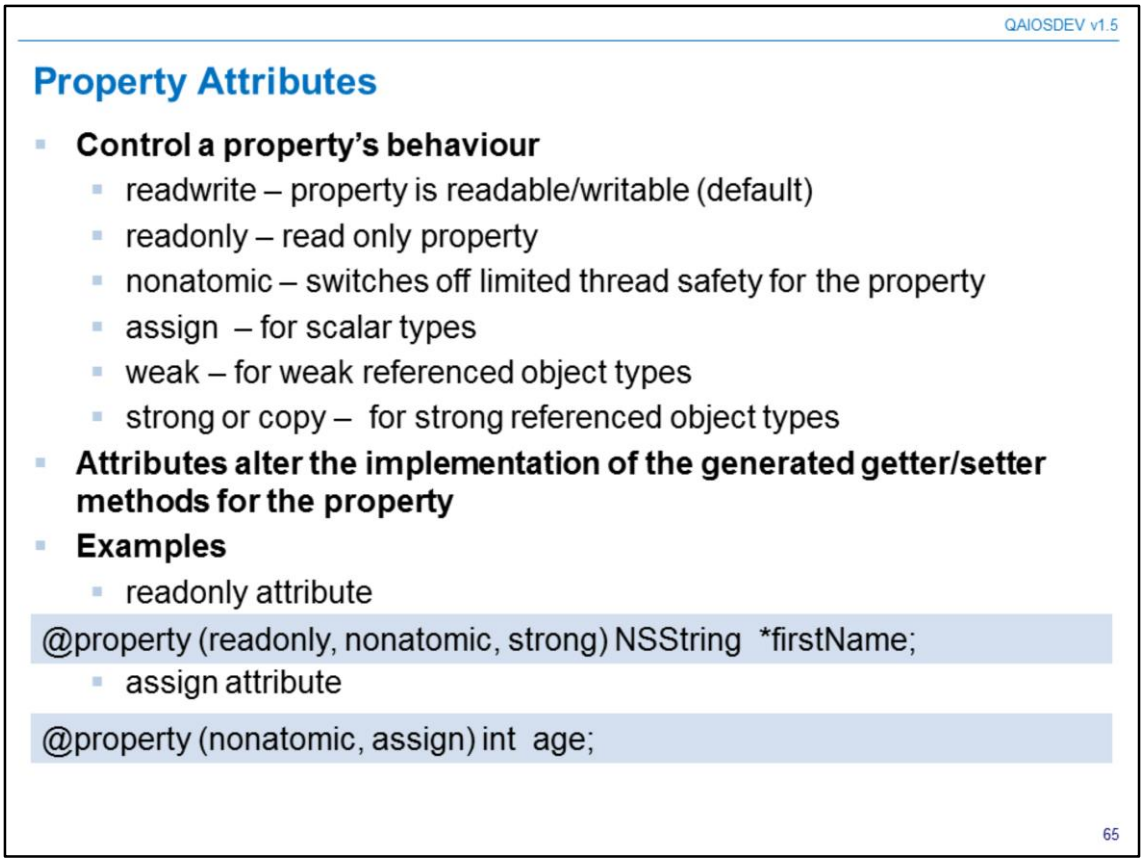
```
person *p=[person new];
p.firstName=@"Fred";
```

- **Properties generate getter/setter methods**
  - Access properties via methods

```
NSString *s=[p firstName];

[p setFirstName:@"Fred"];
```

64

Using a property is really simple. Just add a "." after your variable name followed by the property name and that's it.




Property attributes control the auto generated getter/setter code.

If you mark a property readwrite, which is the default, both a getter and setter will be generated. readOnly creates only the getter.

The nonatomic flag switches of some limited thread synchronization code that can help multi threaded access to properties. If you don't switch it off, there will be some locking involved in your property access, which is more overhead than you need if you aren't running multi threaded.

Assign, strong, weak and copy relate to memory management. Assign is used for scalar type properties. strong is used for object properties that will retain a reference to objects assigned to them. weak properties won't retain a reference.

Retain will always ensure that when an object is assigned to a property the property becomes it's owner. Lets' leave it there for now.

## Inheritance in Objective-C

- **Supports Single Inheritance**
- **Ultimate Base Type**
  - NSObject
- **Defined in the interface section (.h)**

```
#import "animal.h"

@interface person: animal
{
    NSString *firstName, *lastName;
    BOOL enabled;
    int age;
}
@end
```

66

Objective-C supports single inheritance, which means a class can only inherit from a single base class. Some OO languages allow classes to inherit from multiple classes at the same time.

For Mac and iOS applications the base class of all objects is the standard NSObject. Whenever you create a new class, you should make sure you inherit from NSObject first.

To inherit from a base type you just have to modify your class definition (.h) file. Add a colon to the end of your class name followed by the name of the base class you want to inherit from and your done.

Make sure to import the .h file for the class your inheriting from first though.

## Overriding Methods

```
@implementation person

-(void)delete
{
    [self executeDeleteFromDatabase];
}

@end
```

```
@implementation employee // inherits from Person

-(void)delete
{
    [self sendHRNotification: self.employeeId];
    [super delete];
}
@end
```

67

A derived class can add as many new methods as it needs to. One of the key features of inheritance is the ability to override existing methods as well.

Overriding simply involves redefining a method that already exists in the base type. In the example above, the delete method on the person class is overridden by the employee class that inherits from it.

The overridden method can perform its own specialist behaviour and then if it needs to call the functionality of the base type by calling:

[super delete];

Super just means either this class's parent type or one of its ancestors. So calling [super delete] is calling the delete method on the person class as well thereby reusing its functionality. You don't have to do this but it makes sense if you want to reuse base class functionality.

## Protocols

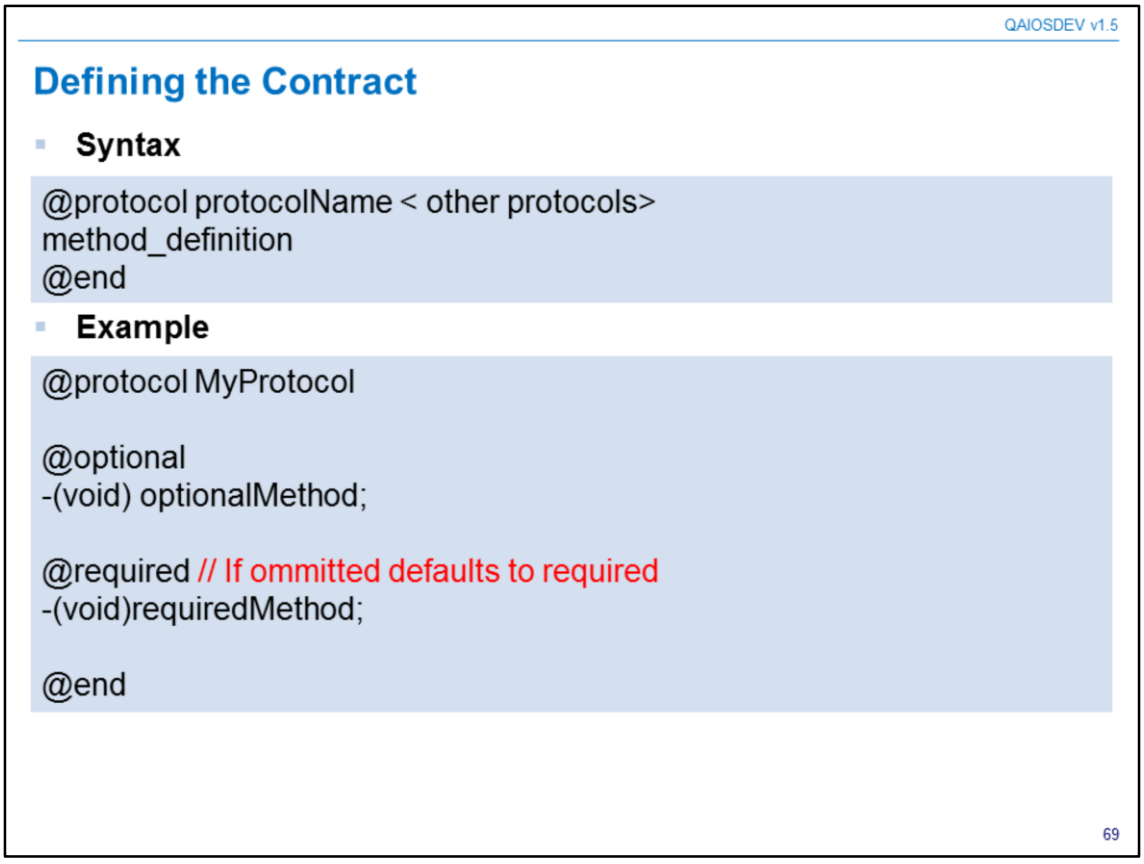
- **Defines a set of methods for a class to adopt**
- **Acts as a kind of contract with a class**
  - If you agree to adopt me then you need to add these methods
- **Similar idea to interfaces in other languages**
  - Objective C 2.0 protocols can have optional methods
  - A class can conform to a protocol without being formally declared to implement it
- **Classes use protocols to advertise**
  - Messages they handle
  - Messages they send (events generated eg mouseMove)
- **Protocols and delegates are used to create and handle events between classes**



A protocol is a definition of a set of methods that a class can formally adopt. It acts as a kind of contract in that the class agrees to implement at least the required methods of the protocol.

Objective C protocols can have optional methods and a class doesn't have to be formally attached to a protocol to conform to it as long as it implements its methods.

Classes use protocols to describe the messages they can handle and the messages the can send to other objects.




You define a protocol within in a .h file using the **@protocol** statement

As part of the definition you can also define a list of other protocols that form part of the one you are defining. By doing this, your protocol is taking on the contract requirements of the other protocols.

In the example, MyProtocol defines an optional method and a required method defined using the standard method definition syntax.

## Implementing a protocol

- **Add protocol to class definition**

```
#import "MyProtocol.h"
@interface newClass: NSObject <MyProtocol>
{
    NSString *name;
}
@end
```

- **Add methods to the implementation**

```
@implementation newClass
-(void)optionalMethod {}
-(void)requiredMethod {}
@end
```

- **Checking Conformity**

```
BOOL isOk=[myObject conformsToProtocol:@protocol(MyProtocol)];
```

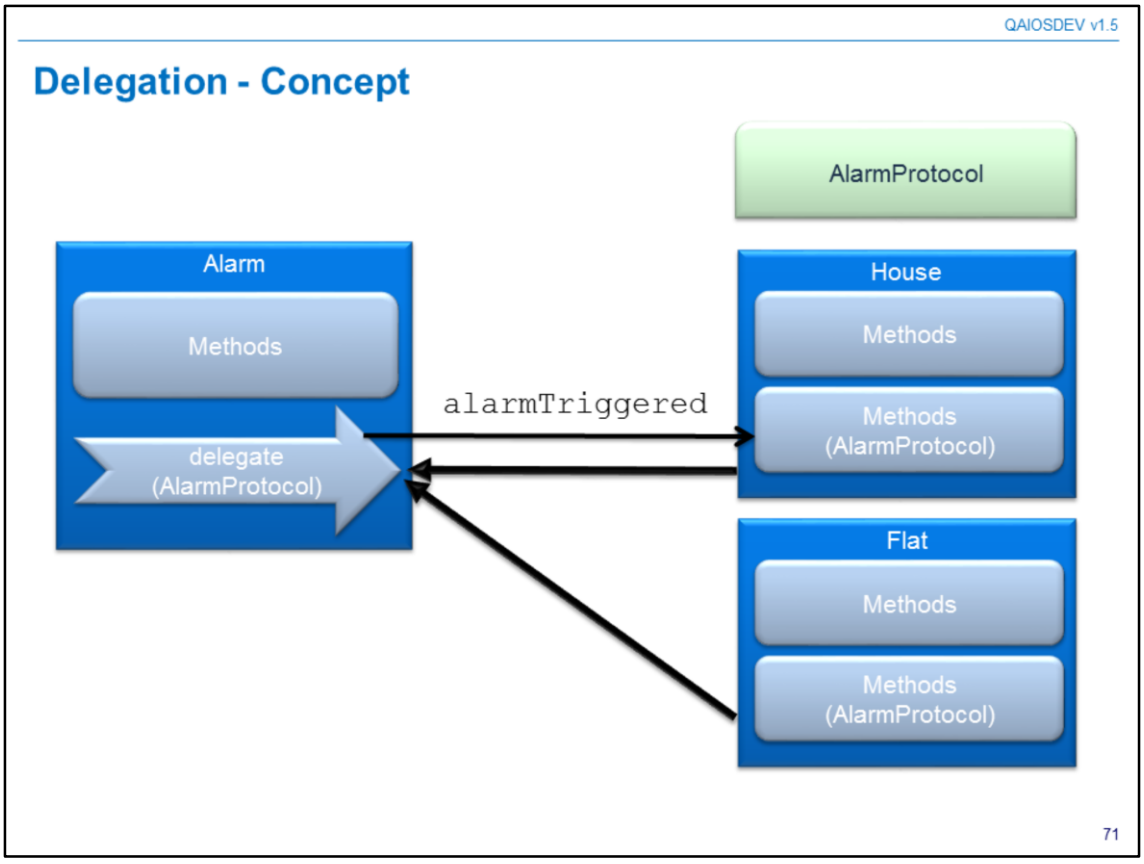
70

Here newClass class implements myProtocol and inherits from NSObject.

You can see in the implementation file that it has implemented both the optional and required methods.

To check if an object conforms to a protocol call the **conformsToProtocol** method, which returns BOOL;

```
newClas *myObject=[newClass new];
if ([myObject conformsToProtocol:@protocol(MyProtocol)])
{
}
```




Delegation allows a class to invoke method in other classes without it having any knowledge of the implementation of those classes. It gives a class a generic event generation mechanism and is widely used in Objective-C programming.

In the example above, the Alarm class provided a delegate property that can hold a reference to any other class as long as that class implements the AlarmProtocol. It does this so it can call Alarm protocol methods such as alarmTriggered when it wants to “raise the alarm”.

Classes such as House or Flat register themselves with the alarm by assigning themselves to the delegate property allowing Alarm instances to call their AlarmProtocol methods.

## Creating the event source

```
#import "AlarmProtocol.h"
@interface Alarm: NSObject

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

@end
```

```
@implementation Alarm

@synthesize delegate;
-(void)activate{
  sleep(5);
  [self.delegate alarmTriggered];
}
@end
```



Alarm has defined a delegate property called "delegate" with an id data type and the implemented protocol.

In its **activate** method, it sleeps for five seconds then calls the alarmTriggered method in the class that has delegated itself to this class's instance.

Simple delegation in action. Any class that implements the AlarmProtocol can delegate itself to the Alarm class and receive alarm triggered events.

## Delegating to the Event Source

```
@implementation House // implements AlarmProtocol

-(void)setupAlarm
{
    self.alarm =[[Alarm alloc] init];
    self.alarm.delegate=self;
    [self.alarm activate];

}
-(void) alarmTriggered
{
        NSLog(@"Alarm has been raised!");
}
@end
```



Here we are setting up delegation in a class that implements the AlarmProtocol. Notice the alarmTriggered method.

In **setupAlarm** and instance of an Alarm class is created and its delegate property is assigned to self. This means when the **alarmTriggered** event fires it will fire in this class.

Finally, **[self.alarm activate]** is called. The result is a 5 second sleep followed by the **alarmTriggered** method firing and “Alarm has been raised” is written to the output.

## Selectors

- **Wrappers for method calls**
  - Used to pass methods as parameters
  - Call methods dynamically
- **Selectors use the full text representation of the method name**
  - Includes all colons and parameter descriptors

```
-(void)speakThis:(NSString*)speech
```

  - Selector name would be **speakThis:**
- **Creating and calling a selector**
  - Selectors can be passed as parameters to other methods

```
SEL sel=@selector(speakThis:);

Dog *dog=[[Dog alloc] init];

[dog performSelector:sel withObject:@"Hello World"];
```

74

Selectors are kind of wrapper objects for methods. They are a way of making methods transportable so you can pass them to other objects for them to call when they need to.

To identify a method name, you need to use its full name, which includes its name plus its parameter descriptors including colons ":".

Method

-(void)speakThis:(NSString*)speech withAccent:(int)accentId

Would have a full name of speakThis:withAccent:

To create a selector just put your full method name (without quotes) inside the @selector function to return your selector as a SEL variable. You can now pass this variable as a parameter to other methods, assign it to other properties or call it directly on objects.

In the example, the selector is being used to call a method on the Dog object. The performSelector method can be used to call a method on an object using a selector. You can pass a limited number of parameters using withObject.

## Categories

- **Categories logically group class methods together**
  - Intended to reduce class complexity and aid maintainability
  - Each category would be maintained separately
  - Class method definition is built up from each of the categories
- **Used extensively to extend existing classes (even framework classes)**
  - You don't need the source code
  - You don't have to sub class
  - You are just extending the functionality of an existing class



Categories are a really simple concept but extremely useful and powerful. They allow you to extend existing classes, even classes you don't have source code access to without creating a new data type.

The main purpose of categories is to allow you to split the definition of a class into categorised sections to reduce complexity and maximise maintainability. Keeping all of the source for a complex class in just one class can be very difficult to work with during development.

The great side effect of categories is this ability to extend the existing framework classes. Adding additional functionality to existing classes gives enormous flexibility.

## Categories Example

- **Syntax**

```
@interface class_name ( CategoryName )
// method declarations
@end
```

- **Example**

```
@interface NSString (Speech)
-(void)speak;
@end
```

```
@implementation NSString (Speech)
-(void)speak
{
        NSLog(@"Speaking %@", self);
}
@end
```

76

To create a category, declare its interface using the name of the class to be extended followed by the name of the category contained within brackets. Then declare the method extensions.

In the implementation file, re-declare the implementation of the class with the category name and provide the implementation of the extension methods.

Typically, the category would be saved to a file using a filename containing the name of the class to be extended followed by a “+” sign and then the name of the category. In the above example the files would be **NSString+Speech.h** and **NSString+Speech.m**

## Core Framework Classes

- **NSString/NSMutableString**

```
NSString *surname=@"Bloggs";
NSString *s1=[[NSString alloc] initWithString:@"Bloggs"];
NSString *s2=[NSString stringWithFormat:@"%@",@"Bloggs"];
```

- **NSArray/NSMutableArray**

```
NSArray *colors=@[@"Red",@"Blue"];
NSArray *c1=[NSArray arraWithObjects:@"Red",@"Blue",nil];
NSString *color=colors[0];
color=[colors objectAtIndex:1];
```

- **NSDictionary**

```
NSDictionary *dets=@{@"lName":@"Bloggs", @"fName":@"Fred"};
firstName=dets[@"fName"];
```

77

## Summary

- **Objective-C Object Oriented C**
- **Objects send messages to each other**
- **Objective-C Runtime manages messaging**
- **Basic set of data types**
- **Classes allow**
  - Local instance variables
  - Methods
  - Properties
- **Protocols define method sets for a class**
  - Required
  - Optional
- **Delegation allows an eventing mechanism**
- **Selectors allow methods to be parameterised**
- **Categories extend existing classes**

## Pre Compiling Methods

- **Methods are compiled to C Functions**

```
@implementation Person
- (void)saySomething:(NSString*)message
{

}
@end
```

- **Becomes C function**

```
void -[Person saySomething:](id self, SEL _cmd, NSString*
message)
{


}
```



Just as classes are precompiled into instances of structs, methods are precompiled into C Functions. The class name and methods name are combined to produce a C function name where the caller and selector are passed in as parameters along with the original parameters in the methods.

## Class Metadata

- **Class info stored as instances of a C struct**

```
struct objc_class {
    Class isa;
    Class super_class;
    const char *name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
}

typedef struct objc_class *Class;
```



obc_class is a C struct that is used to hold the metadata for classes.

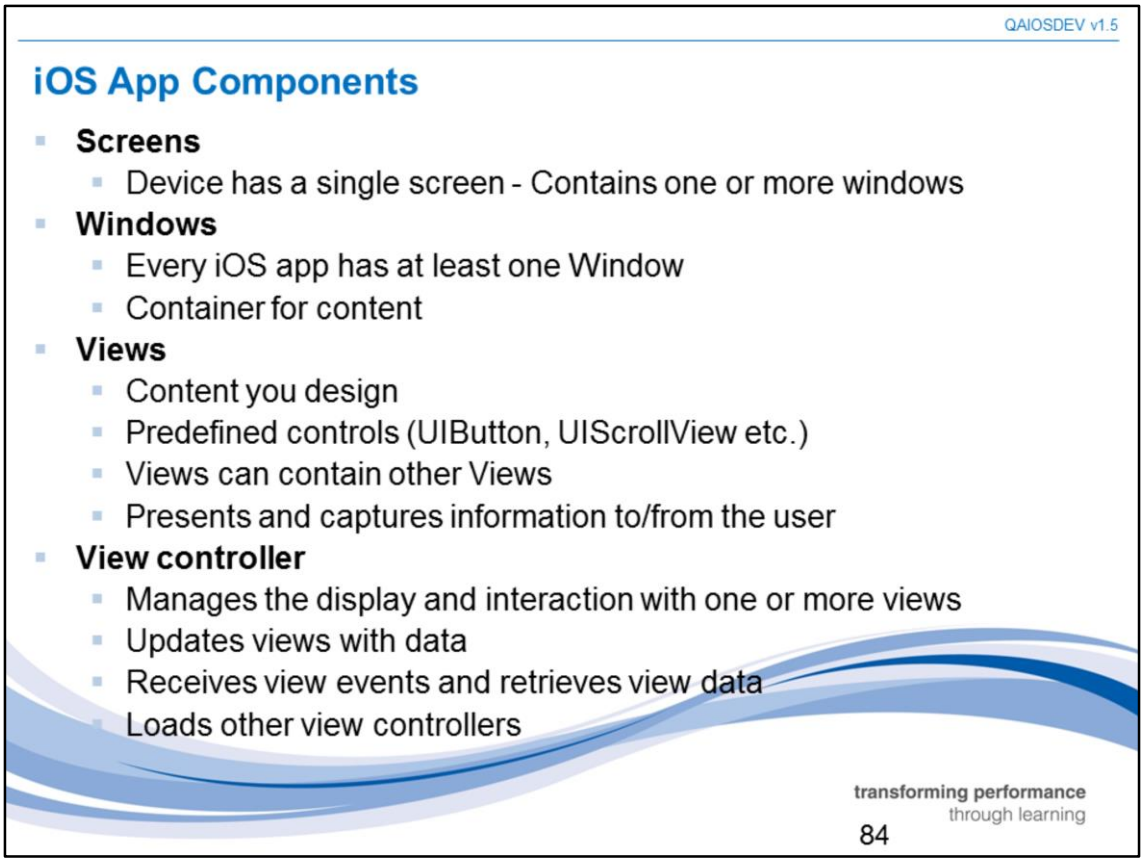
It includes the name, super class, size and lists of ivars and a pointer list of a it's functions.

Notice the class has an isa pointer that points to the classes root base class. This is often referred to as the meta class and in Mac and iOS development will usually be NSObject.


QAIOSDEV v1.5
Basic iOS App Architecture
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - To understand how an iOS app is constructed under the hood
  - To write code to build an app from scratch
- **Contents**
  - iOS Architecture Overview
  - Model View Controller
  - Windows, Views and View Controllers
  - Navigating between View Controllers
  - Handling orientation changes
  - Dealing with low memory events
  - Tab Bar Controllers
- **Hands on Lab**

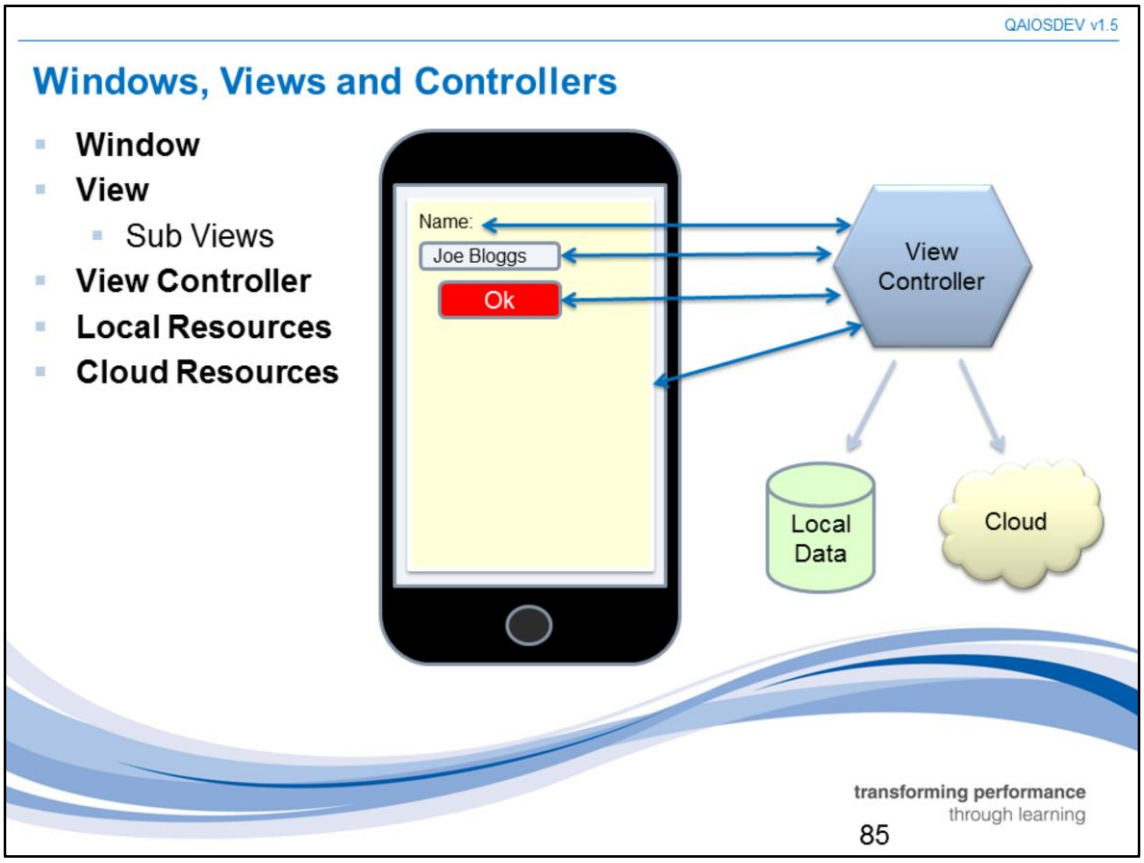



Every iPhone app has at least one window. A window is used to present content on the screen.

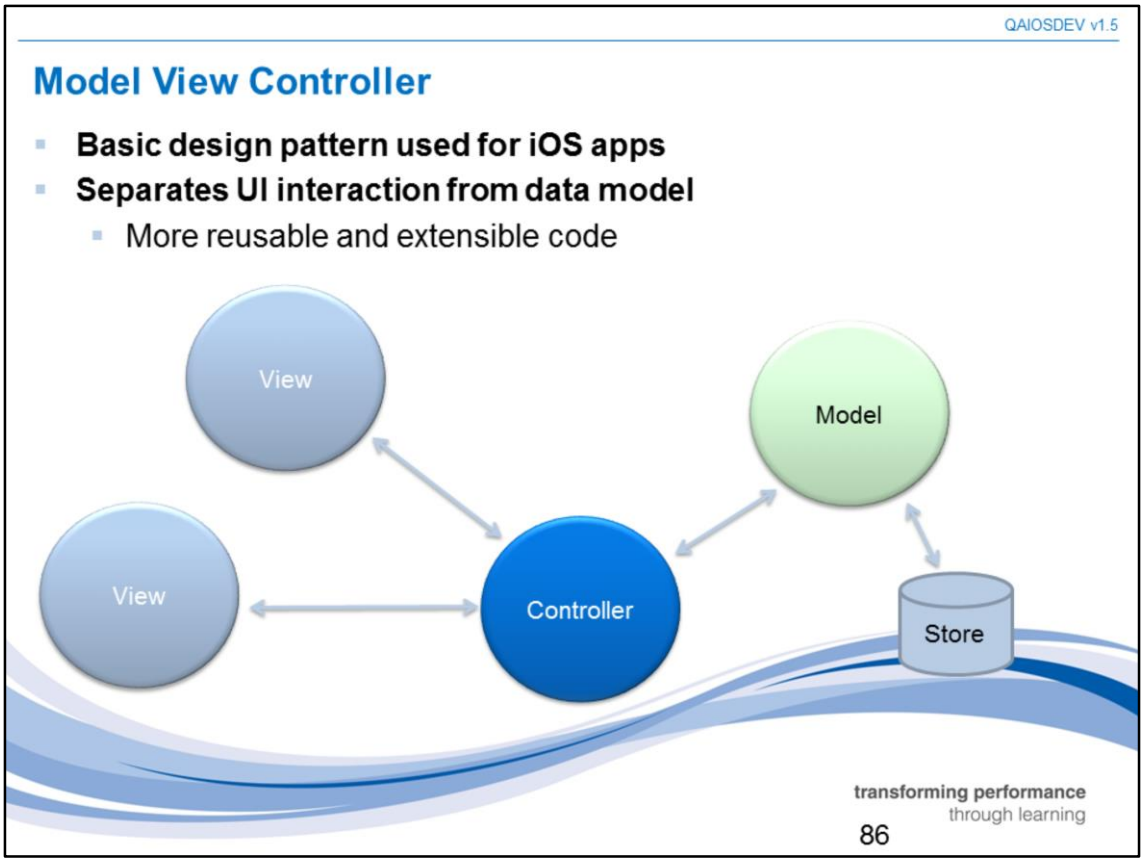
Views are used to present content onto a window and a window can contain multiple views. Views themselves can also contain sub views.

View controllers act as a managing interface between the view and your model (data storage). The view controller will update a view with data and that the view may allow the user to modify data. View controllers receive user events from views such as button clicks and other gestures that may trigger save actions causing data to be persisted to the model.

View controllers also manage system events like phone orientation changes and low memory conditions.




This screen demonstrates how an app hierarchy is build up with a window containing a view with sub views. All of the views are managed by the view controller that in turn may interact with local or cloud data sources.

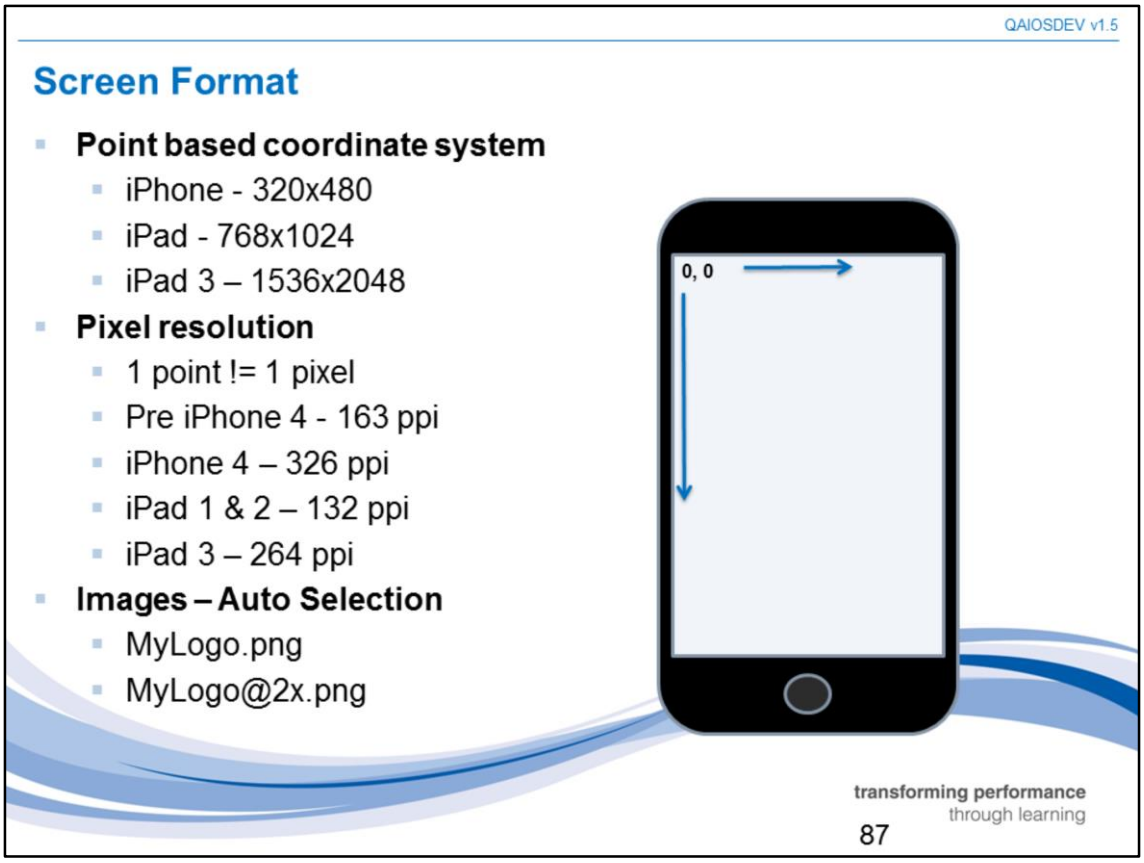



Model View Controller (MVC) separates the presentation logic (View) from the data management logic (Model) using an intermediary called the Controller.

The idea is really simple. A View informs the Controller when a user has changed the View's state, the controller updates the model and can inform the other Views of the change. Equally when the model changes the view can be updated to reflect the change.

The key to MVC is the View and the model don't usually interact directly.

Applications based on MVC tend to be easier to extend and contain code that is more reusable and testable. The controller can be directly unit tested using Xcode's built in unit test framework.




The iPhone coordinates system is zero based at the top left side of the screen.

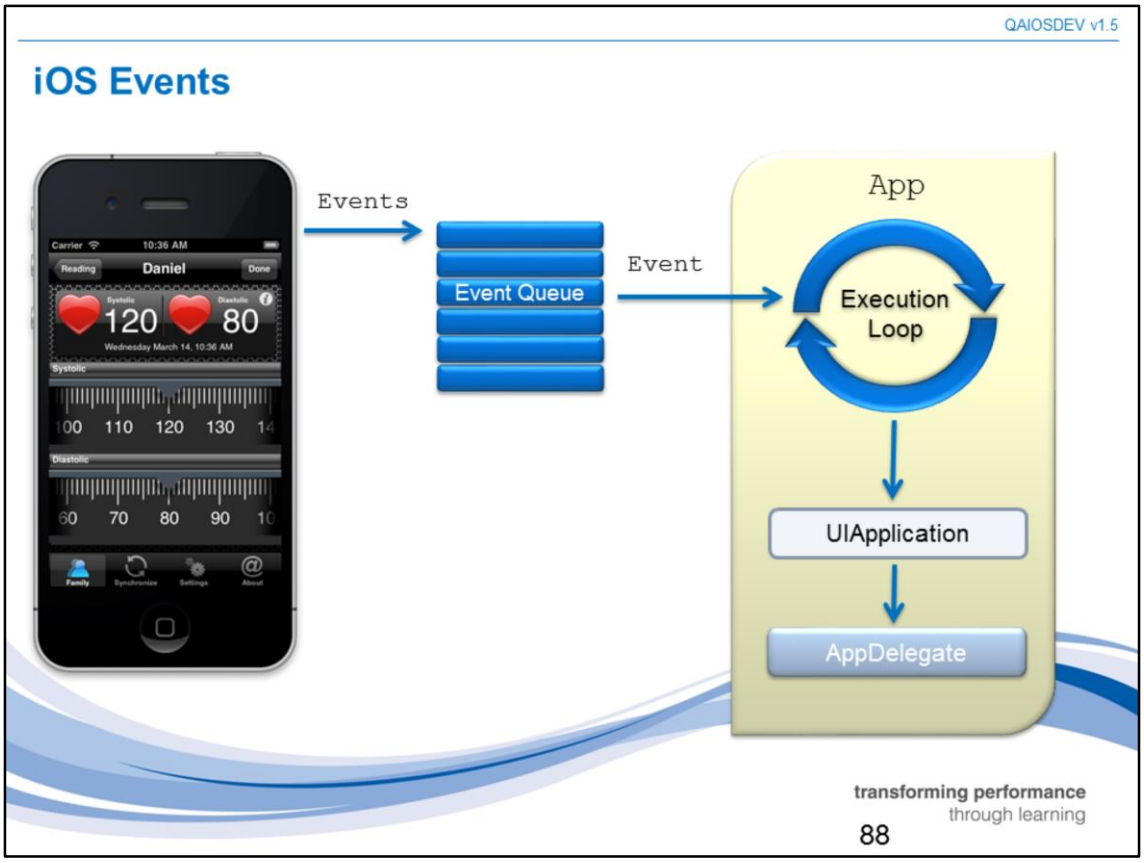
It's point based with the iPhone having 320x480 points and iPad 768x1024.

Points aren't the same as pixels. iPhone 4 has double the pixel resolution of the previous iPhone but the same point resolution. This means that an app written using point coordinates will work on either phone. Only the graphics will display differently.

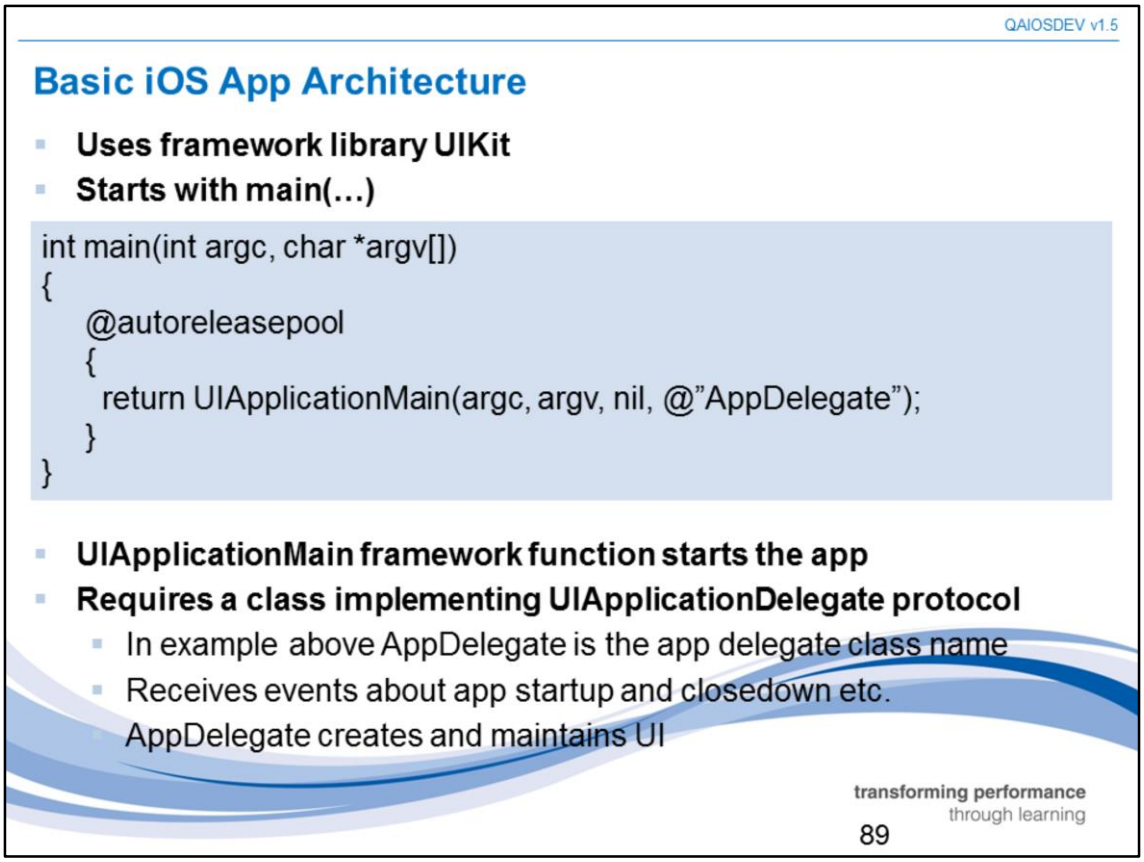
That means that iPhone 4 images need to be rendered at double resolution. All you have to do in your app is include the higher resolution images in your app and append @2x to the image name.

myImage.png would become myImage@2x.png. The SDK takes care of loading the correct image depending on the resolution of the phone the app is running on.

The same is true for iPad 1 & 2 as iPad 3 has double the pixel resolution but equal point resolution.



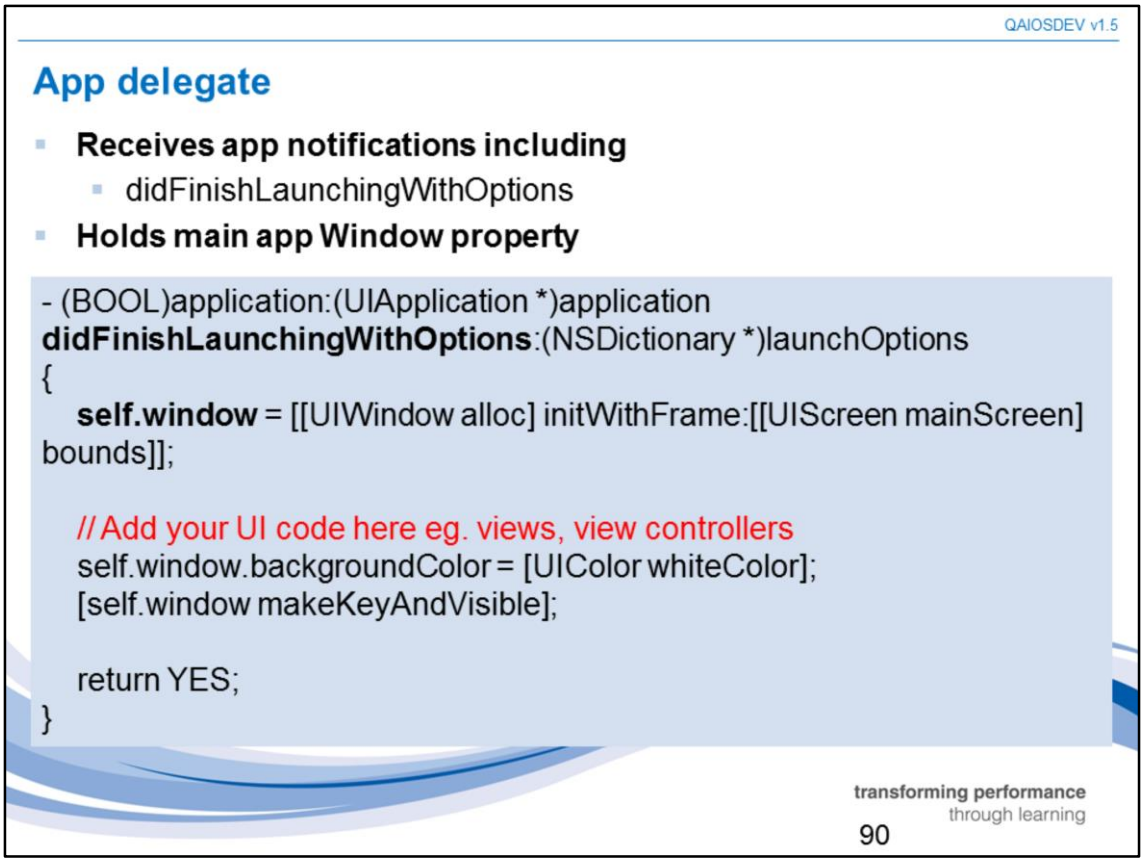

When the user interacts with an app they generate various kinds of events. Those events are directed to an event queue and eventually pass to the apps main execution loop that handles event messages. Events are then notified in the the main application class of all iOS applications, UIApplication, from where they are dispatched to the interested parties in the application. We will see throughout this course how the various events generated by iOS itself should be handled to create a well behaved and responsive app experience.




iOS apps, like all Objective-C apps, start with the main() function. To create the unique UI they use the framework library UIKit that contains many framework classes for the UI elements you are familiar with on your devices.

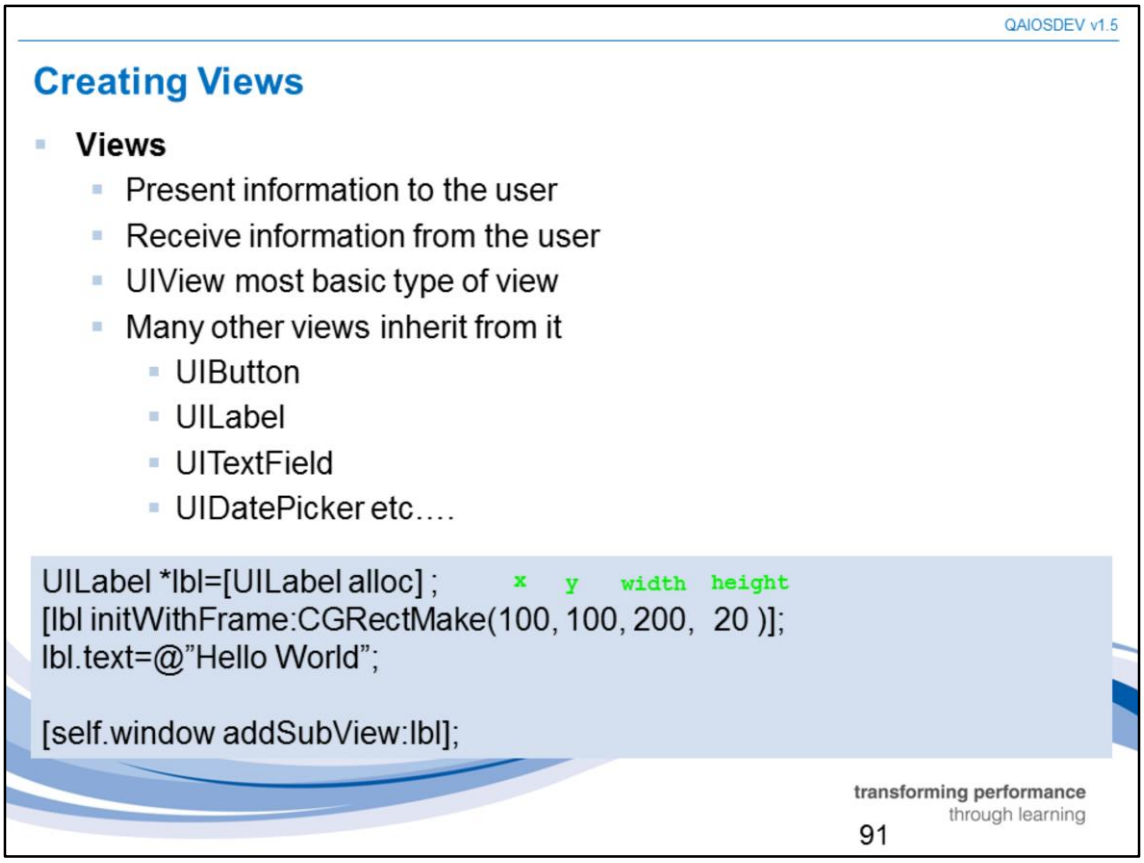
The UIKit framework function UIApplicationMain gets the whole thing started and starts iOS applications running.

You pass it the name of a class that's going to act as a delegate and receive application events, things like startup and shutdown. It's usually called the AppDelegate and is where we place the code that creates the first UI screens.




AppDelegate receives an important event from the application called application:didFinishLaunchingWithOptions and means the app has launched and is ready to display the UI.

It's here that we create a window and then add any views that we want to display.




A view just presents information to a user and allows users to interact. If you can see it and interact with it, it's a view.

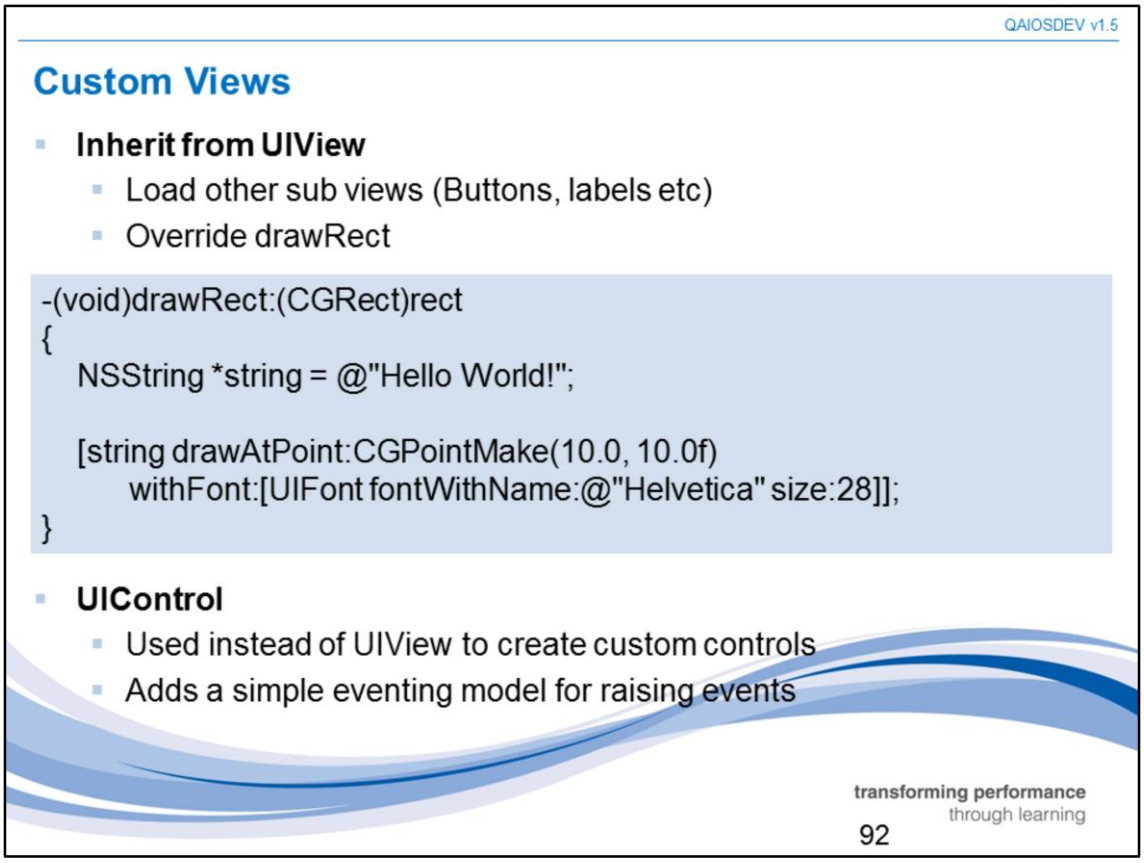
UIView is the base type and many other types of view inherit from it including all of the usual UI controls for text, labels, check boxes, sliders, date pickers etc.

Views can also contain multiple sub views.

In the example code, the traditional "Hello World" application is creating a view for a UILabel control that is located at a specific location on the screen.

The view is then just added to the window using **addSubView**.

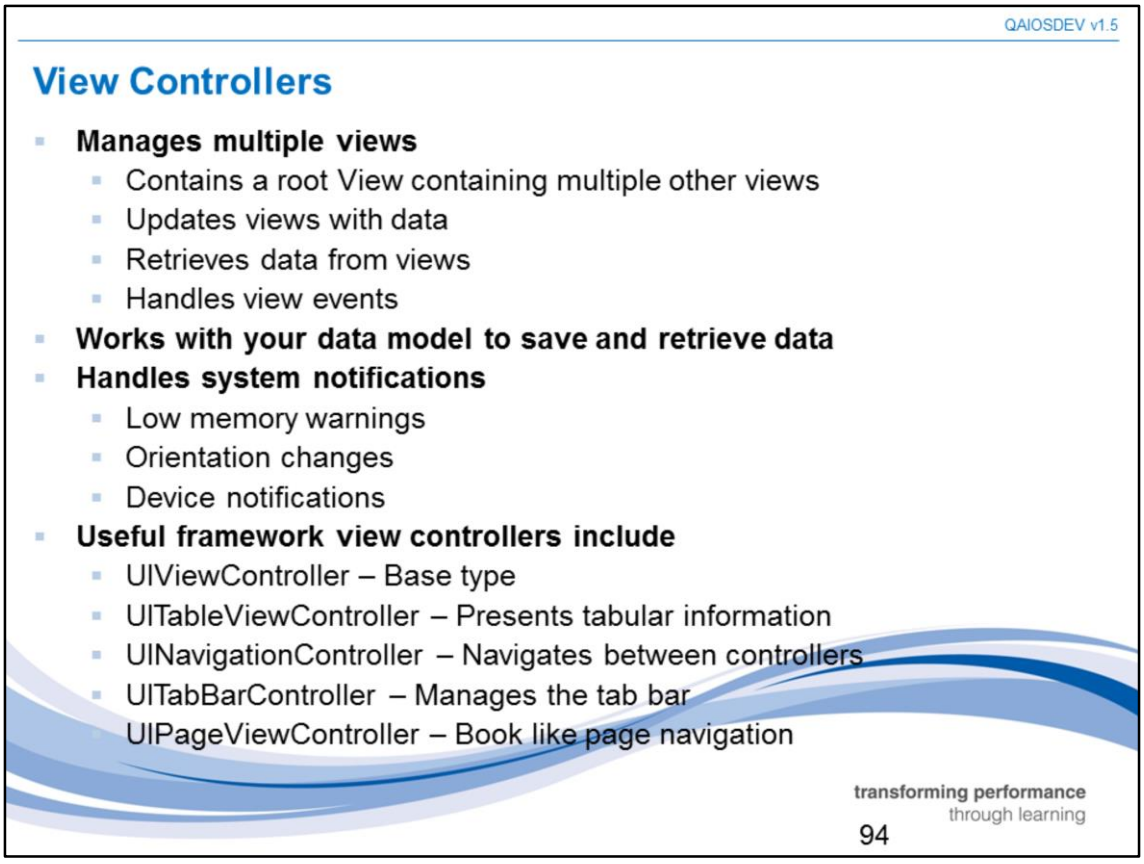
This is the simplest iPhone app.




You may want to create your own customized views. All you have to do is create a class inheriting from UIView or a descendant and override the drawRect method to draw the view content.

You can also add sub views to a customized view to create a reusable UI control. If you are doing this it is advisable to inherit from UIControl instead because it has built in support for event handling and dispatch.


QAIOSDEV v1.5
Demo - Views
Developing Apple Mobile Applications for iOS
transforming performance
through learning




View controllers inherit from UIViewController and are used to manage the display and updating of views. You could use views directly but you would lose a lot of the power of the framework that is built around view controllers.

Model View Controller is an effective and widely adopted design pattern. The view controller sits between the data model and the data display as a manager/coordinator.

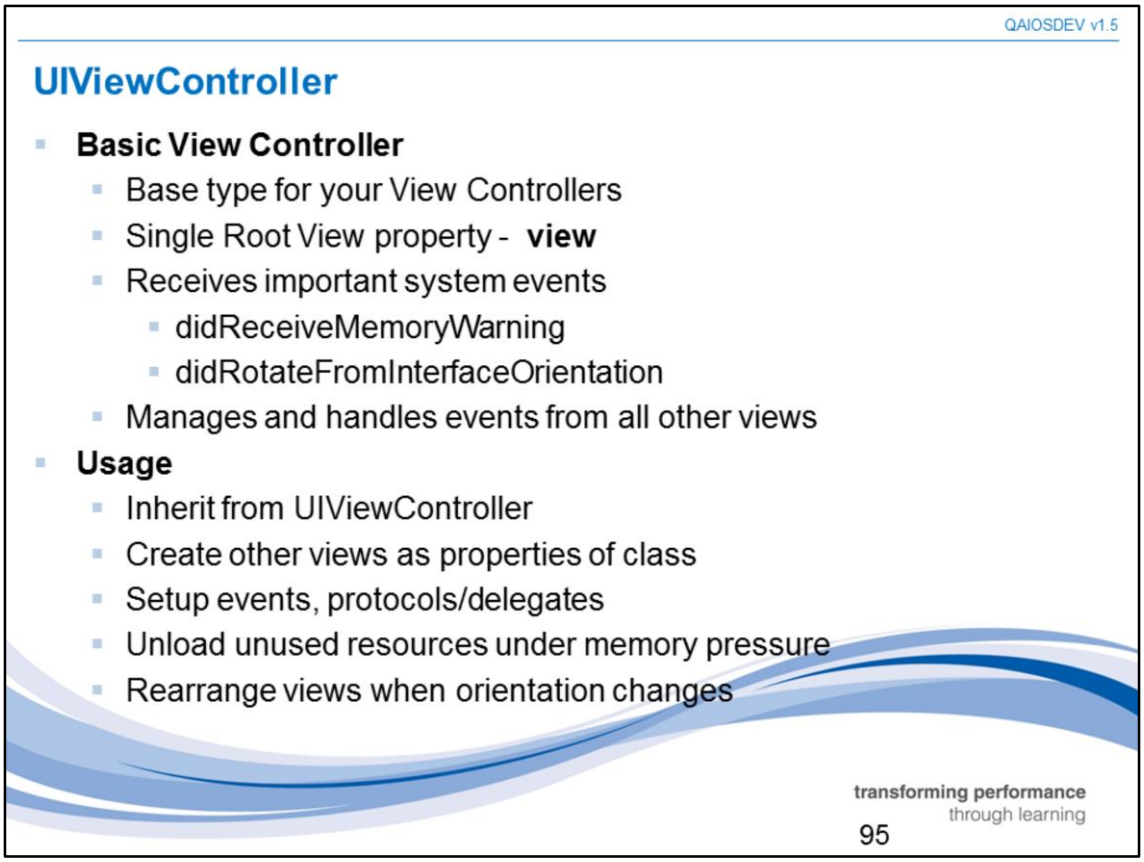
In iOS it has become part of the framework because view controllers receive system messages relating to changes in phone orientation and when memory is low. In both cases your view controller would be expected to take action.

Orientation changes may require an update to the way in which the views are displayed. Low memory warnings require the view to unload any loaded variables that are expendable or are not in use. Failure to take action may result in the entire app being shut down.

As well as you creating your own view controllers the framework provides additional ones like:

- UITableViewController to display tabular data
- UINavigationController manages the sequential display of view controllers in a navigable sequence

- UITabBarController – creates and manages the familiar tab bar at the bottom of iOS apps.



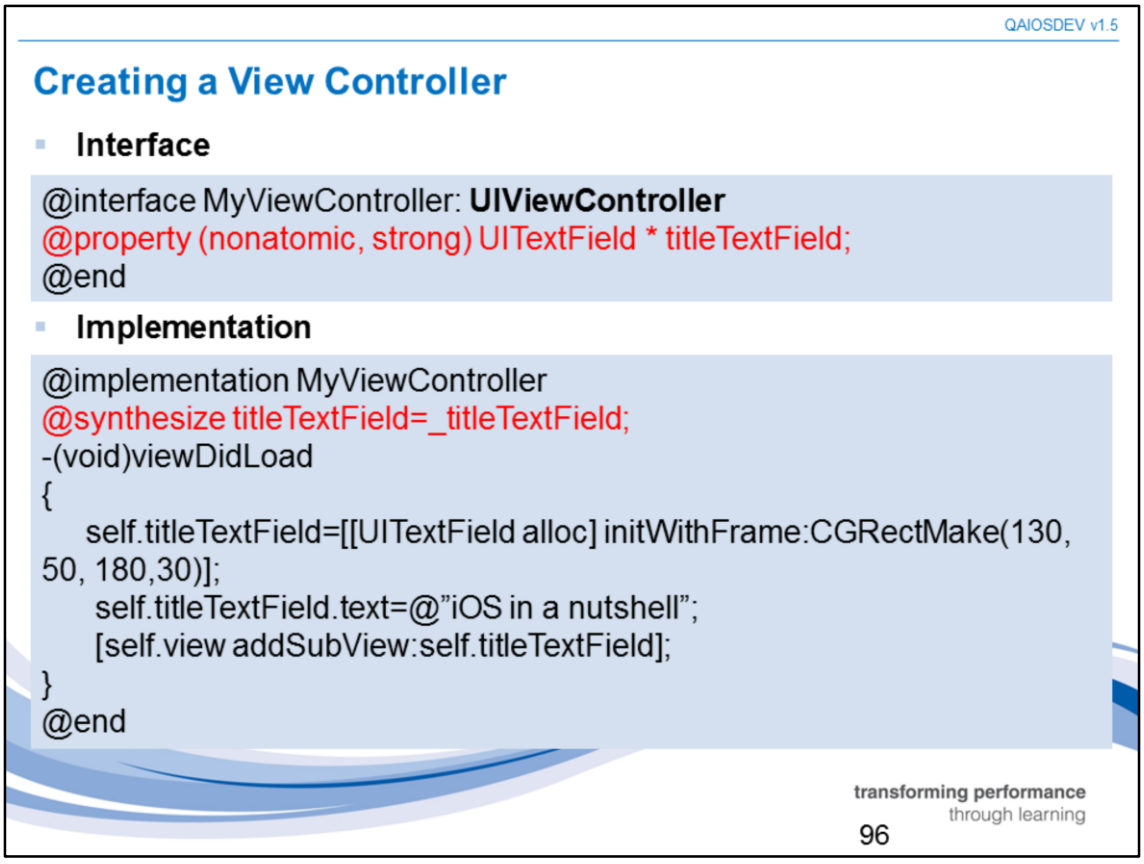

UIViewController is the base view controller type. All other view controllers inherit from as will your view controllers.

It has a single root view property called **view**.

You can add as many subviews as you like to this root view. By default the root view is generated for you but you can override this behaviour by overriding the **loadView** method. Just make sure you don't call [super loadView] as well! **loadView** is often used to load sub views as well.

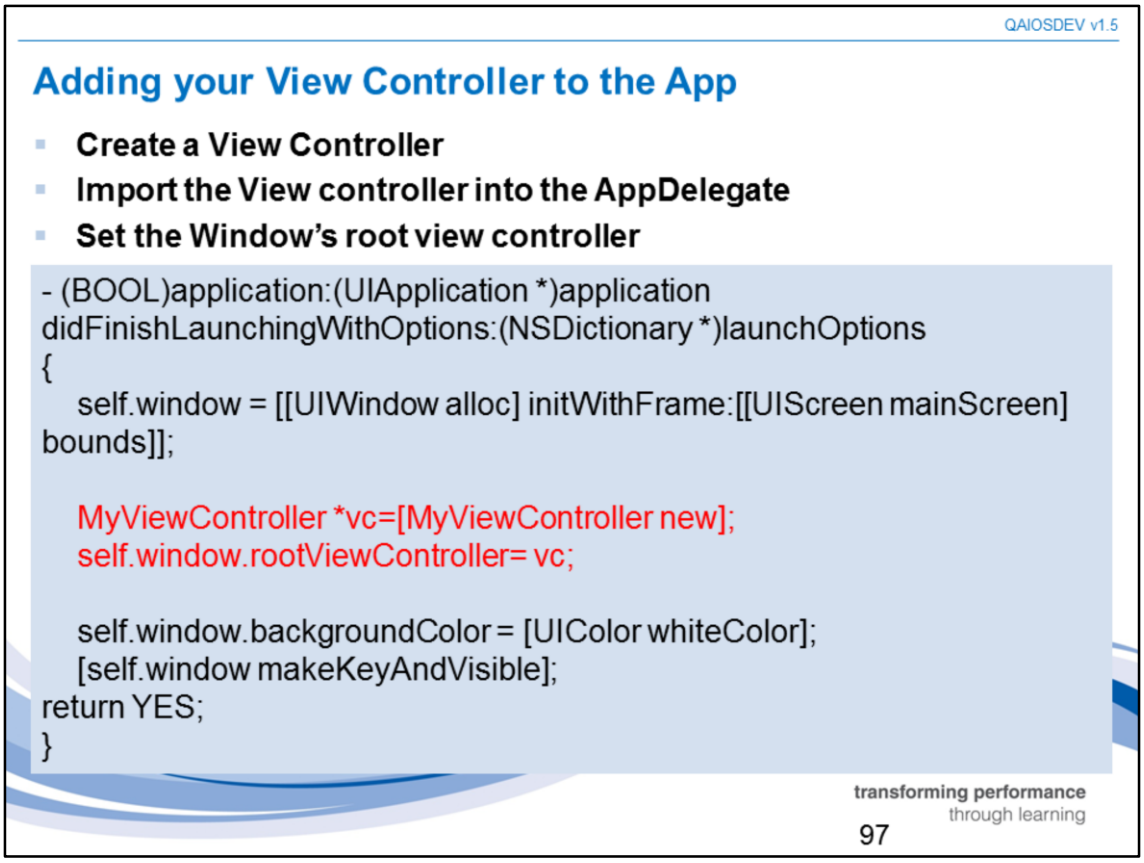
When the root view has loaded the **viewDidLoad** method is called. Here you can create any additional sub views and view dependent data structures.

View controllers automatically receive notifications about device orientation changes and low memory system messages. You need to respond to these events appropriately.



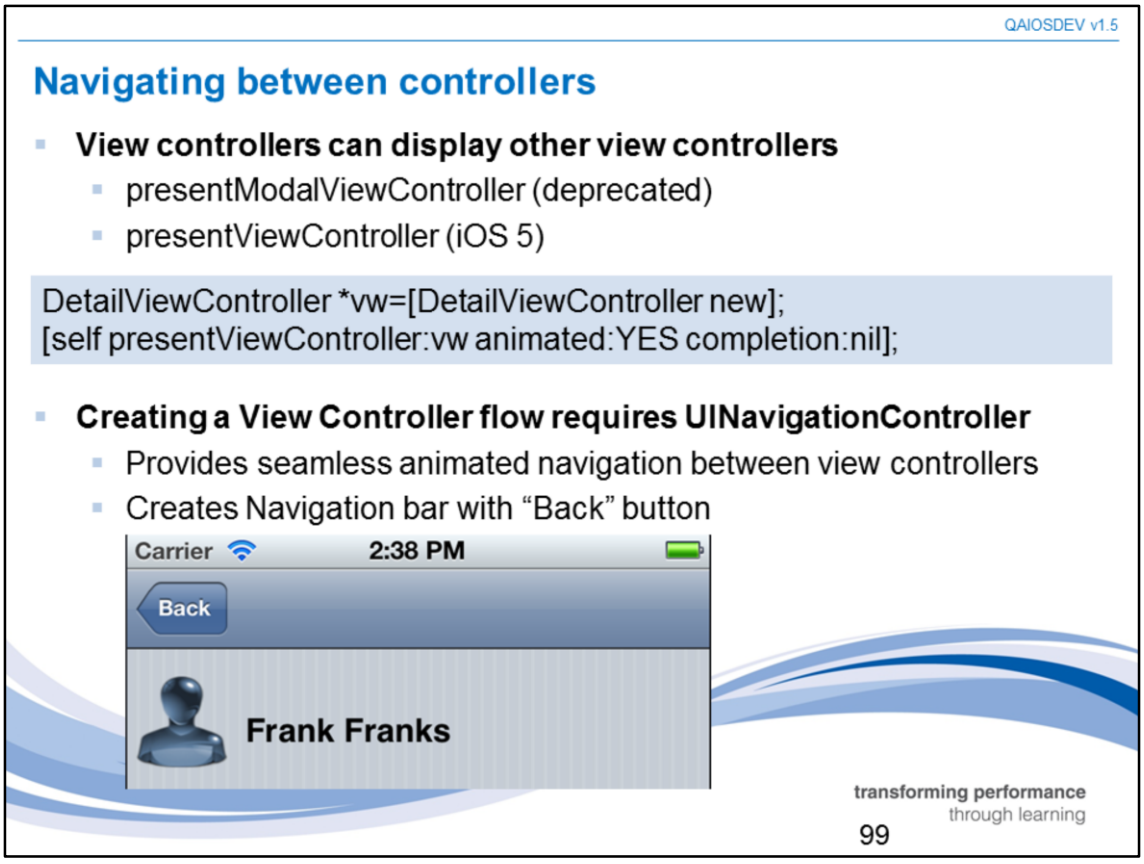

Creating a new view controller is straightforward:

- Create a new class inheriting from UIViewController.
- Setup some UI interaction views as properties
- Add these UI elements to the root view in **viewDidLoad**
- Respond to UI and system events




To now add a view controller to the main app window is really simple. All you need to do is create your view controller and assign it to the root view controller property of the window.


QAIOSDEV v1.5
Demo - ViewControllers
Developing Apple Mobile Applications for iOS
transforming performance
through learning




A view controller on its own is pretty limiting. Most apps allow users to seamlessly move between view controllers to achieve sub tasks or display master/detail information.

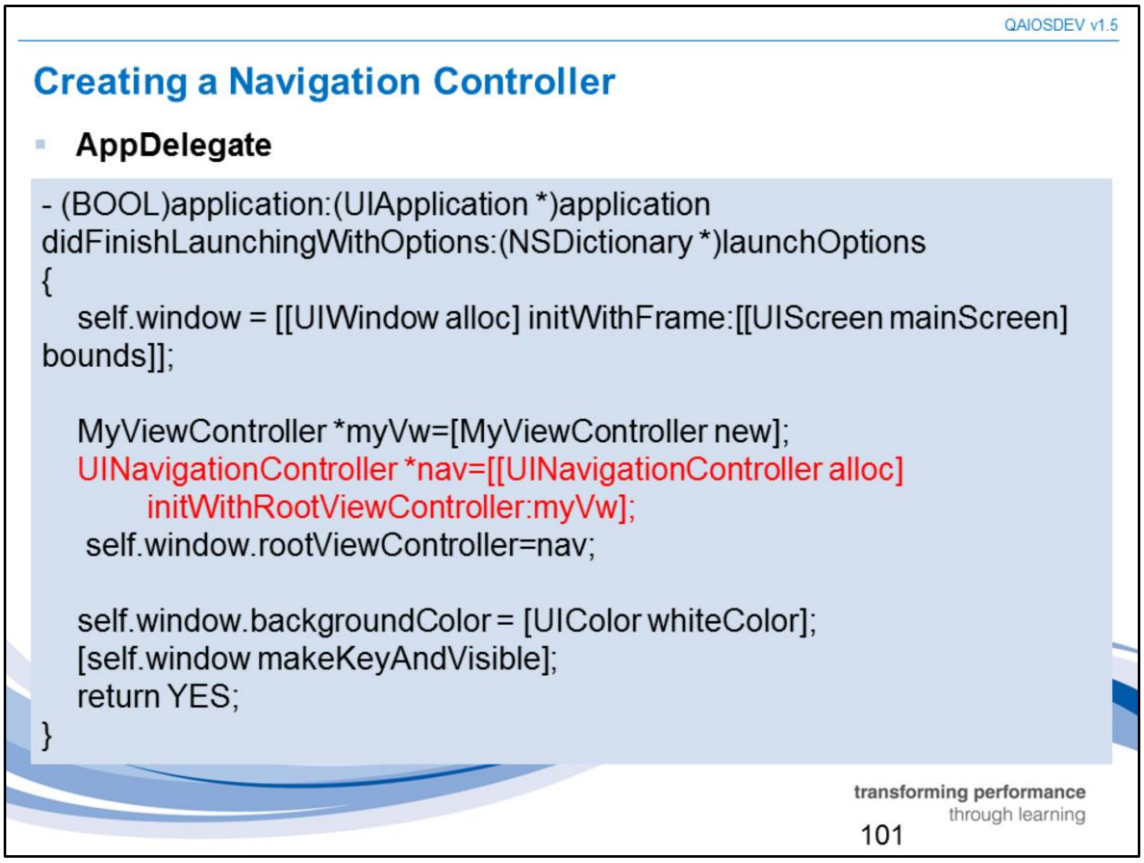
Similar to other types of application iOS allows you to create modal or sequential navigation.

Modal involves displaying a view controller over the top of another and then dismissing it when the task is complete. To present a view controller in this way just call presentModalViewController (pre iOS 5) or presentViewController (iOS 5 onwards).

A modal view controller can dismiss itself by calling:

```
[self dismissModalViewControllerAnimated:YES]; // pre iOS 5
or [self dismissViewControllerAnimated:YES completion:nil]; // iOS 5
```

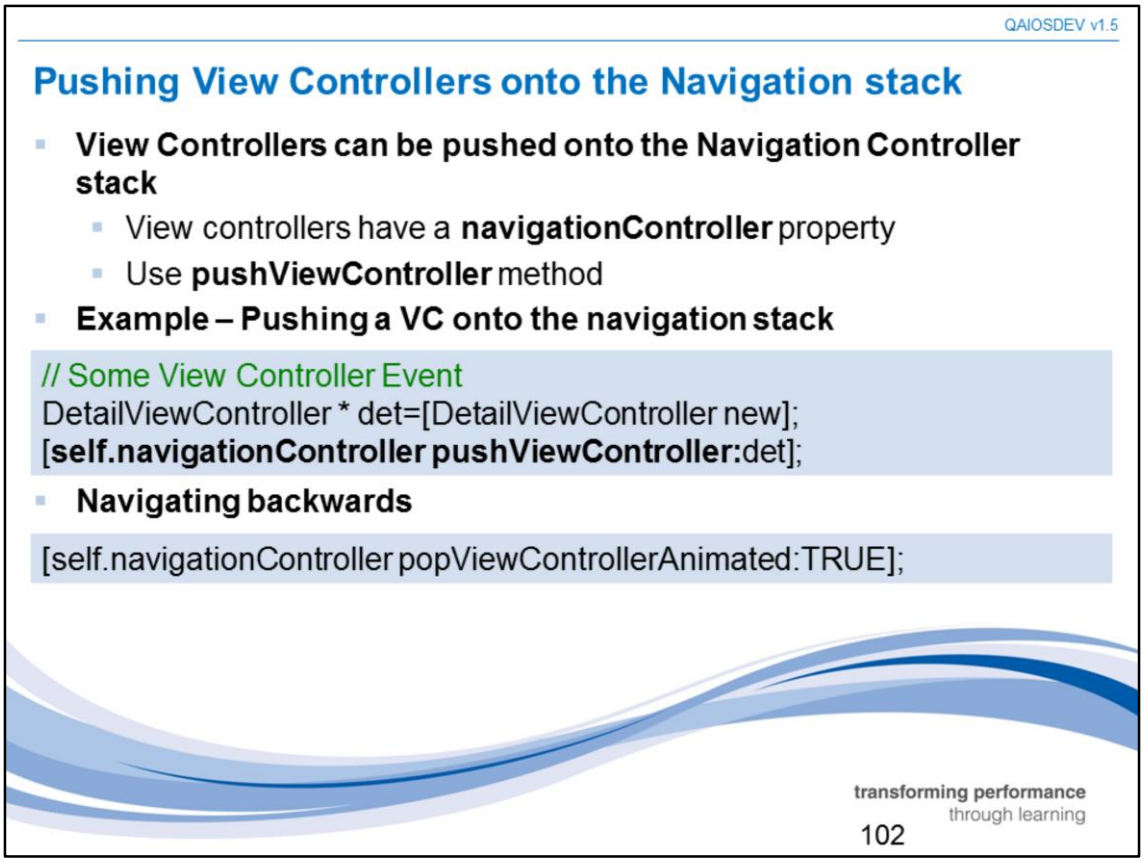
However this isn't the recommended because the original caller isn't informed of the dismissal. Although more complex you should implement a delegate or @selector and effect a callback back to the parent from where you could dismiss the view controller.




Instead of using our view controller directly the navigation controller is going to act as the window's root view controller instead. This allows it to be in control of navigating from one view controller to another.

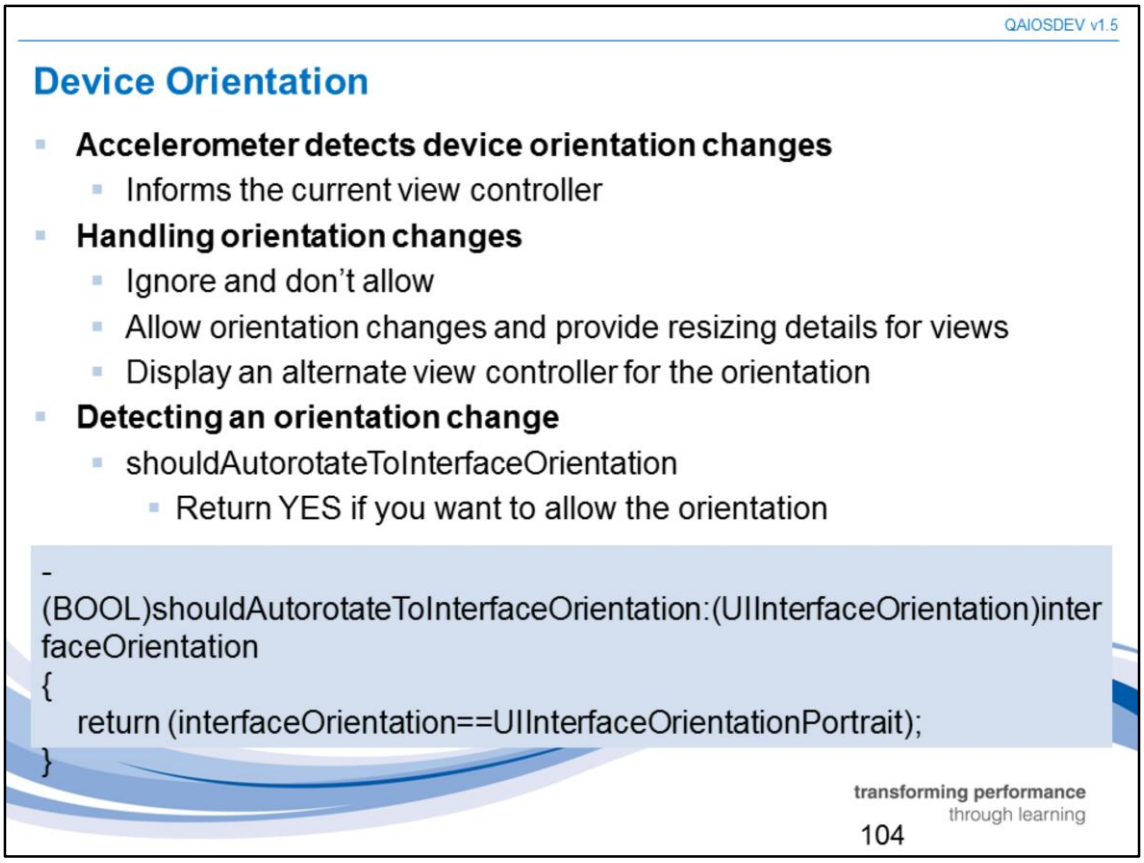
You have to tell it which view controller to use as its starting point (the first view controller to display). You do that at initialization with the call to **initWithViewController**.

And that's it! The navigation controller will now manage the display, transition and navigation of your view controller sequences.




But how do you move from one view controller to another. Simple, just push the new view controller onto the navigation controller stack and it will do the rest. You do that by calling **pushViewController** on the view controller's navigation controller. self.**navigationController** is a property of all view controllers.


QAIOSDEV v1.5
Demo – Navigation Controllers
Developing Apple Mobile Applications for iOS
transforming performance
through learning




When you move from portrait to landscape it effects your UI. The question is how do you handle it?

One option is do nothing and don't respond to orientation changes in which case your app is fixed in one of the orientations.

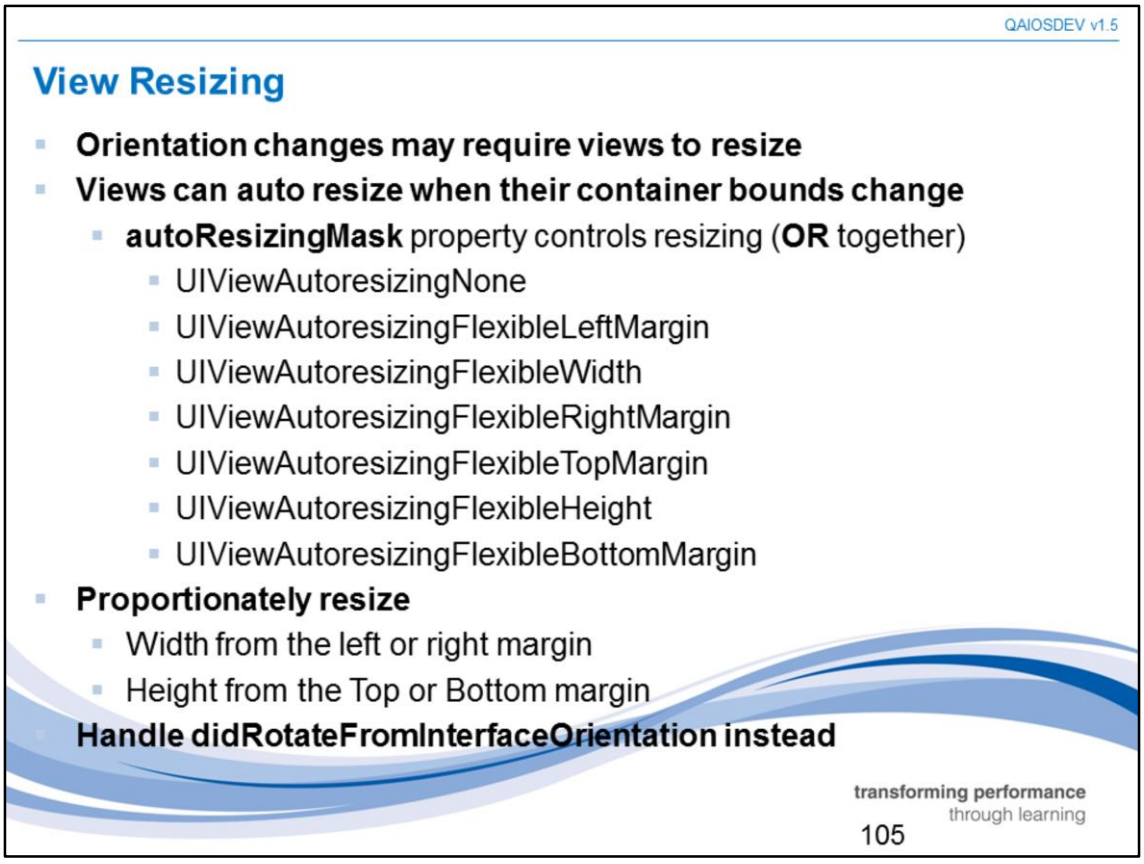
You may want to allow your app to operate in portrait and landscape in upside and downside modes. In this case you will need to adjust your views to cope with the new orientation. This will involve sizing and positioning.

There are four possible orientations, which are self explanatory:

- UIInterfaceOrientationPortrait
- UIInterfaceOrientationPortraitUpsideDown
- UIInterfaceOrientationLandscapeLeft
- UIInterfaceOrientationLandscapeRight

iOS will automatically adjust your views for orientation changes unless you tell it not to. You can detect orientation changes by handling method **shouldAutorotateToInterfaceOrientation.** This passes you the new orientation and you return YES if you want iOS to auto adjust your views for the

new orientation.

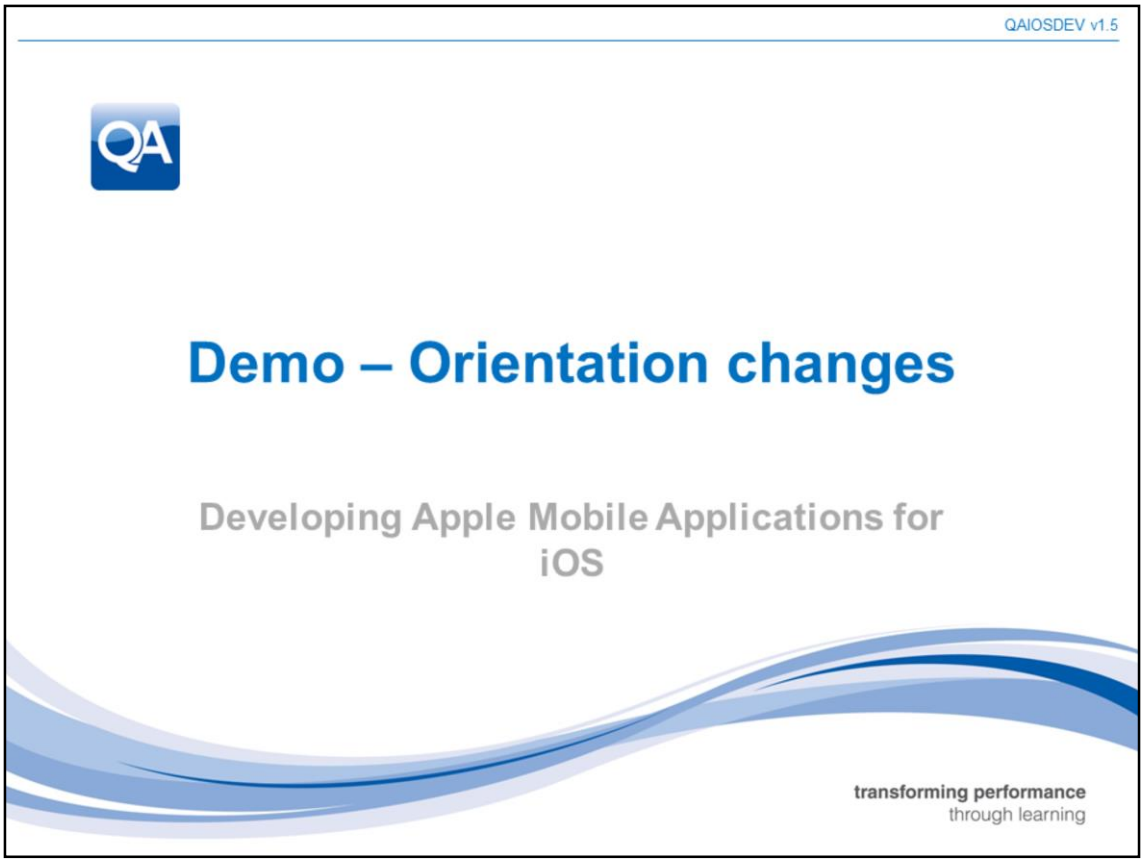


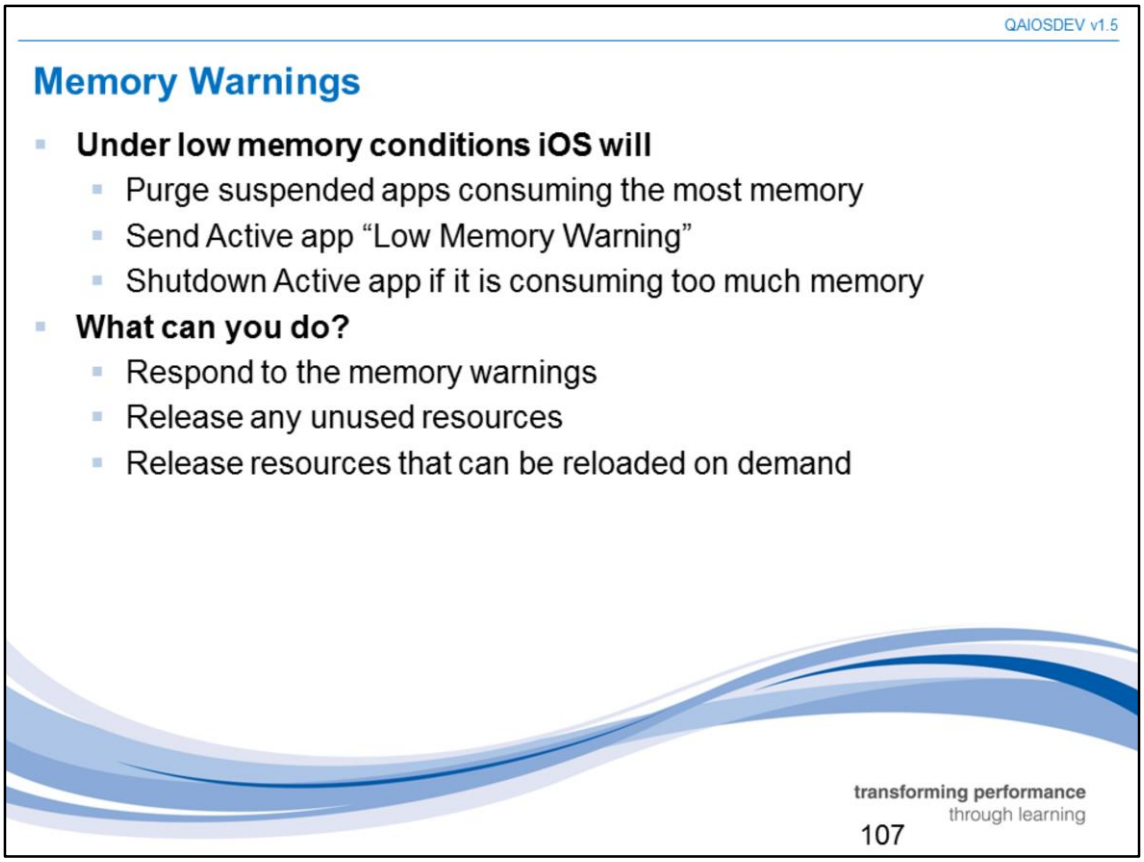

You can precisely control your layout by responding to **didRotateFromInterfaceOrientation** and setting the position and size of your sub views  for each orientation for each device.

That will work but you might also want to consider using the **autoResizingMask** on each of your sub views. This mask allows you to specify what to do when a view is auto resized after an orientation change.

By OR-ing the above contsants together you can come up with a resizing behavior for each of your views.

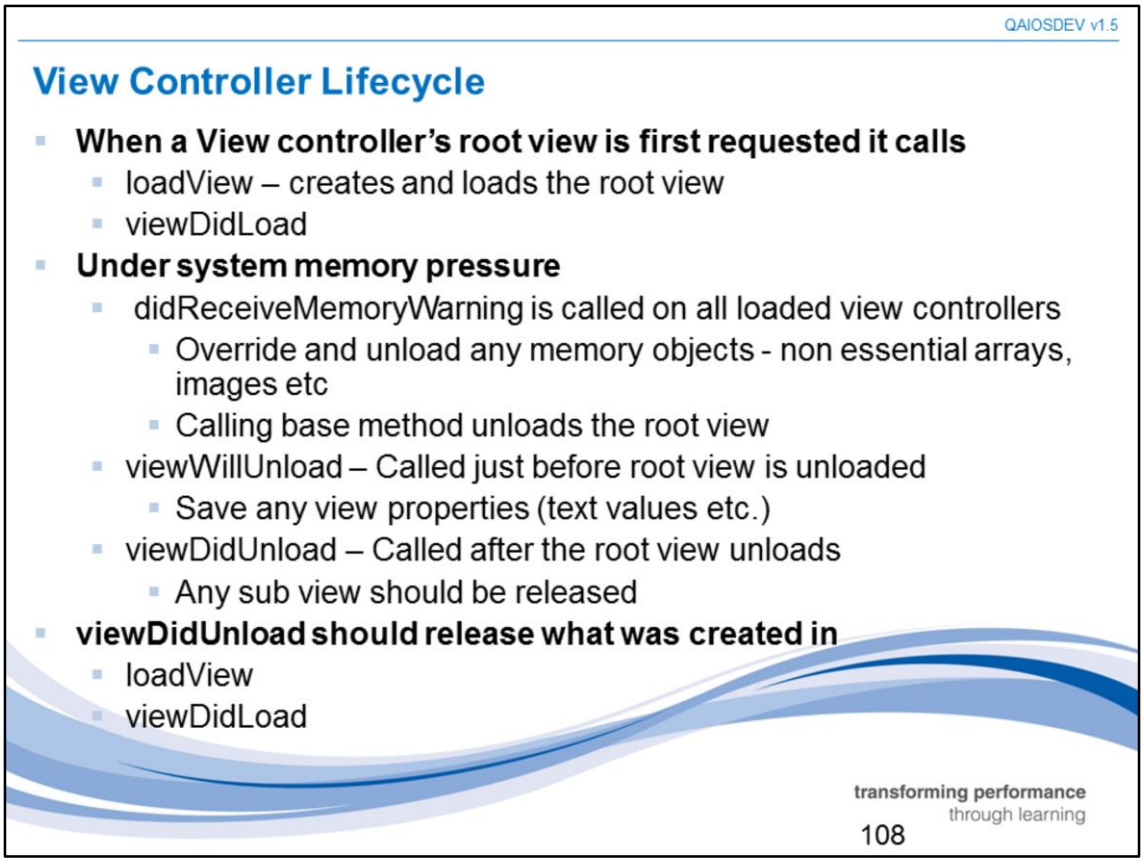
By specifying which dimension (width/height) can be changed (is flexible) and which margin is variable you tell iOS how it can proportionately resize the view.


QAIOSDEV v1.5
Demo – Orientation changes
Developing Apple Mobile Applications for iOS
transforming performance
through learning




When iOS hits low memory conditions it will send the active app and their loaded view controllers low memory warnings. It doesn't do this for suspended apps.

You should respond be releasing all non critical resources from your app delegate and view controllers. Failure to do so may result in your app shutting down.



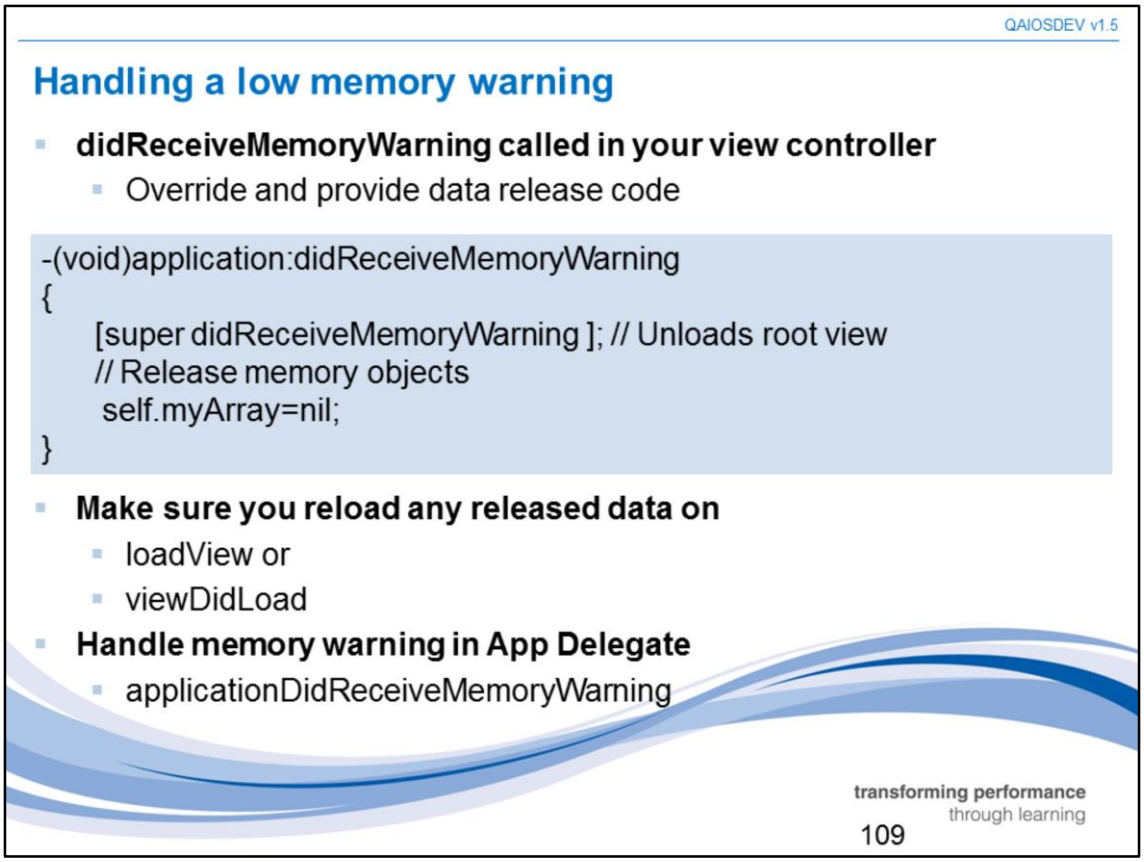

When your view controller's root view is first accessed it calls **loadView** giving you the opportunity to override the creation and loading of the root view. This is followed followed by **viewDidLoad** and is typically where you would load UI controls and UI dependent data structures such as arrays etc.

Under memory pressure the system will call **didReceiveMemoryWarning,** which by default, will unload the view controller's root view. As a result **viewWillUnload** will be called first giving you a chance to persist any UI data held in UI controls. Then viewDidUnload is called after the root view has unloaded.

When the view controller's root view property is next accessed the view will be reloaded with calls to **loadView** and **viewDidLoad.**

The correct response to a low memory warning is to unload all UI properties and UI related data that was created and retained in **loadView** or **viewDidLoad**. The best place to do that is in **viewDidUnload.**

Effectively **viewDidUnload** should reverse what was created in **viewDidLoad** and **loadView**.

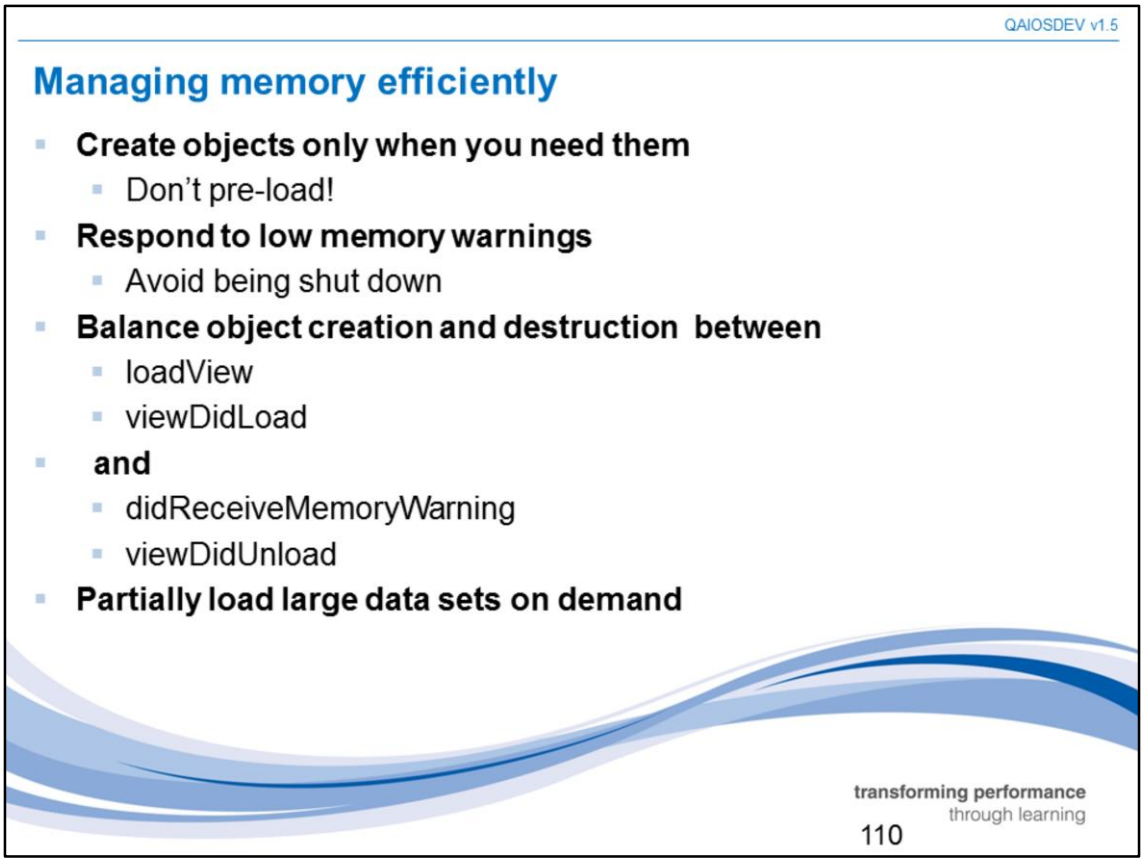



When **didReceiveMemoryWarning** is called on your view controller you need to unload any resources that can be reloaded eg images, caches etc. The base behaviour of didReceiveMemoryWarning is to unload the root view of the view controller. This causes **viewDidUnload** to be called.

in **viewDidUnload** you should release any UI views and UI dependent data structures you loaded in **loadView** and **viewDidLoad**. You may as well because their root view has unloaded anyway.

Low memory notifications are also received in the app delegate in **applicationDidReceiveMemoryWarning.** You should release anything that is non critical and can be reloaded on demand.

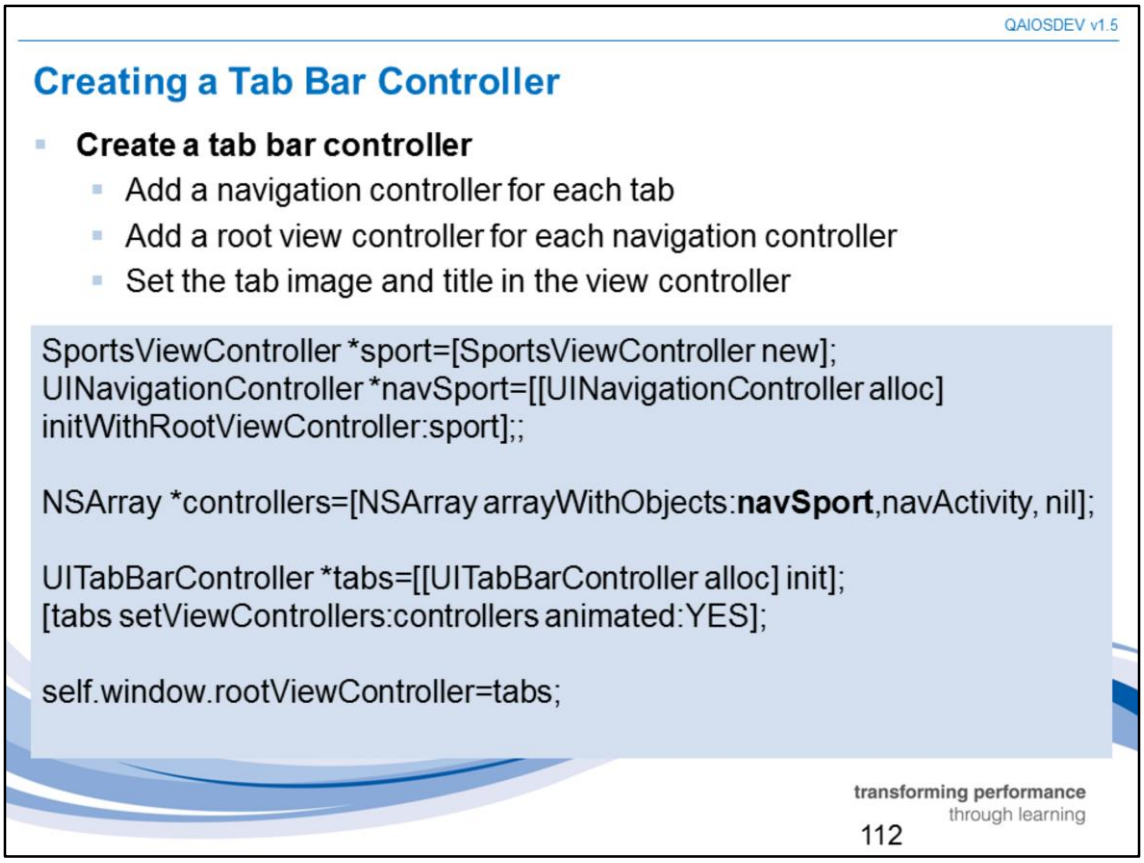
By doing this you maximise the amount of memory you can release when a low memory event occurs and increase the chances of your app surviving




To avoid your app being shutdown make sure you respond appropriately to low memory warnings.


Tab Bar Controllers provide the familiar tab button interface you have seen in many iPhone/iPad apps. They are a specialised type of view controller responsible for providing tabbed navigation between a set of view controllers.

Each tab is actually associated with a navigation controller that's linked to a default (root) view controller.




For each tab you need to create a navigation controller and add a root view controller that will be the root display view controller.

Each navigation controller is then added to an array and the tab bar controller is created from the array.

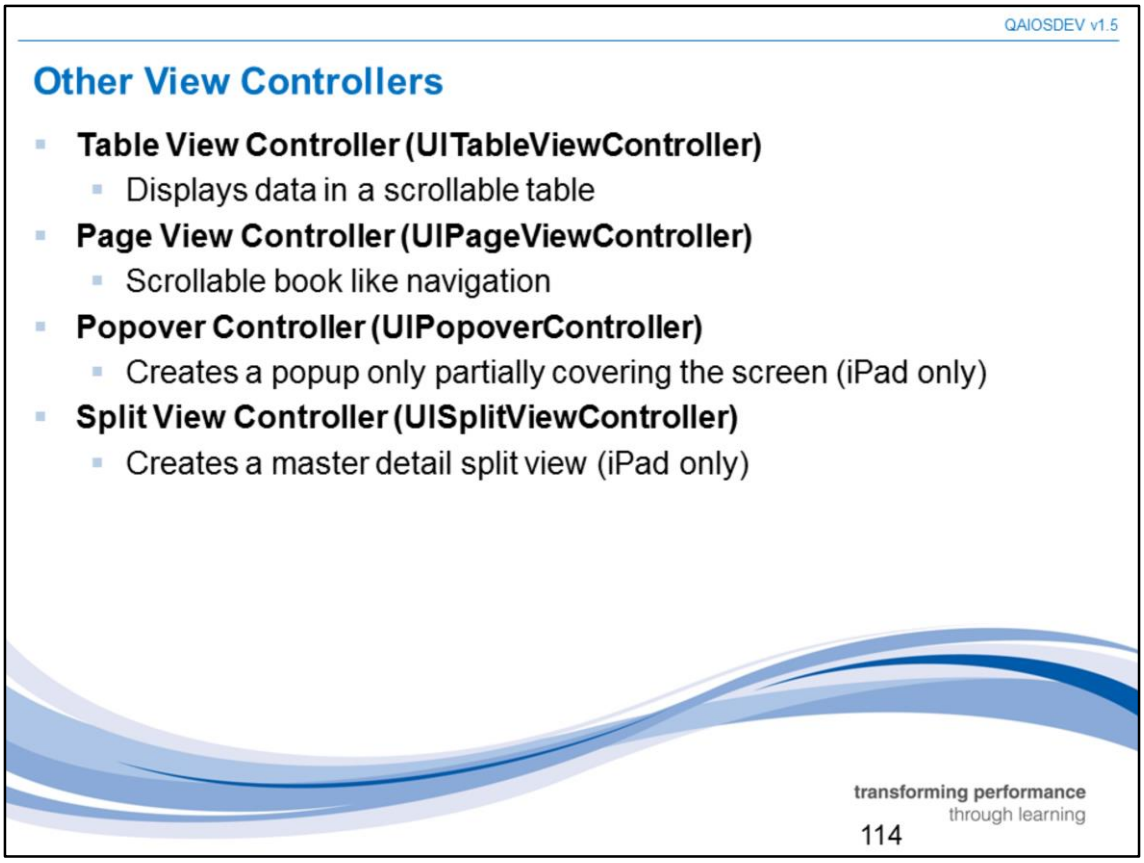
Each root view controller is responsible to specifying the tab bars icon and caption.

Here the root view controller is setting the tabBarItemImage and title inside its **init** method.

```
-(id)init
{
        if (self=[super init])
        {
                // Set the tab bar image and title
                self.tabBarItem.image = [UIImage
imageNamed:@"basketball.png"];
                self.title=@"Sports";

        }
        return self;
```

```
}
```

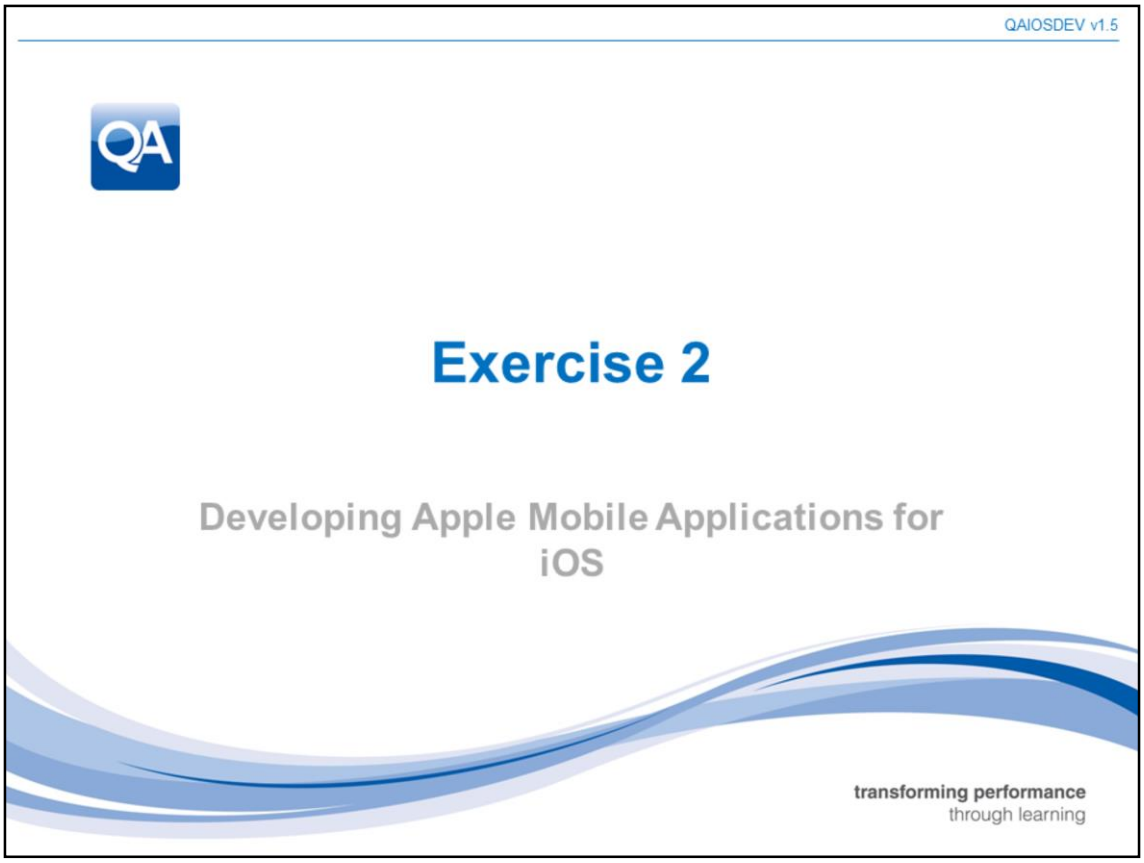

QAIOSDEV v1.5
Demo – Tab bar Controllers
Developing Apple Mobile Applications for iOS
transforming performance
through learning




In addition to the view controllers we have covered already there are a number of additional ones to be aware of.

UITableViewController is an essential view controller for display tabular data and you will it in use in may iOS apps. We will cover it in the next chapter.

There are also a number of view controller specifically designed for the iPad.


QAIOSDEV v1.5
Exercise 2
Developing Apple Mobile Applications for iOS
transforming performance
through learning

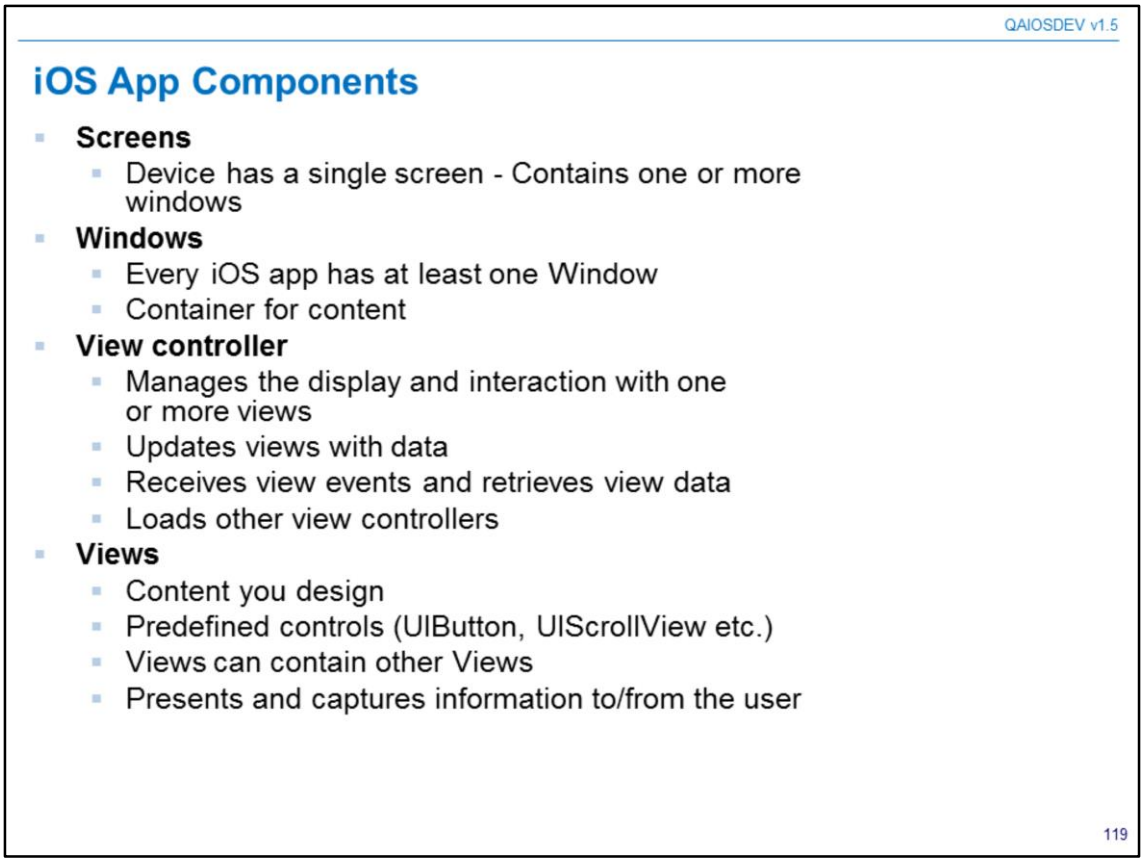
## Summary

- iOS app start with a main() function
- Application events handled by an App Delegate
- An iOS app can have multiple windows
- A window displays multiple views
- View Controllers
  - Manage views and their state
  - Handle Low memory warnings
  - Deal with orientation changes
- UIViewController is base type for all view controllers
- UINavigationController – sequential navigation between controllers
- UITabBarController – tabbed navigation between controllers


QAIOSDEV v1.5
Views and View Controllers
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - To understand how an iOS app is constructed under the hood
  - To write code to build an app from scratch
- **Contents**
  - iOS Architecture Overview
  - Model View Controller
  - Windows, Views and View Controllers
  - Navigating between View Controllers
  - Handling orientation changes
  - Dealing with low memory events
  - Tab Bar Controllers
- **Hands on Lab**




Every iPhone app has at least one window. A window is used to present content on the screen.

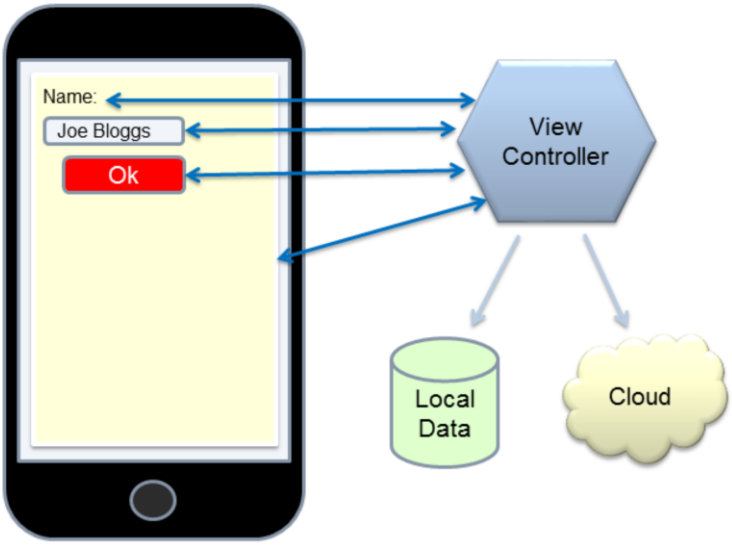
Views are used to present content onto a window and a window can contain multiple views. Views themselves can also contain sub views.

View controllers act as a managing interface between the view and your model (data storage). The view controller will update a view with data and that the view may allow the user to modify data. View controllers receive user events from views such as button clicks and other gestures that may trigger save actions causing data to be persisted to the model.

View controllers also manage system events like phone orientation changes and low memory conditions.

## Windows, Views and Controllers

- Window
- View
  - Sub Views
- View Controller
- Local Resources
- Cloud Resources

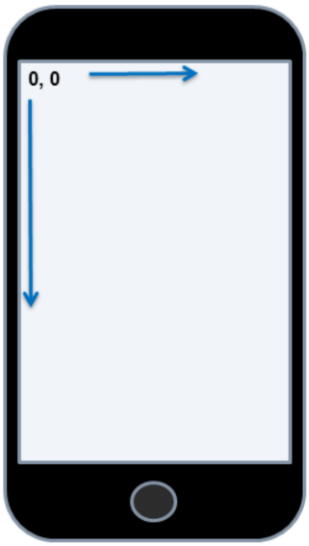





This screen demonstrates how an app hierarchy is build up with a window containing a view with sub views. All of the views are managed by the view controller that in turn may interact with local or cloud data sources.

## Screen Layout

- **Point based coordinate system**
  - iPhone - 320x480
  - iPhone 5 – 320x568
- **Pixel resolution**
  - 1 point != 1 pixel
  - iPhone 3GS - 163 ppi - iPhone 4 – 326 ppi
    - MyLogo.png – MyLogo@2x.png
- **iOS 6 introduced Auto Layout**
  - Dynamic Constraint based layout system
  - Placement is relative to other views






The iPhone coordinates system is zero based at the top left side of the screen.

It's point based with the iPhone having 320x480 points and iPad 768x1024.

Points aren't the same as pixels. iPhone 4 has double the pixel resolution of the previous iPhone but the same point resolution. This means that an app written using point coordinates will work on either phone. Only the graphics will display differently.

That means that iPhone 4 images need to be rendered at double resolution. All you have to do in your app is include the higher resolution images in your app and append @2x to the image name.

myImage.png would become myImage@2x.png. The SDK takes care of loading the correct image depending on the resolution of the phone the app is running on.

The same is true for iPad 1 & 2 as iPad 3 has double the pixel resolution but equal point resolution.

## Basic iOS App Architecture

- **Uses framework library UIKit**
- **Starts with main(…)**

```
int main(int argc, char *argv[])
{
    @autoreleasepool
    {
      return UIApplicationMain(argc, argv, nil, @"AppDelegate");
    }
}
```

- **UIApplicationMain framework function starts the app**
- **Requires a class implementing UIApplicationDelegate protocol**
  - In the example above, AppDelegate is the app delegate class name
  - Receives events about app startup and closedown etc.
  - AppDelegate creates and maintains UI



iOS apps, like all Objective-C apps, start with the main() function. To create the unique UI they use the framework library UIKit that contains many framework classes for the UI elements you are familiar with on your devices.

The UIKit framework function UIApplicationMain gets the whole thing started and starts iOS applications running.

You pass it the name of a class that's going to act as a delegate and receive application events, things like startup and shutdown. It's usually called the AppDelegate and is where we place the code that creates the first UI screens.

## App delegate

- **Receives app notifications including**
  - didFinishLaunchingWithOptions
- **Holds main app Window property**

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]];

    // Add your UI code here eg. views, view controllers
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}
```

123

AppDelegate receives an important event from the application called application:didFinishLaunchingWithOptions and means the app has launched and is ready to display the UI.

It's here that we create a window and then add any views that we want to display.

## Creating Views

- **Views**
  - Present information to the user
  - Receive information from the user
  - UIView most basic type of view
  - Many other views inherit from it
    - UIButton
    - UILabel
    - UITextField
    - UIDatePicker etc….

```
UILabel *lbl=[UILabel alloc] ;        x   y   width  height
[lbl initWithFrame:CGRectMake(100, 100, 200,  20 )];
lbl.text=@"Hello World";

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

124

A view just presents information to a user and allows users to interact. If you can see it and interact with it, it's a view.

UIView is the base type and many other types of view inherit from it including all of the usual UI controls for text, labels, check boxes, sliders, date pickers etc.

Views can also contain multiple sub views.

In the example code, the traditional "Hello World" application is creating a view for a UILabel control that is located at a specific location on the screen.

The view is then just added to the window using **addSubView**.

This is the simplest iPhone app.

## View Controllers

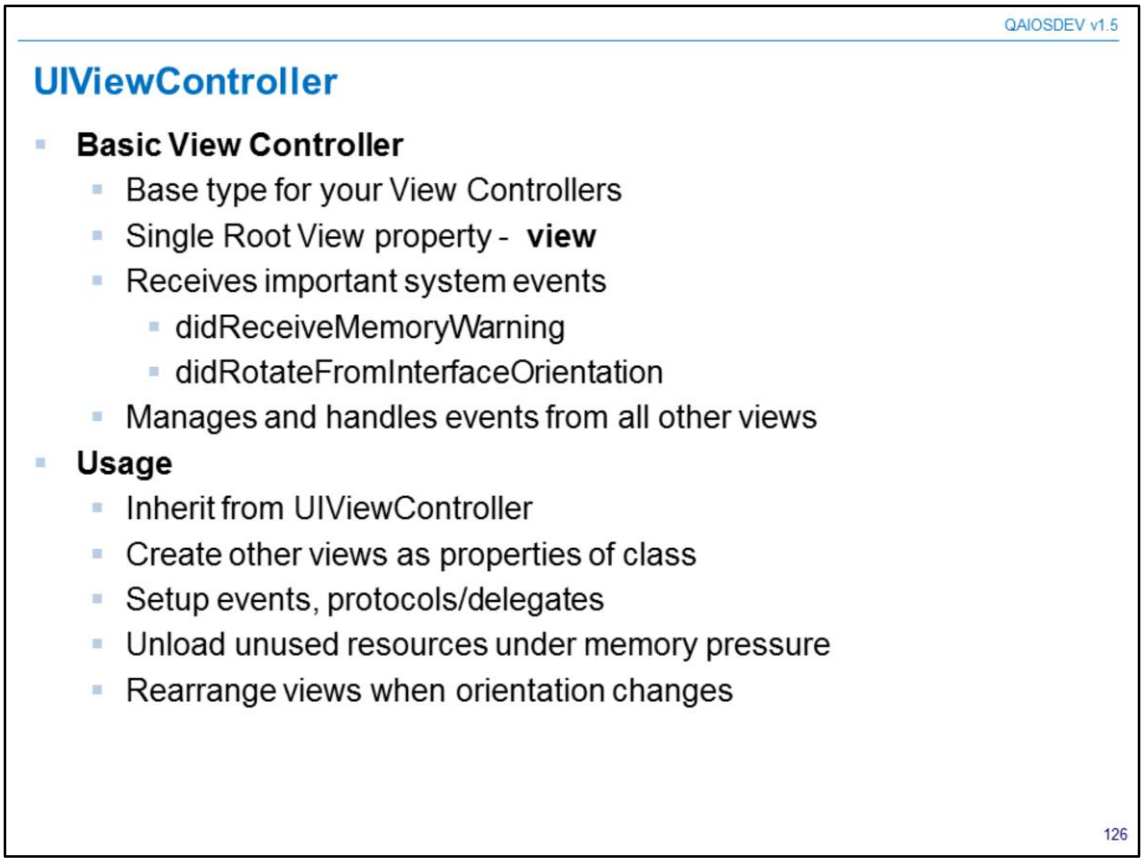
- **Manages multiple views**
  - Contains a root View to which you add other views
  - Updates views with data
  - Retrieves data from views
  - Handles events received from views
- **Works with your data model to save and retrieve data**
- **Handles system notifications**
  - Low memory warnings
  - Orientation changes
  - Device notifications
- **Useful framework view controllers include**
  - UIViewController – Base type
  - UITableViewController – Presents tabular information
  - UINavigationController – Navigates between controllers
  - UITabBarController – Manages the tab bar
  - UIPageViewController – Book like page navigation
  - UICollectionViewController – Tabular based display

125

View controllers inherit from UIViewController and are used to manage the display and updating of views. You could use views directly but you would lose a lot of the power of the framework that is built around view controllers.
Model View Controller is an effective and widely adopted design pattern. The view controller sits between the data model and the data display as a manager/coordinator.
In iOS it has become part of the framework because view controllers, receive system messages relating to changes in phone orientation and when memory is low. In both cases, your view controller would be expected to take action. Orientation changes may require an update to the way in which the views are displayed. Low memory warnings require the view to unload any loaded variables that are expendable or are not in use. Failure to take action may result in the entire app being shut down.

As well as you creating your own view controllers the framework provides additional ones like:

- UITableViewController to display tabular data
- UINavigationController manages the sequential display of view controllers in a navigable sequence
- UITabBarController – creates and manages the familiar tab bar at the bottom of iOS apps



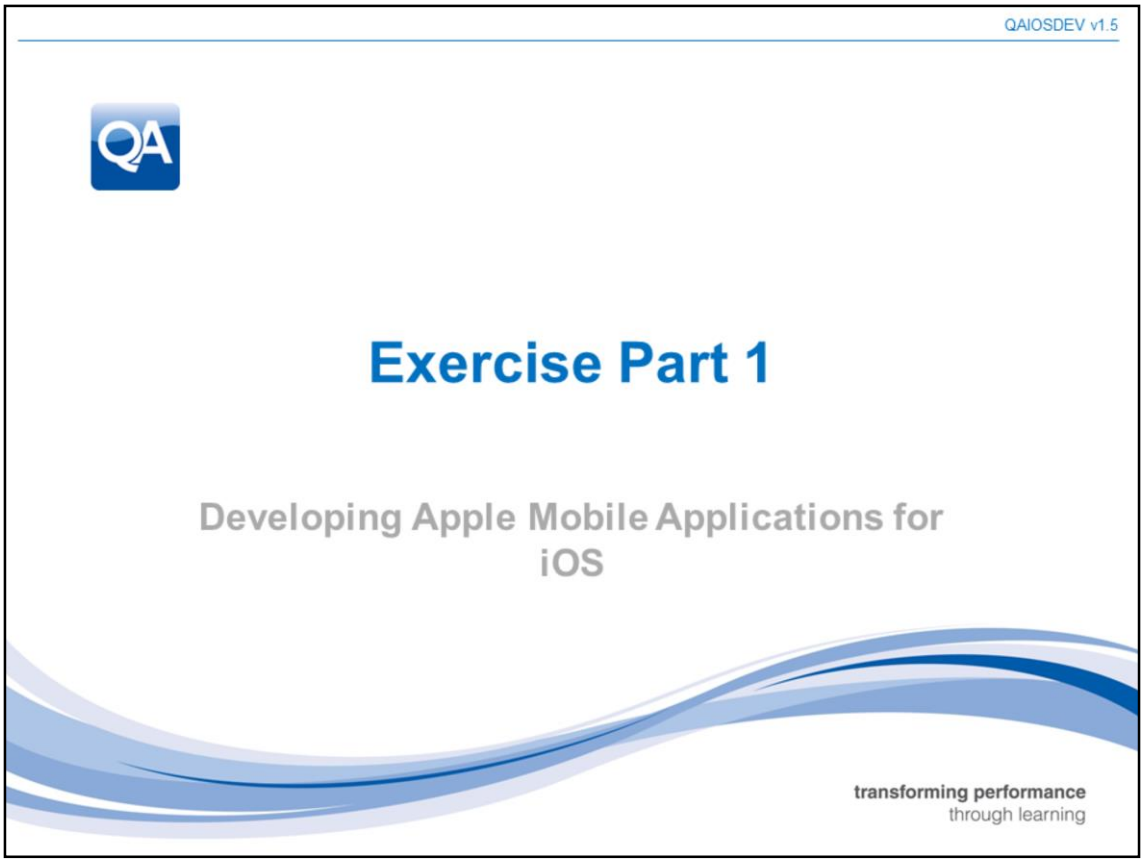

UIViewController is the base view controller type. All other view controllers inherit from as will your view controllers.

It has a single root view property called **view**.

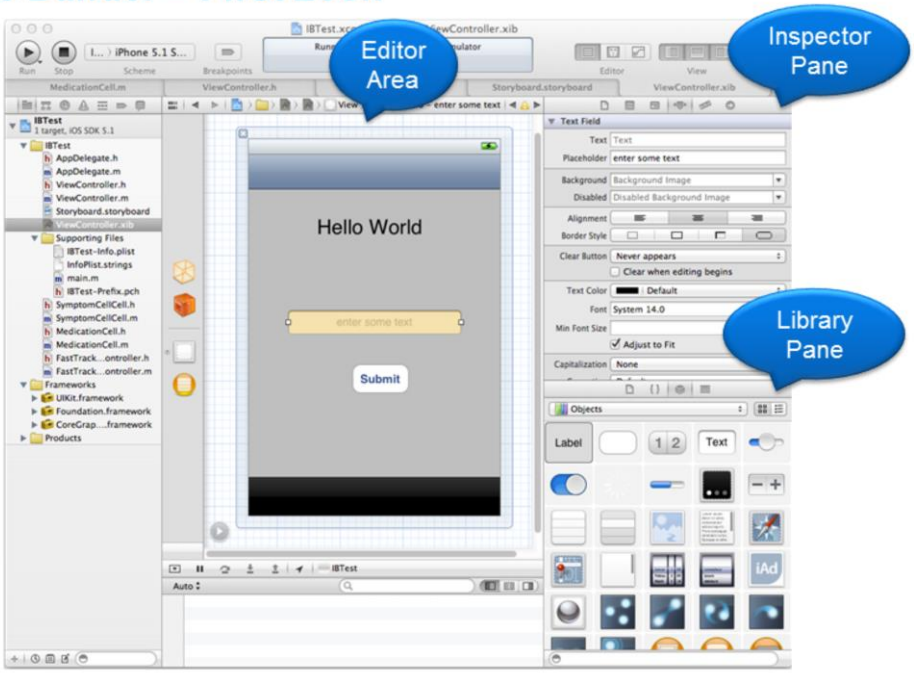
You can add as many subviews as you like to this root view. By default the root view is generated for you but you can override this behaviour by overriding the **loadView** method. Just make sure you don't call [super loadView] as well! **loadView** is often used to load sub views as well.

When the root view has loaded the **viewDidLoad** method is called. Here you can create any additional sub views and view dependent data structures.

View controllers automatically receive notifications about device orientation changes and low memory system messages. You need to respond to these events appropriately.


QAIOSDEV v1.5
Exercise Part 1
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Interface Builder – First Look





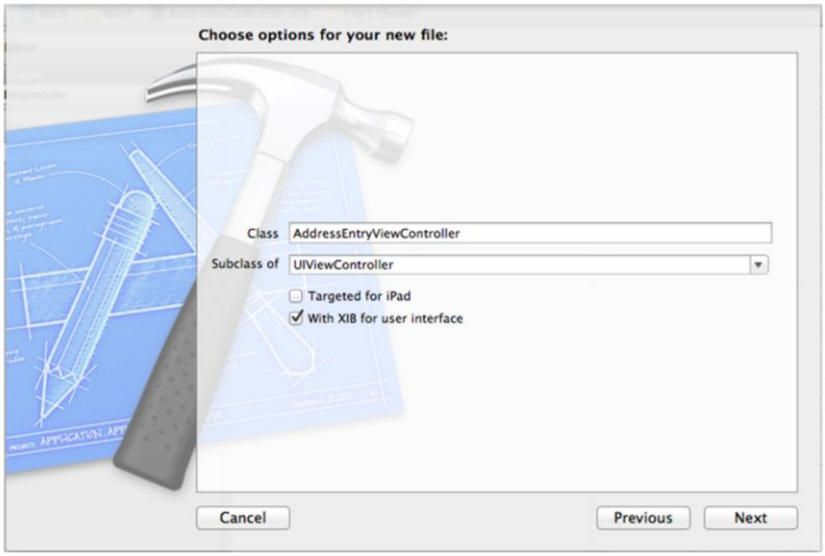

Interface Builder is integrated into Xcode 4. It's a standard visual designer made up of an editor window that holds the screen design, a library pane containing the user interface controls and an inspector pane allowing you to set properties on controls and views.

Interface Builder also lets you hook your UI elements up to code and you can create properties and event methods automatically by simply dragging an dropping onto the code window.

Interface builder files are saved as XIB files, which are XML object graphs representing the UI.

## Creating a View Controller with Interface Builder

- **Create a new View Controller**
  - Choose "With XIB for User Interface"





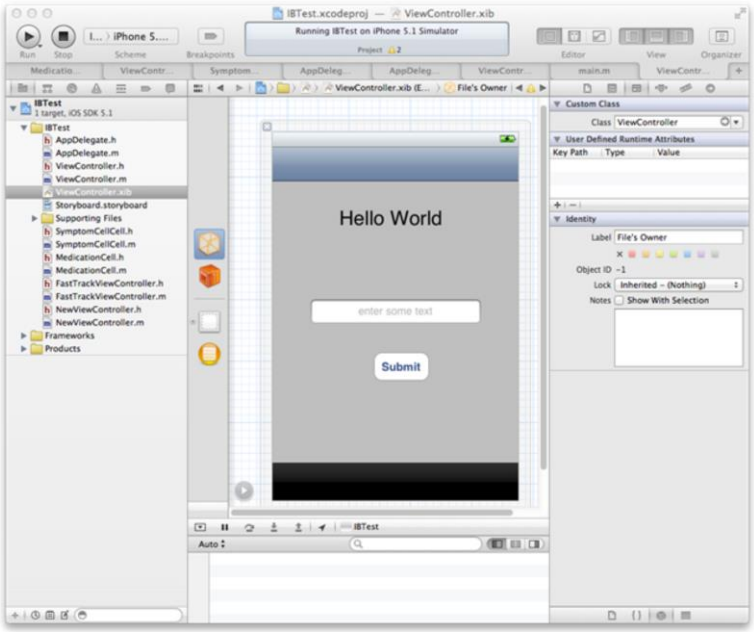

All you have to do to create a view controller with an accompanying XIB file is make sure you check the check box "With Xib for user interface" when creating an new View Controller from the file menu.

A XIB will be created with the same name as the View Controller.

QAIOSDEV v1.5

## Connecting a XIB to a View Controller Class

- **Files Owner**
  - Links xib to class
- **Setup automatically**
- **Change class using**
  - Identity Inspector




130

A XIB is linked to it's handling class within the identity inspector, which is one of the tabs within the inspector pane. To view it, you to first select the File's Owner button that is to the right of the Edit window and looks like a yellow see-through cube.

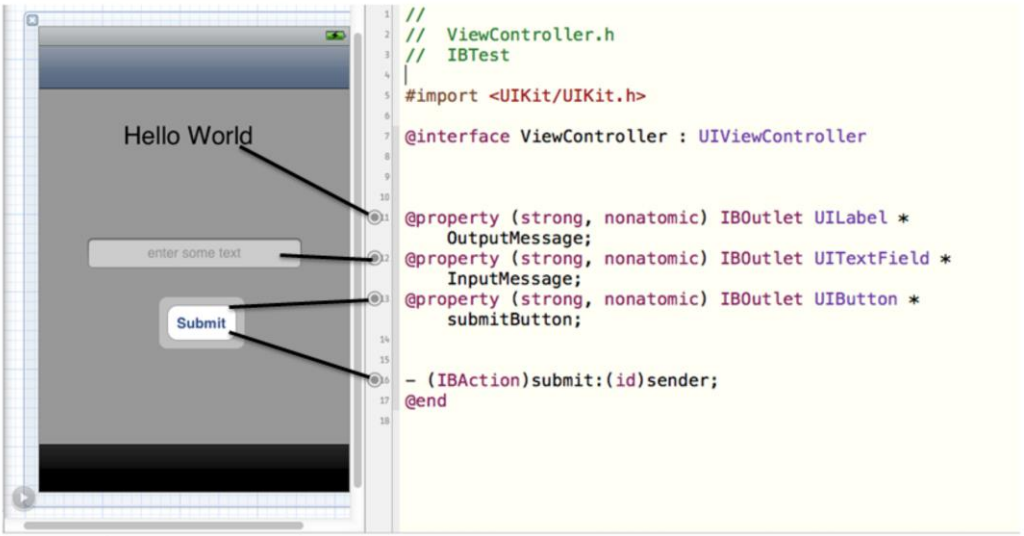
The class handling the XIB is identified in the Class dropdown within the Custom Class section.

File's Owner is the link from the XIB file, which is just an XML file, to the actual Objective-C class that is going to handle its events and properties.

QAIOSDEV v1.5

## Connecting UI elements up to code

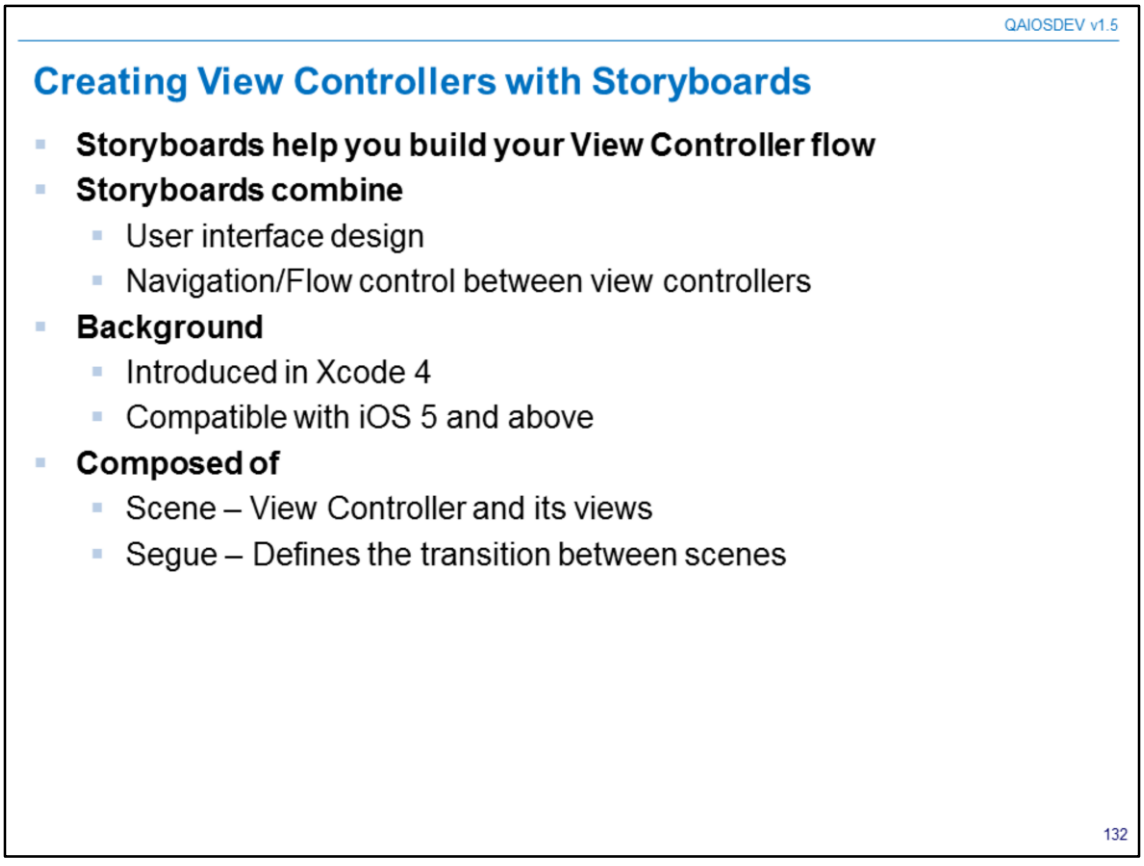
- Connect properties to UI Elements – IBOutlet (Drag&Drop)
- Connect UI actions to methods – IBAction (Drag & Drop)




131

Any UI elements in the XIB file that need to accessible in code should have corresponding properties defined in the view controller. To make sure they can, link up together each of the properties need to be marked with a flag, **IBOutlet,** which indicates that the property can act as an interface builder outlet.

The good news is that Interface Builder (IB) will do most of the work for you. If you switch your editor to assistant view whilst editing your UI in IB your view controller code will also be visible. All you now need to do is press the CTRL key and drag/drop one of your controls from your UI design onto the interface file (.h) of your view controller. A dialog will popup asking you to name the property after which a property will be created for your with an IBOutlet flag already attached.




Storyboards are possibly the most significant advance in iOS development. With storyboards you can design an entire application's screen flow together with the user interfaces for each of the screens. So much can be done in design that it means there is less code for you to write and less code means fewer bugs.

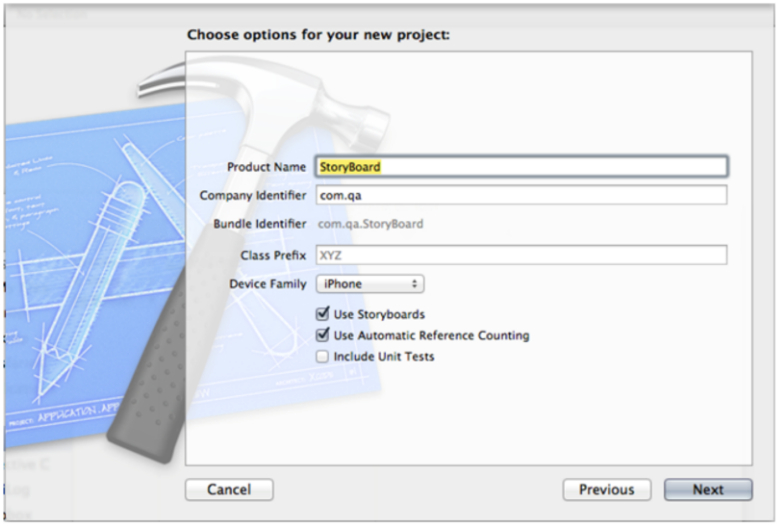
Storyboards were introduced in Xcode 4 and are compatible with iOS 5 onwards.

They are based on two concepts. The Scene is simply a view controller and a Segue is a defined transition from one scene to another.

One other thing to note about story boards is their ability to utilise iOS 5's static and dynamic table views. You can now design table views that will appear at runtime precisely as you designed them without having to write reams of code.

## Creating a Storyboard

- **Create a new project with a Storyboard**



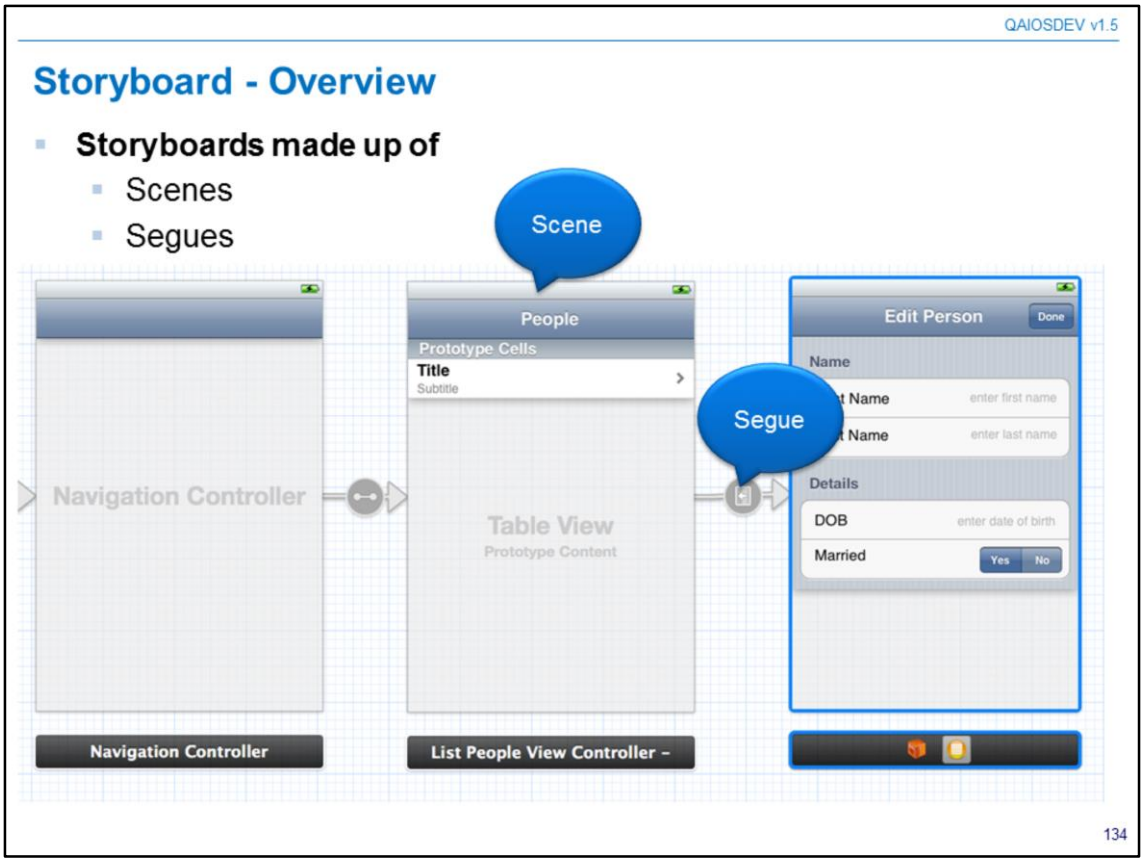

- **Create a new Storyboard within an existing project**



The simplest way to start using storyboards is to create a project that automatically use them.

When you create a project, make sure you check the Use Storyboards checkbox.

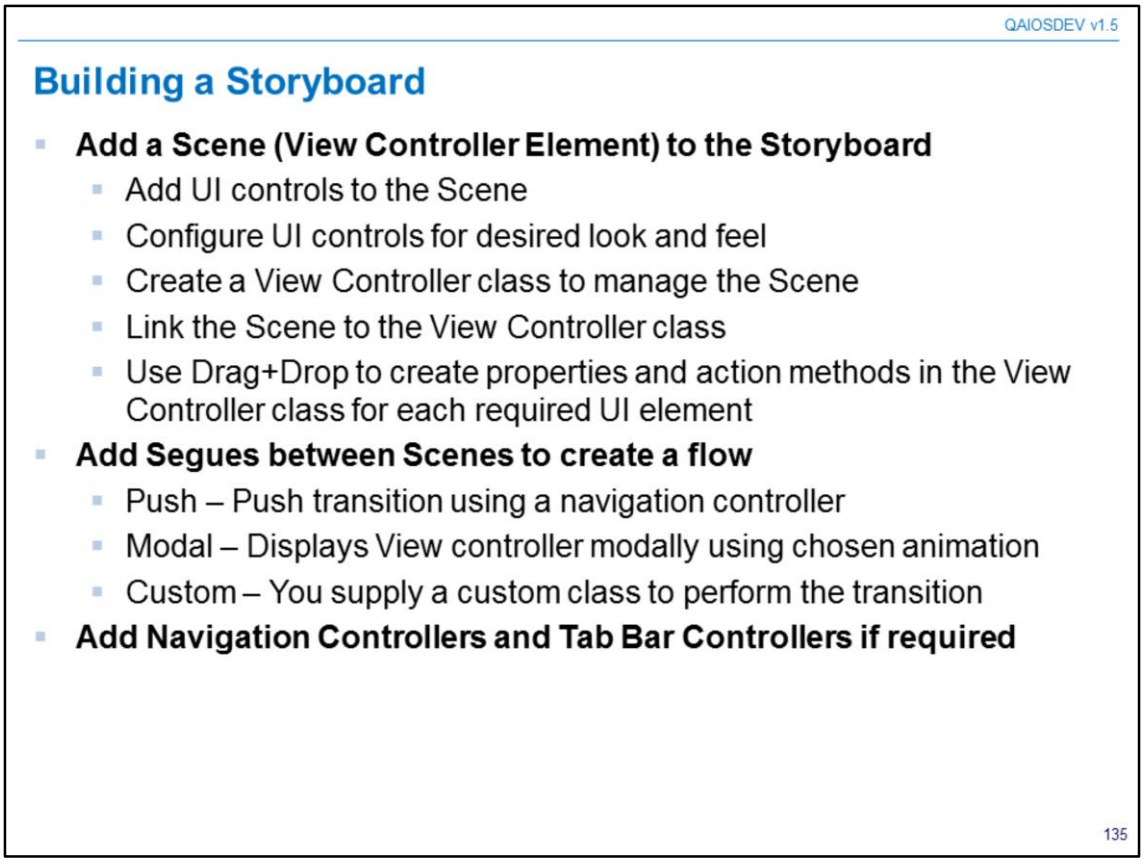
You can also create new storyboards within existing projects from the File->New menu under the User Interface tab.




Storyboard files have a .storyboard extension. When you open them, you will go straight into Interface Builder storyboard editor. From here, you can add new View Controllers (Scenes) from the object browser.

To each view controller you can design its interface exactly as you would when creating a XIB file.

The last step is to link the view controllers together to define a flow. The good news is that's just a case of drag dropping too and we will cover that in a later slide.




Creating a storyboard involves adding scenes (view controllers) and adding the UI elements for each scene.

Its not just about the design because eventually you need to link up all of those UI elements (like text boxes, segmented controls, switch controls etc ) up to code. Unfortunately, at the moment that bit is partially a manual task because even though you have added a scene (view controller) to your storyboard you also need to create a dedicated view controller class, to manage it. Even after you have created a new view controller class you then have to set the scene's class identity property in the identity inspector to the name of the dedicated view controller class you just created.

Once you have setup the scene and the view controller class, you can now start drag/dropping controls onto the code window to create properties and action methods as for XIB files.

Finally, you are ready to link up each of the view controllers into a flow. To do that, you just need to choose a UI element or gesture that will cause the flow to occur e.g. a button, press the CTRL key over it and drag the mouse over to the view controller you want to transition to. You will be asked to choose a Segue, one of Push, Modal or Custom.

## Segues

- **Segues define transitions between Scenes**
  - One View Controller displays one or more other view controllers
  - Push, Modal or Custom
- **Before Storyboards transitions were performed manually in code**
  - Set properties on target view controller to pass data
  - Called **pushViewController** or **presentModalViewController**
- **How do we pass data using segues in a storyboard?**
  - Handle **prepareForSegue** in source view controller

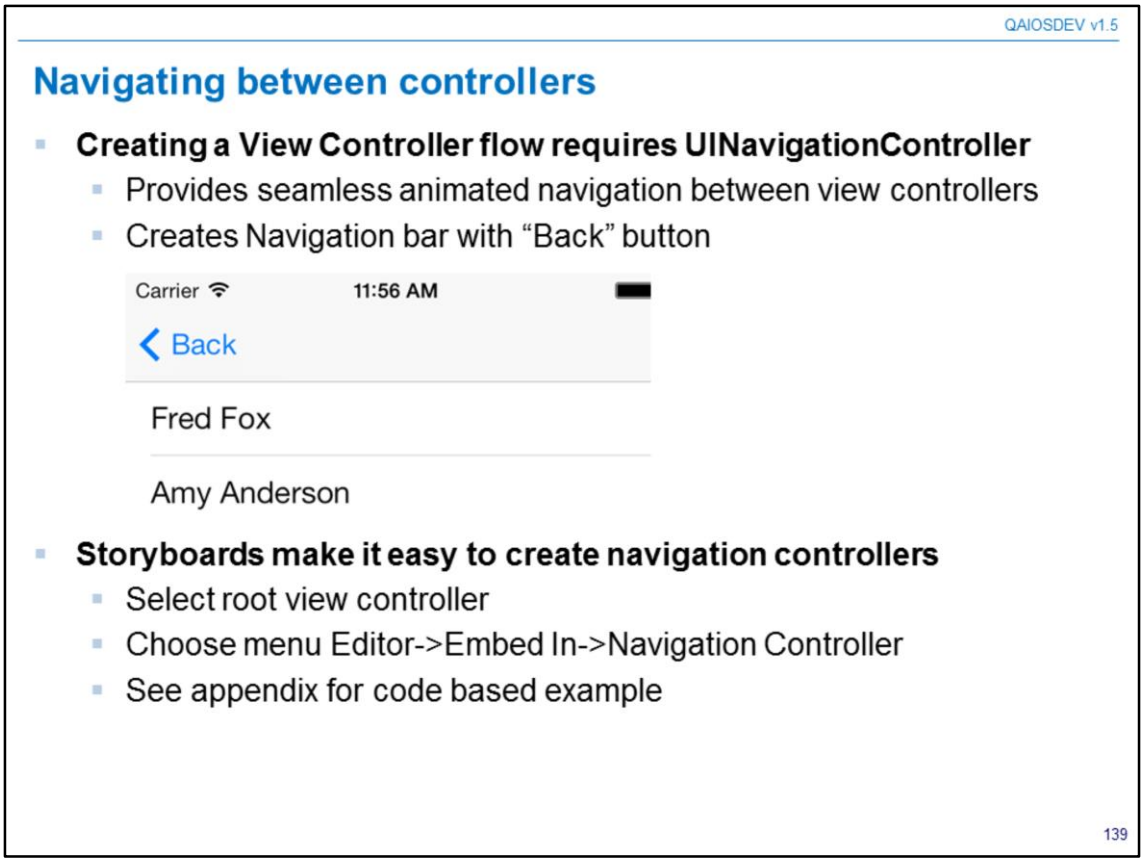
```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
   int idx=[self.tableView indexPathForSelectedRow].row;
   Person *person=[_people objectAtIndex:idx];
   PersonDetailsViewController *vc=segue.destinationViewController;
   vc.currentPerson=person;
}
```

137

As we have seen, segues define transitions between view controllers. Before storyboards, we did this manually in code using the navigation controller method **pushViewController** and the view controller methods **presentViewController** and **presentModalViewController** called from events such as **tableView:didSelectRowAtIndexPath**.

The advantage of this approach was that it was easy to pass data between view controllers prior to transition by just setting properties on the target view controller.

With storyboards, there isn't an event to handle so no opportunity to set up target properties. To get around this, Apple have provided a view controller method called **prepareForSegue**. This method provides access to the destination view controller and allows you to set any properties on it that you need to prior to the transition taking place.




A view controller on its own is pretty limiting. Most apps allow users to seamlessly move between view controllers to achieve sub tasks or display master/detail information.

Similar to other types of application, iOS allows you to create modal or sequential navigation. Modal involves displaying a view controller over the top of another and then dismissing it when the task is complete.

Sequential as the name suggests allows the user to move from one view controller to another sequentially and move backwards through the chain of controllers. This is a very common pattern in iOS apps and requires the use of a navigation controller class called UINavigationController to manage the list of view controllers and the navigation between.

From within a storyboard, select of root view controller and Choose menu Editor->Embed In->Navigation Controller.

## Tab Bar Controllers

- Specialised UIViewController to provide the familiar tab bar interface
- Each tab holds a single root view controller




Tab Bar Controllers provide the familiar tab button interface you have seen in many iPhone/iPad apps. They are a specialised type of view controller responsible for providing tabbed navigation between a set of view controllers.

Each tab is actually associated with a navigation controller that's linked to a default (root) view controller.

## Other View Controllers

- **Table View Controller (UITableViewController)**
  - Displays data in a scrollable table
- **Collection View Controller**
  - Similar to a table view gives greater control and flexibility over layout
- **Page View Controller (UIPageViewController)**
  - Scrollable book like navigation
- **Popover Controller (UIPopoverController)**
  - Creates a popup only partially covering the screen (iPad only)
- **Split View Controller (UISplitViewController)**
  - Creates a master detail split view (iPad only)

141

In addition to the view controllers, we have covered already there are a number of additional ones to be aware of.

UITableViewController is an essential view controller for display tabular data and you will it in use in may iOS apps. We will cover it in the next chapter.

There are also a number of view controller specifically designed for the iPad.

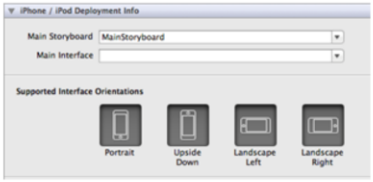
## iOS 7 Full Screen Layout

- **iOS 7 uses Full Screen Layout**
  - Content can fill entire screen and extend under status bar
- **iOS 7 design goal is to make content king**
  - Allow content to shine and be prominent
- **You can choose to extend content**
  - Under top bar
  - Under bottom bar
  - Under opaque bars
- **Set-up in IB or code**
- **View Controllers can fill the entire screen**

142

## Orientation – iOS 6

- shouldAutoRotateToInterfaceOrientation – deprecated
- Instead supported with view controller methods:
  - supportedInterfaceOrientations
  - shouldAutoRotate
  - preferredInterfaceOrientationForPresentation
- Can set Globally supported orientations at project level
  - Logical & with supportedInterfaceOrientations






In iOS 6 viewWillUnload and viewDidUnload are no longer called. You can still handle didReceiveMemoryWarning which you can use to unload views by setting self.view=nil;

Orientation management has also changed in that **shouldAutoRotateToInterfaceOrientation** has been deprecated and replaced with calls to **supportedInterfaceOrientations** and **shouldAutoRotate.**

**supportedInterfaceOrientations** returns an OR-ed mask of supported orientations one of:

- UIInterfaceOrientationMaskPortrait
- UIInterfaceOrientationMaskLandscapeLeft
- UIInterfaceOrientationMaskLandscapeRight
- UIInterfaceOrientationMaskPortraitUpsideDown
- UIInterfaceOrientationMaskLandscape
- UIInterfaceOrientationMaskAll
- UIInterfaceOrientationMaskAllButUpsideDown

A view controller can also specify its preferred orientation which must be one of its supported orientations. A view controller playing video for example will usually prefer landscape.
Orientation can be set global from the project/target settings. These settings or logically AND-ed together with the view controller supported orientations.

## View Controller Orientation

- **Controlling Orientation**

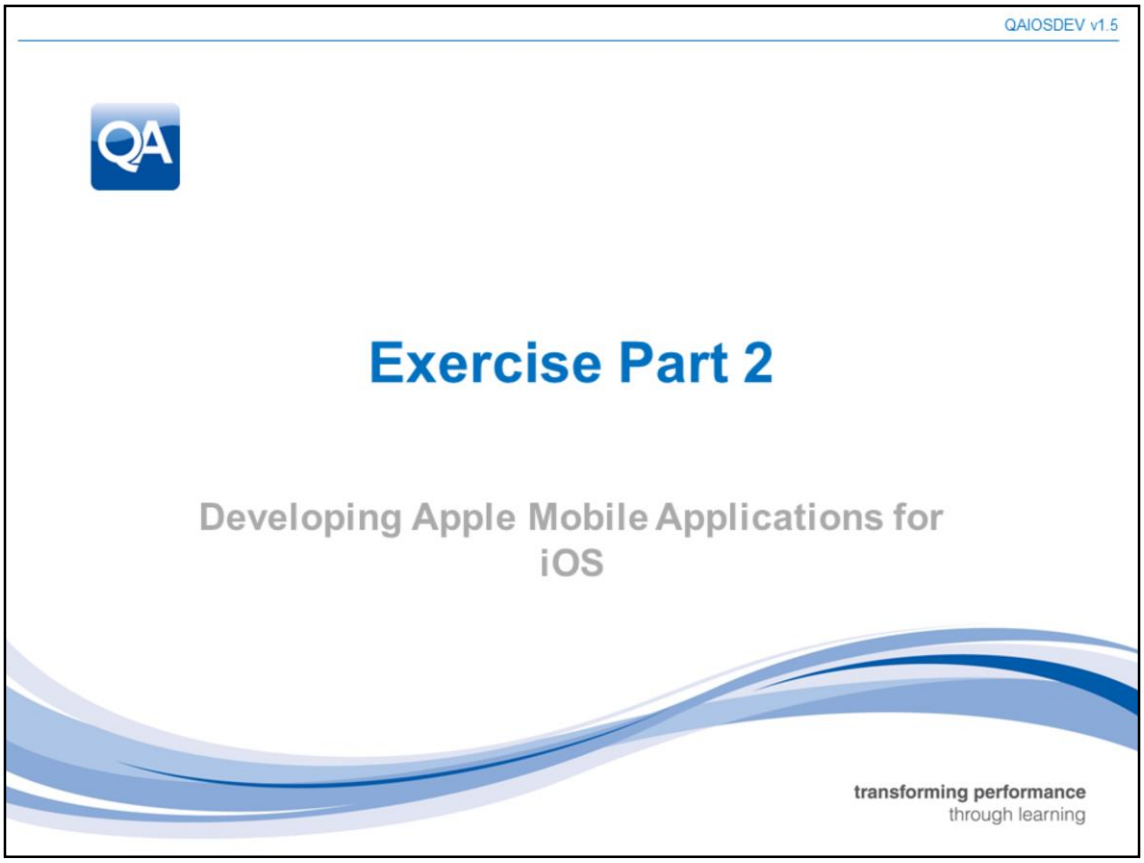
```
-(NSUInteger)supportedInterfaceOrientations
{
  return UIInterfaceOrientationMaskPortrait |
UIInterfaceOrientationMaskLandscapeLeft ;
}
-
(UIInterfaceOrientation)preferredInterfaceOrientationForPresentation
{
  return UIInterfaceOrientationPortrait;
}
-(BOOL)shouldAutorotate
{
  return YES;
}
```

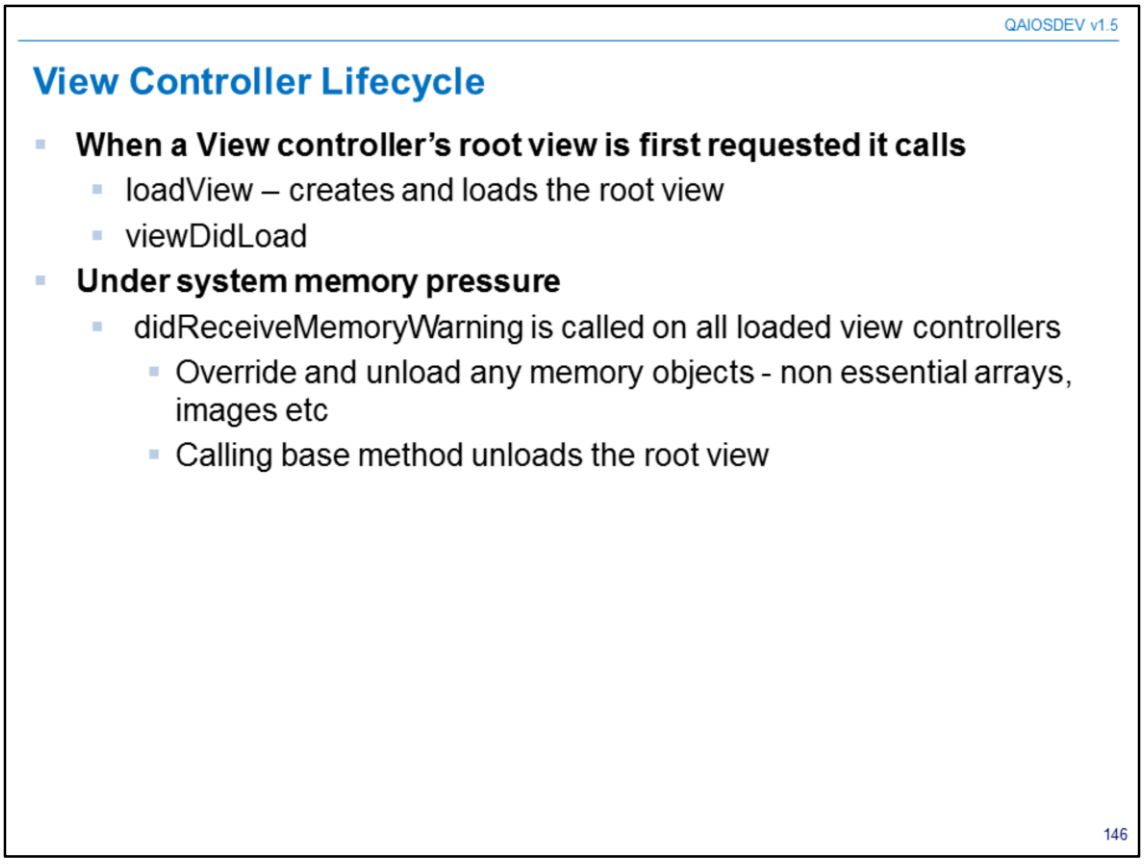


In the example code above, the view controller supports Portrait and Landscape left but prefers portrait.

Notice the returned orientations differ from the masks returned from supportedInterfaceOrientations

- UIInterfaceOrientationPortrait
- UIInterfaceOrientationLandscapeLeft
- UIInterfaceOrientationLandscapeRight
- UIInterfaceOrientationPortraitUpsideDown


QAIOSDEV v1.5
Exercise Part 2
Developing Apple Mobile Applications for iOS
transforming performance
through learning




When your view controller's root view is first accessed, it calls **loadView** giving you the opportunity to override the creation and loading of the root view. This is followed by **viewDidLoad** and is typically where you would load UI controls and UI dependent data structures such as arrays etc.

Under memory pressure, the system will call **didReceiveMemoryWarning,** which by default, will unload the view controller's root view.

When the view controller's root view property is next accessed, the view will be reloaded with calls to **loadView** and **viewDidLoad.**

The correct response to a low memory warning is to unload all UI properties and UI related data that was created and retained in **loadView** or **viewDidLoad**.

## Handling a low memory warning

- **didReceiveMemoryWarning called in your view controller**
  - Override and provide data release code

```
-(void)application:didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning ]; // Unloads root view
    // Release memory objects
     self.myArray=nil;
}
```

- **Handle memory warning in App Delegate**
  - applicationDidReceiveMemoryWarning



When **didReceiveMemoryWarning** is called on your view controller, you need to unload any resources that can be reloaded e.g. images, caches etc. The base behaviour of didReceiveMemoryWarning is to unload the root view of the view controller.

Low memory notifications are also received in the app delegate in **applicationDidReceiveMemoryWarning.** You should release anything that is non critical and can be reloaded on demand.

By doing this you maximise the amount of memory you can release when a low memory event occurs and increase the chances of your app surviving

## Summary

- **iOS app start with a main() function**
- **Application events handled by an App Delegate**
- **An iOS app can have multiple windows**
- **A window displays multiple views**
- **View Controllers**
  - Manage views and their state
  - Handle Low memory warnings
  - Deal with orientation changes
- **UIViewController is base type for all view controllers**
- **UINavigationController – sequential navigation between controllers**
- **UITabBarController – tabbed navigation between controllers**

QAIOSDEV v1.5

## Creating a View Controller

- **Interface**

```
@interface MyViewController: UIViewController
@property (nonatomic, strong) UITextField * titleTextField;
@end
```

- **Implementation**

```
@implementation MyViewController
@synthesize titleTextField=_titleTextField;
-(void)viewDidLoad
{
    self.titleTextField=[[UITextField alloc] initWithFrame:CGRectMake(130,
50, 180,30)];
     self.titleTextField.text=@"iOS in a nutshell";
     [self.view addSubview:self.titleTextField];
}
@end
```

149

Creating a new view controller is straightforward:

- Create a new class inheriting from UIViewController.
- Setup some UI interaction views as properties
- Add these UI elements to the root view in **viewDidLoad**
- Respond to UI and system events

## Adding your View Controller to the App

- **Create a View Controller**
- **Import the View controller into the AppDelegate**
- **Set the Window's root view controller**

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]];

    MyViewController *vc=[MyViewController new];
    self.window.rootViewController= vc;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
return YES;
}
```



To now add a view controller to the main app window is really simple. All you need to do is create your view controller and assign it to the root view controller property of the window.

## Creating a Navigation Controller

- **AppDelegate**

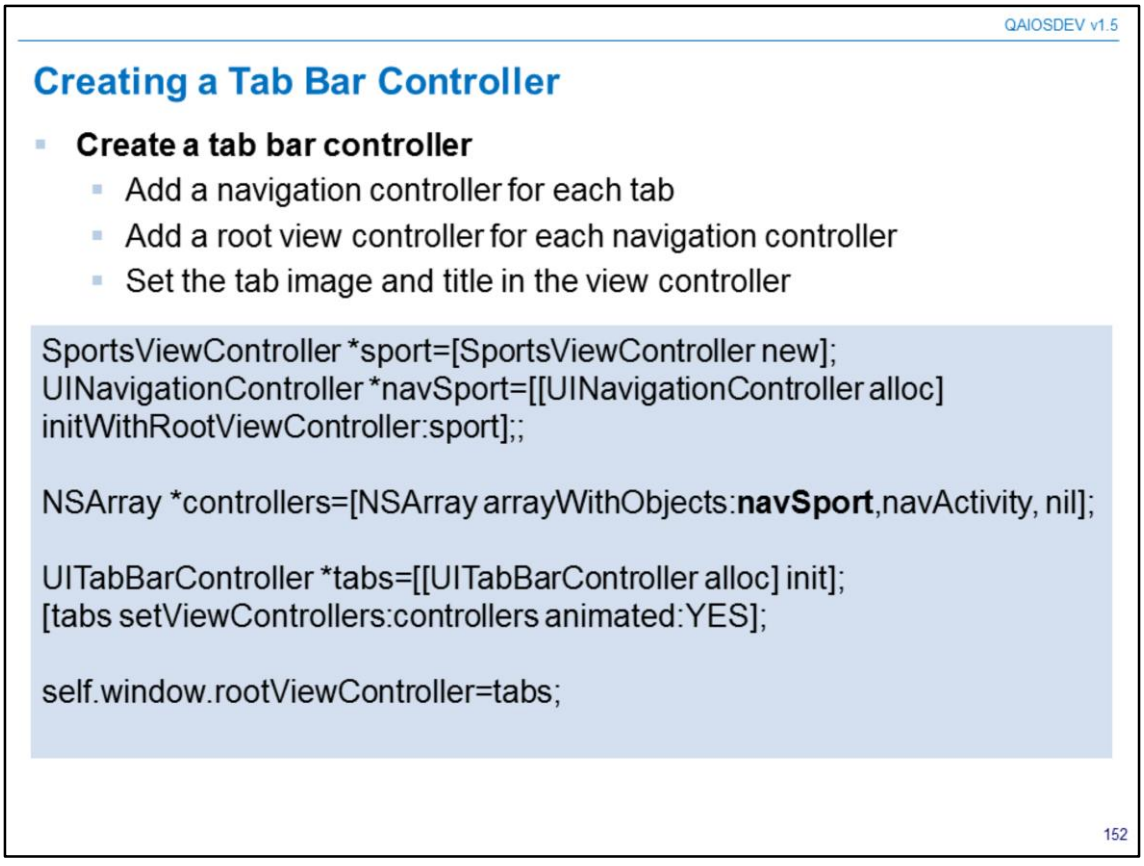
```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]];

    MyViewController *myVw=[MyViewController new];
    UINavigationController *nav=[[UINavigationController alloc]
          initWithRootViewController:myVw];
     self.window.rootViewController=nav;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

151

Instead of using our view controller directly, the navigation controller is going to act as the window's root view controller instead. This allows it to be in control of navigating from one view controller to another.

You have to tell it which view controller to use as its starting point (the first view controller to display). You do that at initialization with the call to **initWithViewController**.

And that's it! The navigation controller will now manage the display, transition and navigation of your view controller sequences.




For each tab, you need to create a navigation controller and add a root view controller that will be the root display view controller.

Each navigation controller is then added to an array and the tab bar controller is created from the array.

Each root view controller is responsible to specifying the tab bars icon and caption.

Here the root view controller is setting the tabBarItemImage and title inside its **init** method.

```
-(id)init
{
        if (self=[super init])
        {
                // Set the tab bar image and title
                self.tabBarItem.image = [UIImage
imageNamed:@"basketball.png"];
                self.title=@"Sports";

        }
        return self;
}
```

## Pushing View Controllers onto the Navigation stack

- **View Controllers can be pushed onto the Navigation Controller stack**
  - View controllers have a **navigationController** property
  - Use **pushViewController** method
- **Example – Pushing a VC onto the navigation stack**

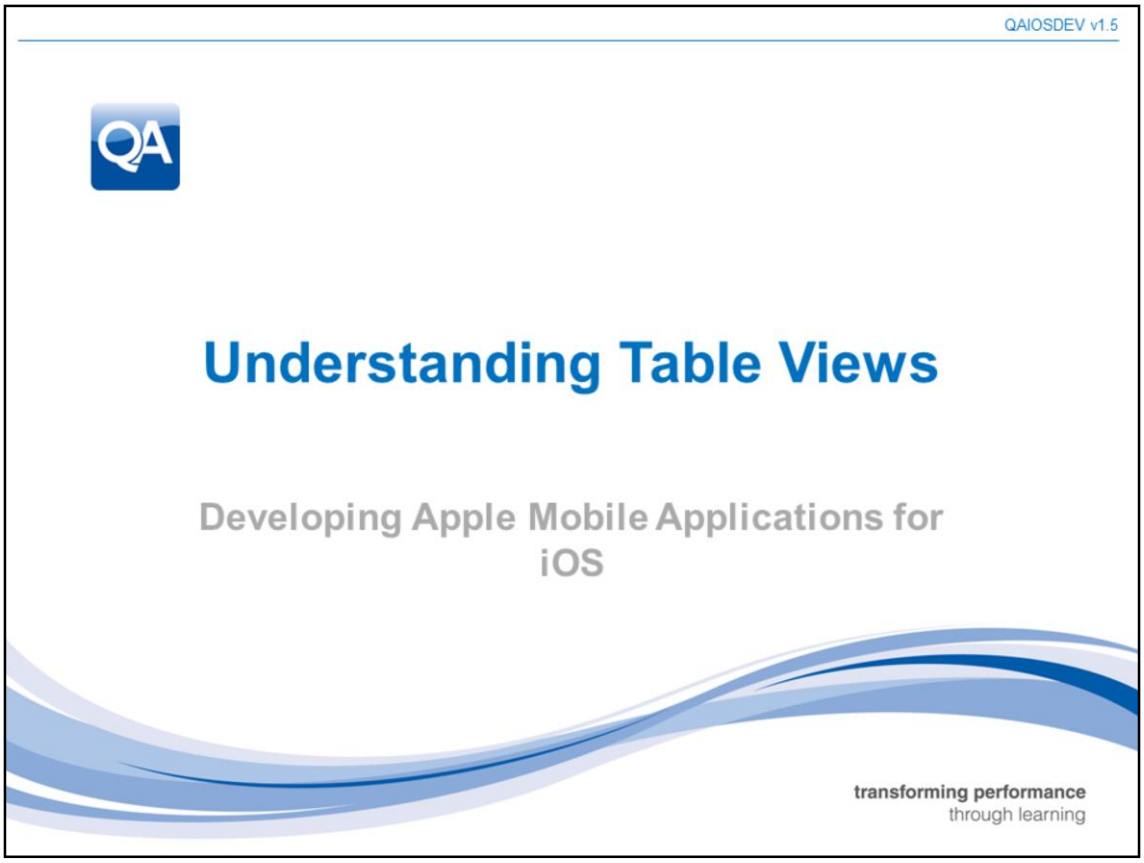
```
// Some View Controller Event
DetailViewController * det=[DetailViewController new];
[self.navigationController pushViewController:det];
```

- **Navigating backwards**

```
[self.navigationController popViewControllerAnimated:TRUE];
```

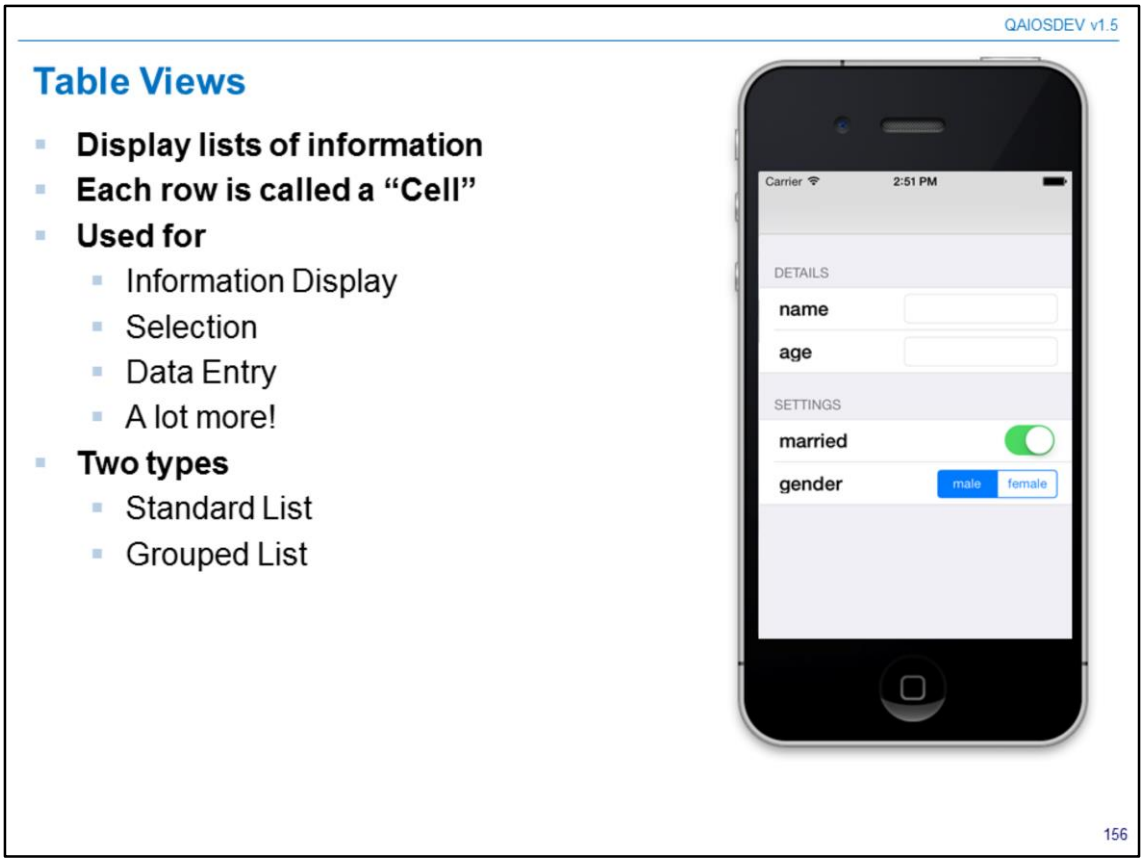
153

But how do you move from one view controller to another? Simple, just push the new view controller onto the navigation controller stack and it will do the rest. You do that by calling **pushViewController** on the view controller's navigation controller. self.**navigationController** is a property of all view controllers.


QAIOSDEV v1.5
Understanding Table Views
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - To understand the importance of table views
  - To be able to work with table views to create UIs
- **Contents**
  - Overview of Table Views
  - Table View Events
    - Data Source Events
    - Delegate Events
  - Creating table views
  - Simple and Grouped styles
  - Cells and Sections
  - Responding to interaction
  - Customizing a table view
    - Direct cell manipulation
    - Sub classing table cells
  - Table View Controllers
- **Hands on Labs**




Displaying data in a table format is a useful UI technique. iOS has a table mechanism called table views that provide a relatively easy way to present data to a user.

You can use table views to display lists of information or to host user input controls for data entry. The iPhone Settings app uses table views as its primary interface with the UI control held within.

There are two types of table view:

- Simple List that displays a sequential list of data
- Grouped list that creates a grouped display

When you create a table view you set the type you want to create.

You can also produce an indexed list that allows you associate an index with the table view and to navigate within the list using the index. Just provide implementations for:
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *) **and**
- (NSInteger)tableView:(UITableView *)tableView
sectionForSectionIndexTitle:(NSString *)title atIndex:(NSInteger)index

## Table View Sections

- **Table Views can have multiple sections**
- **Each Section can have**
  - Header
  - Footer
- **Advantages**
  - Improved usability
  - Creates context
  - Enhances list browsing


157

Table views can be sub divided into sections and as we have already seen you specify the number of sections in **numberOfSectionsInTableView.**

Each section has a header and a footer and you can control the text of both. You can actually control the view displayed for both as well by implementing the correct methods (later).

Sections help give context to a list and aid usability. They also make list browsing much easier.

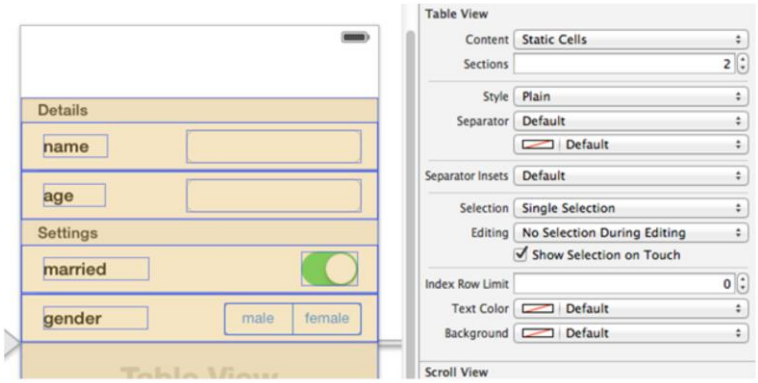
## Types of Tables View

- **Two types of table view**
  - Static
  - Dynamic
- **Static Table Views**
  - Each row (cell) design/configured at design time
  - Ideal for Form based user input capture
- **Dynamic Table View**
  - Generated at runtime
  - Great for displaying retrieved data in a list
- **Created from within a storyboard**

QAIOSDEV v1.5

## Static Table View

- **Statically Defined UI**
  - Set Table View Content to Static Cells
  - Set Style,Sections, Rows
  - Add UI Elements and Connect to View Controller outlets
  - Remove from View Controller Class data source methods




159

To create a static table view you first have to set its content type to “Static Cells”. Having done that you can then define the table style and set up sections and rows all of which will display in the designer.

Now you can add the UI element you need to the table cells, connect the UI elements to the view controller by creating properties and action methods and you’re done.

Finally, make sure you remove the following methods from the view controller:

- tableView:cellForRowAtIndex
- tableView:numberOfRowsInSection
- numberOfSectionsInTableView

# Static Table View Connections

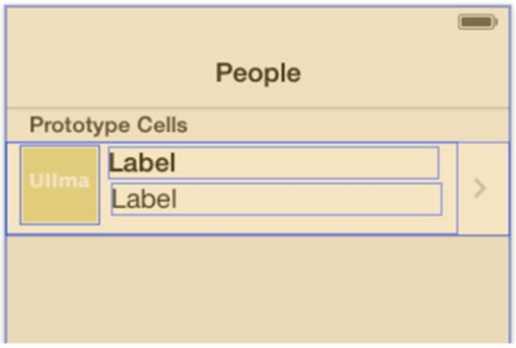
- **Wire up a Static Table View just like any UI**
  - Connect UI elements to to properties

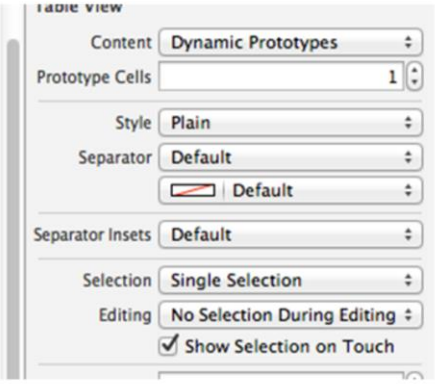

```
//  PersonDetailsViewController.h
//  StaticDynamicTableViews
//
//  Created by Christopher Farrell on
    06/04/2012.
//  Copyright (c) 2012 QA.com. All rights
    reserved.
//

#import <UIKit/UIKit.h>
#import "Person.h"
@interface PersonDetailsViewController :
    UITableViewController
@property (strong, nonatomic) IBOutlet
    UITextField *firstName;
@property (strong, nonatomic) IBOutlet
    UITextField *lastName;
@property (strong, nonatomic) IBOutlet
    UISegmentedControl *isMarried;
@property (nonatomic) Person *
    selectedPerson;
@property (strong, nonatomic) IBOutlet
    UISegmentedControl *gender;
- (IBAction)donePressed:(id)sender;
@end
```

160

## Dynamic Table View

- **Data Driven Table View**
  - Cells created in code at runtime
  - Set Table View Content to Dynamic Prototypes
  - Create one or more prototype cells

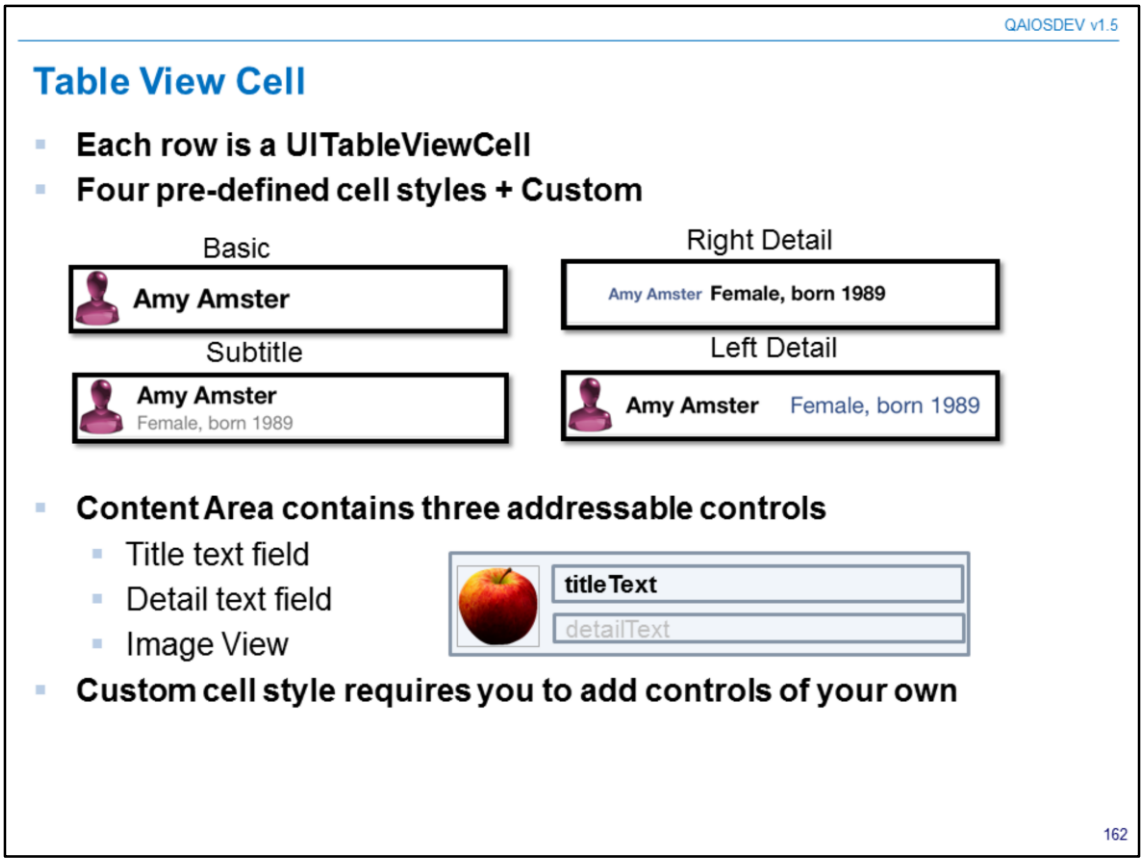



- **Prototype cells are a template you can use to create a cell at runtime**
- **Four Predefined templates supplied**

161

To create a dynamic table, view set the content type of the table view to “Dynamic Prototypes” and choose the number of prototype cells you are going to define. If you are only going to create one customised cell type then leave it at one.




Each row in a table is a UITableViewCell. A table view cell can be in one of two modes depending on the mode of the table view:

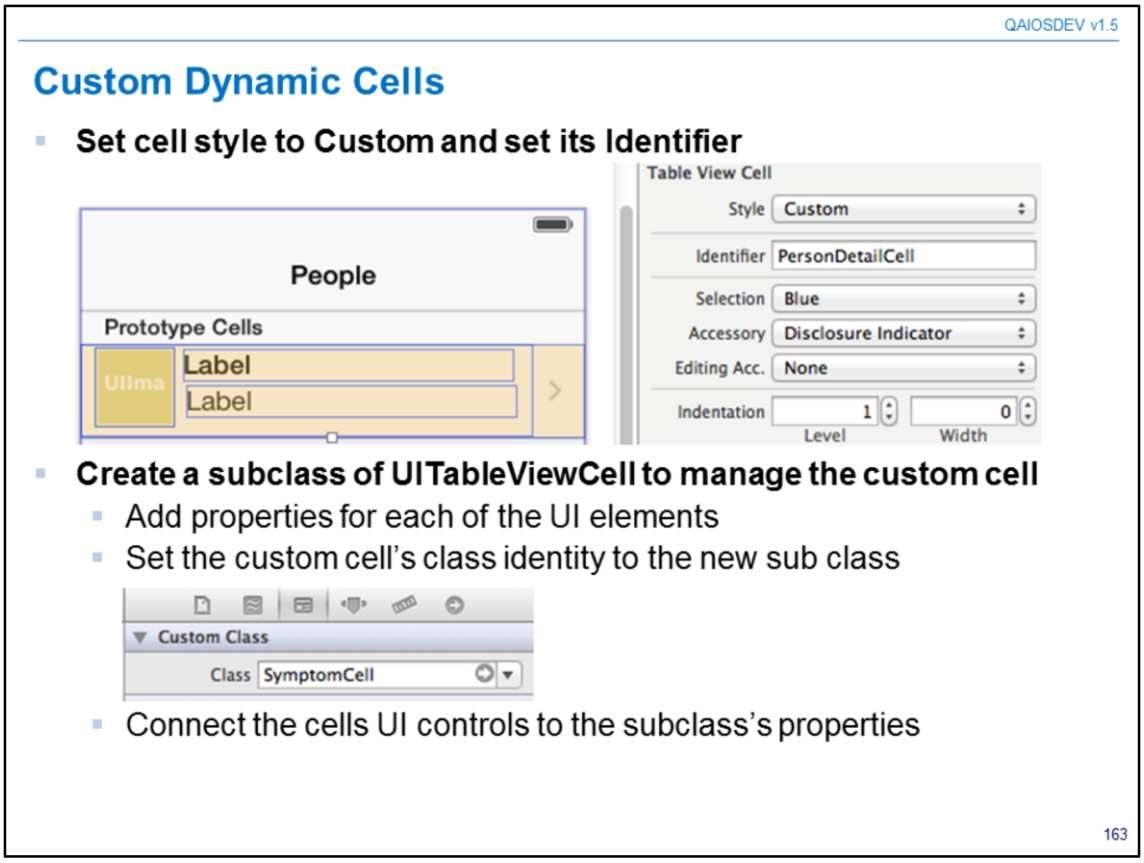
- View Mode
- Edit Mode

In view mode, the full content area is available and a small area to the left is reserved as the accessory view where an accessory button can be displayed. This button usually indicates that the row is part of a master detail relationship and has related to it. Selecting the row should take the user into the detail view.

In edit mode, table cells can be deleted or moved so the view changes to allow for the display of a delete disclosure button on the left hand side and a move row button on the left. As a result, the content view size is reduced.

The content view of a table view cell has a pre configured layout containing a title text field an image field and a detail text sub field. The basic layout of these three display controls can be controlled by setting the cell style. The images above give the best indication of how the four different styles available look.

When you create a table cell, you specify the type of cell to create.




You're now ready to customise the individual prototype cells. By clicking on a cell and inspecting its attributes you can set the cell's style;

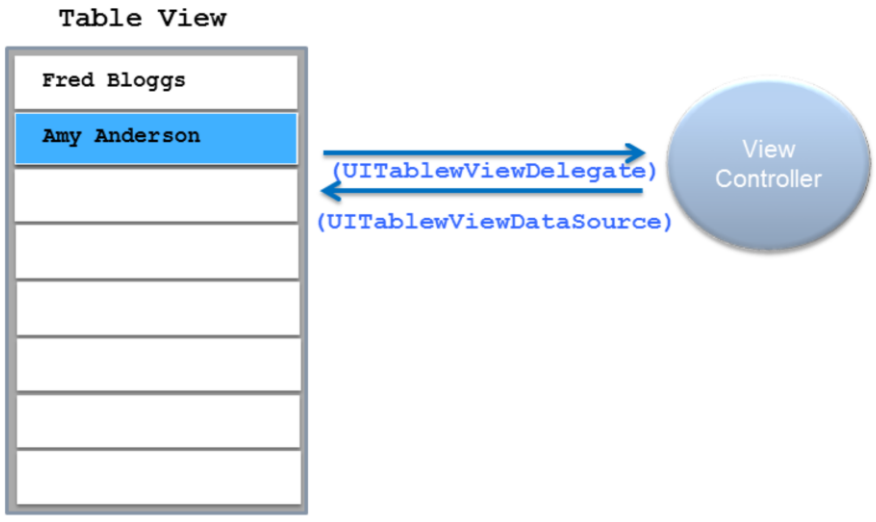
You can chose from
- Basic
- Subtitle
- Left Detail
- Right detail
- Custom

The first four are predefined layouts displaying a textField, detailTextField and an imageView in different positions. Basic doesn't display the detailTextField.

Custom allows you to define your own content using all of the available UI elements.

Once you have defined the UI, set the identifier attribute for the cell to a unique name because we will need to use it later on.

## Table View Delegates – Overview






When a table view executes it asks, the host view controller how many rows of data it needs to display. Then for each row it ask the controller to create a table cell filled in with all of the information to be displayed.

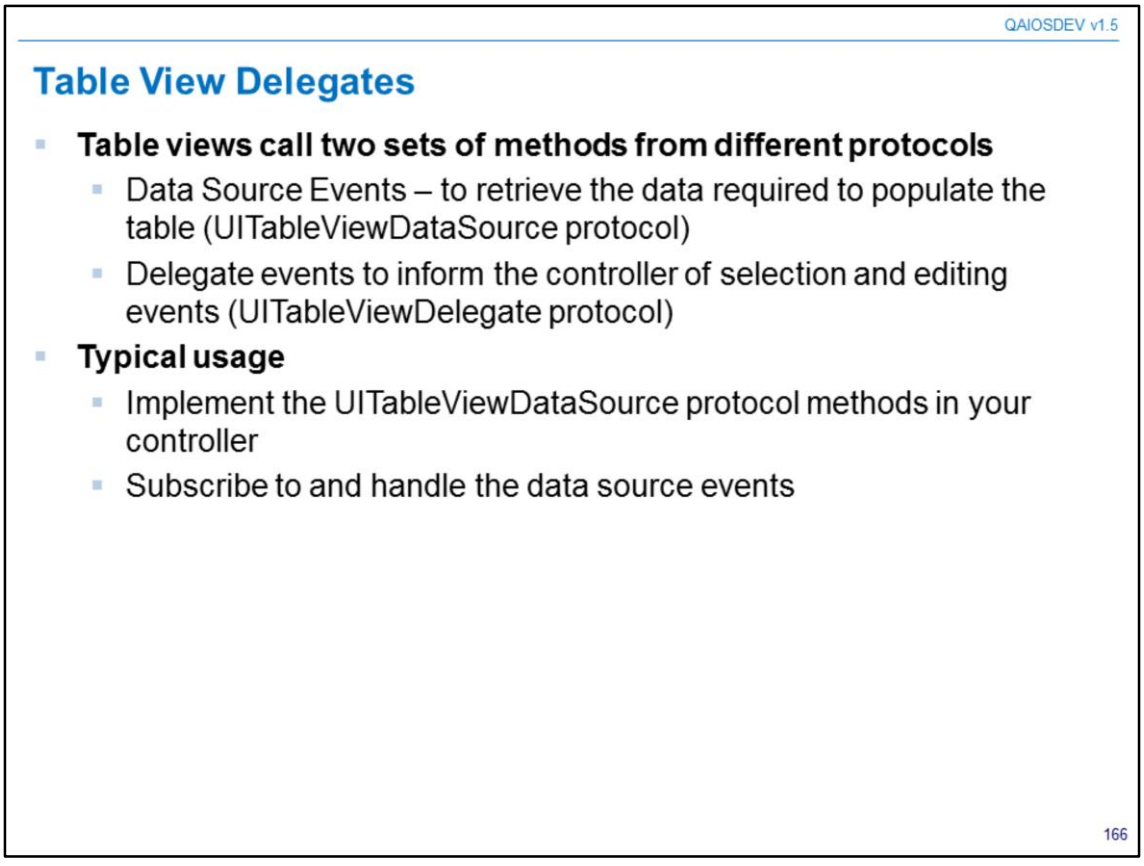



Table views request data from their host controller using the standard iOS eventing model (protocols and delegates).

There are two types of events:

- Data Source events that requests the data to display from the host
- Delegate events that inform the host of events that have occurred on the list e.g. item selection/deletion etc.

DataSource events form part of the UITableViewDataSource protocol and table view host controllers need to implements some key methods within this protocol for the table view to work properly.

Delegate events require the host to implement UITableViewDelegate methods.

## Key Data Source Events fired by a Table View

- **numberOfSectionsInTableView:**
  - Asks the controller how many sections the table view should have

```
-(int)numberOfSectionsInTableView:(UITableView *)tableView
{
        return 1;
}
```

- **tableView:numberOfRowsInSection:**
  - Asks the controller how many rows a section has

```
-(int)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
        return 15;
}
```



A table view can display one or more sections that allow you to group items in your list under separate headers. This can help to break the list up and give a little more context to a long list.

It works for both simple and grouped lists. You set the number of sections in a table view by returning the number from the **numberOfSectionsInTableView** method in the table view's host controller. To set the title of a section you return it as a string from method **tableView:titleForHeaderInSection:**

## Generating Cells at Runtime

- **Handle tableview:cellForRowAtIndex**
  - Use the cell prototype identifier to generate a cell
  - Fill in the cell fields

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    static NSString *CellIdentifier = @"MyCell";
    UITableViewCell *cell = [self.tableView
dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];

    cell.textLabel.text= @"Some Info";
    cell.detailTextLabel.text=@"Detail Info";
    cell.imageView.image=[UIImage imageNamed:@"MyLogo.png"];
    return cell;
}
```

  - Cells scrolled off screen are recycled on subsequent requests



All that is left to do is to generate the table cells at runtime. This follows exactly the same mechanism as for standard table views using the UITableViewDataSource delegate. The only difference is in the way **tableview:cellForRowAtIndex** is handled.

This time cells are created by calling **dequeueReusableCellWithIdentifier** and passing the prototype identifier for the cell prototype you want to create. That will return a table cell of the correct sub class type. Using the returned cell you can fill in its custom fields.

Although it is a bit fiddly to set up, it does allow you to create a very intricate and complex table design with the UI managed within Interface Builder.

## Generating Cells at Runtime

- **Handle tableview:cellForRowAtIndex**
  - Use the cell prototype identifier to generate a cell
  - Fill in the cell fields

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    static NSString *CellIdentifier = @"MyCell";
    UITableViewCell *cell = [self.tableView
dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];

    cell.textLabel.text= @"Some Info";
    cell.detailTextLabel.text=@"Detail Info";
    cell.imageView.image=[UIImage imageNamed:@"MyLogo.png"];
    return cell;
}
```

  - Cells scrolled off screen are recycled on subsequent requests



All that is left to do is to generate the table cells at runtime. This follows exactly the same mechanism as for standard table views using the UITableViewDataSource delegate. The only difference is in the way **tableview:cellForRowAtIndex** is handled.

This time cells are created by calling **dequeueReusableCellWithIdentifier** and passing the prototype identifier for the cell prototype you want to create. That will return a table cell of the correct sub class type. Using the returned cell, you can fill in its custom fields.

Although it is a bit fiddly to set up, it does allow you to create a very intricate and complex table design with the UI managed within Interface Builder.

## Responding to Table View Selection Segue

- **Performing a Segue**

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    int idx=[self.tableView indexPathForSelectedRow].row;
    Person *person=_people[idx];
    PersonDetailsViewController *vc=segue.destinationViewController;
    vc.currentPerson=person;
}
```



Here is how to respond to selected row by performing a segue. You can also do this by handling didSelectRowAtIndexPath (see appendix).

## Managing TableView Sections

- **Setting Header Titles**

```
-(NSString*)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
        return [NSString stringWithFormat:@"Section %i", section];
}
```

- **Managing Cells in different sections**
  - Row index starts at zero for each section!

```
-(UITableViewCell*)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    …
    if (indexPath.section==1)
    {
    }
}
```

171

To control the text inside the header and footer for each section you simply return strings from the following methods:

- **tableView:titleForHeaderInSection**
- **tableView:titleForFooterInSection**

If you want the header/footer to have a specific look, you can create a view instead. To do this you need to implement methods:

- **tableView:viewForHeaderInSection**
- **tableView:viewForHeaderInSection**

Both methods return a UIView and will override what you have done with titleForHeaderInSection and titleForFooterInSection.

When you have a multi section table, you will need to know the section number requested as well as the row number. That's because when a row is requested from **cellForRowAtIndexPath** for a new section the row index starts from zero.

## Deleting Rows

- **Implement commitEditingStyle**

```
-(void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {

    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        [_books removeObjectAtIndex:indexPath.row];
        NSArray  *paths=[NSArray arrayWithObject:indexPath];
       [self.tableView deleteRowsAtIndexPaths:paths
                                     withRowAnimation:YES];
}
```



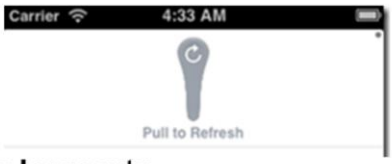
To delete a row implement method **commitEditingStyle**. You need to make sure that the number of items in your data array and the number of items in the table view match when modifying either.

In the above example a book has been removed from both the data array and the table view. Failure to do this will result in an error.

QAIOSDEV v1.5

## Table View Controllers Refresh Control – iOS 6

- **Table Views now have a refresh control**
  - Drag down to cause an auto refresh
  - Create UIRefreshControl control
  - Set Table View Controller's **refreshControl** property




```
// viewDidLoad
UIRefreshControl *refresh = [[UIRefreshControl alloc] init];
[refresh addTarget:self  action:@selector(refreshView:)
                forControlEvents:UIControlEventValueChanged];
self.refreshControl = refresh;

-(void)refreshView:(id)sender
{
   // Reload Data
   [self.refreshControl endRefreshing];
}
```

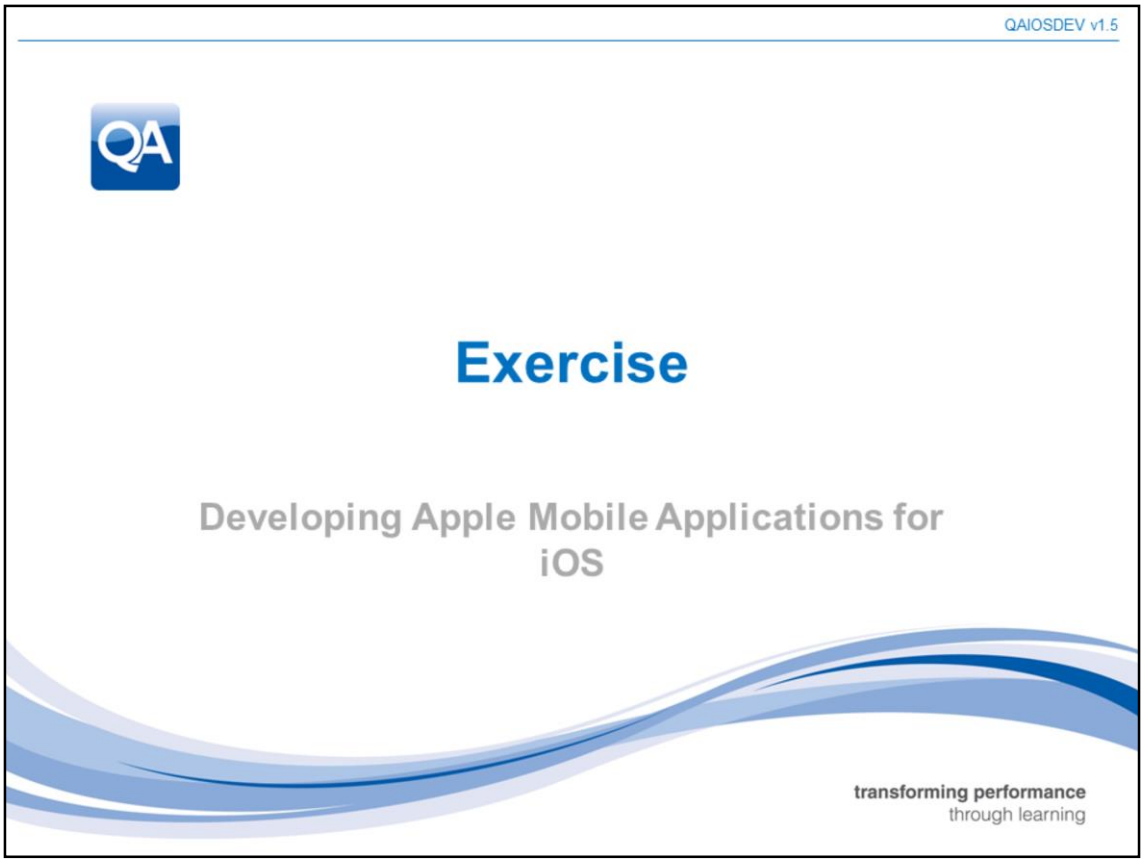
173

Table View Controllers now have a refresh control. You can use it to give the familiar pull down to refresh interface.

You just need to create an instance of a UIRefreshControl and assign it to the table view controllers **refreshControl** property. You also need to add a target/action event for **valueChanged** to handle when a refresh event occurs.

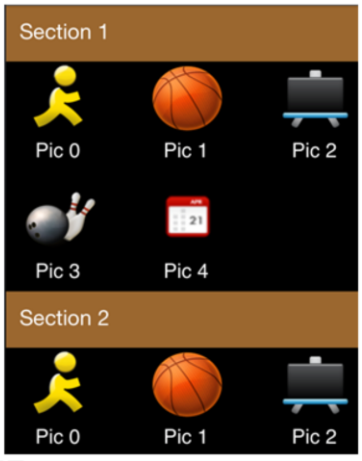
The code above handles a refresh event using a **refreshView** method. When the refresh has taken place you need to call [self.refreshControl endRefreshing] to hide the refresh control.

A Refresh Control can also be added to a Table View from within Interface Builder by setting the Refreshing property of the Table View Controller to enabled. You still have to wire up the target/action event in code though.


QAIOSDEV v1.5
Exercise
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Collection Views – iOS 6

- **Displays rows and columns of data**
  - Create Data Grids, List Views etc
  - UICollectionViewFlowLayout controls layout (can be overriden)
- **Uses same principles as Table Views**
  - UICollectionViewDataSource
  - UICollectionViewDelegate
- **You Design custom Cell types in IB**
  - Create cell of correct type at relevant position in the flow
- **You Design custom Section Header Types in IB**
  - Create correct type of section header when asked






Collection Views are designed to allow for the display of data grids of information. They allows both columns and rows of data to be displayed using a standard flow layout mechanism.

They basically follow the same pattern as table views supporting a DataSource and Delegate protocol pattern.

You design your custom cells in IB and when asked create and assign data to the correct cell type when asked.

You can also create custom section headers.

## UICollectionView - Controlling Item Display

- **Managing Section and Row counts**
  - UICollectionViewDataSource

```
-(int)numberOfSectionsInCollectionView:(UICollectionView
*)collectionView
{
        return 3;
}

-(int)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section
{
        return 5;
}
```

176

UICollectionViewSource protocol is used to implement the data source methods.

numberOfSectionsInCollectionView and numberOfItemsInSection control the number of sections to display and the number of items in each section.

QAIOSDEV v1.5

## Collection Views – Creating Cells

```
-(UICollectionViewCell*)collectionView:(UICollectionView *)collectionView
cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
  ItemCell *cell;
  cell=[collectionView
        dequeueReusableCellWithReuseIdentifier:@"itemCellTemplate"
        forIndexPath:indexPath];

  cell.dataLabel.text=@"Pic 0";

  cell.imageView.image=[UIImage imageNamed:@"Pic0.png"];
  return cell;
}
```

177

cellForItemAtIndexPath is the method that returns the genrated cell. In this case ,we have previously created a cell in IB called **itemCellTemplate** associated with a custom cell class ItemCell (inherited from UICollectionViewCell).

After setting the cells custom properties, it is returned and displayed in the collection view.

## Collection View Headers

```
- (UICollectionReusableView *)collectionView: (UICollectionView
*)collectionView viewForSupplementaryElementOfKind:(NSString *)kind
atIndexPath:(NSIndexPath *)indexPath {

  MyHeader *hw ;
  hw = [collectionView
                dequeueReusableSupplementaryViewOfKind:
                                    UICollectionElementKindSectionHeader
                withReuseIdentifier:@"HeaderTag"
                forIndexPath:indexPath];

  headerView.backgroundColor=[UIColor brownColor];
  headerView.title.text=@"Section";
   return headerView;
}
```



To create a custom header a reusable UICollectionReusableCell is created in IB and returned from **viewForSupplementaryElementOfKind.**

## Summary

- **TableViews present multiple rows of information**
- **Can be plain or grouped**
- **Can have one or more sections**
- **Communicate using delegate methods for protocols**
  - UITableViewDataSource
  - UITableViewDelegate
- **Request data from their controller**
  - Number of rows, sections an cell formatting
  - Row sizes and editing status
  - Header and foot information
- **Inform their controller of user events**
  - Row selected, edit, delete or insert events
- **Table Cells have a default format that can be customized**
- **Cells can be subclassed**

QAIOSDEV v1.5

## Responding to Table View Events

- **UITableViewDelegate protocol defines table view event and property methods**
  - tableView:**didSelectRowAtIndexPath**
  - tableView:**willDisplayCell**
  - tableView:**accessoryButtonTappedForRowWithIndexPath**
  - tableView:**heightForRowAtIndexPath**
- **Example**

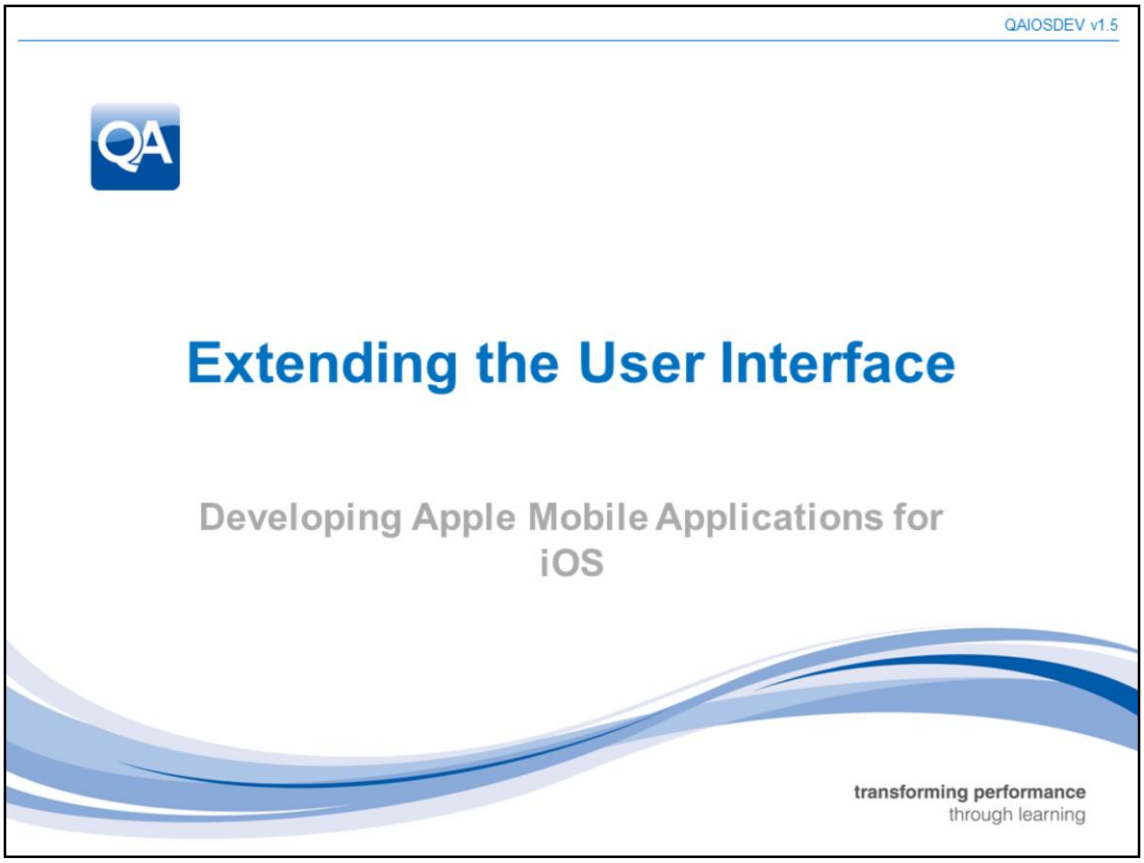
```
-(void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    Book *book=[self.books objectAtIndex:indexPath.row];
    BookViewController *bookVC=[BookViewController  new];
    bookVC.currentBook=book;
    [self presentViewController:bookVC animated:YES completion:nil];
}
```

180

As well as retrieving data from the host controller, the table view also needs to inform the host of table events. The UITableViewDelegate protocol has a number of methods that are used by UITableView to fire events to the host and also to retrieve certain table view properties.

- **tableView:didSelectRowAtIndexPath** is fired when a user touches a specific row. It passes the indexPath describing the row and section that was selected
- **tableView:willDisplay** is called just before a cell is displayed
- **tableView:accessoryButtonTappedForRowWithIndexPath** fired when the accessory button is pressed
- **tableView:heightForRowAtIndexPath** can be used to set the height of individual rows

This is not a definitive list and there are events and property methods spread between both UITableViewDataSource and UITableViewDelegate to control all aspects of a tableview's display and operation. As always refer to the Apple SDK documentation for the full list.


QAIOSDEV v1.5
Extending the User Interface
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - To build on existing Interface Builder skills
- **Contents**
  - iOS 7 Font and Colour
  - Auto Layout
  - UI Controls
  - Handling Text Input
  - Dealing with Date and Time Entry
  - Pick lists
  - Other Controls
  - Unwind Segues
- **Hands on Labs**



This chapter is all about extending the user interface using the built in controls within UIKit.

## iOS 7 Design Changes

- **iOS 7 has a flattened UI with fewer 3D graphical design effects**
- **Design should be deferential to the Content**
- **Apps can use colour to help give a distinctive look**
- **tintColor is used to control colour of toolbar buttons etc**
  - Used for interactivity and selection state
- **Setting**
  - window.tintColor=[UIColor redColor];
- **Will set tintColor for the whole app as it is now inherited by all sub views**
- **Defaults to system default color**

## Image Rendering Modes – iOS 7

- **UIImage has a method**
  - imageWithRenderingMode
- **Can be used to render an image as a template and apply the tint color**

```
cell.imageView.image=[[UIImage imageNamed:@"ReadingIcon2.png"]
imageWithRenderingMode:UIImageRenderingModeAlwaysTemplate];
```

- **Results**


184

## Autolayout – iOS 6

- **Defines your layout intention**
  - What you want to achieve
  - Not the coordinates of where a control should go
  - Autolayout determines the coordinates
- **Based on Constraints**
  - Place control 5pt from the bottom of the screen
  - Centre control horizontally
  - Make control a fixed width
  - Make these two controls a fixed width and set the left control to be a fixed distance from the left border
- **Constraints can have priorities**
  - One constraint can override another based on its priority
  - Autolayout works out how best to display it

185

Autolayout allows you to define the relationships between views rather than their absolute positions. That means the autolayout engine can determine at runtime the actual positioning of view based on the defined relationships. You specify intent rather than location.

This comes in at just the right time because iPhone 5 has a completely different screen size to all other iPhones, so dynamic positioning avoids the need to write complex code to detect device types so that dynamic view positioning can be applied.

Autolayout is based on Constraints that allow you to define simple facts such as:
This view should be 5pt from the left of the screen.
These two view should be spaced apart by 5pt.

Views can be pinned to fixed positions relative to the top, left, bottom or right of the screen. They can be aligned horizontally or vertically and resize in the same directions.

## Autolayout

- **Aligning**
  - Edges (left, right, top, bottom)
  - Centers (Horizontal, vertical)
  - Horizontal and Vertical alignment within container
- **Pinning**
  - Width, Height
    Horizontal/Vertical Spacing
    Leading space to superview
  - Trailing space to superview
  - Top/Bottom space to superview

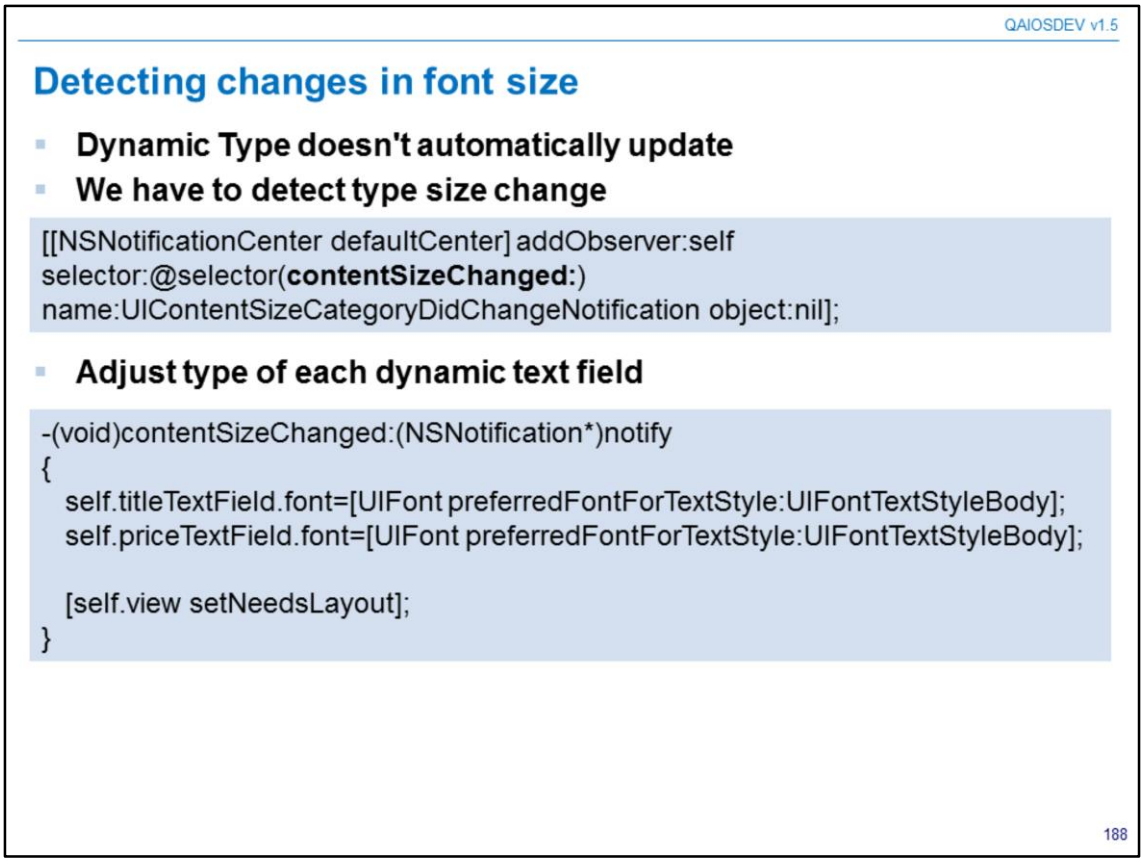


With aligning and pinning, you can ensure views are position relative to their superviews edges and to each other. They can also be made to automatically adjust to changes in the size in their container view cause by orientation changes.

## Dynamic Type

- **Typography has been improved in iOS7**
  - To ensure content is clear and distinct at all sizes
  - Optimized for clarity at all font sizes
- **Instead of choosing a font and size we can now choose a style**
  - Specify fonts semantically
- **User specifies size in iOS Settings**
- **Similar to styles in word**
  - Includes Headline , SubHead, Body, Caption1, Caption2, Footnote etc
  - Accessible from IB and code
- **Use in conjunction with AutoLayout**
  - As sizes change Auto Layout will cope with changes

187

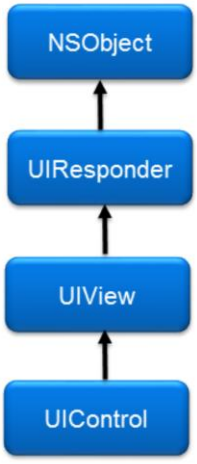



Dynamic text doesn't automatically update in the app and a change by the user in the settings app must be detected and each dynamic text field updated to reflect the change.

QAIOSDEV v1.5

## UI Controls

- **All controls inherit from UIControl**
- **Common set of properties**
  - Enabled – YES allows touches
  - Highlighted – set when a touch event enters
  - Selected – is the control in a selected state?
  - State – bitmask (enabled | highlighted | selected)
- **Uses Target Action pattern to raise events**
  - Host registers to receive control events
  - Control fires event when condition is met
  - Host uses **addTarget** to register



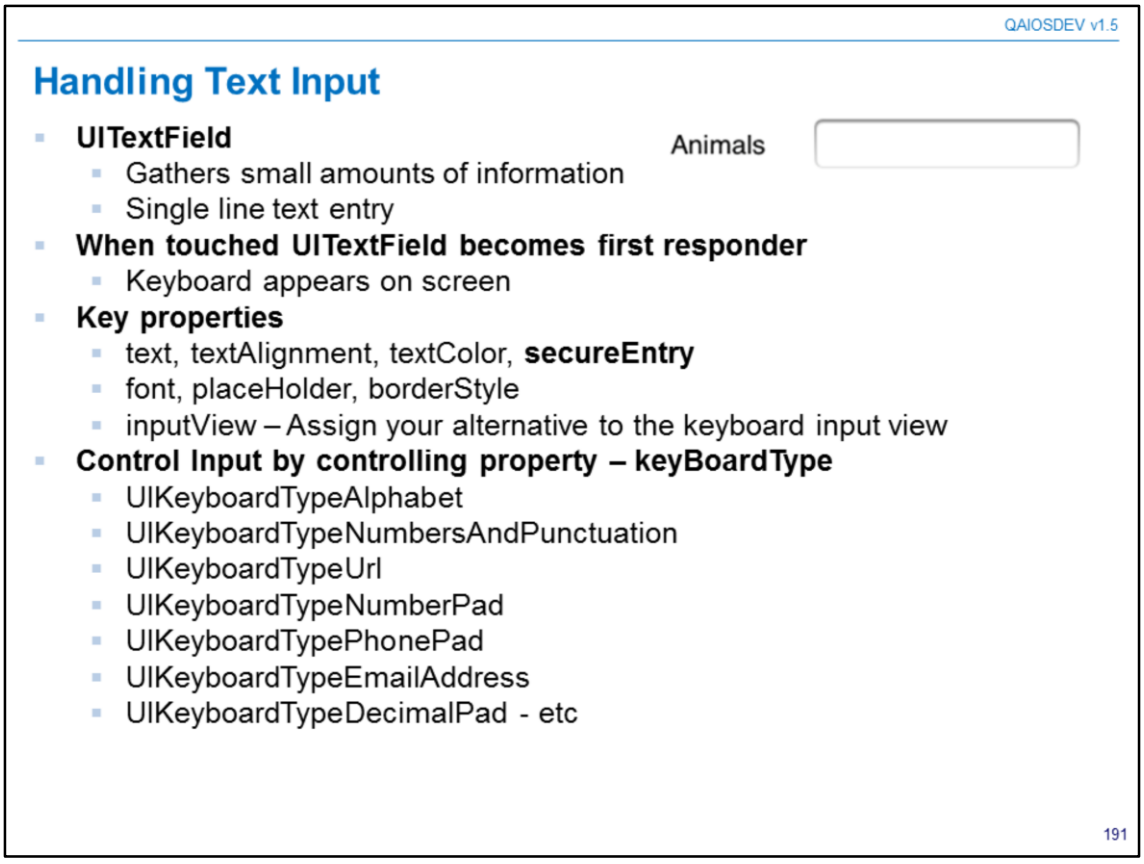

```
UIButton *btn=[self createButton];
[btn addTarget:self  action:@selector(btnPressed)
      forControlEvents:UIControlEventTouchUpInside]
```

189

User controls all inherit from the same base class UIControl. There's a simple reason for that, it provides the target action mechanism that allow a controls container to sign up for specific events.

UIControls also have a standard set of properties:

- Enabled determines if a control is enabled
- Highlighted indicates if a control is in the process of being touched. When the touch exits the highlighted state is reset
- Selected stores a control's selected status, which is relevant only to certain controls like switches/buttons etc. To all others the default value is FALSE.
- Tracking is TRUE if the controls is in the process of tracking touch events
- State is a bit mask made up of the following possible constants
  - UIControlStateNormal
  - UIControlStateHighlighted
  - UIControlStateDisabled
  - UIControlStateSelected
  - UIControlStateApplication
  - UIControlStateReserved




The primary text input field is UITextField that is designed to gather small amounts of data. It supports single line text entry.

When a text field is touched, it becomes the first responder and the keyboard appears on the screen.

The text field is very flexible and you can control its appearance in many ways including fonts, colours, borderStyles etc. The best way to design it is from within Interface Builder.
The placeholder property allows you to enter some hint text that displays inside the text field when it's empty allowing you to describe to the user what kind of data they should be entering.
If **secureEntry** is TRUE then the text is obscured ideal for password entry.

The inputView allows you to replace the Keyboard as the primary source of data entry with another type of view. Often you would use a date picker or picker view or even a custom designed view of your own.

If you are using the keyboard, you can restrict the entered data by managing the type of keyboard displayed.

## TextField Events

- UITextField supports UITextFieldDelegate protocol
- UITextFieldDelegate raises key editing events

| Event | Details |
|---|---|
| shouldChangeCharactersInRange | Return NO to disallow entered/pasted character(s) |
| textFieldDidBeginEditing | Fires when field becomes first responder |
| textFieldDidEndEditing | Fires when field resigns as first responder |
| textFieldShouldReturn | Fires when Return key pressed |
| textFieldShouldBeginEditing | Return YES to allow editing to start |
| textFieldShouldEndEditing | Return NO to stay on field |
| textFieldShouldClear | Return NO to avoid clearing the field |

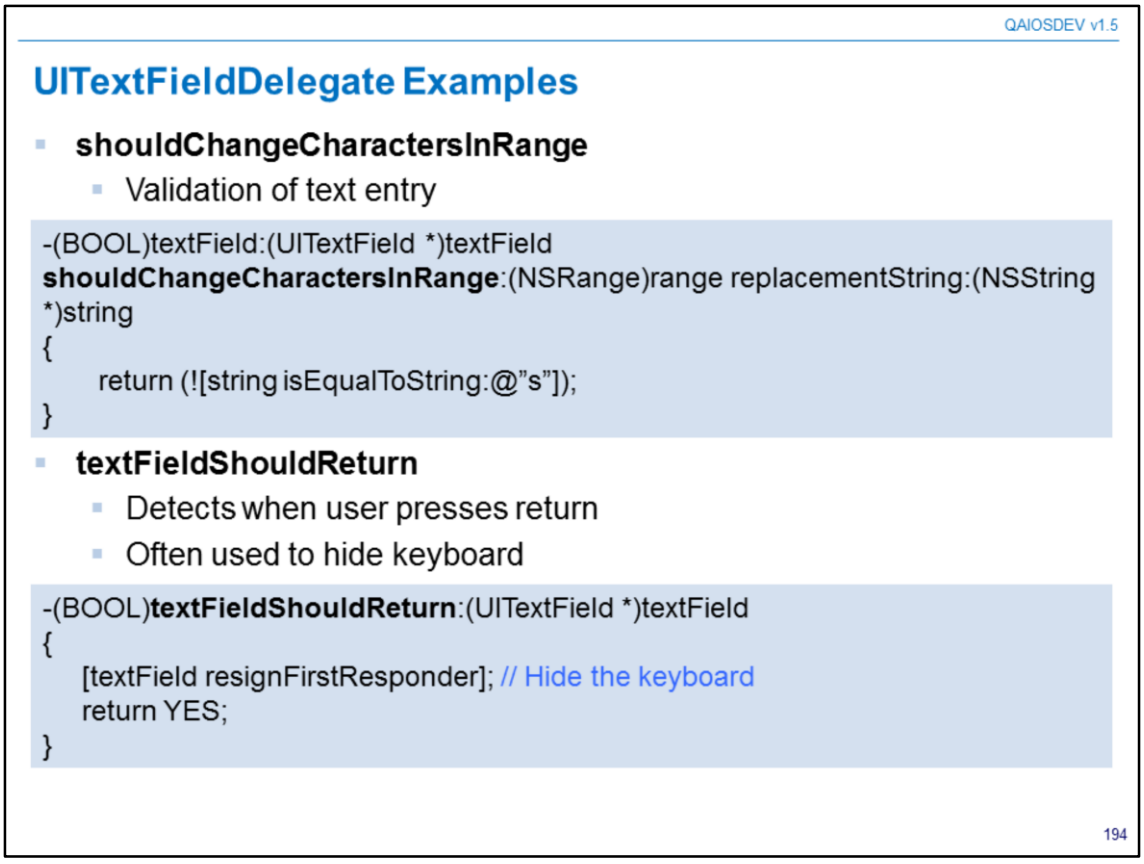
193

To detect key text field events text field supports the UITextFieldDelegate protocol. That means your view controller can implement the delegate and provide implementations for the above methods each of which detect a key editing event.

shouldChangeCharactersInRange is great for carrying out validation checking and max length processing because it allows you to return NO if you are unhappy with the text that has been entered in which case it doesn't make it onto the field.

You can also when a field is accessed (Begin Editing) and left (End Editing) and when the Return key is pressed. You can also prevent Editing or losing editing focus.

NSNotificationCenter can also be used to detect the following notifications UITextFieldTextDidBeginEditingNotification, UITextFieldTextDidChangeNotification and UITextFieldTextDidEndEditingNotification.




Above, are some examples of UITextFieldDelegate implementations.

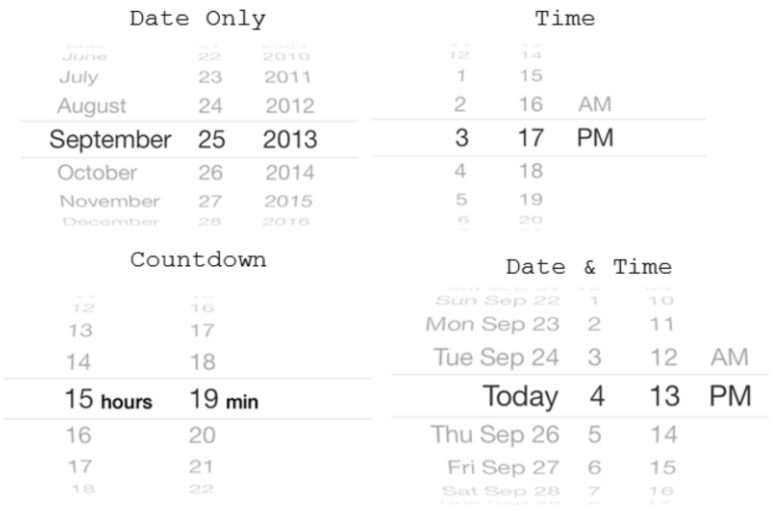
In the example above, **shouldChangeCharactersInRange** is used to validate text entry. The **string** ,which can be more than just single characters if a paste operation was performed, is the changed text not the entire field content. Equally, you could test the length of the entire field. The range parameter indicates the position and length of the changed characters in the entire field string.
Returning YES allows the text entry to be made. NO disallows it.

A more usual validation mechanism would be to use a regular expression.

```
NSError *error;
NSRegularExpression* regex = [NSRegularExpression
regularExpressionWithPattern:@"[,.£$]" options:0 error:&error];

if (error!=nil) return YES;
NSArray *matches=[regex matchesInString:string options:0
range:NSMakeRange(0, string.length)];
return matches.count==0;
```

## Dealing with Dates and Times

- **iOS has a unique date selection mechanism**
  - Date Picker Control – UIDatePicker






iOS has its own unique and iconic date selection mechanism. the date picker.

It can operate in four modes.

- Date Only
- Time Only
- Date and Time
- Countdown

Typically, a text field would be used as a date placeholder. When the user touches the field the Date Picker appears at the bottom of the screen.

## Working with the Date Picker

- **Choose the Date Picker Type**

```
self.datePicker=[[UIDatePicker alloc] initWithFrame:dateRect];

self.datePicker.datePickerMode=UIDatePickerModeDate;
```

- **Handle date selection event**

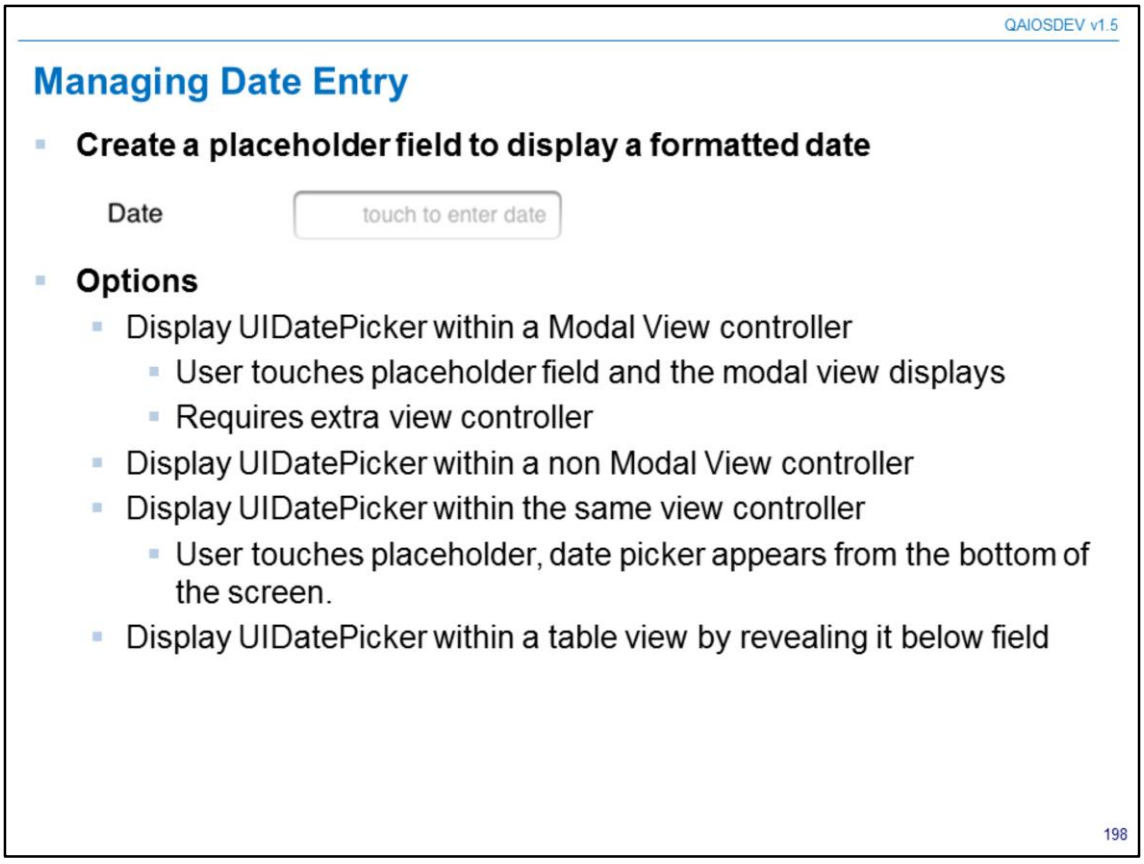
```
[self.datePicker addTarget:self action:@selector(dateChanged:)
forControlEvents:UIControlEventValueChanged];
```

```
-(void)dateChanged:(id)sender
{
    NSLog(@"%@",self.datePicker.date);
}
```

197

To use a Date Picker, you create an instance and set the picker mode. In the example above, we have chosen UIDatePickerModeDate.

So we can detect a change of date we have also implemented a **UIControlEventValueChanged** event that will fire the **dateChanged** method when the user selects a new date.




The standard date entry pattern for iOS apps is:

- A user touches a date field and either a modal or non modal view controller displays into which they can select a date using a date picker
- On selecting a date, they navigate back to the original view controller and the date field will have been updated

This works fine but has the disadvantage of taking the user out of the UI context they were in.

An alternative and increasingly common solution is to display the date picker at the bottom of the screen when the user touches the date field.

## Working with the Date Picker – Simple Solution

- **Text fields are used as date placeholders**
  - When user touches field a date picker should appear
- **Use inputView of UITextField to replace keyboard with Date Picker**

```
// Set the input view of the date text field
self.dateTextField.inputView=self.datePicker;
```

- **Disadvantage**
  - Cursor is active on the text field
- **Alternative**
  - Use an UIActionSheet to host the Date Picker (more complex!)
  - Use a Table View with a Date Cell (much more complex)

199

It is surprisingly easy to display a date picker as a popup on the screen. All you have to do is set the **inputView** property of the placeholder text field with a date picker as above.

You can even add a toolbar to the placeholders **inputAccessoryView** property if you want to add Done and Cancel buttons at the top of the date picker.

The only drawback to this approach is that you get a flashing cursor on the text field. You could code a workaround using superimposed labels during editing etc.

For most people, the actual solution is to use the rather more complex solution using an action sheet. There is a demo available that shows how to do this “DatePickerAdvanced”.

## Formatting Dates

- **Formatters are used in Objective-C to convert objects to and from strings**
- **NSDateFormatter formats dates**

```
-(void)dateChanged:(id)sender
{
    NSDateFormatter *format = [[NSDateFormatter alloc] init];
    [format setDateFormat:@"MMMM dd, yyyy"];
    self.dateTextField.text=[format stringFromDate:self.datePicker.date];
}
```



In order to display a date, it needs to be formatted first. To do that, we need to use a formatter used in Objective-C to convert certain objects like dates and numbers to strings. The NSDateFormatter does such a job.

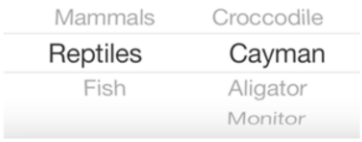
In the example above, we are first detecting when the date picker date selection changes and using that event to update the date display field with a formatted date.

The NSDateFormatter is created and the required date format string is set using method **setDateFormat.**
(The format strings are the standard ICU date format strings)

Y - Year
M - Month
d – day of month
h – Hour (1-2)
H - Hour (0-23)
m – Minute
s – Second

## Picker Views (UIPickerView)

- **Used to select an item from a list**
- **Made up of one or more wheels (components)**
  - Each wheel contains a separate list
- **Each wheel can modify the contents of another**
- **Supports UIPickerViewDelegate Protocol**




| Event | Details |
|---|---|
| numberOfComponentsInPickerView | Return the number of wheels to display |
| numberOfRowsInComponent | Return the number of rows for each wheel |
| titleForRow | Return the Text to display for a row |
| viewForRow | Return a View to display for a row |
| didSelectRow | Fired when a row is selected |



Picker views are used to provide the user with a selectable list of items. They can quickly scroll up and down the list.

Picker views are made up of one or more wheels internally referred to as components. Each wheel can be used as an independent list or one can be used to drive the contents of another.

They support the UIPickerViewDelegate protocol providing a set of methods that you can use to program the contents of each of the wheels.

## Managing a Picker View

```
NSArray * _array=[NSArray arrayWithObjects:@"One", @"Two",@"Three", nil];
....
-(NSInteger)numberOfComponentsInPickerView:(UIPickerView*)pickerView {
   return 1;
}
-(NSInteger)pickerView:(UIPickerView*)pickerView
numberOfRowsInComponent:(NSInteger)component{
   return _array.count;
}
-(NSString*)pickerView:(UIPickerView *)pickerView titleForRow:(NSInteger)row
forComponent:(NSInteger)component{
   return [_array objectAtIndex:row];
}
-(void)pickerView:(UIPickerView *)pickerView didSelectRow:(NSInteger)row
inComponent:(NSInteger)component{
   self.animalTextField.text=[_array objectAtIndex:row];
}
```

202

To set the number of wheels, you implement the **numberOfComponentsInPickerView** method and return the number of wheels.

Next, return the number of rows in each wheel by implementing **numberOfRowsInComponent**. This is called once per wheel and passes the index of the component.

In the example above, it's returning the count of objects from an array that will be displayed.

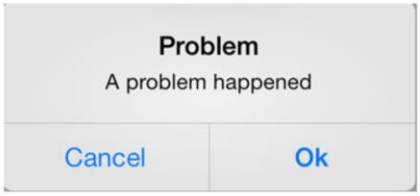
To display the contents of the array in the picker view, **titleForRow** is implemented for each row and component.

Finally, when an item in the picker view is selected **didSelectRow** is called and we just update the contents of a label to reflect the selection.

QAIOSDEV v1.5

## Displaying Alerts

- **Alerts and Action sheets present on top of the current view controller**
- **Alert Views present alert messages**
  - Warnings
  - Errors
  - Information
- **Supports UIAlertViewDelegate protocol**
  - alertView:clickedButtonAtIndex
- **Example**




```
UIAlertView *alert= [[UIAlertView alloc] initWithTitle:@"Transfer Complete"
            message:@"Thanks for submitting your meter reading"
            delegate:nil
            cancelButtonTitle:@"Done"
            otherButtonTitles:nil, nil];
[alert show];
```

203

Sometimes you may need to display and alert message or some information to the user in the form of a popup message. The Alert View is an ideal solution for that.

With it, you can create a popup message with a title a sub message and a set of buttons.  You can also add sub views to an Alert View.

The buttons include a cancel button and zero or more other buttons defined by a comma separated list of button captions terminated with a null.

In the code below, three buttons will be displayed Cancel, Ok and Not Ok. The pressed button can be detected using delegate method clickedButtonAtIndex.

UIAlertView *prompt = [[UIAlertView alloc] initWithTitle:@"Transfer Complete"
message:@"Thanks for submitting your meter reading"

delegate:self

cancelButtonTitle:@"Cancel"

otherButtonTitles:@"Ok",@"Not Ok", nil];

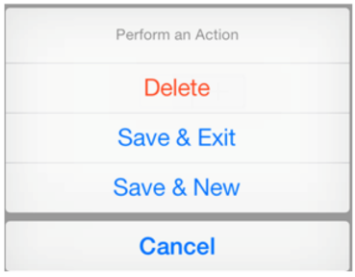
Handle button clicks via delegate UIAlertViewDelegate and method

-(void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex.

QAIOSDEV v1.5

## Requesting Responses

- **Action Sheets ask a user to choose from a number of actions**
  - UIActionSheet
  - Multiple action buttons can be presented
  - Supports UIActionSheetDelegate protocol
  - Can be presented from
    - Tool bar
    - Tab bar
    - View




```
UIActionSheet * sheet=[[UIActionSheet alloc] initWithTitle:@"Perform an Action"
delegate:nil cancelButtonTitle:@"Cancel"      destructiveButtonTitle:@"Delete"
otherButtonTitles:@"Save & Exit",@"Save & New",nil];
[sheet showInView:self.view];
```

- **Action Sheets can have sub views added to them**

205

Action sheets are similar to alert views except they also have a destructive button (red delete style button) and can be presented from the tab bar, tool bar or other view from which they animate their appearance.

Just like with Alert Views, you can control the extra buttons displayed in **otherButtonTitles** and can determine button clicks by implementing the UIActionSheetDelegate method **actionSheet:clickedButtonIndex.**

By passing nil to all of the button titles and then adding your own sub views, Action Sheets can be used as a popup manager platform. This is widely used for managing DatePicker and Picker Views.

QAIOSDEV v1.5

## Other Controls

- **UISwitch**
  - Captures Boolean Conditions
  - **isOn** indicates state
  - Fires event UIControlEventValueChanged (target/action)
- **UIButton – Standard Button**
  - Types
    - System
    - Info Light
    - Info Dark
    - Detail Disclosure
    - Add Contact
    - Custom
  - Set title, image, background, font and text color properties for states
    - Default
    - Highlighted (Pressed)
    - Selected
    - Disabled
  - Handle Event UIControlEventTouchUpInside for click event

Ok

206

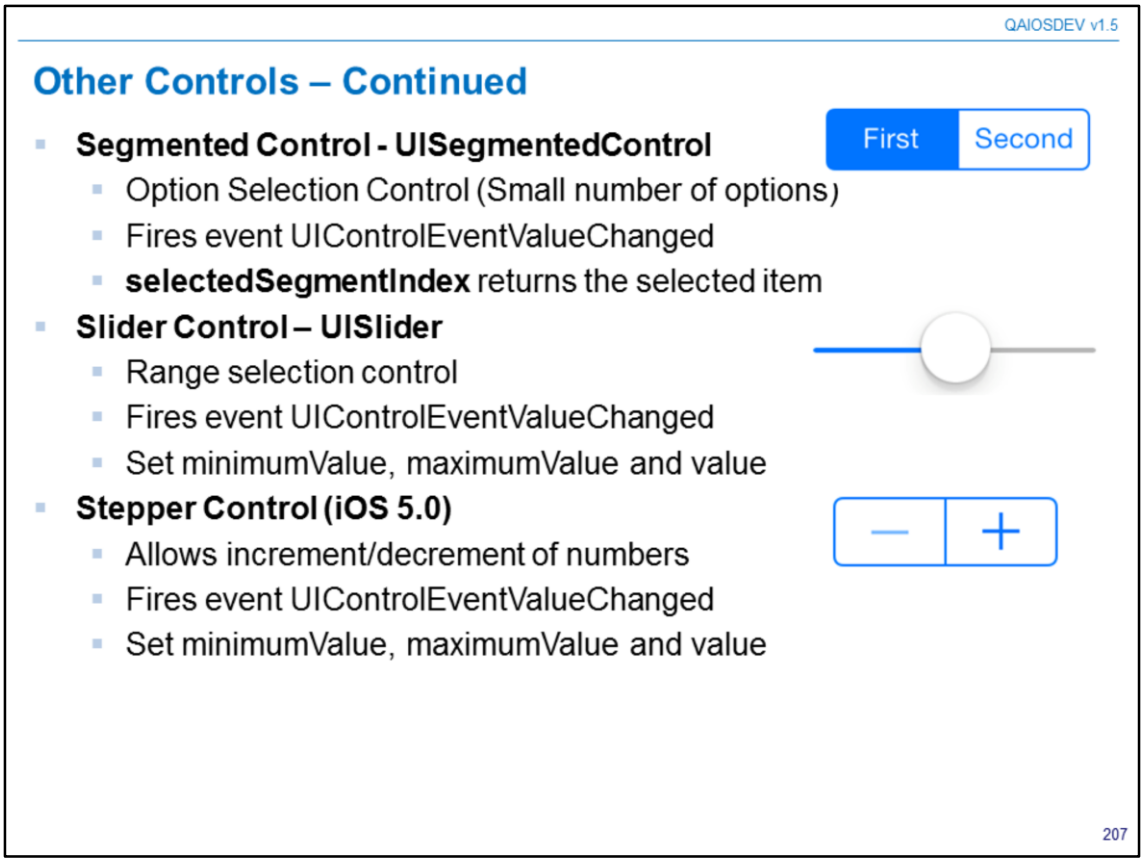
The switch control is a common sight in iOS apps and is used to capture boolean conditions.

The UIButton is also a common control and, as you can see above, there are a number types of button you can create.

The custom button type gives you the flexibility to create your own type of button from scratch using custom images. The easiest way to do this is from within Interface Builder where you can set the look and feel of the button for each of the following states:

- Default
- Highlighted
- Selected
- Disabled

To detect the click event, you handle the UIControlTocuUpInside event.




§The segmented controls is a button like control that allows the user to select from a small number of options. It is the fastest way for a user to make a choice. When they select an option it becomes highlighted.

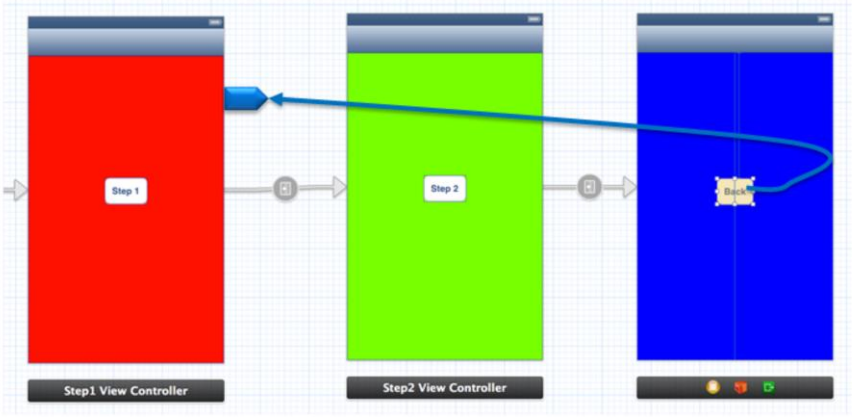
The slider control is ideal if you want the user to set a value based on a range of values. It fires event UIControlEventValueChanged and you can set the min and max values as well as the starting value.

The stepper control gives and similar increment/decrement capability.

QAIOSDEV v1.5

## Unwind Segues

- **How do you navigate backwards in a storyboard?**
  - **popViewController** on navigation controller
- **Unwind Segues allow you to**
  - Define unwind points in your view controller flow
  - Navigate back to that point using an unwind segue




208

Unwind segues allow you to navigate backwards in a storyboard flow to a previous step. Prior to iOS 6 this wasn't possible without writing navigation controller code directly.

In iOS 6, you can define specific unwind segue points by adding methods with a specific signature to your view controllers:

```
-(IBAction)<MethodName>:(UIStoryboardSegue*)sb
```

It's then a simple matter of CTRL dragging from buttons etc down to the new green unwind icon of the scene control bar from where a popup list of all unwind segue methods will be displayed. Just choose the relevant one.

QAIOSDEV v1.5

## Unwind Segue – Unwind Points

- **Unwind Points are defined using a specific method format**

```
-(IBAction)<MethodName>:(UIStoryboardSegue*)sb
```

- **Add to view controllers to act as unwind points**
  - Step 3 unwinds to step 1
    - Add an unwind method to step1 view controller

```
-(IBAction)step1VC_UnwindPoint:(UIStoryboardSegue *)segue {

        NSLog(@"Unwound here");
}
```

- **Setup Segue in Interface builder**
  - Press CTRL and drag from UI control to View Controller's Green Icon 
  - Choose Unwind method names from list (populated from all VCs)

209

Above, is an example of an unwind method defined within a view controller. A view controller further along in the segue sequence could use this method as an unwind point. When the unwind is executed the unwind method step1VC_UnwindPoint would fire and the storyboard would be unwound to its containing view controller.

Notice it takes a UIStoryboardSegue parameter, which means you have access to the source and destination view controller.

## Gesture Recognizers

- **Detect when different gestures occur on the view**
  - Tap
  - Pinch
  - Rotation
  - Swipe
  - Pan,
  - Long Press
- **Integrated into Interface Builder – Makes it easy to use**
  - Configure properties
  - Create segues
  - Create handler methods
- **Supports Multi touch**
  - Set the number of taps/touches
- **You can still write multi touch handlers in code**

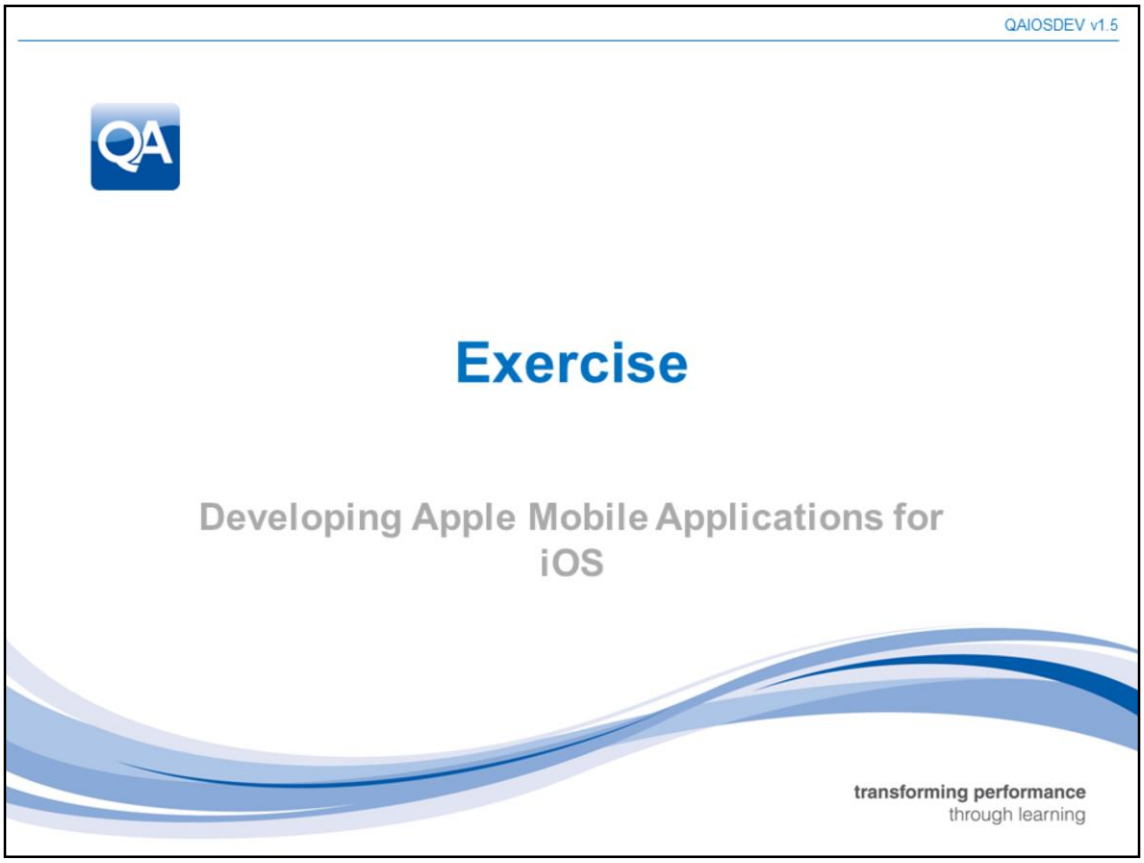


Detecting touches, multiple touches and gestures used to be a difficult programming task but with the built in gesture recognizer classes within Interface Builder you can quickly support these features using drag & drop.

Simply drag the gesture you want to support on to the view controller's scene bar and configure its properties. Properties include such things as the number of taps or touches required.

Within Interface Builder a gesture can be configured to fire an event by calling a specific method.

You can still write code to support touch events like **touchesBegan, touchesMoved, touchesEnded** and **tocuhesCancelled.**


QAIOSDEV v1.5
Exercise
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Summary

- **tintColor and Dynamic Font used to aid clarity of content**
- **AutoLayout allows UI layout to respond to UI changes dynamically**
- **Controls inherit from UIControl**
  - Target Action mechanism
- **Common Controls include**
  - UITextField captures single line text
  - Date Picker (UIDatePicker) - choosing dates
  - Picker View (UIPickerView) – choosing anything else
  - Switch (UISwitch) – Boolean questions
  - Slider (UISlider) – Captures a value with a range
  - Stepper (UIStepper) – Increment/Decrement of a number
- **UIAlertView - Displays alerts and info**
- **UIActionSheet - Presents user with a set of actions to choose from**
- **Unwind Segues**
- **Resources**
  - http://www.cocoacontrols.com - Control repository

## UI Responders

- **UI Controls must respond to user interactions (events)**
  - touch
  - motion
  - action (from target action events eg button clicks)
- **Views, Windows, View Controllers and Controls are all responders**
  - Inherit from UIResponder
- **App determines the responder to handle an event (first responder)**
  - Responder must return YES from **canBecomeFirstResponder** to be considered
- **Events are routed to a First Responder**
  - Touches routed to the view directly overhead
  - Keyboard touches routed to the most recently touched text entry field
- **UIResponder handles key interaction events**
  - touchesBegan, touchesEnded, touchesMoved, touchesCancelled
  - motionBegan, motionEnded, motionCancelled
- **Controls can resign as first responder - resignFirstResponder**

213

UI Controls have to respond to user interaction otherwise they are pointless. To do that they inherit from UIResponder, which is the main class that handles the key interaction action events for both touch and motion.

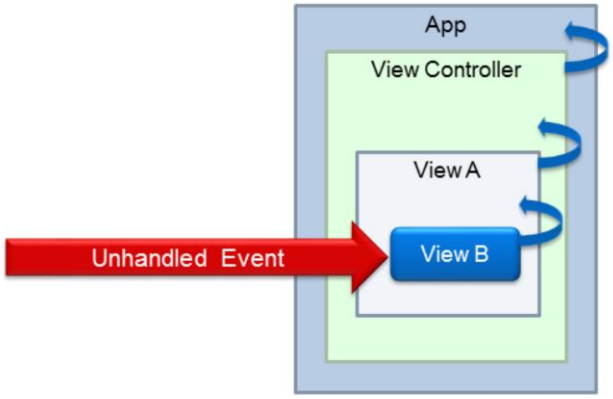
When an event occurs, the app has to decide which object is going to handle the event first, “first responder”. To do that, it considers the type of event and its context. For a touch event, the first responder will be the control or view directly over the touch.

To be considered as a first responder, a control must return TRUE/YES to **canBecomFirstResponder**.

When a control has finished responding, it can resign as first responder by calling **resignFirstResponder**. This is particularly relevant for controls like text fields that reveal the keyboard when active as first responder. To hide the keyboard again they just need to resign as first responder.

## Responder Chains

- **What happens if a control can't handle an event?**
  - Exception?
  - Is it ignored?
- **Event Chaining**
  - Unhandled events are transferred up through the hierarchy






If a control can't handle an event, it is passed up the hierarchy until a suitable handler is found. If no handler is found, the event is ignored. This is called event chaining.

## Manual Gesture Recognizers

- **Creating UIGestureRecognizers in code**

```
UITapGestureRecognizer * recognizer = [[UITapGestureRecognizer alloc]
initWithTarget:self action:@selector(handleTap:)];

recognizer.numberOfTapsRequired=2;

[self.footBallImage addGestureRecognizer:recognizer];

- (void)handleTap:(UITapGestureRecognizer *)recognizer {

    self.footballImageView.hidden=YES;

}
```

215

You can create gesture recognizers in code as the slide demonstrates. Just create an instance of the required recognizer. The following gesture recognizers are available:

- UITapGestureRecognizer
- UIPinchGestureRecognizer
- UIRotationGestureRecognizer
- UISwipeGestureRecognizer
- UIPanGestureRecognizer
- UILongPressGestureRecognizer

Next, provide a selector method to handle the gesture event.

Finally, you just need to add the recognizer to the view of interest.

## Managing Styles

- **iOS 5.0 introduced UIAppearance**
  - Controls the global styling and appearance of UI elements
  - Retrieves a proxy for all class instances

```
[[UITabBarItem appearance] setTintColor:[UIColor blackColor]];
```

- **Manage styling based on a control's container**
  - Using method **appearanceWhenContainedIn**

```
[[UITabBarItem appearanceWhenContainedIn:[MyClassA class], [MyClassB class], nil] setTintColor:[UIColor blackColor]];
```

- **UI Controls implementing UIAppearanceProtocol can be controlled**
  - **Properties must be decorated with** UI_APPEARANCE_SELECTOR

```
@property(nonatomic,strong) UIColor *tintColor UI_APPEARANCE_SELECTOR;
```



Managing multiple styles within an iOS app has always been a difficult thing to achieve. This was mainly because everything had to be done at the instance level. There was no way to create a global style.

In iOS 5.0, the UIAppearance protocol class was introduced allowing you to control the appearance of any UI property decorated with the UI_APPEARANCE_SELECTOR definition. (Just look at a classes header file to determine if this is attached).

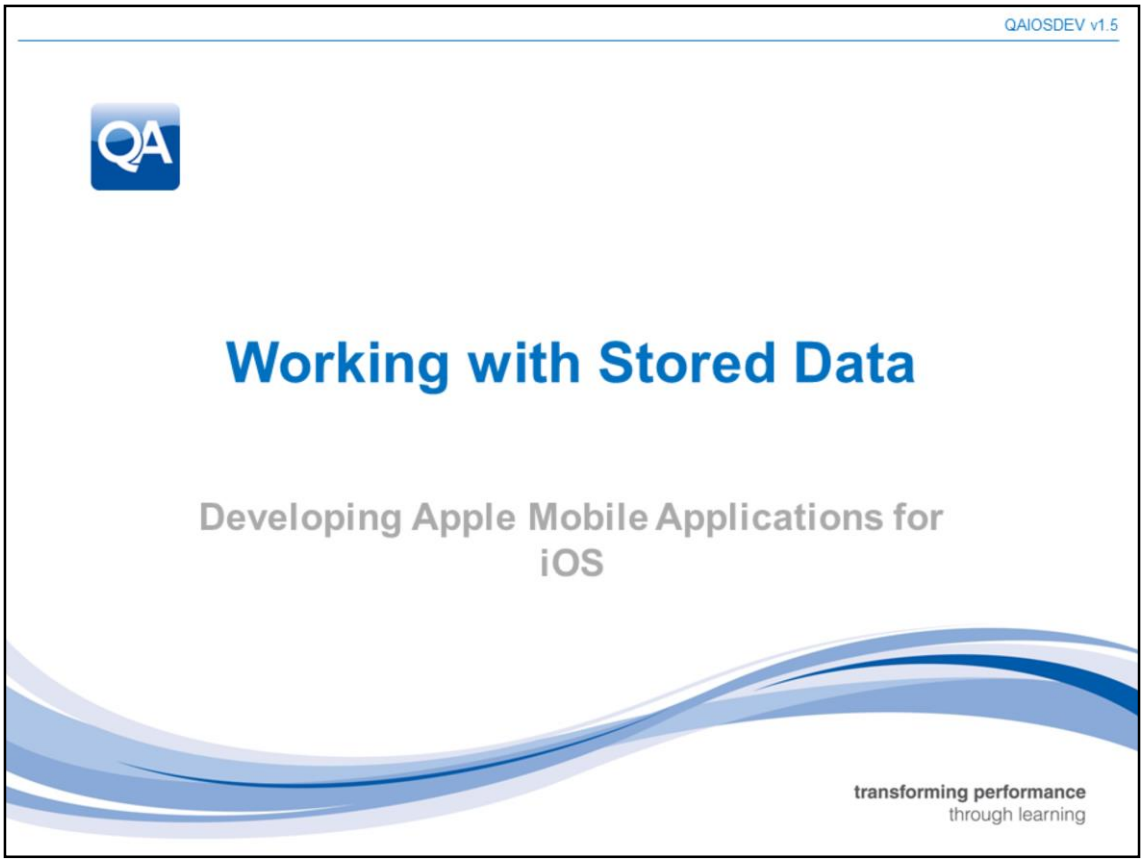
Using the **appearance** method to retrieve a proxy that is then used to send **set** messages to all instances of the targeted class.

In the example below, the appearance proxy for the UITabBarItem class is retrieved. All instance of UITabBarItem have subscribed to this appearance proxy.

[[UITabBarItem **appearance**] setTintColor:[UIColor blackColor]];

Next the setTintColor method is called using the proxy. This is called on all instances of UITabBarItem.

```
[[UITabBarItem appearance] setTintColor:[UIColor blackColor]];
```


QAIOSDEV v1.5
Working with Stored Data
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - To learn how to store local data
- **Contents**
  - Overview
  - iOS File System
  - Defaults Database
  - Key Chain Storage
- **Hands on Labs**



This chapter is all about extending the user interface using the built in controls within UIKit.

## Overview

- **Many apps need to store and retrieve data on device**
- **iOS has four main options**
  - User Defaults database
    - Key Value pairs
  - pLists
    - Key value pairs accessed via NSDictionary
    - Ideally store small volumes of data
  - SQLite
    - SQL DBMS (SQLite dialect)
  - Core Data (Next chapter)
    - Object graph storage with multiple storage options
      - In memory – No backing store
      - SQLite – Uses SQLite as a backing store
      - Atomic – Your own custom storage mechanism
    - iCloud

## iOS File System and the Sandbox

- **Apps run within their own "Sandbox"**
  - Apps have limited access to file system
  - Apps can't access each others Sandbox
- **At installation**
  - App directory is created (Sandbox) and app is copied in
  - **Documents** directory is created - Critical data backed up by iTunes
  - **Library** is created – Non user data files such as databases, caches, etc. Backed up by iTunes
  - **tmp** is created as a temporary storage area for the app. Not backed up.




When an app is installed, it is placed inside its own base directory that becomes its own private domain called a "Sandbox". Apps have access to files and directories within their own sandbox in which they can create, update and delete files and folders.

Apps also have access to public interfaces such as the Address Book and Music but no access to each others sandboxes.

At installation, the app is placed in the base directory and three sub directories are created:

- Documents – should be used by the app to store data generated by the user like reports etc. It is backed up by iTunes and the contents can be made available via iTunes file sharing
- Library – Used to store application specific data like SQLite databases, downloaded images, caches, object archives, application support data etc. Also backed up by iTunes backup
- tmp – Temporary storage area that is not backed up, should be used to generate short lived files that are periodically deleted

QAIOSDEV v1.5

## The App Bundle

- **An iOS app is actually a bundle of files containing**
  - App executable
  - Resources
  - NIB/XIB Files
  - Icons
  - Launch Images
  - Database files
  - Property Lists
  - Info.plist - Config info


- **Certain resources have to be transferred to the Sandbox before use**
  - e.g. SQLite Databases
- **Bundle file paths**

```
NSString *path = [[NSBundle mainBundle] pathForResource:@"Customers"
ofType:@"sqlite"];
```

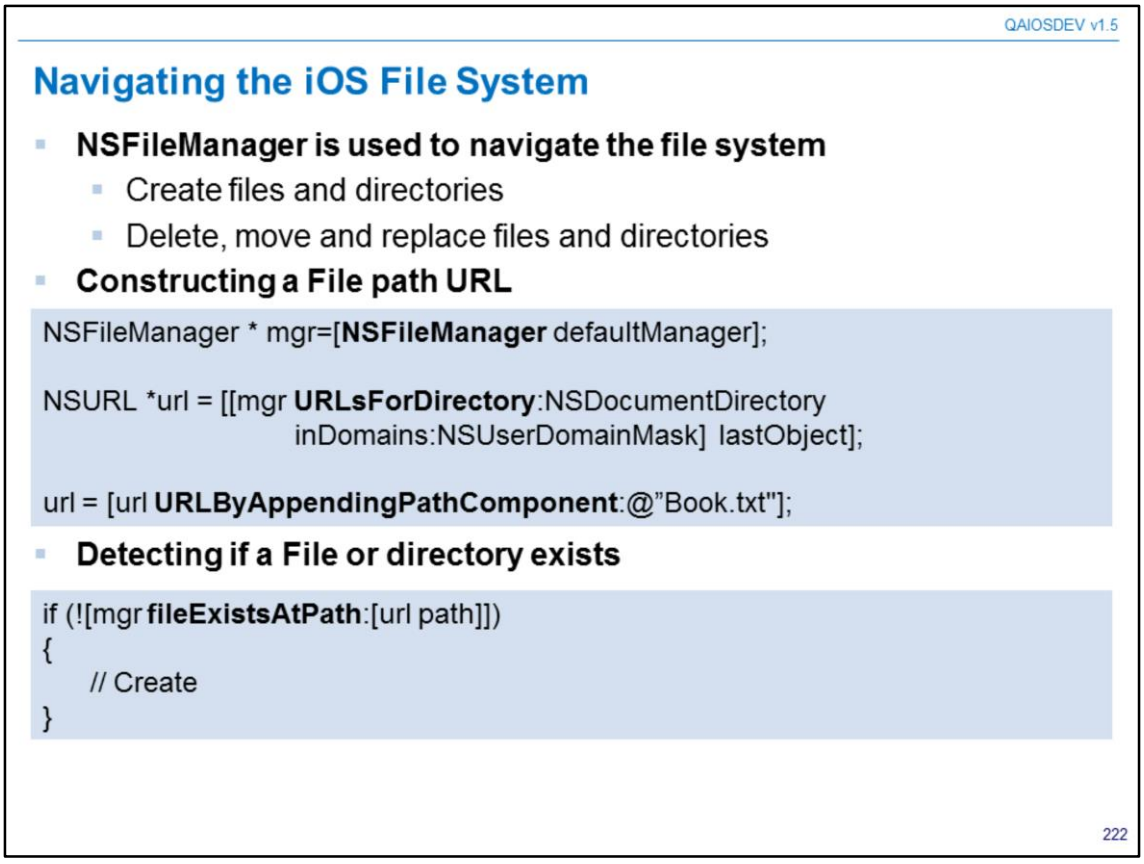
221

iOS apps are actually bundles of files containing not only the executable but also other resources including icons, nib files launch images, database files etc.

Files like SQLite databases need to be transferred from the bundle to the app's home directory before they can be accessed by the app. This is usually one of the first tasks when an app runs for the first time.

To access the path of a file within a bundle, the NSBundle class is used together with the mainBundle method to retrieve the current apps main bundle. Calling **pathforResource** and passing the name of the resource and its type will return its file path within the bundle.

This path can be used together with NSFileManager to transfer a bundle file to a specific file system path (using NSFileManager's copyItemAtPath method)




To navigate through the iOS file system you use NSFileManger. It does everything from checking if files and directories exist to creating, deleting, moving and replacing them.

Probably, the first thing you need to do is construct file paths to specific locations within the app's sandbox. The preferred way to do this is using URLs. You can use local UNIX file paths but with the advent of iCloud it's more flexible to get used to using URLs from the start.

Using the shared NSFilemanager defaultManager, you can retrieve the URL for a specific directory type by calling **URLsForDirectory** and passing a search path enum for directories including:

- NSDocumentDirectory
- NSLibraryDirectory – SQLite Databases, cached data etc
- NSCachesDirectory – Cached Info, downloaded graphics etc

Once you have a URL, you can append paths to it to create extended file paths using method call **URLByAppendingPathComponent**.

## Creating and Deleting Folders and Files

- **Creating a Folder**

```
NSError *err;
BOOL ok=[mgr createDirectoryAtURL:booksURL  withIntermediateDirectories:YES
                 attributes:nil error:&err];
```

- **Deleting a folder/file**

```
[mgr removeItemAtURL:booksURL  error:&err];
```

- **Creating a File**

```
NSData *data=[@"Hello world" dataUsingEncoding:NSUTF8StringEncoding];
[mgr createFileAtPath:[bookURL path] contents:data attributes:nil];
```

- **Reading a File**

```
NSData *data=[mgr contentsAtPath:[bookPath path]];
NSString *str=[[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];
```

224

NSFileManager can also create and delete files and folders.

To create a folder, simply call **createDirectoryAtURL** passing it a URL and YES to the withIntermediateDirectories parameter if you want it to also create any intermediate directories in the path that don't already exist.
The attributes parameter is a dictionary in which you can set ownership and permissions. Passing nil sets the default options for the current user process, which for iOS apps is fine.

**removeItemAtURL** can be used to delete a folder or file.

**createFileAtPath** creates a file using the contents of an NSData object. In the example we have created, an NSData object from a string. (NSData is just a useful byte container). Again we are setting the attributes to nil.
NSData also has file manipulation methods including **writeToUrl** that can be used to write additional data.

**contentsAtPath** simply reads data in from a file into an NSData object. From here we are converting the NSData object to a string.

## File Security

- **Files can be set to be encrypted on disk until device is unlocked**
  - Set on file creation

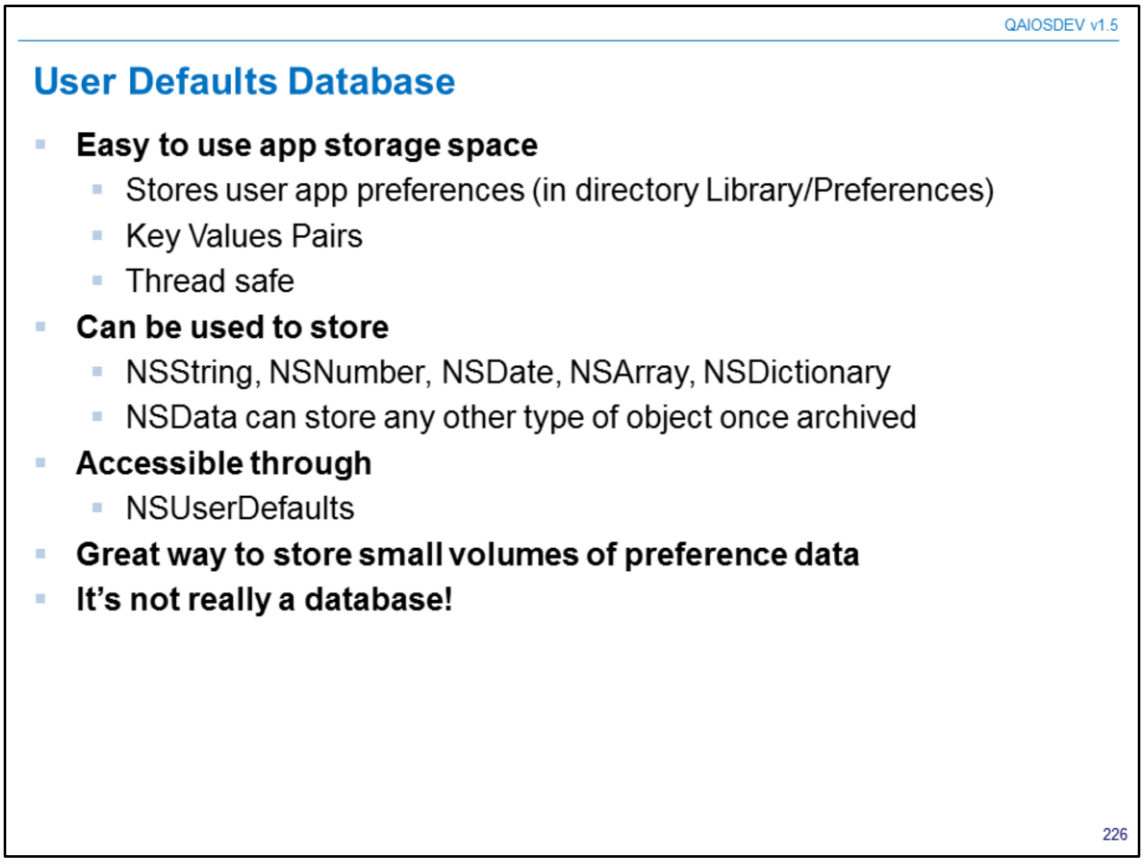
```
[[NSFileManager defaultManager] createFileAtPath:[self filePath]
        contents:[@"super secret file contents"
                        dataUsingEncoding:NSUTF8StringEncoding]
        attributes:[NSDictionary dictionaryWithObject:NSFileProtectionComplete
                            forKey:NSFileProtectionKey]];
```

  - After Creation

```
NSError *err;
NSDictionary *secAttrs = [self attributesOfItemAtPath:self.filePath error:&err];
if(![secAttrs ["NSFileProtectionKey"] isEqual:NSFileProtectionComplete])
{
      secAttrs = @{NSFileProtectionKey:NSFileProtectionComplete};
      BOOL success = [self setAttributes:secAttrs
                                  ofItemAtPath:self.filePatherror:&err];
}
```

225

Files can be set to encrypted on disk until the device is unlocked with the passcode. You can also setup security for the app using entitlements.




Working with the file system directly can be a bit of an overhead especially if you just want to store some basic data.

The user defaults database is a great place to store app specific preference data. It's easy to use and simply stores key value pairs of the following types:

- NSString
- NSNumber
- NSDate
- NSArray
- NSDictionary
- NSData

The data is actually stored in the Library/Preferences folder.

It's not a database, however, so you shouldn't store large volumes of data here because performance will suffer.

## NSUserDefaults in action

- **Retrieving items from User Defaults**
  - Call **standardUserDefaults** to retrieve NSUserDefaults
  - Access items using:
    - **objectForKey, stringForKey, doubleForKey …..**
    - **arrayForKey, stringArrayForKey, dictionaryForKey …..**

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

NSString * firstName=[defaults stringForKey:@"firstName"];
```

- **Create or update new key/values**
  - **setObject:forKey:, setBool:forKey, setDouble:forKey….**

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

[defaults setObject:@"Bloggs" forKey:@"lastName"];

[defaults synchronize];
```

227

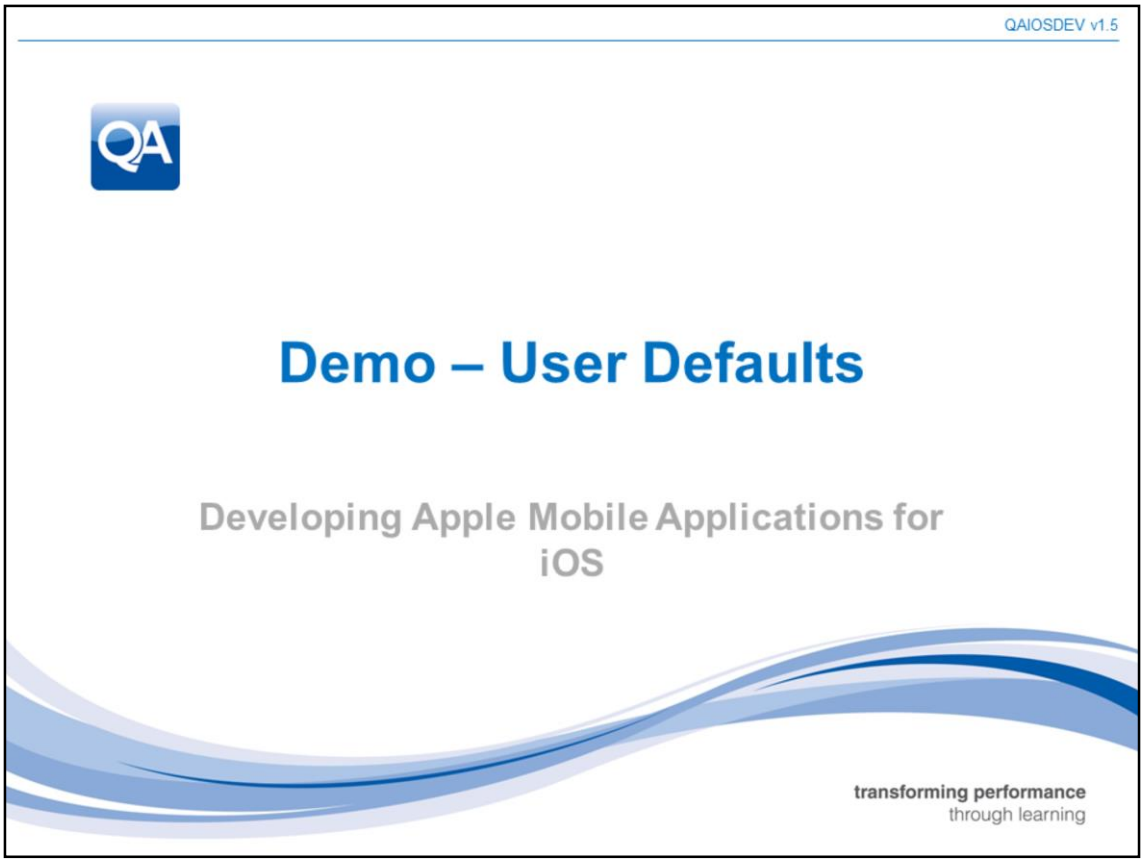
To access the user defaults database, you use NSUserDefaults by calling the class method standardUserDefaults to return the shared defaults object.

From here, you can set and retrieve values using the various setters and getters for each of the allowed types.

Setters include setBool:forKey, setFloat:forKey, setDouble:forKey, setObject:forKey, setInteger:forKey and setURL:forKey. You supply the value you want to create or update followed by the key you will use to identify it.

To retrieve a stored key value, use one of the getter methods that take the key name as a parameter and include:

- stringForKey
- arrayForKey
- boolForKey
- dictionaryForKey
- floatForKey


QAIOSDEV v1.5
Demo – User Defaults
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Secure Key/Value Storage of

- **Key Chain is a secure database for encrypted key/values**
  - Uses Low level C-API
- **Requires Security.framework**
  - #import <Security/Security.h>
- **Adding unique items to the key chain**

```
NSData* secret = [@"some data" dataUsingEncoding:NSUTF8StringEncoding];
NSDictionary* dict = @{
  (__bridge id)kSecClass: (__bridge id)kSecClassGenericPassword,
  (__bridge id)kSecAttrService: @"com.qa.myService",
  (__bridge id)kSecAttrAccount: @"some service",
  (__bridge id)kSecValueData: secret };

OSStatus status = SecItemAdd((__bridge CFDictionaryRef)dict, NULL);
```

230

The key chain is an encrypted storage database to which apps can securely store small pieces of data. Provides a low level C api.

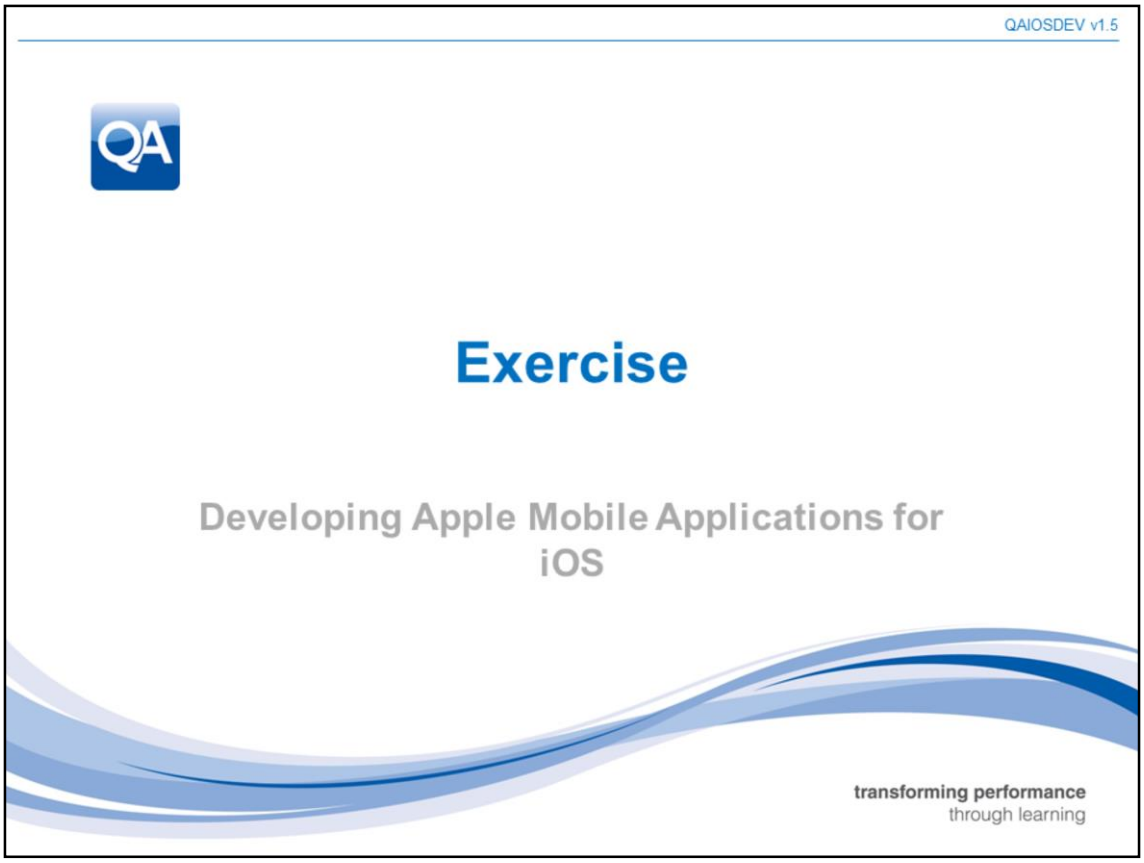
## Retrieving Items from the Key Chain

- **Use SecItemCopyMatching to query the key chain**

```
NSDictionary* query = @{
  (__bridge id)kSecClass: (__bridge id)kSecClassGenericPassword,
  (__bridge id)kSecAttrService: @"com.qa.myService",
  (__bridge id)kSecAttrAccount: @"some service",
  (__bridge id)kSecReturnData:@YES
 };

CFTypeRef inTypeRef;
OSStatus status = SecItemCopyMatching((__bridge CFDictionaryRef)query,
&inTypeRef);

NSData *data=(__bridge NSData *)(inTypeRef);

NSString *secret=[[NSString alloc] initWithData:data
                                        encoding:NSUTF8StringEncoding];
CFRelease(inTypeRef);
```

## Additional Key Chain calls

- **To modify an item you must use**
  - SecItemUpdate
  - Re-Adding will fail
- **To delete use**
  - SecItemDelete
- **Always check the status code – Zero is Good!**
- **Keychains are app specific and accessible when device is unlocked**
  - Can configure app to share keychain across app from same teams
  - See Keychain Sharing in App Entitlements


QAIOSDEV v1.5
Exercise
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## SQLite Data Storage

- **Relational database management system**
  - SQLite 3
  - Transactional (ACID transactions)
  - SQLite SQL Syntax – http://www.sqlite.org
- **Ideal when a full relational, transactional database is required**
- **Doesn't directly support iCloud synchronisation**
- **Requires access via low level C API**
- **Third party wrappers available**
  - FMDB - https://github.com/ccgus/fmdb
  - iActiveRecord - https://github.com/AlexDenisov/iActiveRecord
- **Requirements**
  - You have to deploy the database to the Documents directory
  - App updates must also apply any database updates via database definition language (DDL) to avoid user data loss



SQLite 3 is a full RDBMS giving you access to a lightweight fast and powerful SQL database. SQLite supports transactions and is ACID (Atomicity, Consistency, Isolation, Durability) compliant.

SQLite has its own SQL dialect that is a subset of standard ANSI SQL. See http://sqlite.org for more info.

When you need a fully relational, SQL based database.

To access it, you need to use a low level C API although there are a number of libraries available that wrapper the api and make life a lot easier.

Alternatively, it's a good idea to write your own wrapper.

SQLite databases need to be included in the app bundle and deployed to the app Library folder on first run.

Also, when you alter the database schema, you will need to add code to your

app that updates the database schema of existing users.

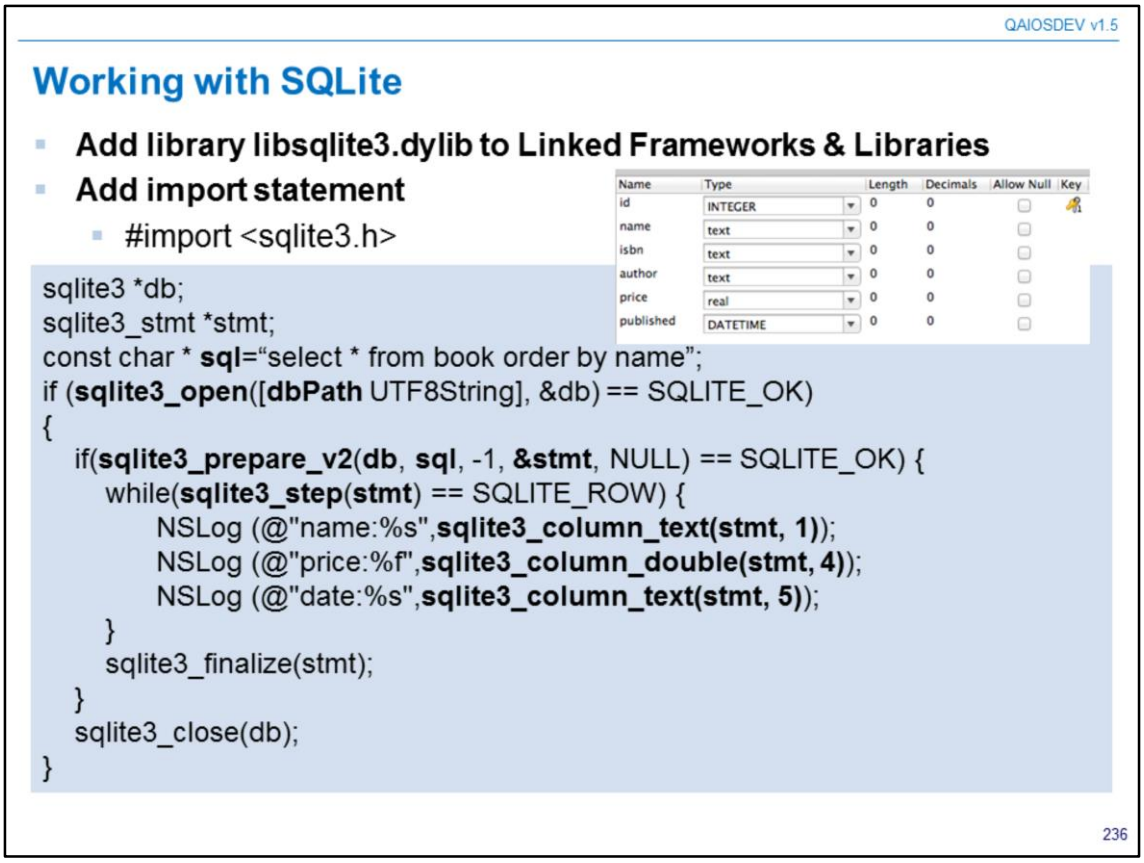
## Creating and Managing SQLite Databases

- **Xcode doesn't have a SQLite Editor**
  - Create Database manually using DDL?
    - create table..
    - create index..
    - create view..
    - insert into…
- **Third Party Managers are available**
  - Firefox Addons – SQLite Manager
  - Mac App Store Apps
    - NaviCat for SQLite
    - SQLiteManager



Xcode doesn't have a SQLite manager, so you will need to either write code to generate the database using SQL data definition language or use a third party tool.

There are plenty of tools out there. Firefox has an add-on called SQLite Manager that works well. There are also some good app store apps most notable Navicat and SQLite Manager.




To access the SQLIte library, you need to add it to your project. To do that in Xcode select your project icon in the solution explorer and once the Target Summary pane has displayed scroll down to the Linked Frameworks and Libraries section.
It's now a simple matter of adding the C library **libsqlite3.dylib**

To use the SQLite functions in your code import file:

**#import <sqlite3.h>**

Because we are dealing with a C library its functions are expecting C parameters so no NSStrings etc.

There are two important objects to work with, the **sqlite3** object that represents a database connection and a **sqlite3_stmt** object which is a SQL statement.

In common with most database libraries, the first step is to open a connection to the database. Function **sqlite3_open** achieves this taking a SQLite database file path as a parameter and a **sqlite3** connection instance.
Once the connection has opened (SQLITE_OK returned), the SQL statement we want to execute can be prepared.

## SQLite Data Manipulation

- **Uses SQL Templates**
  - Create a SQL Template for insert, update or delete
  - Bind columns to template

```
const char * sql="insert into book (name, price) values (?,?)";
if (sqlite3_open([dbPath UTF8String], &db) == SQLITE_OK)
{
  if(sqlite3_prepare_v2(db, sql, -1, &stmt, NULL) == SQLITE_OK)
  {
    sqlite3_bind_text(stmt, 1, "New Book", -1, SQLITE_TRANSIENT);
    sqlite3_bind_double(stmt, 2, 25.99);
    if (sqlite3_step(stmt) != SQLITE_DONE)
    {
            NSLog(@"Error Occured %s", sqlite3_errmsg(db));
    }
    sqlite3_finalize(stmt);
   }
   sqlite3_close(db);
}
```

238

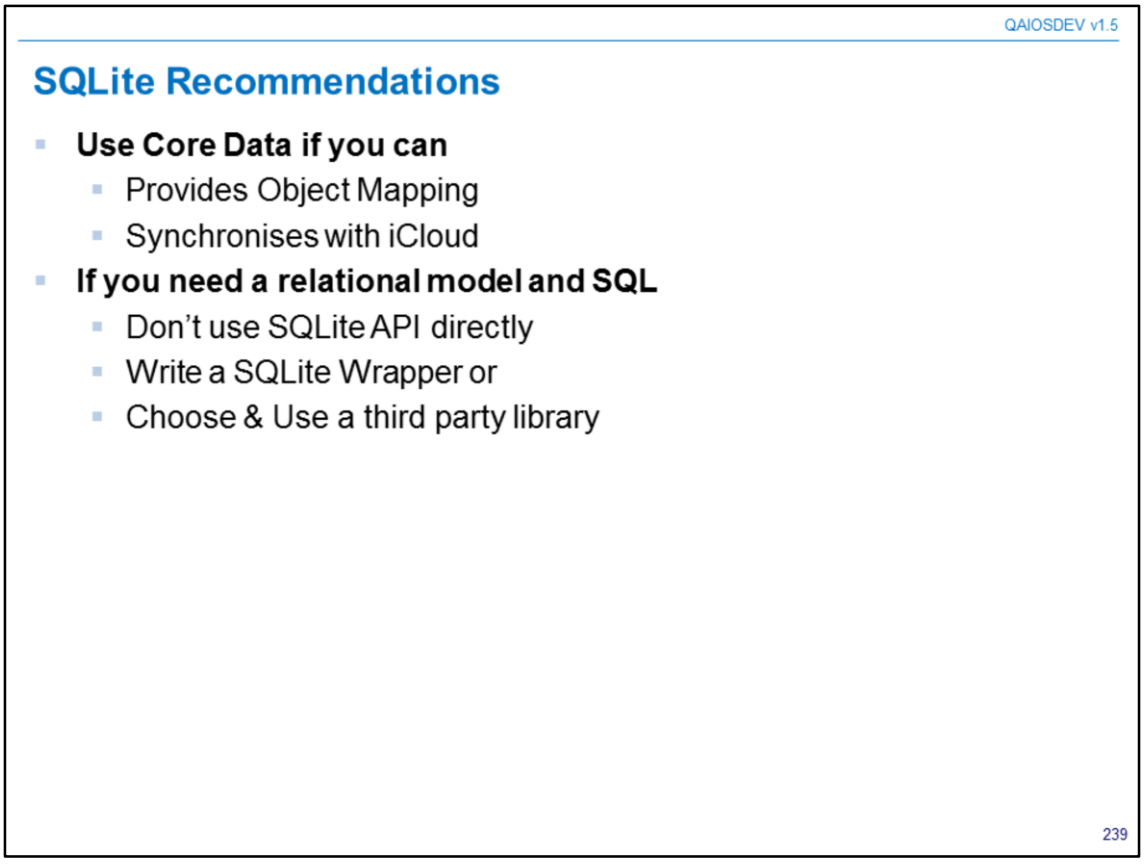
Inserting, Updating and deleting data follows the same pattern using the standard SQL syntax modified to include ? placeholders in place of the data values.

The SQLite API allows you to bind data to the prepared SQL statement using the order in which the ? appear. If a question mark appears at position 1 in the SQL string you can bind a data value to position 1.

Each of the placeholder are bound to data using **sqlite3_bind_** functions. You specify the placeholder position in the prepared statement and pass the value to be bound.

In the code above it is binding “New Book” to the **name** column and 25.99 to the **price** column. Type binding functions include:

- sqlite3_bind_text
- sqlite3_bind_double
- sqlite3_bind_int
- sqlite3_bind_int64
- sqlite3_bind_blob
- sqlite3_bind_null
- sqlite3_bind_zeroblob




If you need a relational SQL based database then SQLite is your only option. It's a fast powerful lightweight database.

If you do use, find a third party library or write one of your own to hide the C library complexity. It can be prone to leaks if not used correctly so implementing it correctly in one place will minimise risk.

Generally, it is advisable to use CoreData in all other circumstances. CoreData performs object mapping persistence and as such isn't relational. There is no access to SQL querying with a code based predicate languages available to perform queries against data.

CoreData hides a great deal of complexity from the developer and provides very easy to use data access capability once the initial library has been learned. It also natively supports iCloud synchronisation something that SQLite on its own never will.

## Summary

- **App bundle contains a mini file system of app resources**
- **Apps run within their own file "Sand Box"**
  - Isolated file access – Security
- **NSFileManager navigates and manipulates the iOS file system**
- **Store Local Data**
  - Local files (text, XML, JSON)
  - NSUserDefaults
  - Key Chain Storage
  - Core Data (Next chapter)

## Property List (plist) Data Storage

- **Xcode has a designer to create plist files in your app**
- **Great for low data volumes**
- **Store key value pairs for data data types**
  - NSNumber, NSString, NSDate, NSArray, NSDictionary, NSData
- **Query and manage plists using NSMutableDictionary**

```
NSMutableDictionary *data;

data= [[NSMutableDictionary alloc] initWithContentsOfFile: path];

NSString *firstName=[data objectForKey:@"firstName"];

[data setValue:[NSNumber numberWithInt:33] forKey:@"Age"];

[data writeToFile:path atomically:NO];
```

241

Another way to store and retrieve small amounts of data is using property lists (plists). They can be designed in Xcode and saved as part of your app bundle. You will need to deploy them from the app bundle to the file system first before use.

You can store key value pairs with data types including NSNumber, NSString, NSDate, NSArray, NSDictionary and NSData.

To load a plist into an NSMutableDictionary, call **initWithContentsOfFile** passing it the path of the plist.

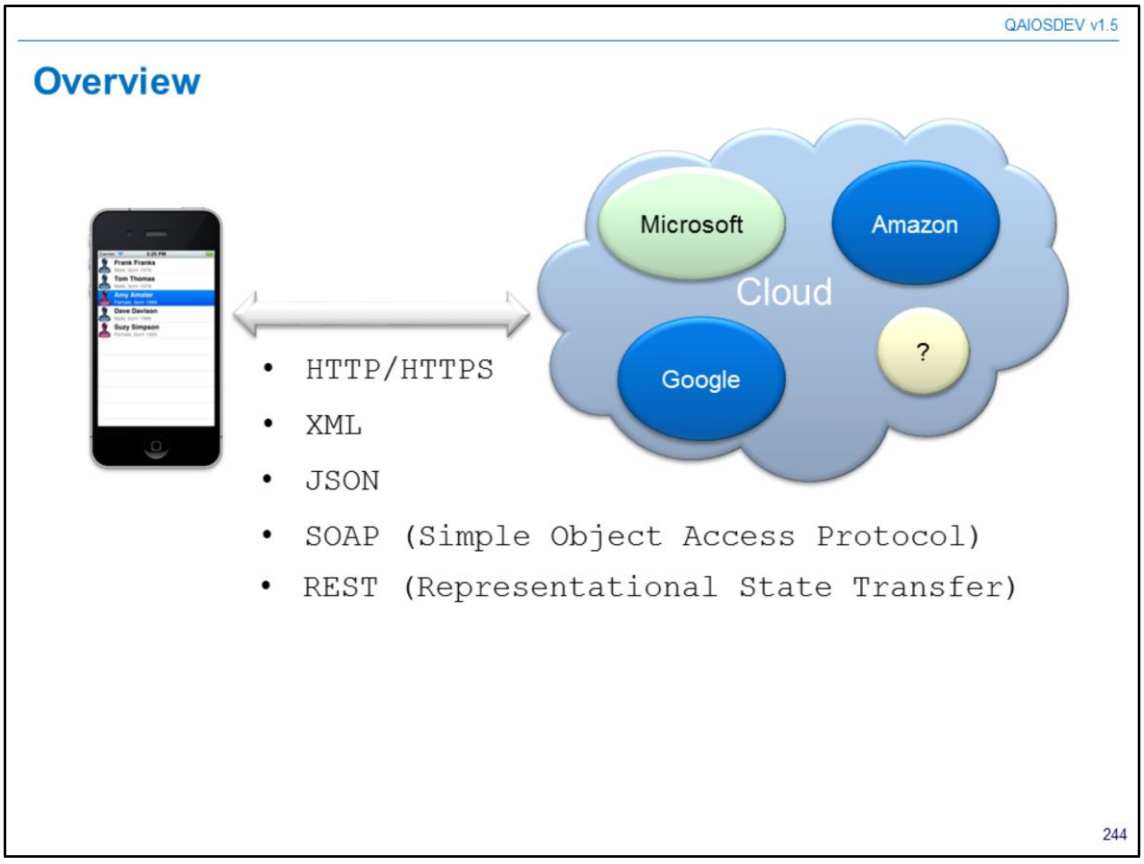
From there, you can retrieve data as for any NSDictionary object using objectForKey.

Adding/updating data just involves calling method **setData** and passing both the value to set and the key of the value to the **forKey** parameter.


QAIOSDEV v1.5
Accessing Web Data Services
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - To learn how to send HTTP requests and receive responses
  - To understand how to process XML and JSON data
- **Contents**
  - Overview of HTTP Request/Response
  - Retrieving data in Objective-C
  - Synchronous/Asynchronous Requests
  - Posting Data
  - Parsing XML with events
  - XML Parsing in memory
  - JSON Parsing
  - Other Parsing Libraries

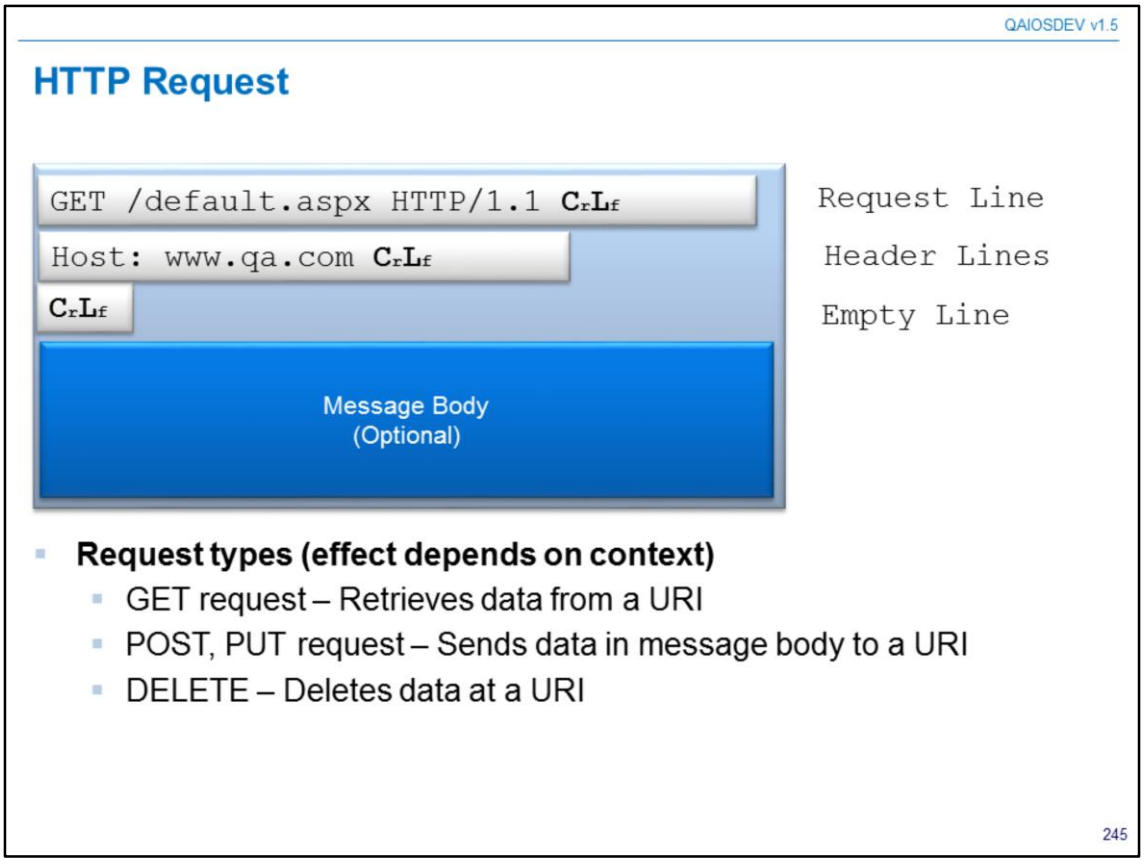



There are a whole host of data services out there in the cloud. From the well known ones provided by Amazon, Microsoft and Google, through Facebook, twitter and a whole host of other providers.

Underlying all of these services are a common set of technologies that include the transport mechanism HTTP/HTTPS and a common data mark-up syntax usually either XML (extensible mark-up language) or JSON (JavaScript object notation).

Cloud services tend to be more highly structured and use more generic protocols to ease the task of exchanging data. Simple Object Access Protocol (SOAP) describes an XML based protocol in which each of the services are described in detail using an XML file that clients can freely request. Using this document, they can discover the available services and their interfaces allowing them to construct XML payloads to make and receive HTTP requests for services.

REST has a much simpler protocol for service requests relying on the HTTP mechanism itself to carry request information within the URL, header and the request body.




HTTP requests are just text messages with a specific structure.

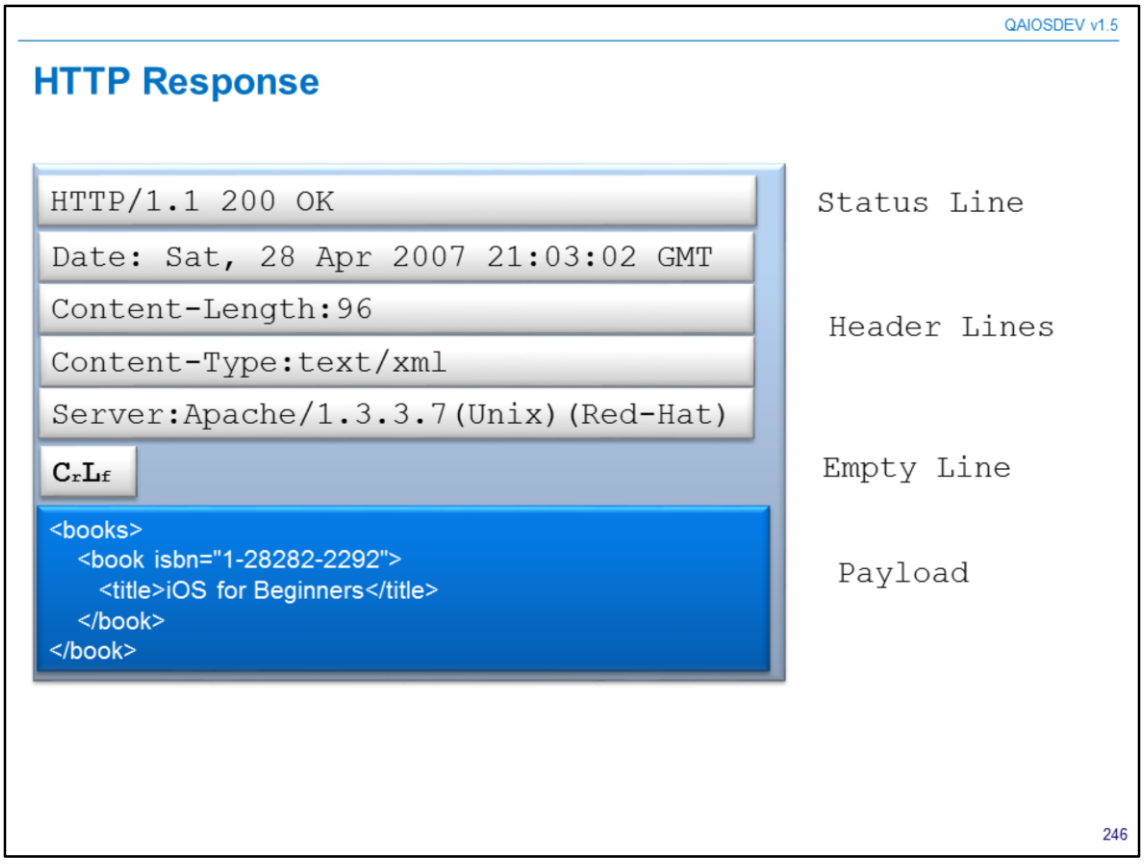
The first line contain the request information indicating the request type GET, POST, PUT or DELETE. GET requests perform a query. In the above example, the request is looking for fie **default.aspx.**

Request line always terminate with CrLf.

Subsequent lines are for header information, which send information about the request and browser etc.

There is always then an empty line followed by the request body, which can contain data to be sent to a server if this is a POST or a PUT request.

When a request is sent we then wait for a response.




The HTTP response comes back after a request has been sent. The first line of the response is the status line. Ideally, we want as status of 200 or OK.

Next, we have a series of response headers with details of the response e.g. server info etc.

A blank line is followed by the response body that contains the payload we have been waiting for which is the data we requested.

## Retrieving Data

- **NSData can retrieve data from a URL**
  - dataWithContentsOfURL:(NSURL*)url

```
NSString *sUrl = @"http://feeds.bbci.co.uk/news/rss.xml";

NSURL *url = [NSURL URLWithString:sUrl];

NSData *dt= [NSData dataWithContentsOfURL:url];

NSString *xml=[[NSString alloc] initWithData:dt encoding:NSUTF8StringEncoding];

NSLog(@"%@",xml );
```

- **Issues**
  - Synchronous
  - No access to headers
  - No Status codes or Error reporting

247

iOS has some simple to use classes you can use to retrieve data from the web. The easiest of which is **NSData's** method **dataWIthContentsOfURL**. All you need to pass it is a URL and it will return an NSData object filled with the requested data.

The code above illustrates its use. First we construct an NSURL form the string using **URLWithString**. Next we load the NSData object using **dataWIthContentsOfURL** passing the URL.

Finally, we convert the NSData object to an NSString by calling **initWithData** on an **NSString** class giving it an encoding of UTF8 as XML is usually UTF8 encoded.

The main problem with this simple approach is it is synchronous and so blocking. It provides no access to the request or response headers and we can't look at status code or errors.

## Managing Requests - Synchronous

- **NSURLRequest gives more control**
- **NSURLConnection submits the request**

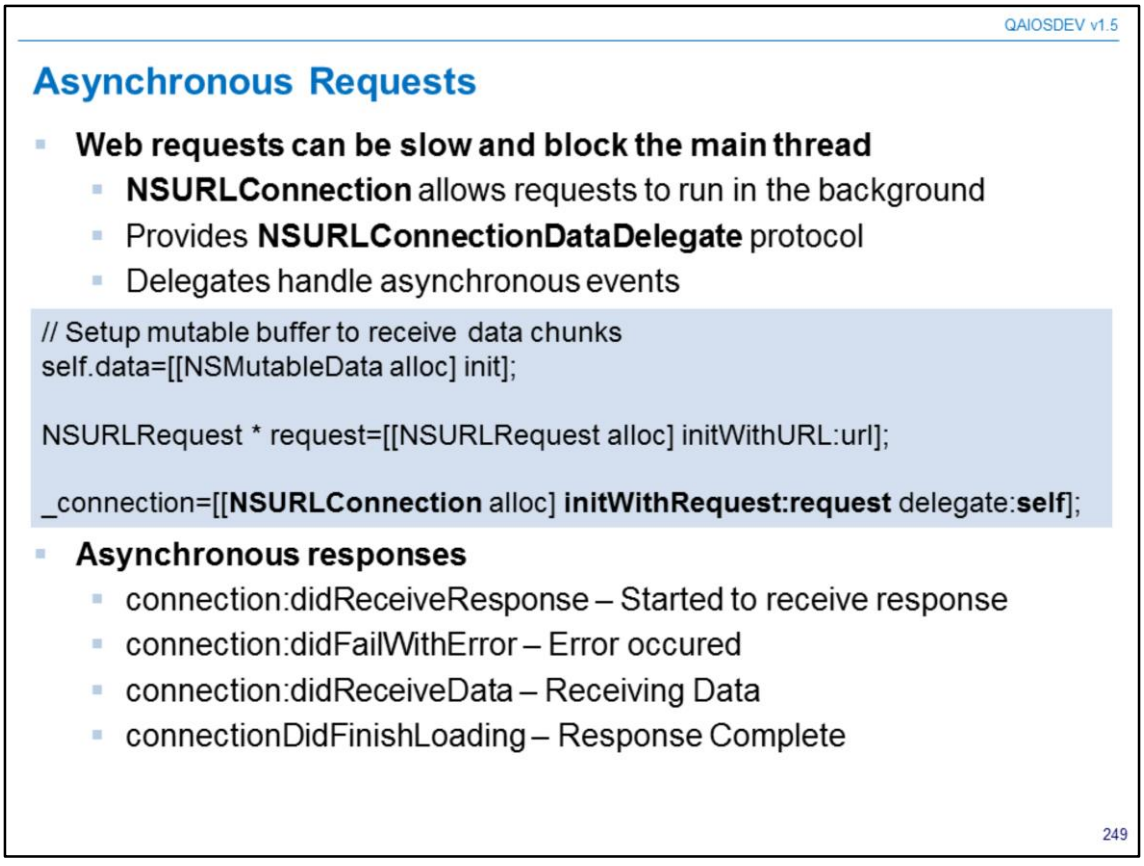
```
NSString *sUrl = @"http://feeds.bbci.co.uk/news/rss.xml";
NSURL *url = [NSURL URLWithString:sUrl];
NSURLRequest * request=[[NSURLRequest alloc] initWithURL:url];

NSHTTPURLResponse *response=nil;
NSError *err;

NSData *dt=[NSURLConnection sendSynchronousRequest:request
                                 returningResponse:&response error:&err];
// If all OK HTTP Status Code is 200
if (response.statusCode==200)
{
    // Get the Content-Type header
    NSLog(@"%@",[[response allHeaderFields] objectForKey:@"Content-Type"]);
}
```

248

To gain control of the request, we need to use NSURLRequest to submit requests directly. Just create an instance of one and supply the required URL.

Next, submit the request using the class method of NSURLConnection **sendSynchronsousRequest.** That returns an NSData object, a NSHTTPURLResponse object and reports an error if one occurs.

We now have all of the required information for both the request and the response. Only trouble is it's still synchronous and therefore blocking.




Web requests can be really slow and will block the main thread if you let them. You need to allow them to run in the background until they have returned their data so asynchronous processing is vital.

NSURLConnection provides that capability. All you need to do is initialise it with your request but also provide a delegate class eg **self** that implements protocol **NSURLConnectionDataDelegate.** NSURLConnection will now fire asynchronous event calls to the delegate class when:

- It first receives a response
- If an error occurs
- When data is received
- When all data has been received

These events will run on the same thread as the NSURLConnection was called on.

Data will be received in chunks asynchronously so we would usually setup an **NSMutableData** buffer to store the data as we receive it.

The **didReceiveResponse** event method is called when we first receive a response after making a call. Usually, this is a good time to reset the data buffer.

## Asynchronous Request – Setting up to receive data

- **Handle Initial Response**

```
-(void)connection:(NSURLConnection *)connection didReceiveResponse:
(NSURLResponse *)response {

    [self.data setLength:0];
}
```



When a response is first detected **connection:didReceiveResponse** fires. This is a good place to reset data storage buffers etc.

## Asynchronous Request – (continued)

- **Dealing with Errors**

```
- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError
*)error {
        NSLog(@"Error retrieving data %@",[error localizedDescription]);
}
```

- **Processing Results**

```
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
   [self.data appendData:data];
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    // Parse Data
}
```



If an error occurs **didFailWithError** will fire.

**didReceiveData** will fire every time some data is received and all we need to do is add it to the buffer.

**connectionDidFinishLoading** fires when the request is complete and that's the time to parse the data in the buffer.

## Asynchronous requests with blocks – iOS 5

- **iOS 5 introduced a block method on NSURLConnection**
  - **sendAsynchronousRequest**

```
+ (void)sendAsynchronousRequest:(NSURLRequest *)request
        queue:(NSOperationQueue *)queue
        completionHandler:(void (^)(NSURLResponse*, NSData*, NSError*))handler
```

- **Example**

```
NSOperationQueue *q=[[NSOperationQueue alloc] init];
[NSURLConnection sendAsynchronousRequest:request queue:q
                    completionHandler:^(NSURLResponse* resp,
                      NSData* data, NSError* err)
{
        if (err!=nil)
        {
            // Process NSData & Access the UI on the main thread!!!
        }
}];
```

252

**NSURLConnection** provides a block method **sendAsynchronousRequest** simplifies your code to a great extent removing the need for the delegate methods and protocol declarations etc. Everything is in one place.

You need to provide an NSOperationQueue then call class block method **sendAsynchronousRequest** on NSURLRequest and provide a completions handler. The completion handler runs when the request has completed and provides the response, the received data and an NSError object.

One thing you do have to remember to access is the UI, you need to run on the main thread!

## Posting Data

- **Posting data requires the following actions**
  - You must use an NSMutableURLRequest
  - Set the request's HTTP method
  - Set the requests BODY with a payload
- **Setting the HTTP Method**

```
NSMutableURLRequest * request=[[NSMutableURLRequest alloc] initWithURL:url];
[request setHTTPMethod:@"POST"];
```

- **Add a payload to the request body**

```
NSString *xml=@"<books/>";

NSData *xmlData=[xml dataUsingEncoding:NSUTF8StringEncoding];
[request addValue:@"text/xml" forHTTPHeaderField:@"Content-Type"];
[request setHTTPBody:xmlData];
NSData *dt=[NSURLConnection sendSynchronousRequest:request
                                 returningResponse:&response error:&err];
```

253

To POST data to a server, you must use an NSMutableURLRequest because you need to make changes to it like setting its HTTP method to POST. You also need to add data to the request body.

Creating the request is identical to previous examples.

Next you call **setHTTPMethod** passing string "POST" that tell the receiving server it's a post request.

Finally, you need to set the HTTP body with some data. Call **setHTTPBody** passing an NSData variable.

If you need to set any Headers call **addValue:forHttpHeaderField.**

## NSURLSession – iOS 7

- **Replacement for NSURLConnection**
  - Per connection authentication
  - Global/Session network object storage (cache etc.)
  - Supports Background Transfers
- **Creating a session**

```
NSURLSessionConfiguration *config = [NSURLSessionConfiguration
defaultSessionConfiguration];

NSURLSession *session = [NSURLSession sessionWithConfiguration:config
delegate:self delegateQueue:nil];
```

- **Background session**

```
NSURLSessionConfiguration *config= [NSURLSessionConfiguration
backgroundSessionConfiguration:@"com.qa.bg"];

NSURLSession *bgSession = [NSURLSession sessionWithConfiguration:config
delegate:self   delegateQueue:nil];
```

254

iOS 7 has brought in a replacement for NSURLConnection. It should be used for all >=iOS7 developments.

It has many advantages including per connection authentication rather than per request (as before). It will also work in background mode which means a file can be downloaded as a background task even when the app is no longer active.

## Accessing Data

- **Using Block method – dataTaskWithURL**

```
NSURL *URL=[NSURL URLWithString: @"http://feeds.bbci.co.uk/news/rss.xml"];

[[session dataTaskWithURL:URL
  completionHandler:^(NSData *data, NSURLResponse *response, NSError *error)
{
    NSString *data=  [[NSString alloc] initWithData: data
                                        encoding: NSUTF8StringEncoding]);

}
] resume];
```



Simplest way to access data is with block methods **dataTaskWithURL** (passing a URL) or **dataTaskWithRequest (passing an NSURLRequest)**.

## Downloading Files

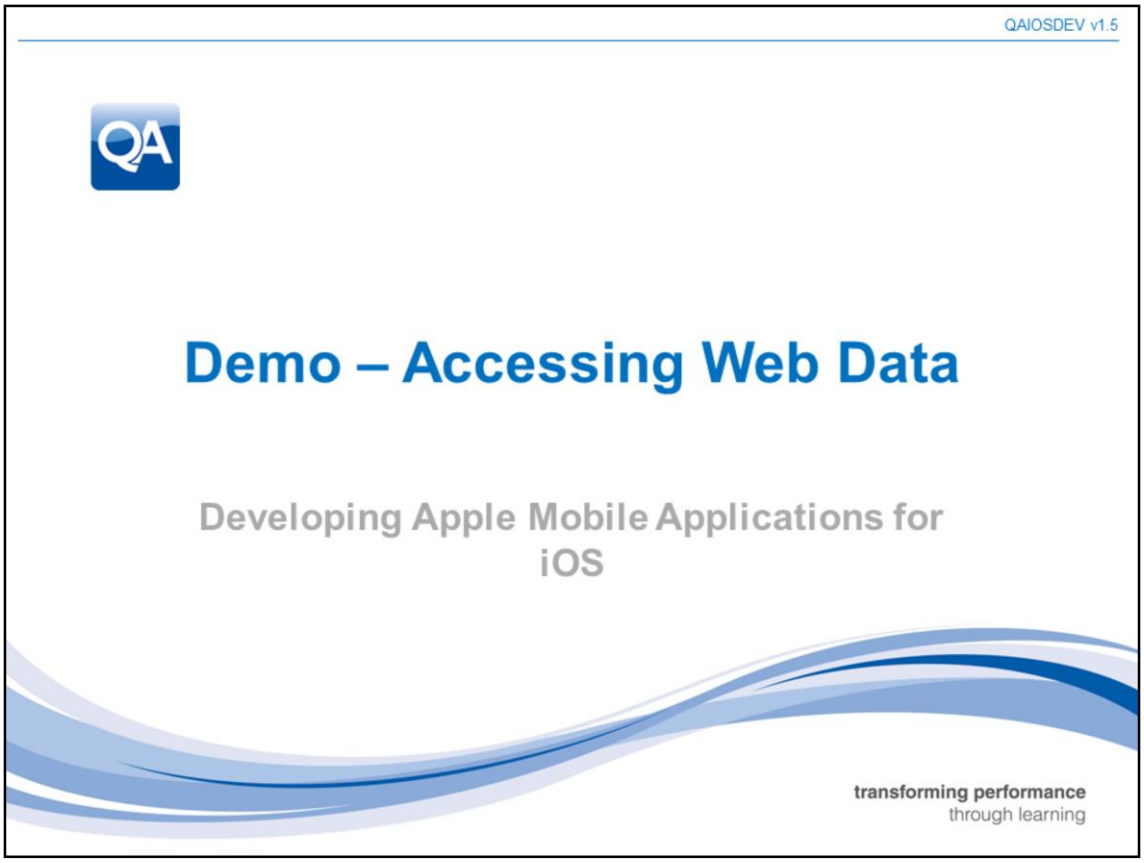
- **Use downloadTaskWithURL/downloadTaskWithRequest**

```
NSURL *URL=[NSURL URLWithString: @"http://sample.com/movie.mp4"];
NSURLRequest *req=[NSURLRequest requestWithURL:URL];

// Could also use downloadTaskWithURL
self.downloadTask=[[session downloadTaskWithRequest:req];
[self.downloadTask resume];
```

- **Handle delegate methods**
  - downloadTask:didWriteData
  - downloadTask:didFinishDownloadingToURL



You can download files **downloadTaskWithURL** or **downloadTaskWithRequest**. Handle delegate methods **didWriteData** to be informed of progress and **didFinishDownloadingToURL** to find out when the download completed.


QAIOSDEV v1.5
Demo – Accessing Web Data
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Processing Results

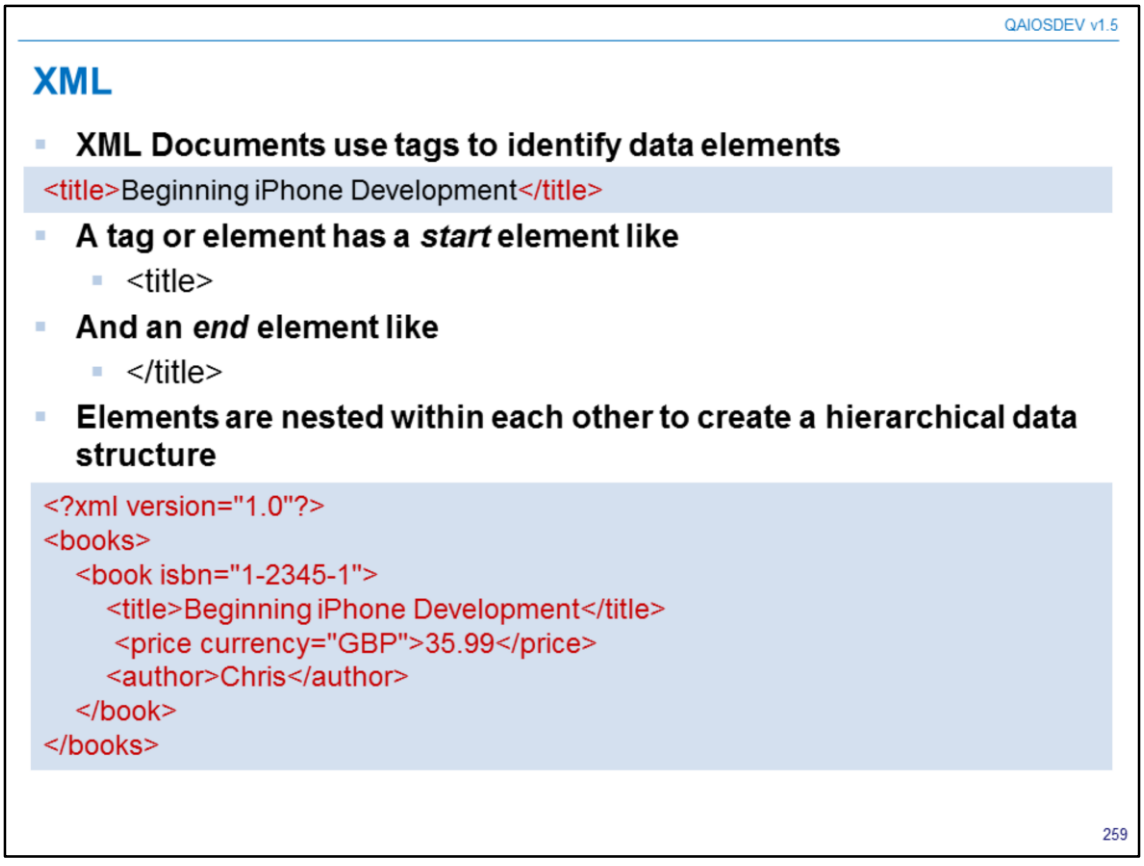
- **Most common data mark-up formats?**
  - Comma Separated Format (CSV)
  - JavaScript Object Notation (JSON)
  - Extensible Mark-up Language (XML)
- **Good storage formats are**
  - Human readable
  - Machine readable (parsers available)
  - Compact
  - Easy to process
  - Verifiable



Most common data mark-up formats are CSV, JSON and XML.

A good mark-up language is human readable, easily machine readable, compact and easy to validate/specify.

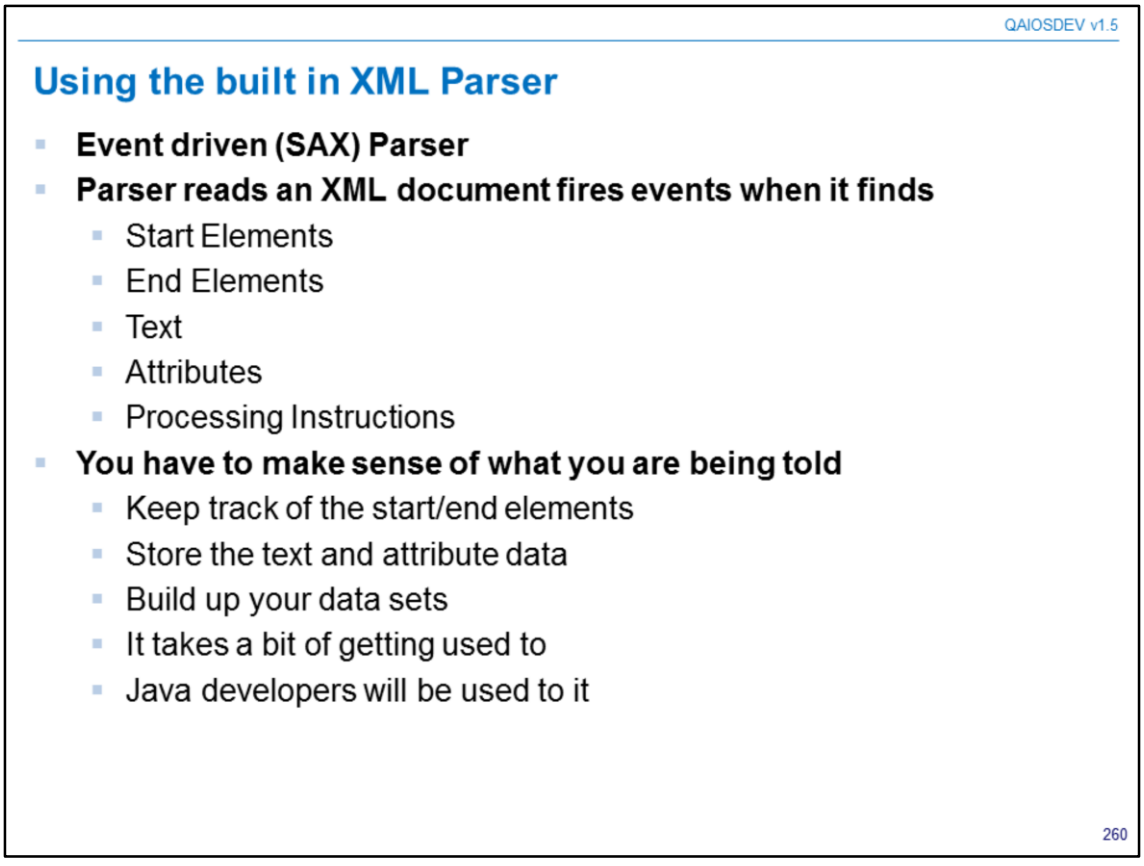
CSV is poor. XML and JSON have their strengths and weaknesses.




XML uses tags or elements to mark data items. They wrap data items between a start tag and an end tag. Tags contain a name within two chevrons. The end tag also contains a backslash.

Elements can contain attributes that reside within the start element and store data within quotes. In the following example, **<book isbn="1234">,** the book element has an isbn attribute containing the data 1234.

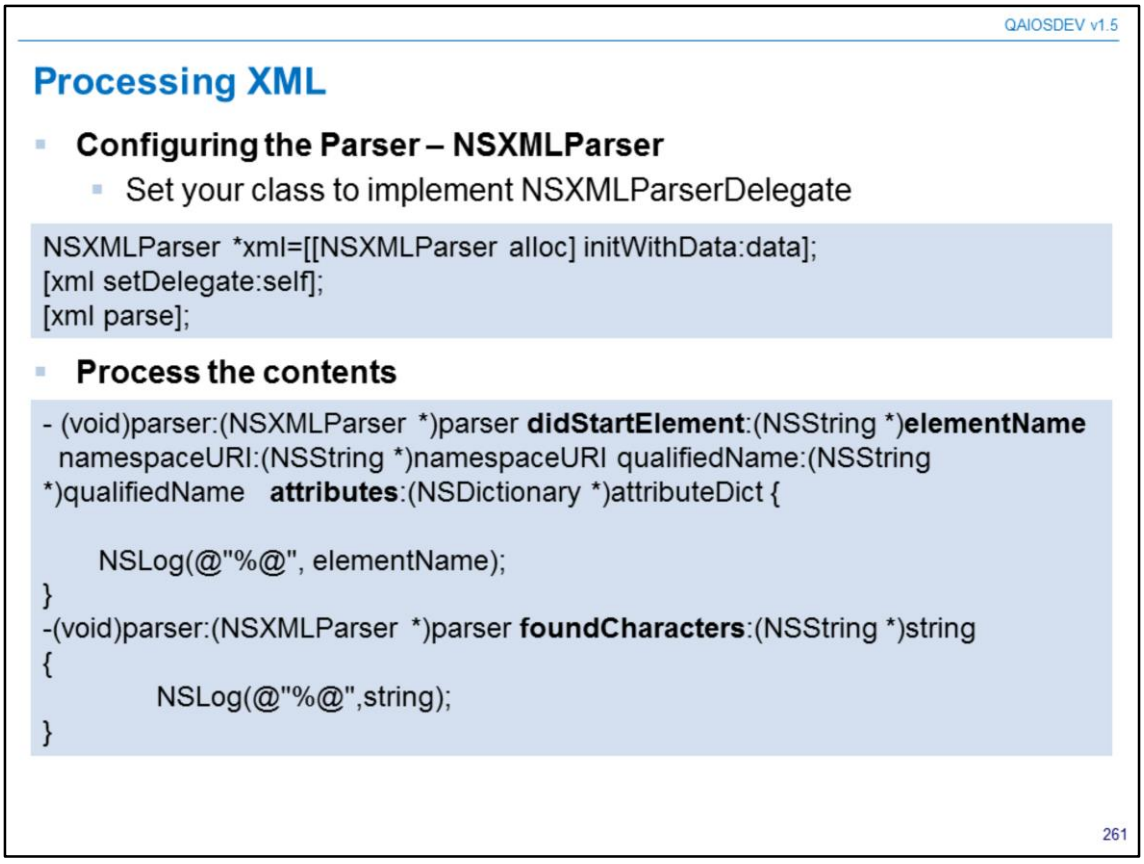
Elements can contain data or other elements. The combination allows you to create a complex hierarchical data structure.




iOS has a build in XML parser. It's an event driven (SAX type) parser.

SAX Parsers read through an XML document and fire events whenever they find some kind of XML structure like an XML start element or attribute or piece of text.

With SAX parsers, you have to write quite a lot of code to process your XML. By contrast DOM (Document Object Model) parsers load the whole document into memory and you are free to query and navigate through it.




First, create an instance of NSXMLParser loading your data.

Next you need to set the delegate to a class that implements the NSXMLParserDelegate protocol. Finally, call **parse** on the NSXMLParser class.

Parsing will commence and the delegate events will start firing.

These delegate events include:

-
(void)parser:(NSXMLParser *)*parser* didStartElement:(NSString *)*elementName*
namespaceURI:(NSString *)*namespaceURI* qualifiedName:(NSString *)*qualifiedName* attributes:(NSDictionary *)*attributeDict*
-
(void)parser:(NSXMLParser *)*parser* didEndElement:(NSString *)*elementName*
namespaceURI:(NSString *)*namespaceURI* qualifiedName:(NSString *)*qName*

-(void)parser:(NSXMLParser *)*parser* foundCharacters:(NSString *)*string*
-(void)parserDidEndDocument:(NSXMLParser *)*parser*
-(void)parserDidStartDocument:(NSXMLParser *)*parser*

Notice didStartElement contains the element name and the attributes.

## JSON

- **JSON is made up of three structures**
  - Key value pair – "name" : "Fred"
  - Object – { "name":"Fred", "age":20}
  - Array – ["Red", "Blue", "Green"]
- **Combined together you can build a complex data structure**
  - Key Value pairs and Objects are serialized NSDictionaries
  - Arrays are serialized NSArrays

```
{   "people" :
    {"person":
      {
      "firstName": "Fred",
      "lastName" : "Bloggs",
      "age"      : 25,
      "address"  : { "street": "21 My Street", "city": "London","postcode":"EC1"}
      }
    }
}
```

262

JSON is a compact mark-up language. It's made up of three structures.

- Key Value pairs are a descriptor in quotes followed by a colon then a quoted text or numeric value
- Objects contain comma separated lists of key value pairs or other objects or arrays delineated by braces
- Arrays contain lists of any of the above delineated by square brackets

Ultimately, these structures translate back into code as dictionaries and arrays, which is the best way to think about them. Braces with key value pairs will give you a dictionary, Square brackets will give you an array.

## JSON Parsing

- **iOS 5 provides NSJSONSerialization**
  - Load data using method **JSONObjectWithData**

```
+ (id)JSONObjectWithData:(NSData *)data options:(NSJSONReadingOptions)opt
error:(NSError **)error;
```

- **Root object - either**
  - NSArray
  - NSDictionary
- **Root object contains any of**
  - NSArray
  - NSDictionary
  - NSString
  - NSNumber



iOS 5 provides the NSJSONSerialization object that makes JSON parsing very straightforward. When parsing a JSON structure, the top level object is either an NSArray or NSDictionary.

The **JSONObjectWithData** method loads JSON data and, following on from JSON structure, returns a root object that will either be an NSArray or NSDictionary. Within the root object, any of the main JSON data types can be contained within including NSArray, NSDictionary, NSString and NSNumber. That just depends on the structure of the JSON data message itself.

Further into the structure the other objects will also include NSString and NSNumber.

## Navigating through a JSON Structure

```
NSString *sURL=@"http://search.twitter.com/search.json?q=objective-c";
NSURL *url=[NSURL URLWithString:sURL];
NSData *data =[NSData dataWithContentsOfURL:url];
NSError *jsonError = nil;

NSDictionary *dict  = [NSJSONSerialization JSONObjectWithData:data
                              options:kNilOptions error:&jsonError];

NSArray *results=[dict objectForKey:@"results"];

for (NSDictionary *dict in results){

    NSString *tweet=[dict objectForKey:@"text"];

     NSLog(@"%@",tweet);
}
```

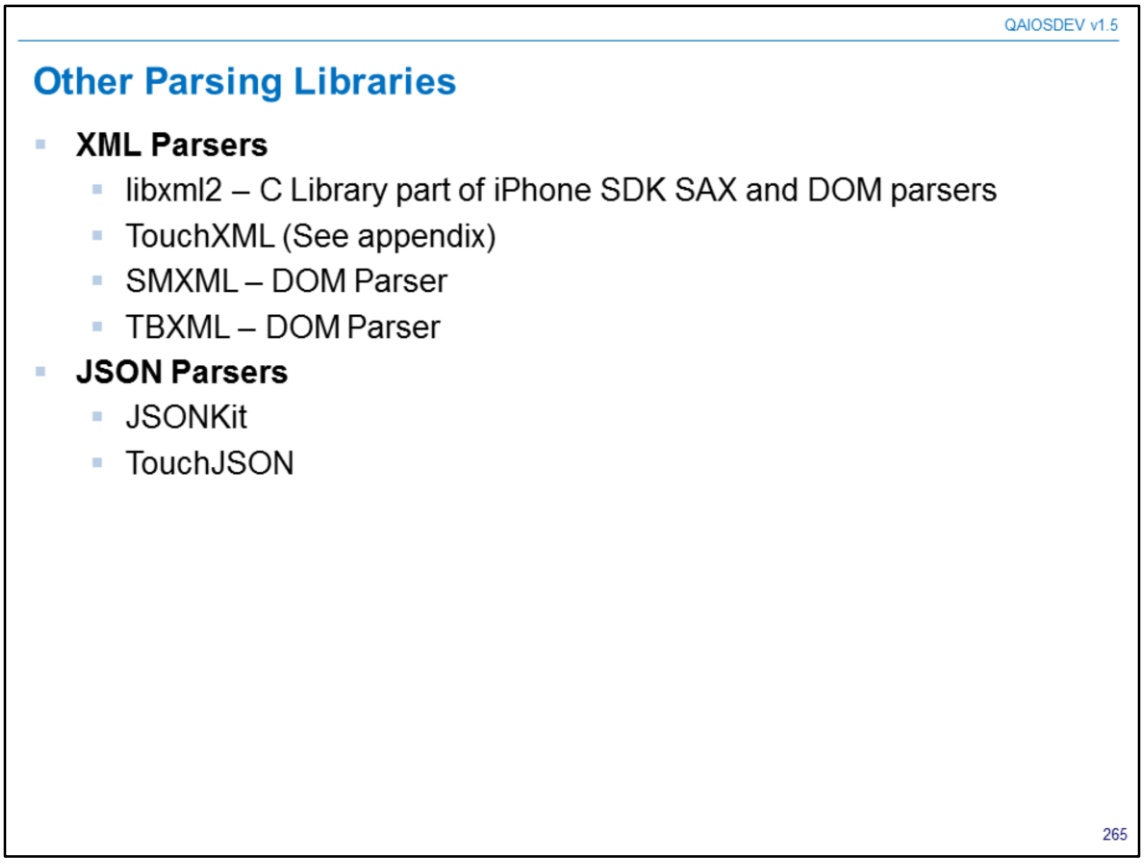
```
{"completedIn":0.199,
"results":[{
"from_user":"fred",
"text":"objective-c rocks"
}]}
```



In the code example above, we start by setting up the URL of the twitter search URL that returns JSON data. We are searching for tweets containing the word "objective-c" then, to keep the code simple, we are using the synchronous NSData loading method **dataWIthContentsOfURL**.

Next, we parse the data with **NSJSONSerialization** class and method **JSONObjectWithData** to load the JSON twitter data. It either returns an NSDictionary or an array. As you can see in the blue box, the JSON data has key value pairs completedIn, results, from_user and text. Key value pairs belong in a dictionary so we can tell that a dictionary will be returned by the method call.

We want to get at the **results** array to obtain the tweets and then the **text** to obtain each tweet. All we need to do is use the NSDictionary's **objectForKey** method to retrieve the text value from the dictionary.




We've looked at some options for both XML and JSON parsing and of course there are many other options including:

For XML, notable ones include the iPhone SDK built in C library libXML2 that contains a SAX and DOM parser.

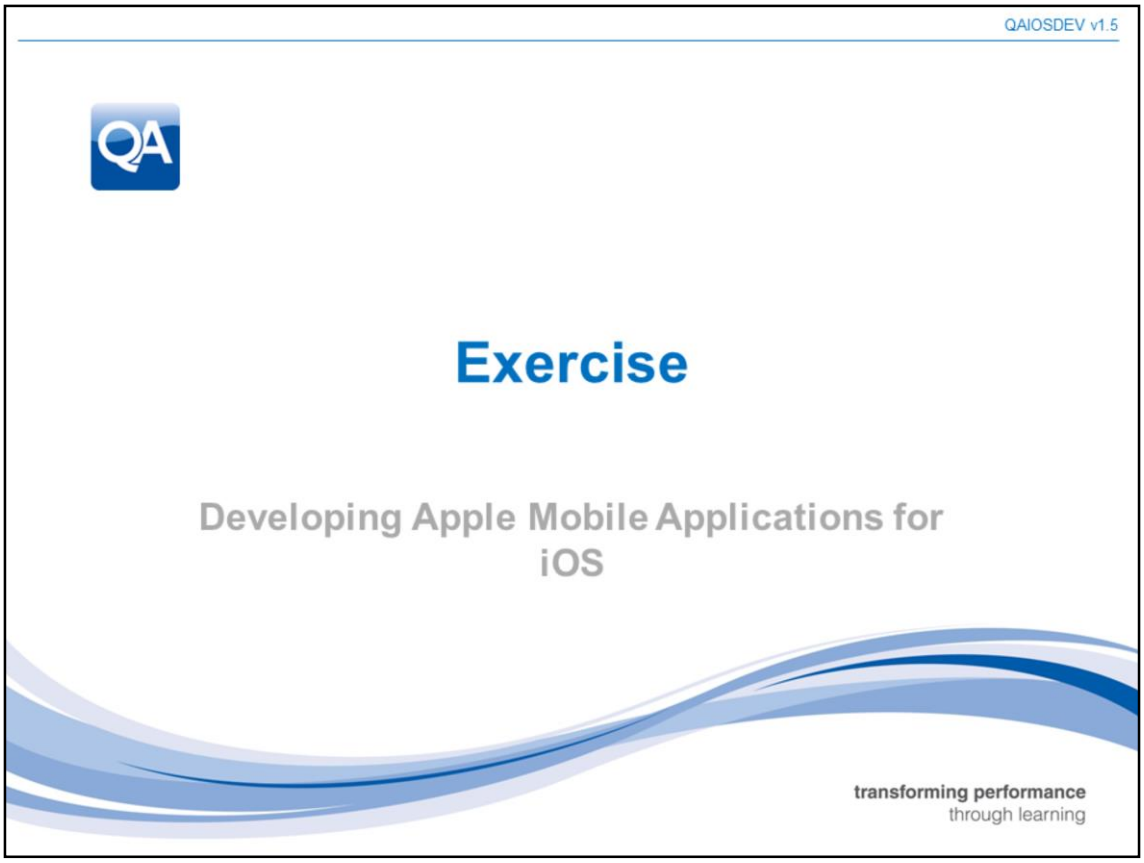
SMXML is a TouchXML like parser that is also well liked by developers. http://github.com/nfarina/xmldocument

TBXML is another lightweight XMLDOM parser http://github.com/71squared/TBXML. It's a pre ARC implementation.

Other, pre iOS 5 JSON Parsers include JSONKit http://github.com/johnezang/jsonkit and TouchJSON http://github.com/touchcode/tochjson

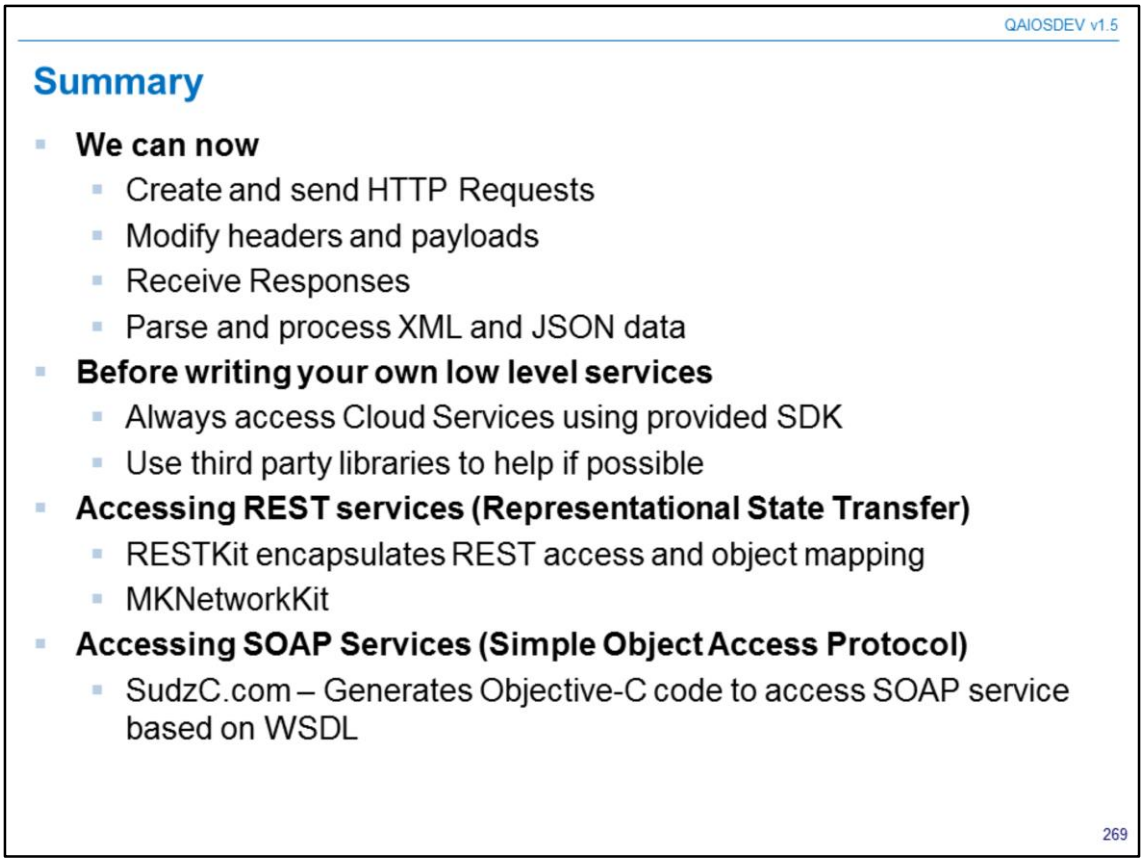
All of the above are possible options for you to evaluate and there are many others.


QAIOSDEV v1.5
Demo – Parsing Data
Developing Apple Mobile Applications for iOS
transforming performance
through learning


QAIOSDEV v1.5
Exercise
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Cloud Resources

- **Parse – Parse.com**
  - Cloud based object database and push notifications
  - Free for startup apps – 1 million free requests per month
- **Stackmob**
  - Cloud Database integrated with Core Data
  - Push Notifications
- **Amazon EC2**
- **Windows Azure Services**
- **All accessed via specific apis**




Using the raw skills learned so far, we can create and send HTTP requests after adding headers and payloads.

We've also seen how to process the returned response and retrieve the status code and the response body.

Processing the payload is always the last step and usually this involves either XML or JSON parsing.

Obviously, before you start all of this low level work, it's a good idea to check if you can use an SDK for the service you are trying to access.

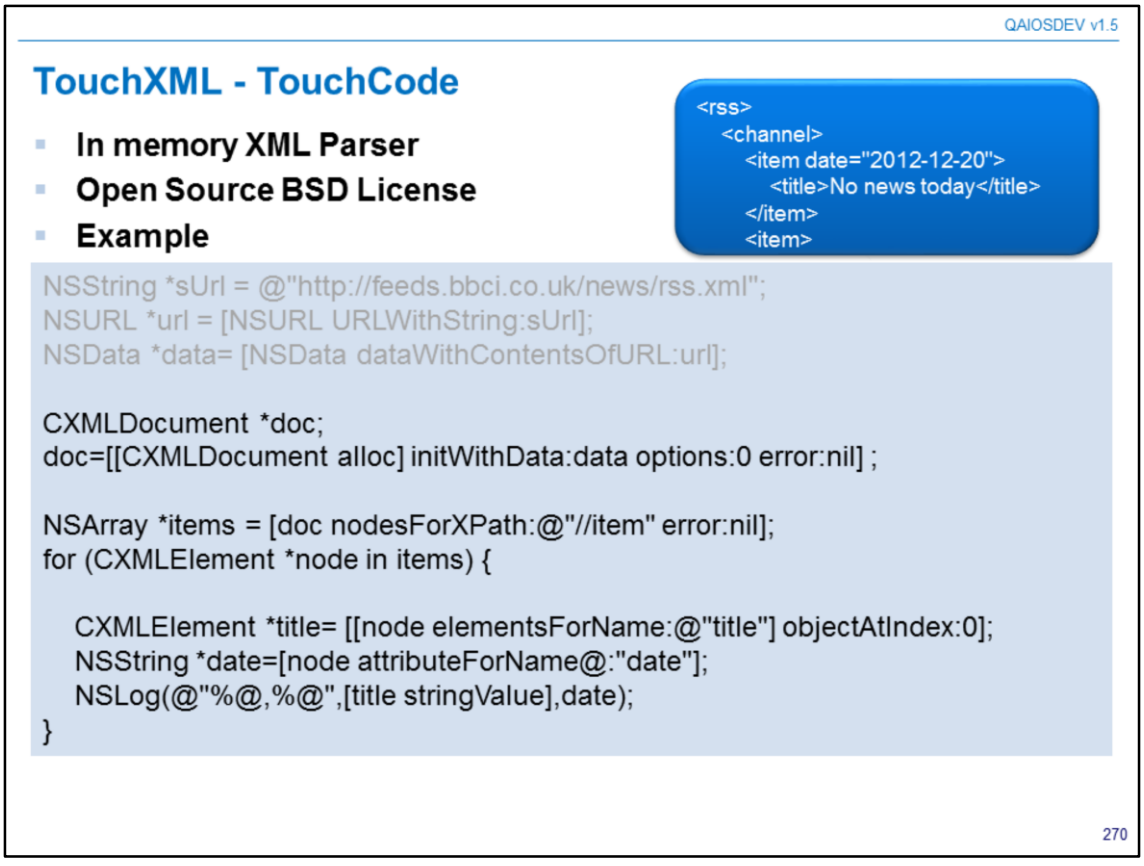
Failing that, if the service is using a standard protocol like REST (Representational State Transfer) or SOAP (Simple Object Access Protocol) then try using one of the frameworks/utilities out there to help with that.

RESTKIT can you you with access to rest services. http://Resttkt.org

http://SUDZ.com will generate Objective-C proxy classes to access webs

service endpoints based on a WSDL URL you supply.

Both options are worth evaluating.




TouchXML is an “In Memory” open source XML Document Object Model (DOM) Parser. You can download it from https://github.com/TouchCode/TouchXML

You can query the document structure using the XML query language XPATH. // is a deep query that looks for all elements in the structure with the element name following it. So //item retrieves all item elements into an array.

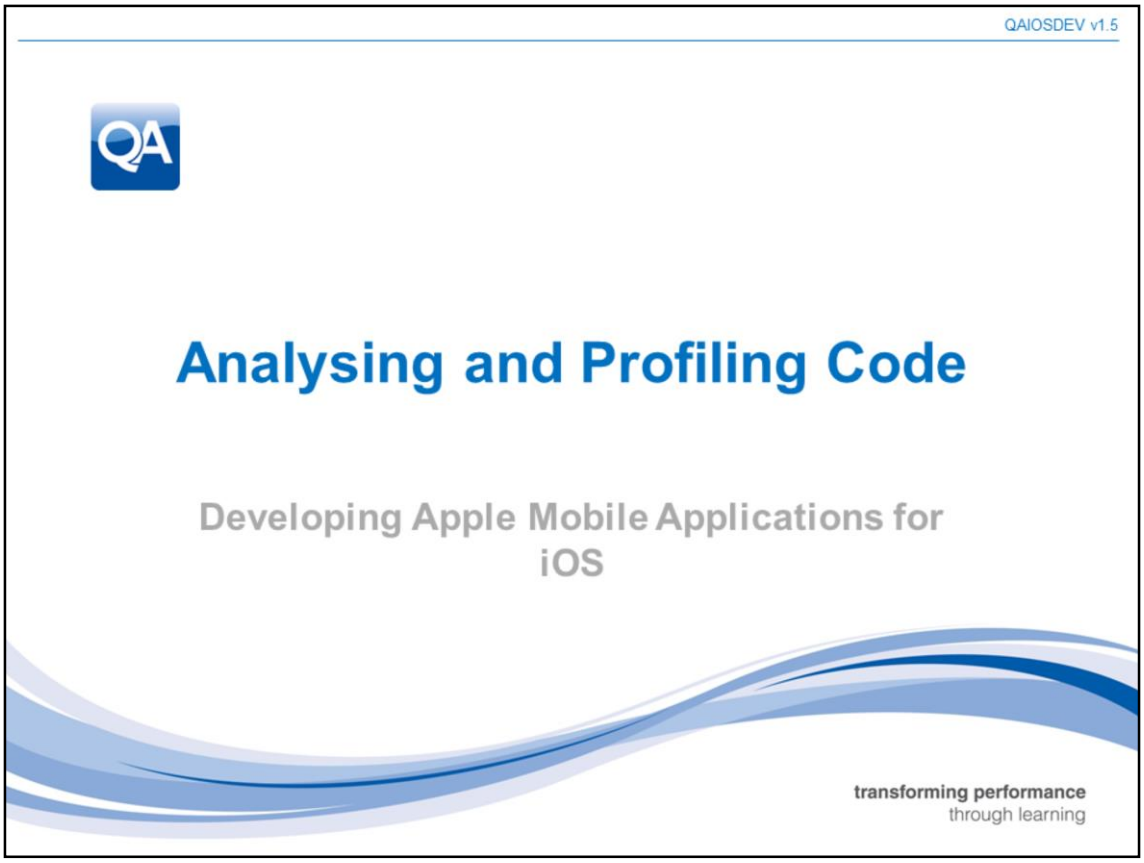
That’s what the code above is doing retrieving all **item** elements from the XML RSS news feed.

First, it loads the XML data into a TouchXML CXMLDocument using **initWithData.** Next, it runs the XPATH query using **nodeForXPath** returning an NSArray of elements.

Next, it loops through each item element and retrieves the **title** element using **elementsForName**. As there is only one, it retrieves the first one in the array into another CXMLElement object. Finally, it retrieves the text inside the element using **stringValue.**

If you’re wondering "why didn’t we just run the XPATH query" //title, you’re right. We could have done that but you would have seen less functionality!

Notice we are also retrieving the date attribute using **attributeForName**.


QAIOSDEV v1.5
Analysing and Profiling Code
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Contents

- **Objectives**
  - Understand how to use the Code Analyser to find and fix potential code issues
  - Learn how to use Instruments to find performance and memory problems
- **Content**
  - Analysing your code
  - Performance profiling an application
  - Analysing an application's memory usage

## Analysing Code

- **Static Code Analysis**
  - First Line of defence
  - Identify problems even before you test
- **Detects "potential" problems in code**
  - Logic problems
  - Unused variables
  - Potential Memory issues
  - Misuse/omission of standard coding practices



Xcode has a built in static code analyser that you can use to highlight some key coding issues including logic problems, unused variables, potential memory issues and misuse or omission of expected coding conventions.

It's not perfect but it is a good first line of defence and is worth running even before you begin your own testing.

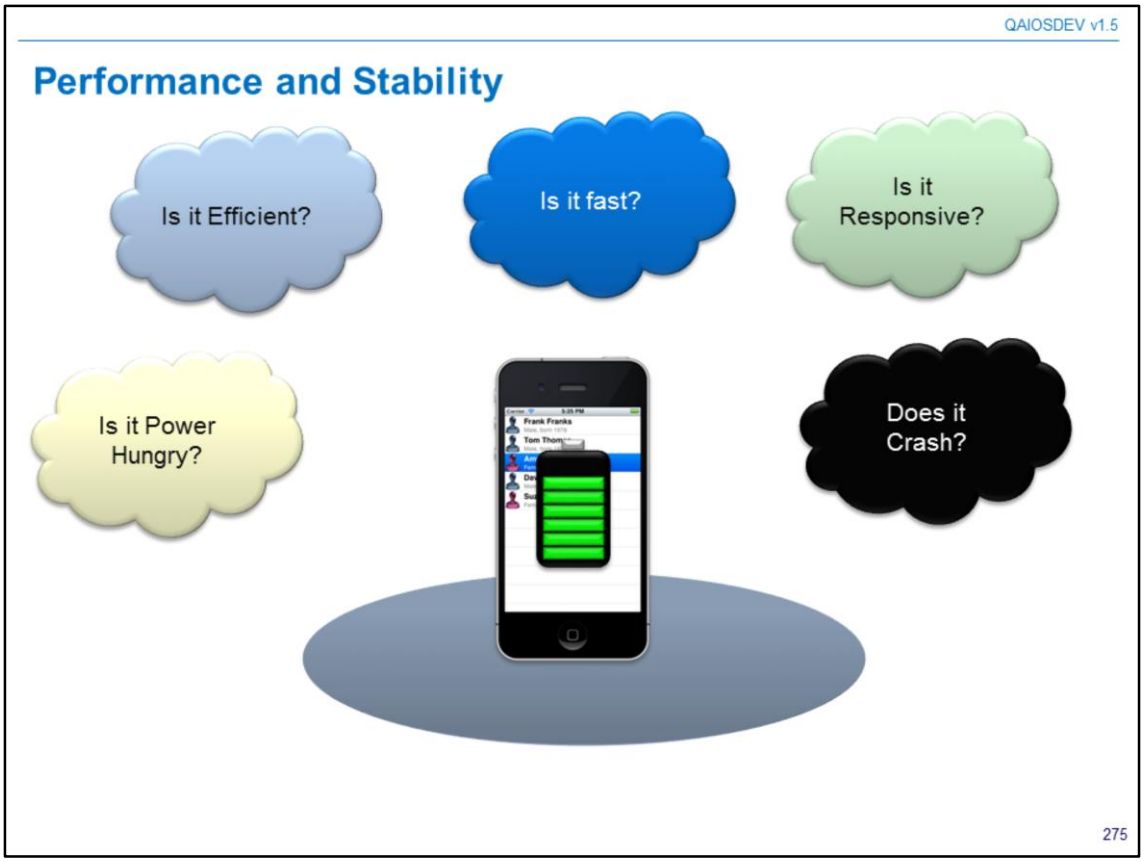
## Static Analyzer

- **Uses the Clang Static Analyzer**
  - Access it from Xcode menu Product->Analyze
  - Reports issues in the Error and Code Windows

```
29 - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
     interfaceOrientation
30 {
31     return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
32 }
33
34 - (IBAction)causeLeakPressed:(id)sender {
35
36     static int i=0;
37     i++;
38     NSString *str=[[NSString alloc] initWithFormat:@"Test %i",i];
39     NSLog(@"%@",str);
40     // Why is this a leak
41     str=[[NSString alloc] initWithString:@"World"];   Value stored to 'str' is never read
42
43 }   Potential leak of an object allocated on line 38 and stored into 'str' 2
44 @end
```

274

Accessible from the Product Menu within Xcode issues are reported within the code window itself as well as the standard Error window in common with compile time error and warning messages.



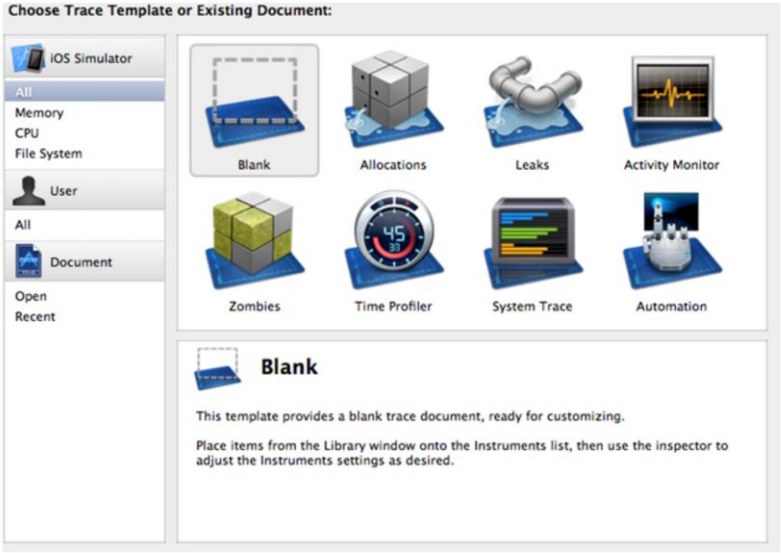

We all want our apps to be both fast and stable.

Is it efficient, fast, responsive and power hungry? Does it crash? Are all questions that need a definitive answer and do we need profiling tools and time to test to gain a degree of confidence in the answer?

Apple have provided **Instruments** as the analysis tool of choice and it provides a host of analysers that can profile many aspects of performance and stability from CPU time profiling to memory leak detection and energy usage analysis.

## Instruments

- Profiles CPU, Memory, IO, Graphic performance
- Creates a Trace Document containing recorded trace data






Instruments provides a suite of different analysers that can be used to trace performance. To make things a little easier, it provides templates that group related analysers (or instruments) together into useful analysis sets.

For example, the Leaks Template includes the Allocations Instrument and the Leaks Instrument. Both give insight into to common memory issues.

## Profiling Code with Instruments

- **Performance Analysis**
  - Time Profiler – CPU time
  - Disk Monitor – Disk I/O
  - Network activity Monitor – Network I/O
  - Energy Diagnostics – Battery Usage
- **Memory Analysis**
  - Allocation – Objects allocated and not destroyed
  - Leak detection – Memory leak detection
  - Zombie detection "over release" (MRR)



There are many available instruments for profiling. Some of the ones in use are listed above.

The key tools are Time Profiler, Allocation and Leak Detection.

## Profiling Strategy

- **Analyse your main flows**
  - Design and focus your test case
  - Profile, optimise, re-profile, compare
- **Work with a realistic environment**
  - Data set size
  - Network speed (WiFi, 3G, EDGE)
  - Intended device targets
- **Don't test performance on the simulator!**
  - Memory testing is ok
- **Key Instrument – Time Profiler**
  - Instrument Strategy – Overall CPU time
  - CPU Strategy – CPU Utilisation
  - Thread Strategy – Thread Utilisation

278

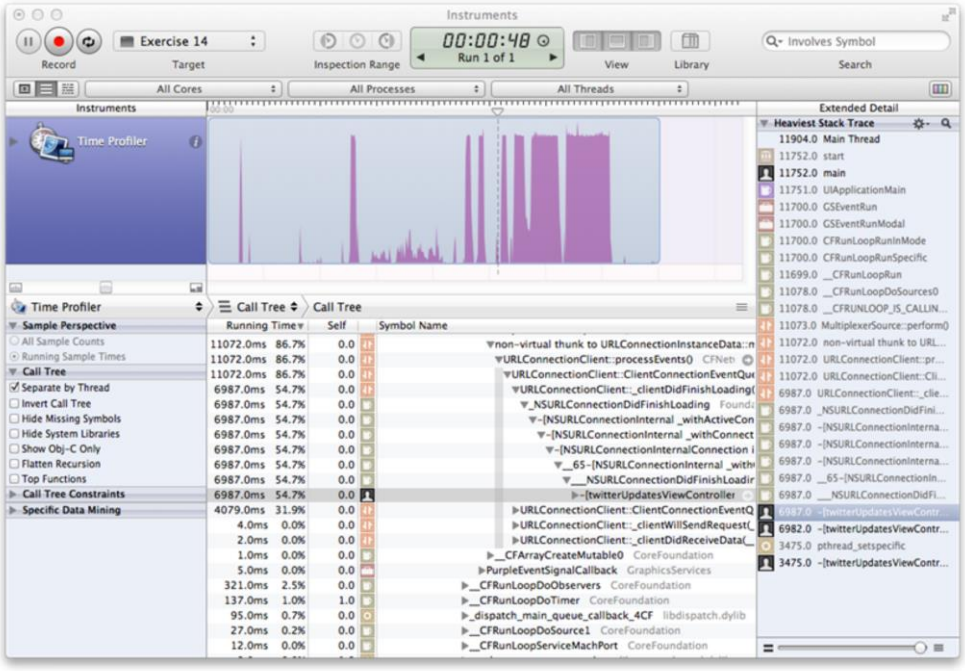
Before you start profiling, you should always plan your test cases first. You need to be able to repeat the test case exactly because and essential part of the strategy is to optimise and compare. Ideally, the optimisation will have made things faster.

You should always work within a realistic environment, which includes testing on the devices you will be releasing to using data sets and networks speeds likely in the live environment.

Don't test for performance on the simulator.

The Time Profiler will be a key instrument here and you should make sure you use the CPU Strategy View to ensure all of the CPUs are utilised.

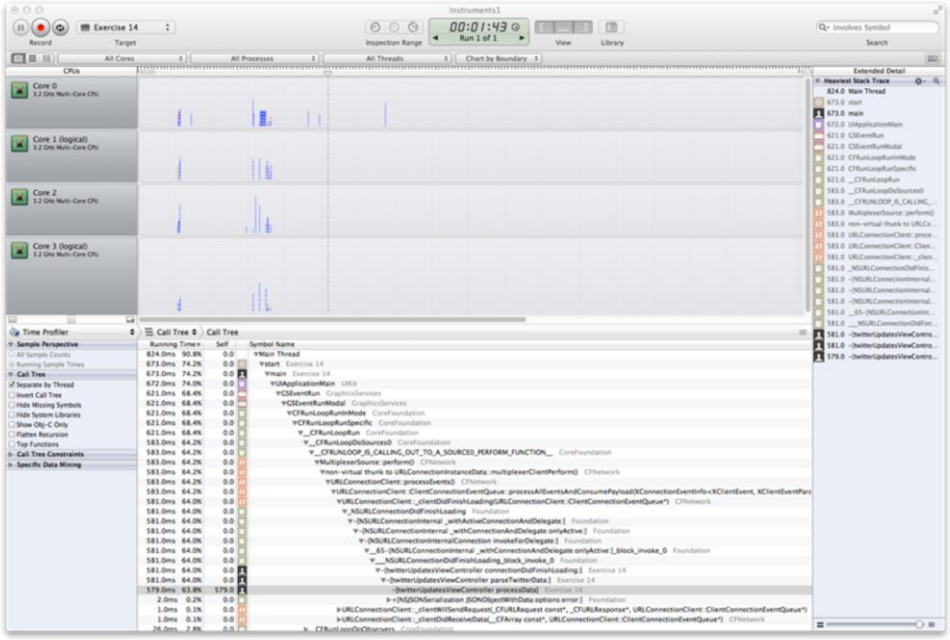
# Time Profiler – Instrument Strategy




279

Time Profiler allows you to carry out Call Tree analysis where you can drill down into the critical performance path to determine which is the method responsible for the majority of the CPU execution time.

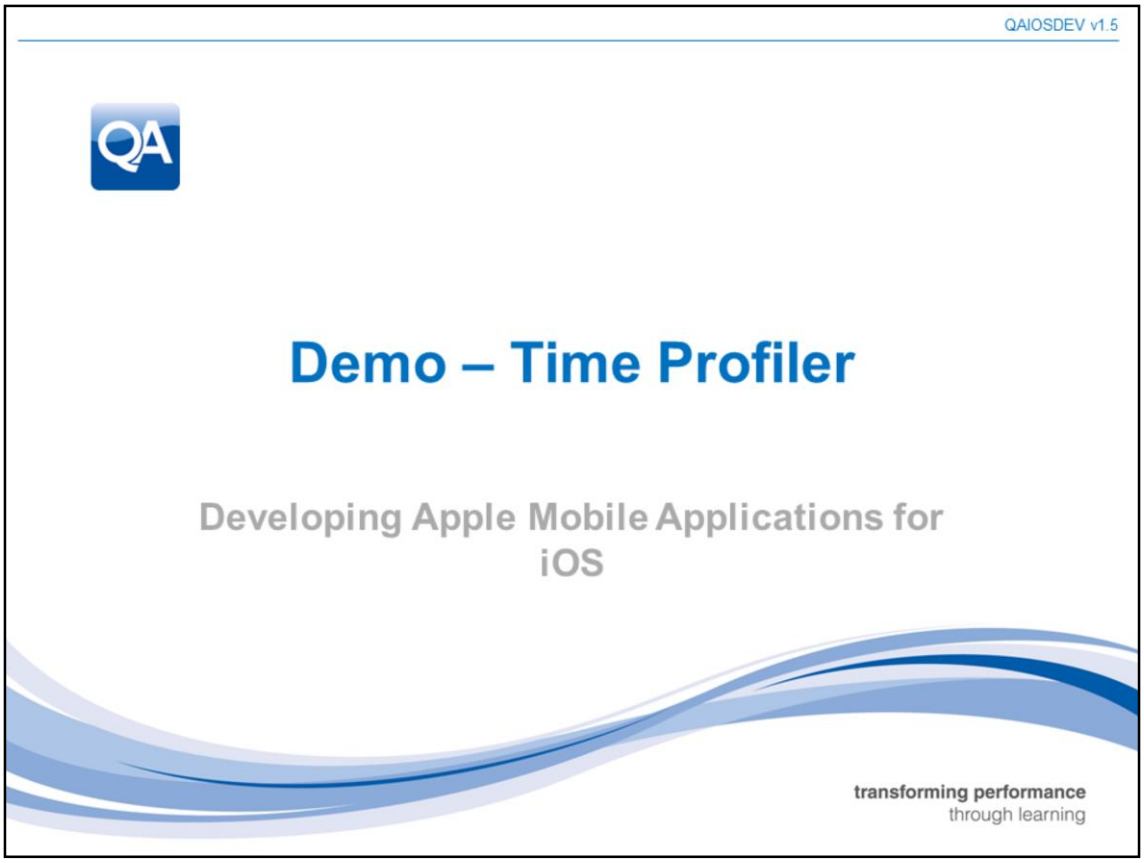
## Time Profiler – CPU Strategy





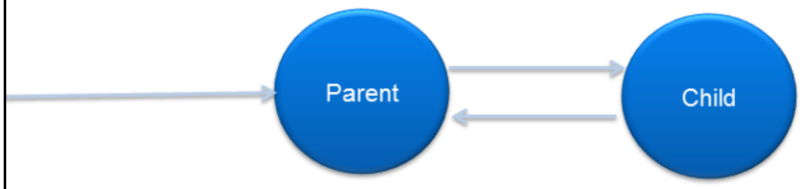
You can also look at how the CPU/cores have been exercised and how well you have achieved concurrency.

A profile displaying unbalanced core usage is a profile to look out for. Cores in heavy use whilst other cores are quiet indicate possible concurrency optimizations can be made.


QAIOSDEV v1.5
Demo – Time Profiler
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Memory Issues

- **Object Retention**
  - Objects loaded too early
  - Objects are retained for longer than necessary
  - Objects retained indefinitely
  - Increases the memory footprint of an app!
  - **Jetsam** will shutdown largest apps first
- **Memory Leak**
  - Objects left on the heap due to a missing release (Manual Retain Release)
  - Objects caught in a retain cycle






If you load objects too early, you increase the memory footprint of you app. That's not a good idea because iOS has a background process called **jetsam,** which, under memory pressure, looks for apps to shut down. It chooses apps with the the largest memory footprint first.

Loading too early, retaining for longer than necessary or indefinitely will increase your footprint.

The other memory issue is of course the leak. Failure to balance your alloc, new, copy, retain statements with your release statements in Manual Retain Release mode will cause leaks. Even in ARC, you will need to avoid retain cycles.

## Detecting memory issues

- **Leaks Template provides**
  - Allocation Instrument – Detects retained allocations
  - Leaks Instrument – Detects leaked allocations
- **Allocation Instrument**
  - Detects objects created, not yet destroyed
  - Heapshot can compare test runs to display differences!
  - Allocated instances can be traced to code
- **Leaks Instrument**
  - Highlights when a leak occurs
  - Leaked objects can be traced to code
- **Strategy**
  - Look for memory issue that grows larger with each test iteration
  - Small unintended retentions can be hard to see



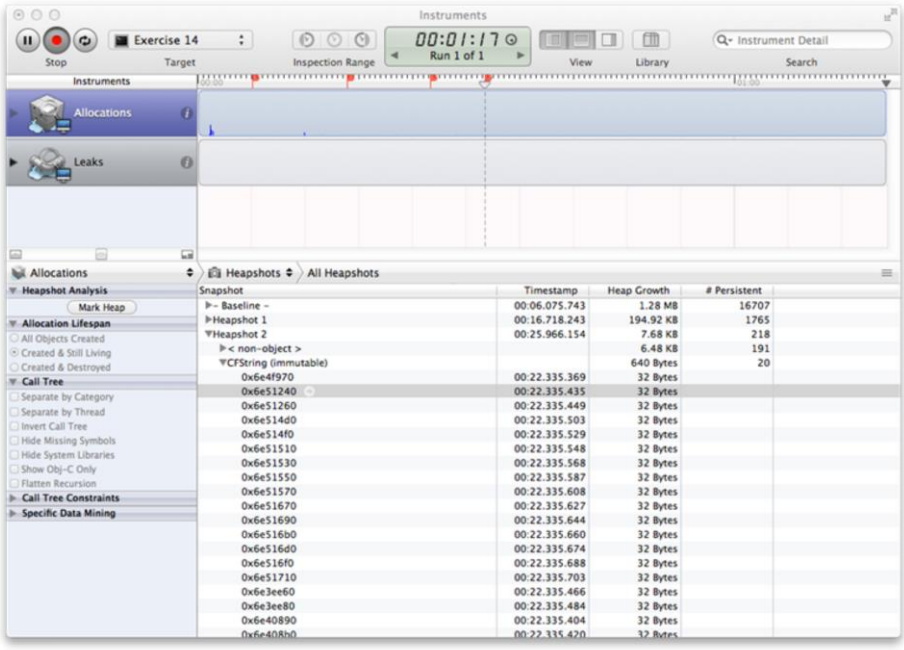
Instruments provides the Leaks templates that gives you both the allocations and leaks instruments.

The Leaks instrument will almost immediately highlight when a leak has occurred from where you can track the offending leak.

The Allocation instrument allows you to create Heapshots that are snapshots of the heap. Taken after a test, you can compare them and identify the differences. Those differences are likely to be the retained object instances.

Memory problems tend to grow larger with each test iteration. Identifying this worsening state finding the objects involved then tracing those objects back to where they were allocated is the key to solving the problem.

## Allocation and Leak Instruments




284

In the above example, a Heapshot is used to identify an allocated retained object between test runs. Double clicking on the object will highlight the location in code where the object was allocated.

## Memory Tips

- **Load Data when you need it**
- **Release memory as soon as you can – set it to nil!**
- **Don't load data in large "chunks" – filling a table view?**
  - Creates memory spikes
  - Load as required
- **Use ARC if you can**
- **Watch out for retain cycles**
  - Delegate properties
  - NSNotifications
  - Block references



Load memory at the point of it's use. Don't preload it first as you may do in other application types for efficiency. iOS apps literally live and die by their footprint.

Release memory when not in use and respond to low memory warnings.

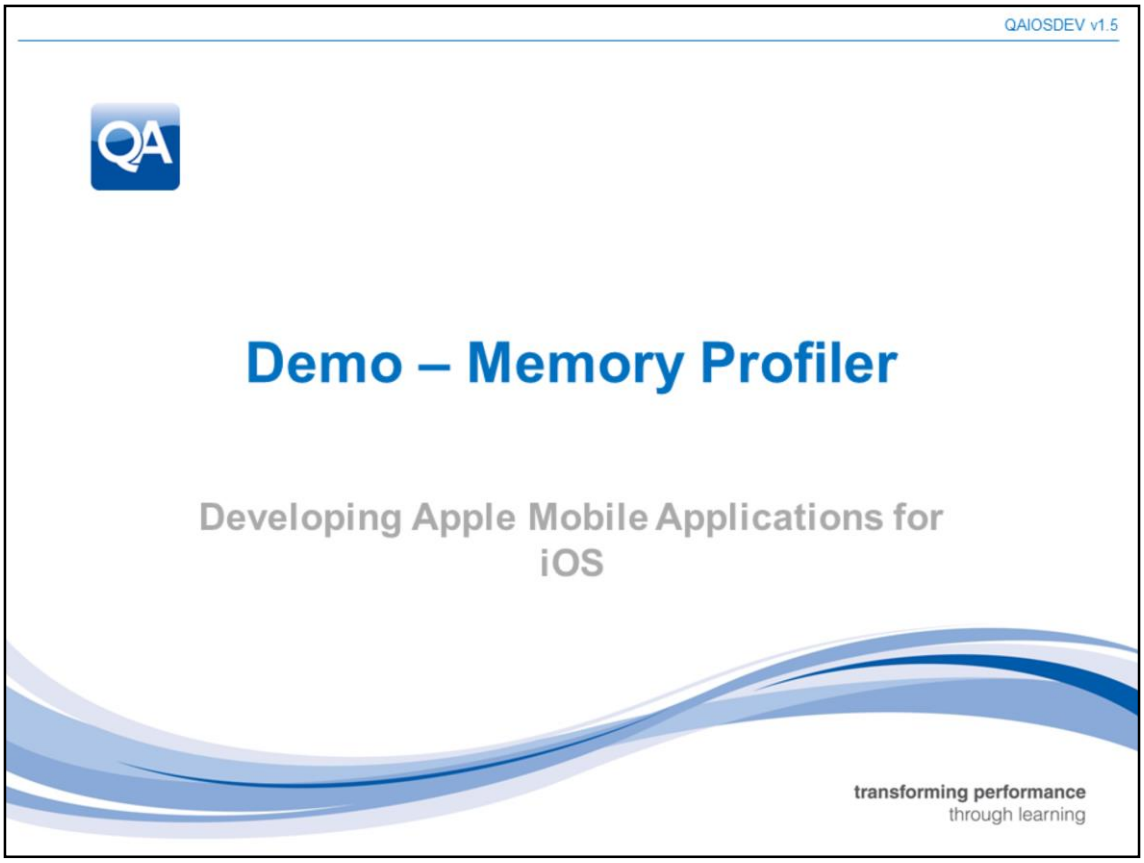
Load large data sets for example table views in pages chunks as required rather in one go.

Use ARC if you can is it is a provides more efficient memory control than you will achieve manually.

Avoid retain cycles.

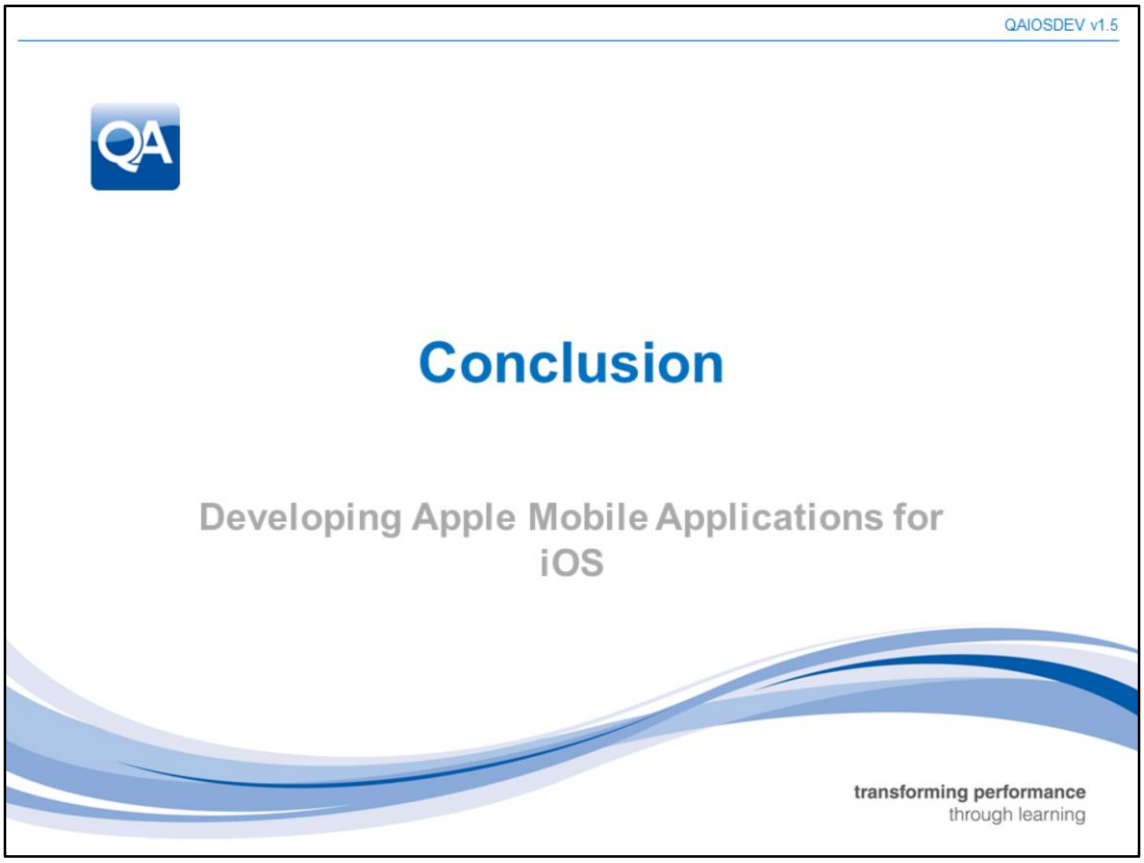
## Other key Instruments

- **Energy Usage**
  - Run Wirelessly to identify impact of your test cases on battery
- **Network Activity**
  - Identify Network access profile of app. Chatty app can impact performance and battery (bursting)
- **Disk Monitor**
  - Monitors Disk I/O
- **VM Tracker**
  - Dirty Resident memory <100Mg
- **Automation**
  - Record an Playback UI actions


QAIOSDEV v1.5
Demo – Memory Profiler
Developing Apple Mobile Applications for iOS
transforming performance
through learning

## Summary

- **Static Analysis**
  - First line of Defence
  - Potential Problems
- **Performance Profiling**
  - CPU Profiling – Time spent on CPU
  - CPU/Core Utilisation
- **Memory Profiling**
  - Object retention/Footprint Analysis
  - Leak Detection
  - Jetsam
- **Other Considerations**
  - Energy
  - Network I/O
  - Disk I/O


QAIOSDEV v1.5
Conclusion
Developing Apple Mobile Applications for iOS
transforming performance
through learning

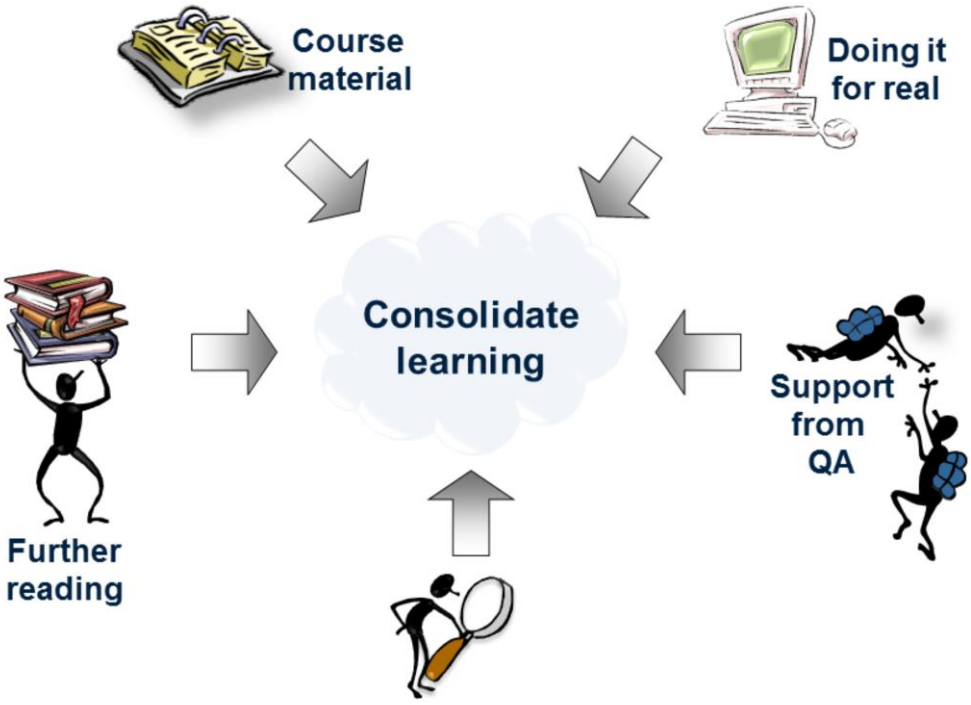
## Overview

- Checking objectives
- References
- What next?

## Review aims and objectives

- **We hope you will be able to do the following:**
  - Write an iPhone, iPad and universal app
  - Create a UI using just code
  - Use Interface Builder to create a UI
  - Work with the main user interface controls
  - Make effective use of View Controllers and Table Views
  - Create and use Data Stores
  - Work with Core Data and iCloud
  - Use multithreading techniques (indirectly)
  - Apply Animation
  - Access Cloud Data
  - Handle Notifications
  - Handle multitasking and background tasks
  - Analyse apps for performance and stability

## What next?


Course material
Doing it for real
Consolidate learning
Support from QA
Further reading

## Help from QA

- **Ever increasing range of courses**
  - Objective-C
  - C Programming
  - Advanced C
  - C++ For C Programmers
  - Object Oriented Design for C++
- **Web site**
  - www.qa.com
  - Social networking
  - Blogs

## In closing...


Remember to make a copy of your work to take away

Please complete your course evaluation

294