COMP327

Mobile Computing

Session: 2014-2015

Lecture Set 2 - iOS Basics

In these Slides...

- **We will cover...**
  - App Lifecycle and an introduction to UIKit
  - The Model-View-Controller (MVC) Pattern
  - View Fundamentals
  - Building Interfaces
    - Nib files and Outlets/Actions
  - Events and the Target-Action design pattern

Getting Started with iOS

These slides will allow you to develop a simple application, understand basic touch events, create bespoke views, and design interfaces using Xcode.

UNIVERSITY OF LIVERPOOL

App LifeCycle

Lecture Set 2 - iOS Basics

Anatomy of an Application

- An application is a bundle of files:
  - Compiled code
    - Your code
    - Frameworks
    - Info.plist configuration file
  - Storyboard and Nib files
    - UI elements and other objects
    - Details about object relationships
  - Resources (images, sounds, strings, etc)
    - Asset Libraries

UIKit Framework

- Defines the main layer of classes an application needs to construct and manage its interfaces
  - Supports underlying behaviours:
    - **UIResponder** defines interface and default behaviour for event-handling methods
  - This in turn defines the family of **UIView** and **UIViewController** classes
    - **Controls**, such as buttons, sliders, switches, pickers etc
    - **Modal Views**, such as action sheets or alert boxes
    - **Scroll Views**, for scrolling over larger areas
    - **Toolbars, Navigation Bars, split views, view controllers**, etc
  - UIGestureRecogniser defines single and multi-touch gestures
- Defines **UIApplication**, which orchestrates app lifecycle

Inside an iOS app

- Like C programs, Objective-C programs start in the **main()** function
  - In effect, the **`UIApplicationMain`** is the real main entry point !!!

- In an iOS app, the job of **main()** is simple:
  - Set up a core group of objects
  - Hand over control to those objects,

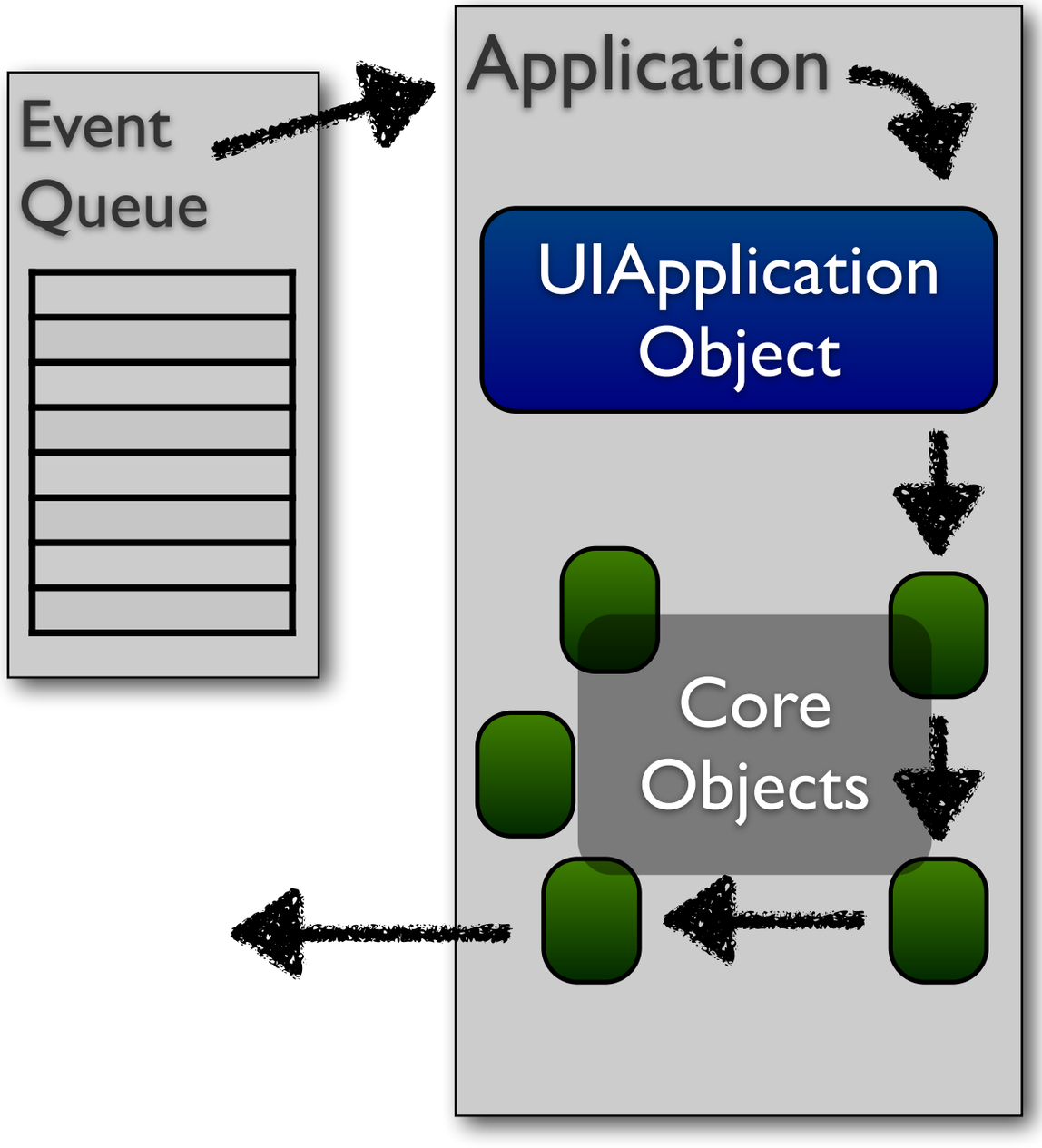
```
int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
                                 NSStringFromClass([AppDelegate class]));
    }
}
```

Set up the top level autorelease pool block to managed autoreleased memory objects.

Control of the application is passed to a UIApplicationMain singleton class object

UIApplication

- Every application has a single instance of UIApplication
  - **Singleton** design pattern
    - Only one instance exists, for your application
    - This instance can be easily accessed and queried to the app status
- Orchestrates the lifecycle of an application
  - Dispatches events
  - Manages status bar, application icon badge, etc
  - Rarely subclassed
    - Uses delegation instead



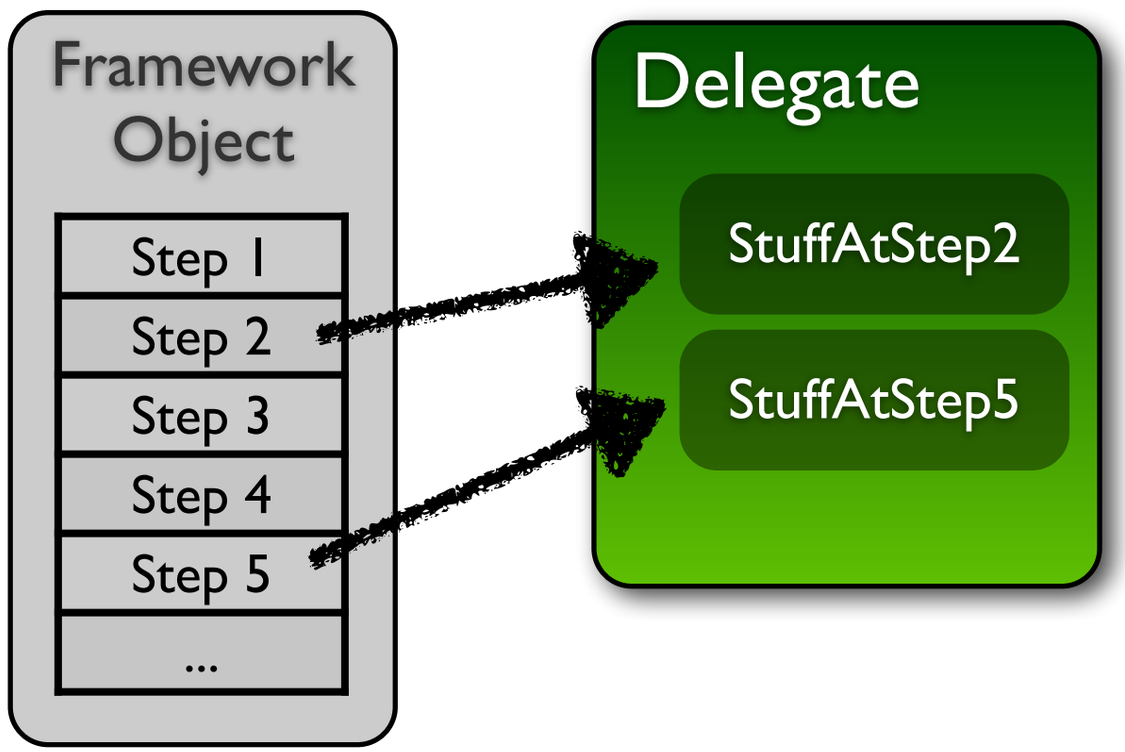
Delegation

- A design pattern where a host object embeds a pointer to another object (the delegate), an sends messages when input is required
  - Allows an "off-the-shelf" object to be extended without being subclassed
    - Messages are defined, which are called given certain events
    - The delegate provides the application specific behaviour for each event
  - Many UIKit classes use delegates
    - UIApplication
    - UITableView
    - UITextField

Protocols

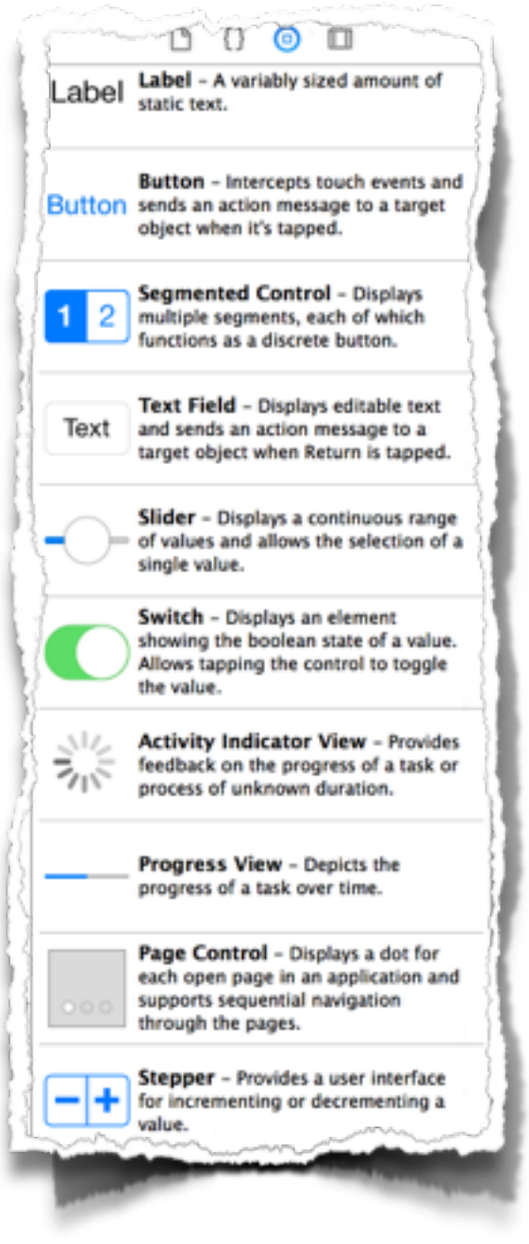
A **protocol** defines what messages the framework object may send the delegate.

We'll cover protocols later ...



View Controllers

- Views are rectangular area on screen, which handles events and displays some content
  - A view could be subclassed as in interface widget
    - such as a label, button or slider etc
  - A view could be subclassed and customised for specific behaviour
    - such as the drawing area in the SketchMe labs.
  - A view could also correspond to the a full "screen" of content
    - **These are typically managed by a viewController!**
- ViewControllers are special controllers that manage views
  - They form the basic building blocks for applications



UNIVERSITY OF LIVERPOOL

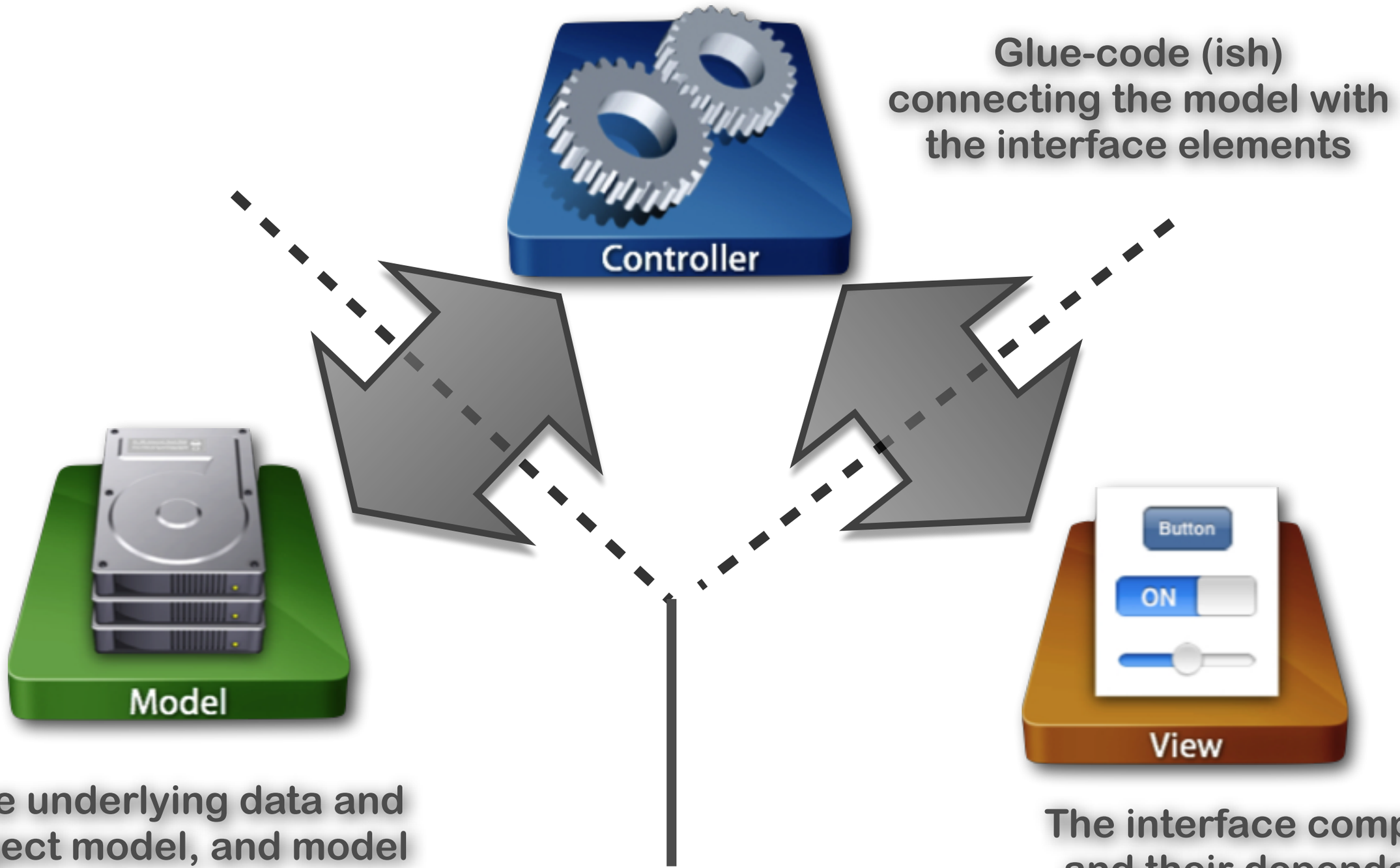
The MVC (Model View Controller) Pattern

Lecture Set 4 - iOS Basics

The Model, View, Controller (MVC) Design Pattern

- Used extensively within Cocoa Touch
  - A high level pattern from the days of Smalltalk
    - Manages the global architecture of the application
    - Classifies objects according to the roles they play
      - **Model Objects** encapsulate data and basic behaviour
      - **View Objects** present information to the user
      - **Controller objects** tie the model to the view
  - This pattern is frequently used in many other environments
- Can lead to reusability
  - By minimising dependencies, you can take a model or view class you've already written and use it elsewhere
  - Think of ways to write less code

Model, View, Controller



Glue-code (ish) connecting the model with the interface elements

The underlying data and object model, and model algorithms, all being interface independent

The interface components, and their dependency on each other

Model, View, Controller

- **Model**
  - **This contains your underlying data**
    - Could correspond to an external data source or some current model
      - iTunes database, stored files, internal state of a game (e.g. tic-tac-toe etc.)
    - Actions on the model manage the app data and its state
    - Not concerned with UI or presentation
      - Leave the interface to the view, and the application logic to the controller
      - Should be completely unaware of the controller or view
    - Same model should be reusable, unchanged in different interfaces
      - iPhone, cmd line, OSX app, etc

- Typically, this is the most reusable component

Model, **View**, Controller

- **View**
  - **This is what you see on the screen**
    - The canvas itself, and the interface elements such as buttons, labels, table views etc
    - It allows user to manipulate data, and displays data sent to it
  - Does not store any data
    - Use the model to maintain data
      - The typical exception is to cache state for optimisation
    - Updates to the model are done through the controller
  - Easily reusable & configurable to display different data
    - Often uses delegation to allow user to configure behaviour without subclassing

- Tends to be reusable - especially if well written

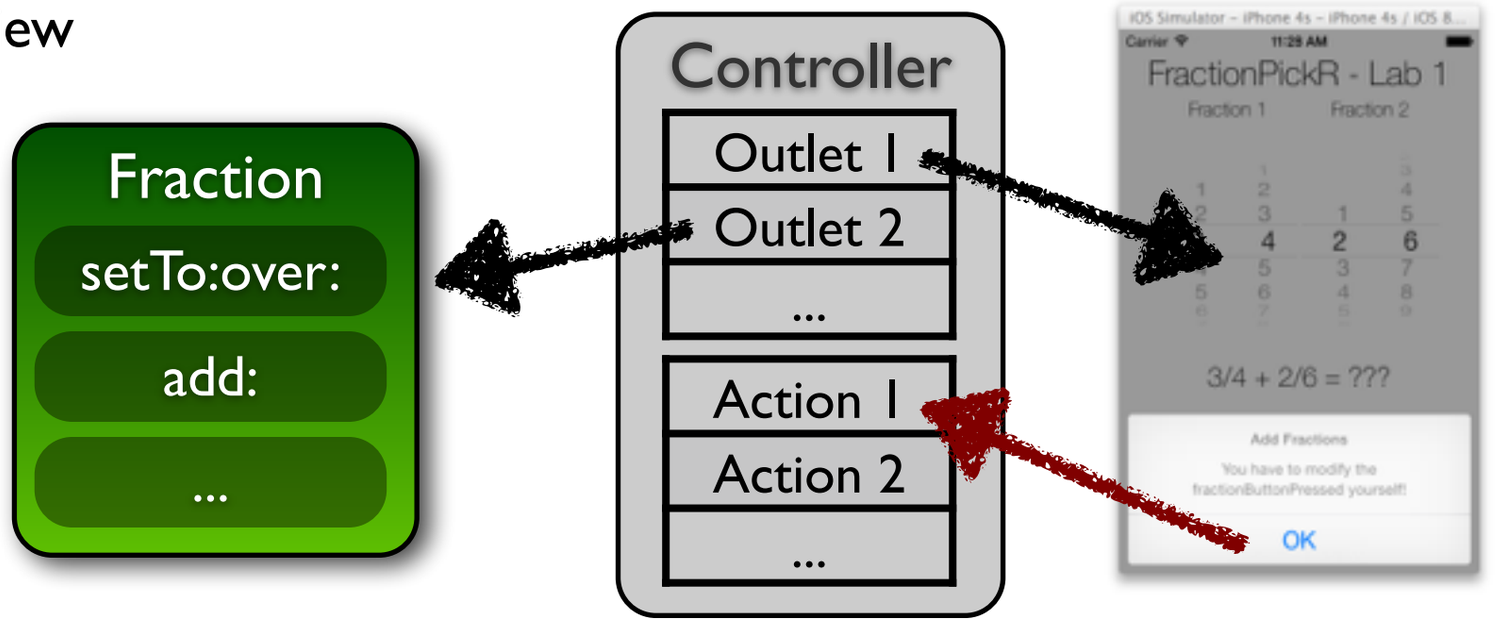
Model, View, **Controller**

- **Controller**
  - **This is the glue that defines what your app does**
    - Knows about both the view and the model
    - Acts as an intermediary between them
    - When the model changes, it lets the view know
    - When data is manipulated by the view or other events are triggered, it updates the model
  - iOS includes several controller types tailored to managing their own views:
    - ***UIViewController*** for managing apps with generic views
    - ***UITabBarController*** for tabbed applications (*e.g. clock*)
    - ***UINavigationController*** for managing hierarchical data (*e.g. email folders*)
    - ***UITableViewController*** for lists of data etc (*e.g. iTunes tracks*)

- Typically app specific, and thus rarely reusable

Model, View, Controller

- Controller typically exposes **outlets** and **actions** which can be used by the model and controller
  - Sometimes, a composite controller may be used
    - ViewControllers manage views, although the views themselves are separate objects
  - Models may actually be objects managed by the controller and shared with the view



UNIVERSITY OF LIVERPOOL

View Fundamentals and Building Interfaces

Lecture Set 2 - iOS Basics

View Fundamentals

- What is a **View** ???
  - A rectangular area on screen, which handles events and displays some content
  - Interface elements are subclasses of views
  - A view is a subclass of UIResponder

- Views are arranged hierarchically
  - every view has one **superview**
  - every view has zero or more **subviews**
  - Views live inside of a window (UIWindow)

- One **UIWindow** for an iPhone app
  - Contains the entire view hierarchy
  - Set up by default in Xcode template project

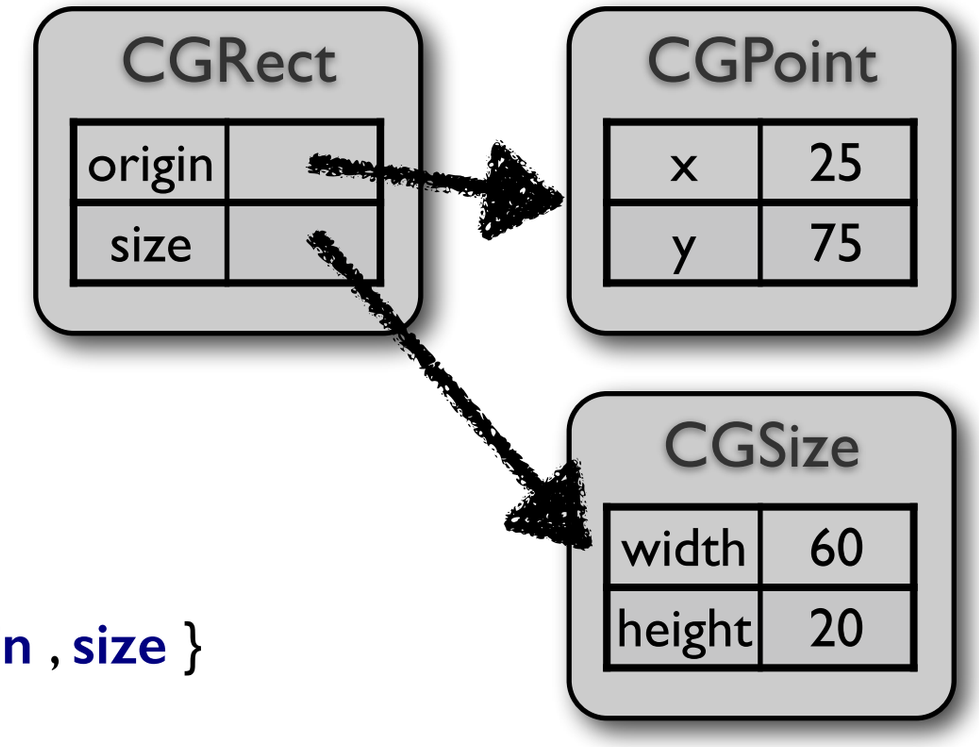
View Hierarchy - Ownership

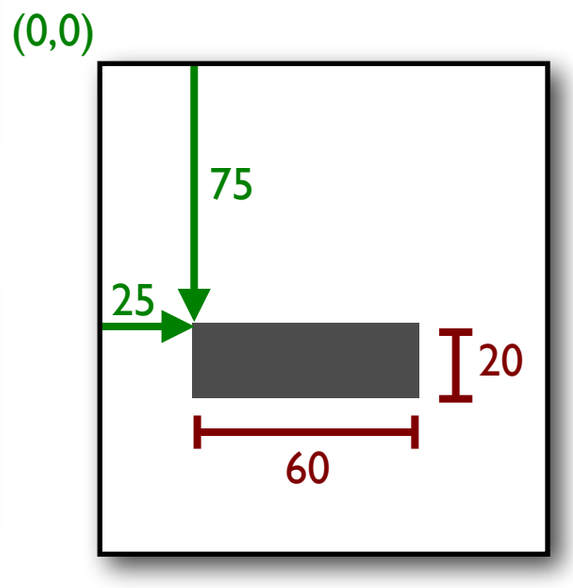
- Superviews retain their subviews
- Not uncommon for views to only be retained by superview
  - Be careful when removing a view!
    - It might have subviews you might want to keep!
  - Retain subview before removing if you want to reuse it
- Views can be temporarily hidden

```
[theView setHidden:YES];
```

View Related Structures

- CGPoint
  - location in space: { **x** , **y** }
- CGSize
  - dimensions: { **width** , **height** }
- CGRect
  - location and dimension: { **origin** , **size** }





Important

These structures are actually C structs, not Objective-C objects!!!

Creation Function	Example

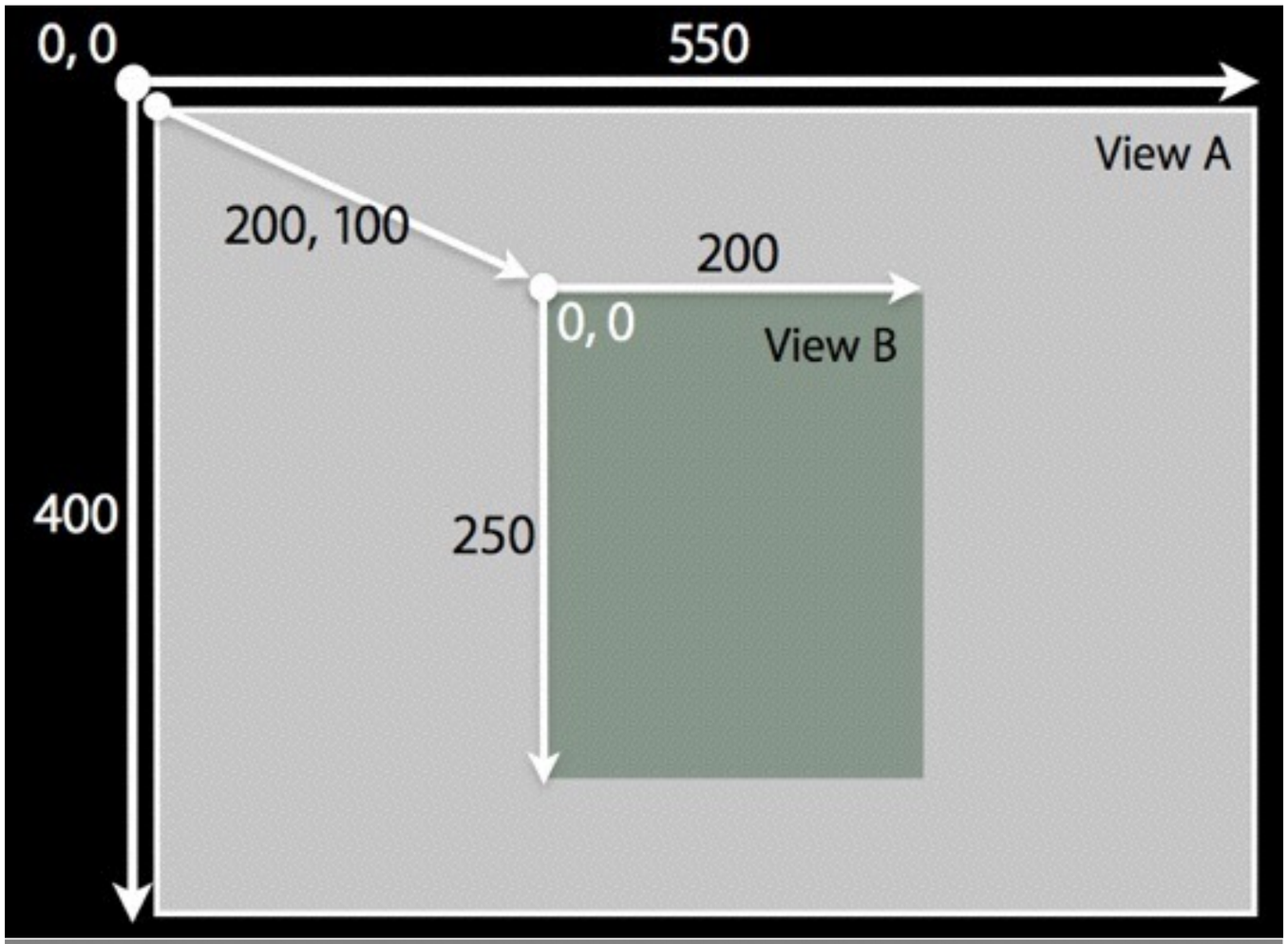
`CGPointMake (x, y)`	`CGPoint point = CGPointMake (25.0, 75.0);`
	`point.x = 25.0;`
	`point.y = 75.0;`

`CGSizeMake (width, height)`	`CGSize size = CGSizeMake (60.0, 20.0);`
	`size.width = 60.0;`
	`size.height = 20.0;`

`CGRectMake (x, y, width, height)`	`CGRect rect = CGRectMake (25.0, 75.0, 60.0, 20.0);`
	`rect.origin.x = 25.0;`
	`rect.size.width = 60.0;`

Location and Size

- Origin is in the top left corner - with y axis growing downwards
- View's location and size expressed in two ways
  - **Frame** is in superview's coordinate system
  - **Bounds** is in local coordinate system



View A frame:
origin: 0, 0
size: 550 x 400

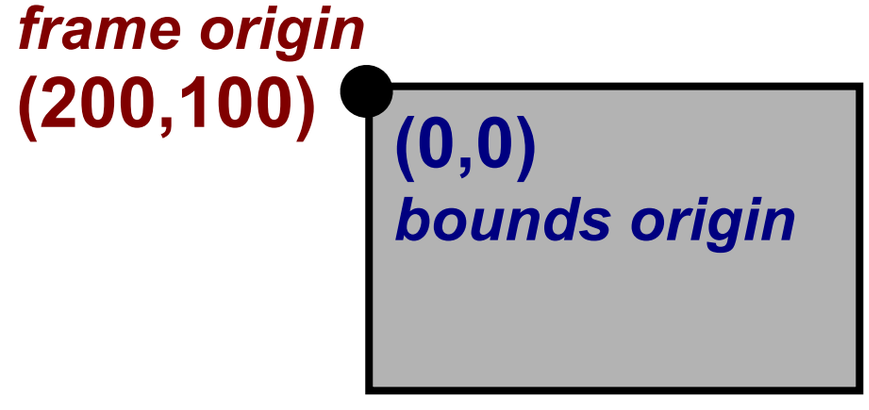
View A bounds
origin: 0, 0
size: 550 x 400

View B frame:
origin: 200, 100
size: 200 x 250

View B bounds
origin: 0, 0
size: 200 x 250

Frame and Bounds

- Which to use?
  - Usually depends on the context
- If you are ***using a view***, typically you use frame
- If you are ***implementing a view***, typically you use bounds
- Matter of perspective
  - From outside it's usually the frame
  - From inside it's usually the bounds



- Examples:
  - Creating a view, positioning a view in superview - use frame
  - Handling events, drawing a view - use bounds

Drawing Views

- **- (void)drawRect:(CGRect)*rect***
  - **-[UIView drawRect:]** does nothing by default
  - If not overridden, then backgroundColor is used to fill
- Override **- drawRect**: to draw a custom view
  - ***rect*** argument is area to draw
- **drawRect**: is invoked automatically
  - Don't call it directly! Let the framework decide
  - When the app **thinks** the view needs to be redrawn, use:

```
- (void)setNeedsDisplay;
```

- For example, in your controller:

```
- (IBAction)changeLineWidth:(id)sender{

    lineWidth = [lineWidthSlider value];
    [penView setPenValues:lineWidth];

    [penView setNeedsDisplay];
}
```

Being lazy is good for performance

When the framework needs to render the views, it determines which are visible, and only draws those. For each view that should be redrawn, its drawRect method will be called.

This can significantly improve performance!!!

Graphics Context

- All drawing is done into an opaque graphics context
  - **Normally set up automatically, so no need to set this up explicitly**
- Graphics context setup automatically before invoking drawRect:
  - Defines current path, line width, transform, etc.
  - Access the graphics context within drawRect: by calling

    `(CGContextRef)UIGraphicsGetCurrentContext(void);`
  - Use CG calls to change settings such as colour, line width, etc
- Context only valid for current call to drawRect:
  - Do not cache a CGContext!

```
- (void)drawRect:(CGRect)rect {
    CGRect bounds = [self bounds];

    // Get the current graphic context
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextSetLineWidth(context, (2*myLineWidth));

    CGContextBeginPath(context);
    CGContextMoveToPoint(context, (bounds.size.width-1)/2, bounds.size.height/2);
    CGContextAddLineToPoint(context, (bounds.size.width-1)/2, bounds.size.height/2);
    CGContextStrokePath(context);
}
```

Drawing in views

- UIKit offers very basic drawing functionality

```
UIRectFill(CGRect rect); // filling the rectangle
UIRectFrame(CGRect rect); // stroking the rectangle
```

- CoreGraphics - Drawing APIs
  - CG is a C-based API, not Objective-C
- CG and Quartz 2D drawing engine define simple but powerful graphics primitives, including:
  - Graphics context
  - Transformations
  - Paths
  - Colours
  - Fonts
  - Painting operations

Setting the App's Main View

- All view controllers have a view
  - UIViewController instances have a view property
    - represents the root view for the view controller's hierarchy
- An App needs to know what view is its root view of the view hierarchy
  - Normally the view of the main (root) view Controller
  - This is typically determined within the App Delegate method

```
- (BOOL)application: didFinishLaunchingWithOptions:
```

  - This tells the delegate that the application has launched, and it is about to be moved into the "running" state

- **Pre iOS 4.0**
  - The *view* should be added as a subview to the window in this method, and then the window should become the key window.

```
[window addSubview:[myVC view]];
[window makeKeyAndVisible];
```

    - App delegates have the property **`window`**, which is of type UIWindow
    - The subview is added to this window

- **Post iOS 4.0**
  - The *view controller* should be added as the root view controller of the window, prior to making the window the key window.

```
[[self window] setRootViewController:myVC];
[[self window] makeKeyAndVisible];
```

    - As properties should now be accessed through getters/setters, we obtain **`_window`** using the object's **`window`** method.

The App Delegate

- This is the main class that defines what the app will do at certain stages of the app lifecycle, for example:
  - once the application has launched
  - when the app goes into the background
  - when the app is about to terminate

The method `application:didFinishLaunchingWithOptions:` is called once the application has started launching. This is typically where the view hierarchy is initialised.

The app's window is created, based on the size of the device's screen.

```
- (BOOL)application:(UIApplication *)application
                    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

The method `makeKeyAndVisible` causes it's owner (the window) to be loaded, which in turn causes any other views attached to it to be loaded.

To create your own view, create a View Controller and make it the root view controller **here** in the code.

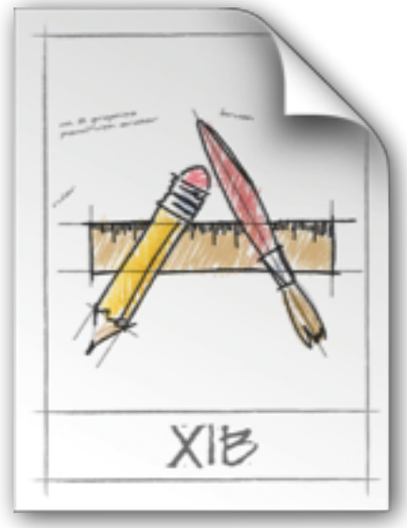
App Delegate Source File:

Here, we add code after the "override" point as we are not explicitly using Storyboards.
A new view controller is created, and used to define the root View Controller.

```
//  AppDelegate.m
#import "AppDelegate.h"
#import "MyWelcomeVC.h"

@implementation AppDelegate

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

    // ============================================================
    // Override point for customization after application launch.
    MyWelcomeVC *vc = [[MyWelcomeVC alloc] initWithNibName:@"MyWelcomeVC" bundle:nil];
    [[self window] setRootViewController:vc];
    // ============================================================

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

Note: in earlier versions of iOS and Xcode (pre iOS6 and Xcode 4.5), it was common to simply add the view controller's view as a subview to the window. However, this now generates a warning by the compiler.

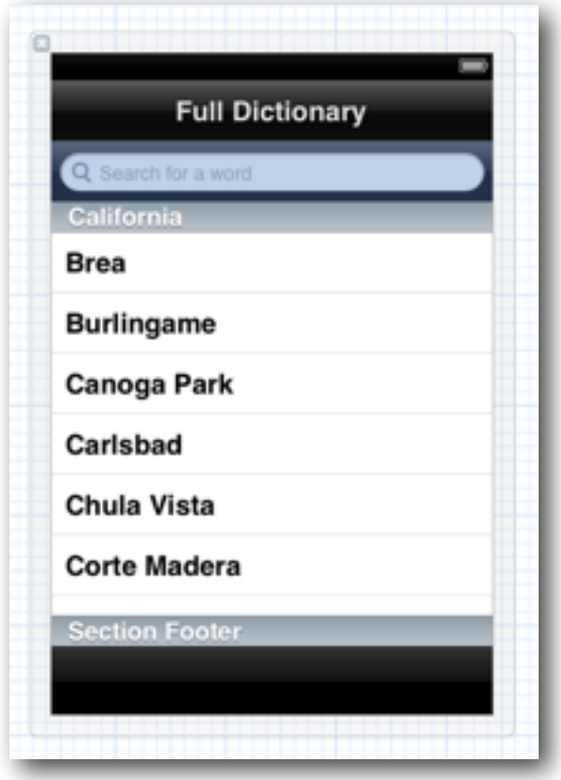
Nib/Xib Files (pre Storyboards)



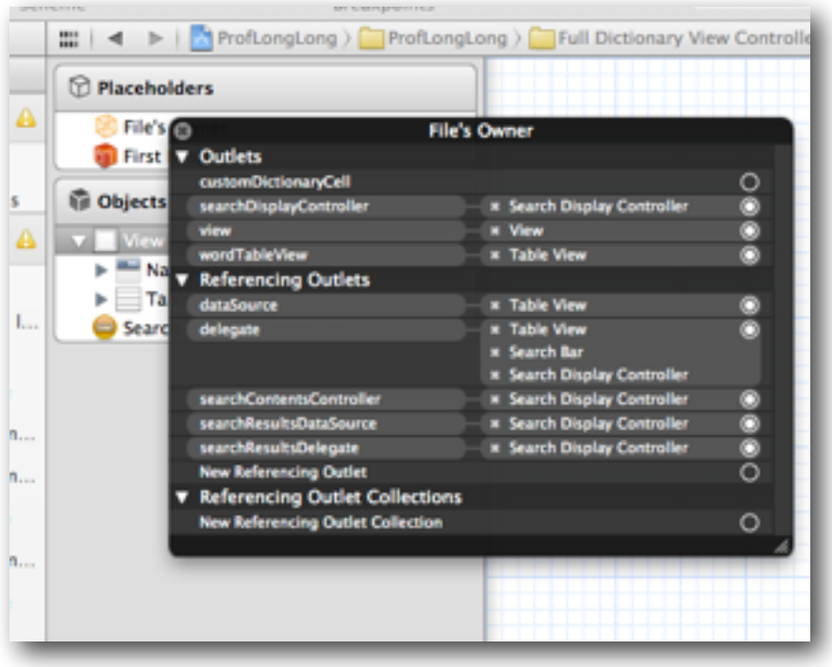
- Helps you design the 'V' in MVC:
  - layout user interface elements
  - add controller objects
  - Connect the controller and UI
- At runtime, objects are unarchived
  - Values/settings in Interface Builder are restored
  - Ensures all outlets and actions are connected
  - Order of unarchiving is not defined

Getting objects from the Nib

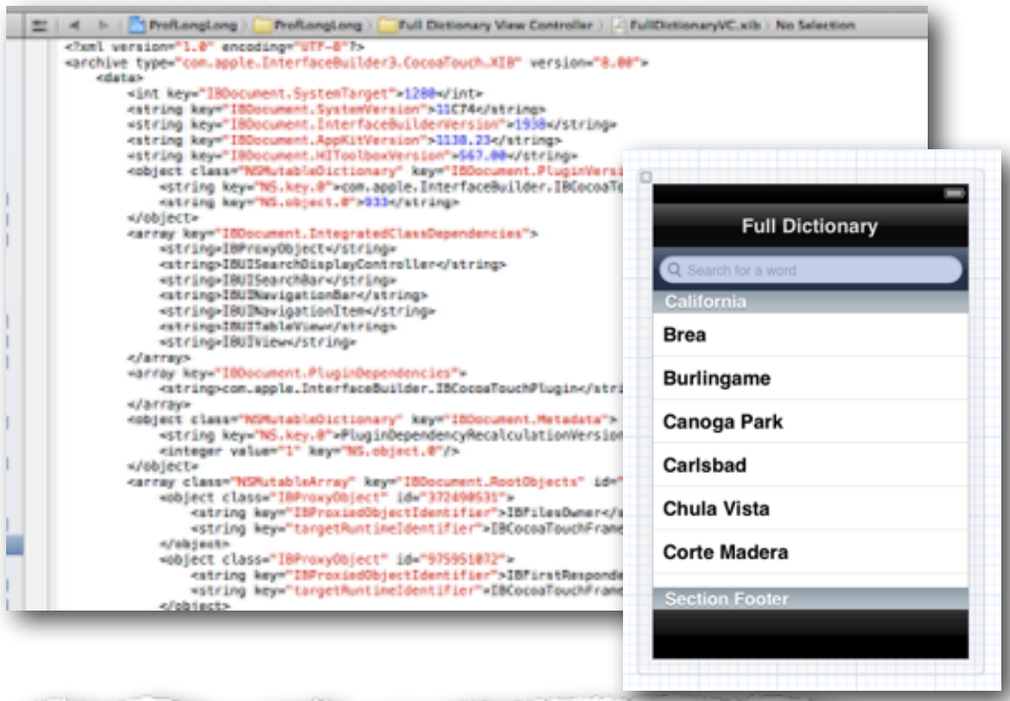
- When the nib is created in interface builder, you attach these to the properties in the associated view or controller
  - File's Owner in the nib corresponds to the object that owns the nib
  - IBOutlet and IBAction properties defined in the object owned by the nib are exposed in InterfaceBuilder
  - Links are made from these properties to the objects that are created by the nib



- Hence - object creation does not appear in any source code, but access to the objects is provided by the IBOutlet properties



From Xib to Nib



1) .xib files are xml-based source that define various objects and their properties.

2) When they are compiled, the objects are actually created in memory, and their property values are set.

Compiler

3) The memory objects are then serialised (or marshalled) and stored as .nib files, within the app bundle.

4) When the application runs, the nib files can be loaded into memory to create the objects directly, rather than having to be alloc/init'd at runtime.

IBOutlets

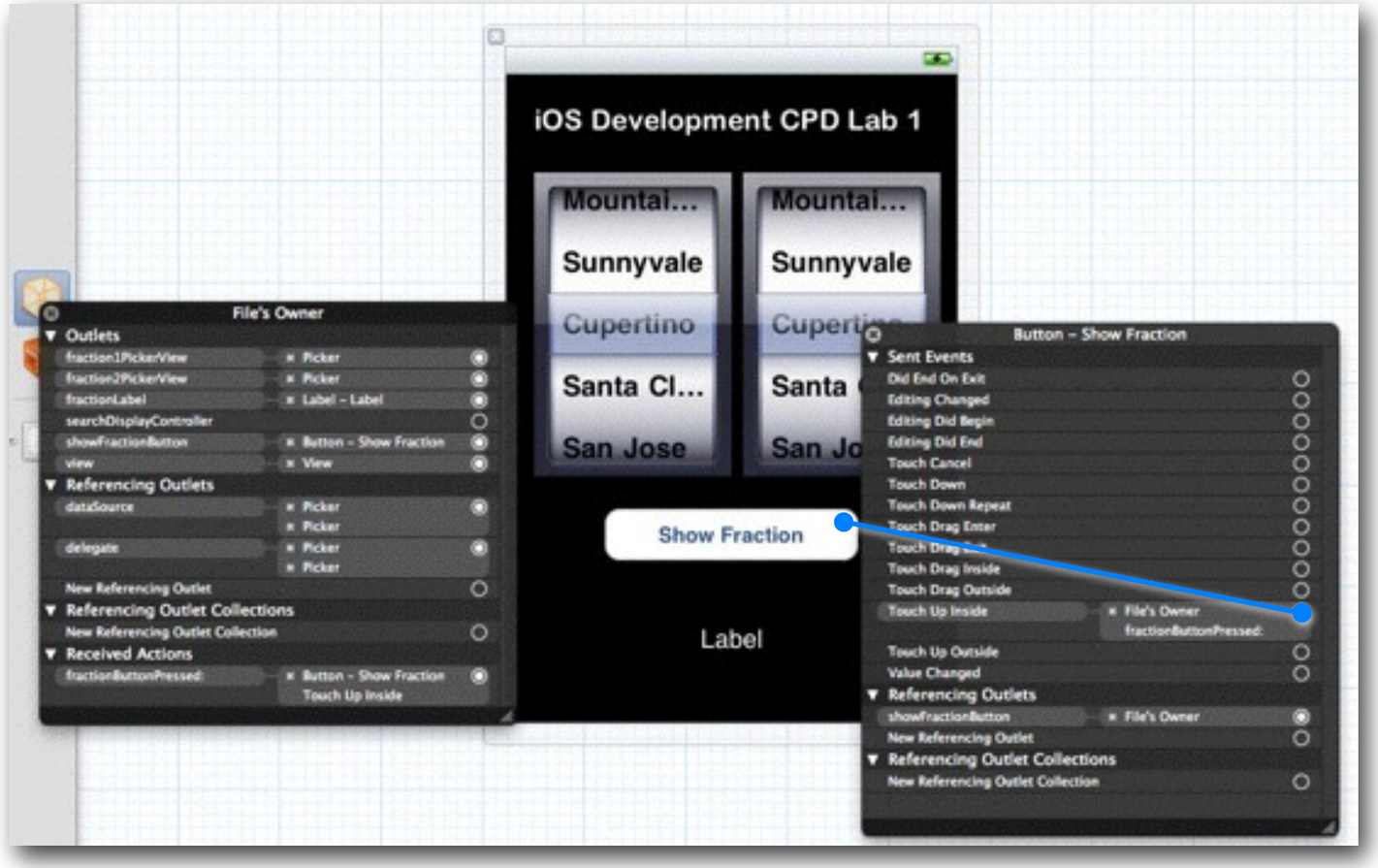
- IBOutlets are properties of a view controller that are exposed to the Interface builder.
  - Objects are created in the Interface Builder
    - Memory is alloc'd for these objects, but they still need to be managed
    - By "linking" objects in the Interface Builder with the exposed setters of the view controller (known as the File's Owner, the controller takes control of those objects
      - This is the reason why many properties to UI objects use the retain modifier on the properties
- IBOutlets do **nothing** programmatically
  - #define IBOutlet

```
@interface FractionPickerViewController :
UIViewController

// Property Definitions
@property (nonatomic, retain) IBOutlet UIPickerView
*fraction1PickerView;
@property (nonatomic, retain) IBOutlet UIPickerView
*fraction2PickerView;
@property (nonatomic, retain) IBOutlet UILabel
*fractionLabel;
@property (nonatomic, retain) IBOutlet UIButton
*showFractionButton;
```

IBActions

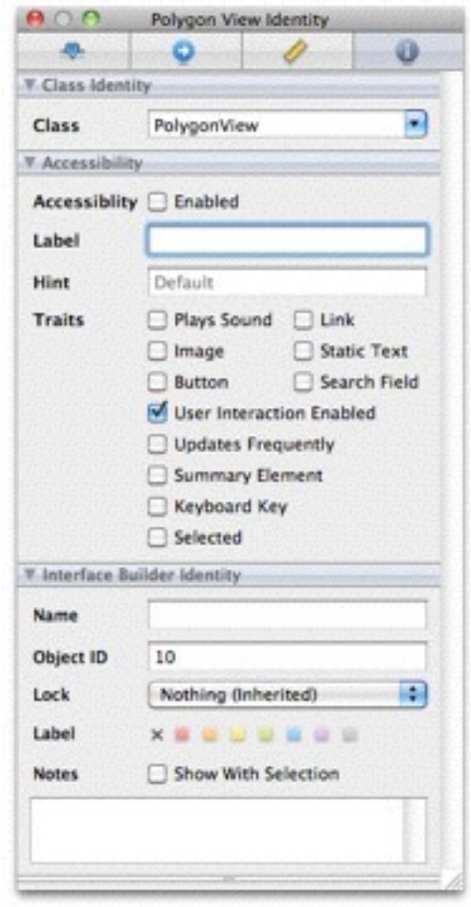
- IBActions identify methods provided by a view controller that can be called when some action occurs within the interface
- These are exposed to the Interface builder, and can be linked to actions performed by an interface object.
  - Events will be generated by an interface object.
    - Those events that are valid for the object can then be linked back to the IBAction exposed by the File's Owner
- IBActions return void
  - There is nothing to return a value to
  - #define IBAction void

```
@interface FractionPickerViewController :
UIViewController {
    ...
}
- (IBAction)fractionButtonPressed:(id)sender;

// ------------

@implementation FractionPickerViewController

- (IBAction)fractionButtonPressed:(id)sender{
    ...
}
```



Defining the size and location of view objects

- **Using NIB files**
  - Drag out any of the existing view objects (buttons, labels, etc) and create desired size
  - Or drag generic UIView and set custom class



- **Programatically**
  - Views are initialised using -initWithFrame:

```
CGRect frame = CGRectMake(0, 0, 200, 150);
UIView *myView = [[UIView alloc] initWithFrame:frame];
```

  - Example:

```
CGRect frame = CGRectMake(20, 45, 140, 21);
UILabel *label = [[UILabel alloc] initWithFrame:frame];

[view addSubview:label];
[label setText:@"Hello World"];
[label release]; // label now retained by window (MRR)
```

Once the view has loaded

- If loading the nib automatically creates objects and order is undefined, how do I customize?
  - For example, to displaying initial state?

- -viewDidLoad
  - Control point to implement any additional logic after the view loading
  - You often implement it in your controller class
    - e.g. to restore previously saved application state
  - At this point, everything has been unarchived from nib, and all property connections have been made

UNIVERSITY OF LIVERPOOL

Events and the Target-Action Design Pattern

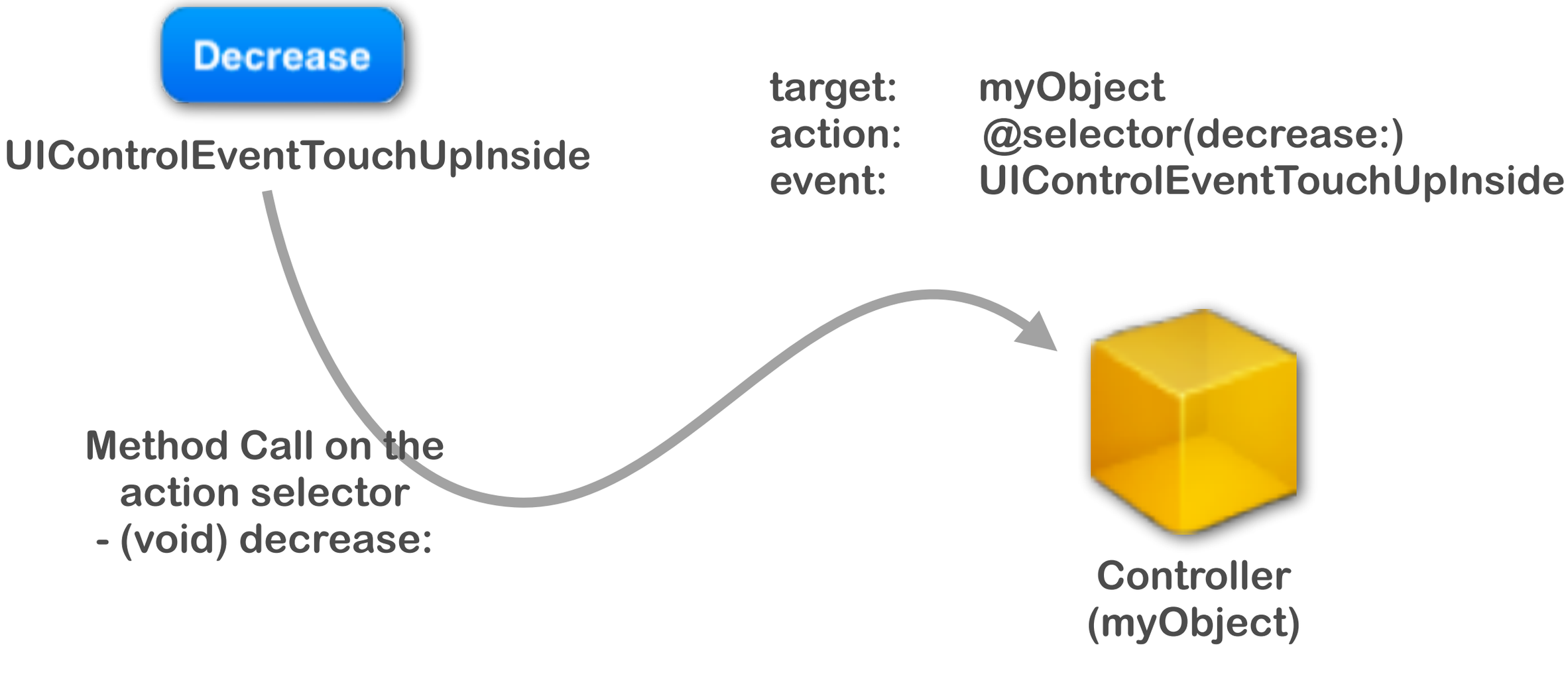
Lecture Set 2 - iOS Basics

Controls - Events

- View objects that allows users to initiate some type of action
- Respond to variety of events
  - Touch events
    - touchDown
    - touchDragged (entered, exited, drag inside, drag outside)
    - touchUp (inside, outside)
  - Value changed
  - Editing events
    - editing began
    - editing changed
    - editing ended

Controls - Target / Action

- When an event occurs, an action is invoked on the target object



Action Methods

- 3 different flavours of action method selector types

```
(void)actionMethod;
```

```
(void)actionMethod:(id)sender;
```

```
(void)actionMethod:(id)sender withEvent:(UIEvent *)event;
```

  - **`(id)sender`** corresponds to the object that created the event
  - **`(UIEvent *)event`** contains details about the event that took place

- Controls can trigger multiple actions on different targets in response to the same event
  - Different events can be set up in the Interface Builder

Action Method Variations

- Simple no-argument selector

```
- (void)increase {
    // Increase the Drawing Line Width
    [lineObj setWidth:[lineObj width] + 1];
}
```

- Single argument selector - control is ‘sender’
  - May need to interrogate the control for additional information

```
// for example, if control is a slider...

- (void)adjustLineWidth:(id)sender {
    [lineObj setWidth:[sender value]];
}
```

Action Method Variations

- Two-arguments in selector (sender & event)
  - UIEvent contains details about the event that took place

```
- (void)adjustLineWidth:(id)sender
              withEvent:(UIEvent *)event
{
    // may want to perform one action if the was a
    // touch-up event, and a *different* action if
    // a drag or edit event occurred.

    // sender allows you to interrogate the interface element
    // event allows you to ask about the event that occurred

}
```

Creating Target-Actions programmatically

- Target Actions can be created using various methods
  - Typically, you would use addTarget:action:forControlEvents:
    - **`addTarget:(id) target`** - corresponds to the instance that receives the message
    - **`action:(SEL)action`** - corresponds to the method selector that should be sent
    - **`forControlEvents:(UIControlEvents)controlEvents`** - the event type

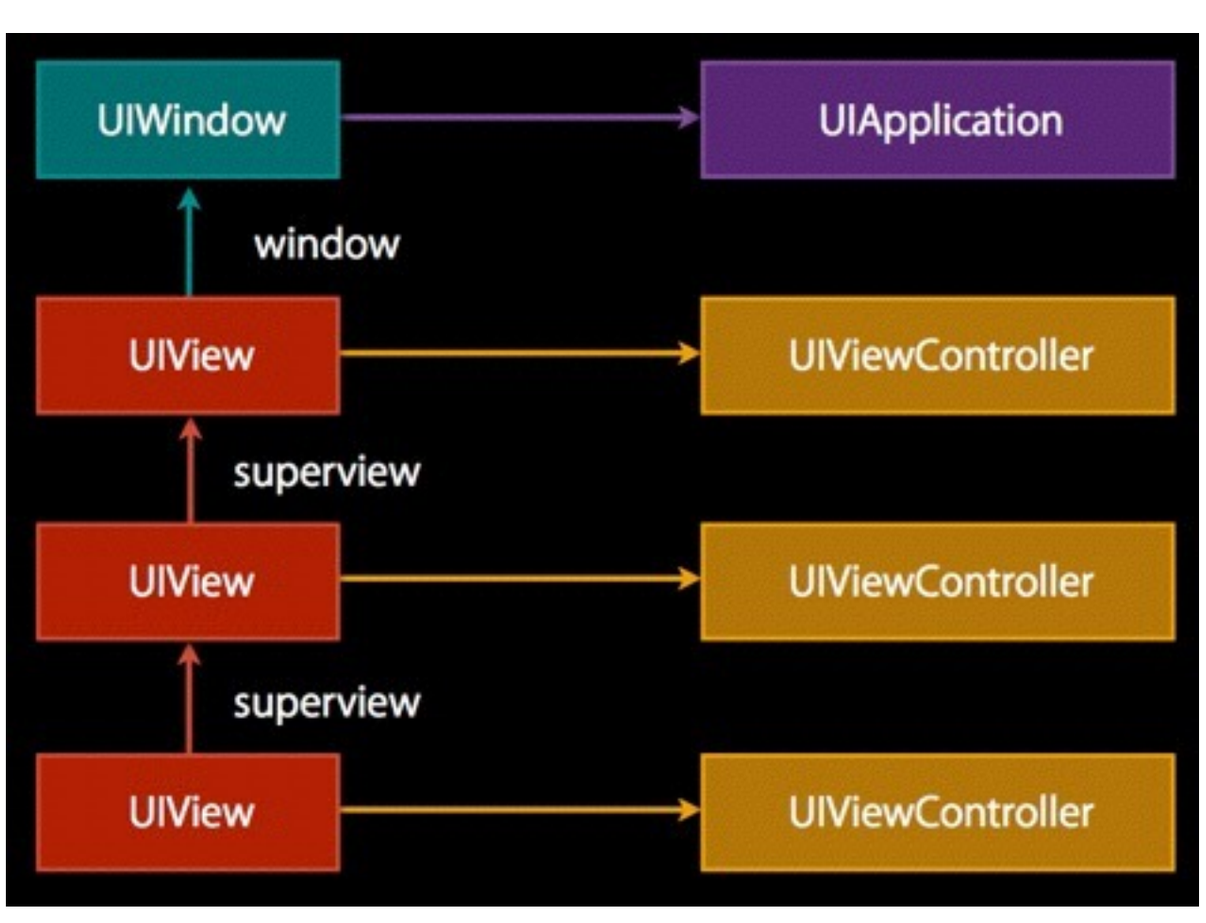
```
- (void)decrease:(id)sender {
    // Note that we don't use sender here, but illustrate a method with one arg.
    myCount -= 1;
}

- (void)setUpButton {
    UIButton *button = [UIButton buttonWithType:UIButtonTypeRoundedRect];

    [button addTarget:self
               action:@selector(decrease:)
     forControlEvents: UIControlEventTouchUpInside];

    [button setTitle:@"Decrease" forState:UIControlStateNormal];
    button.frame = CGRectMake(80.0, 210.0, 160.0, 40.0);
    [view addSubview:button];
}
```

Touch Events

- Handling touch events requires the definition of four methods to manage the different phases of a touch action
  - - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
    - This is called when a touch event is discovered
  - - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
    - This is called when a touch event moves around the view
  - - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
    - This is called when the finger is removed from the display
  - - (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
    - This is called if there is some cancel action, e.g. a call comes in, etc

Touch Events

- It is important to override all these methods...
  - ... **Even if they do nothing!!!**
- Events propagate from the lowest view up, until methods are found
  - However, a class may handle state depending on the different events
  - Hence, all the methods should be handled together. Avoid some events propagating up, when others stay at a lower level!!!



Example code for touchesBegan

1) Extract a single touch event from the touches set.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

    // We only support single touches, so anyObject retrieves just that touch from touches
    UITouch *touch = [touches anyObject];

    UIView *myView = [self mySketchView];

    if ([touch view] == myView) {
        // only process touches occurred in this view only

        // get the touch coordinate wrt this view
        CGPoint touchPoint = [touch locationInView:myView];
        NSString *coordinateMsg = [NSString stringWithFormat:@"Coord: %.0f, %.0f",
                                   touchPoint.x, touchPoint.y];
        NSLog(coordinateMsg);
    }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {

    // Do something similar to touchesBegan
}
```

2) Find out in which view the touch began (**not** where it is now), and if it began in the sketchView, then handle it.

3) Calculate the position of the touch **with respect to the sketchView view**, and display the coordinates.

Example code for touchesEnd or touchesCancelled

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
        NSLog([NSString stringWithString:@"Touch Ended"]);
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
        // Do nothing
}
```

- Remember - it is important to override all these methods...
  - **... Even if they do nothing!!!**

UNIVERSITY OF LIVERPOOL

Questions?

Lecture Set 2 - iOS Basics