ES3 Lecture 4

iPhone development: Resource management, libraries and sensors

Containers: Mutable and Immutable

- Containers can either be **immutable** (cannot change, insert or remove items after creation) or **mutable** (change after creation)

- Mutable versions inherit from immutable ones
  - All built in containers have a mutable and immutable version
  - **Because of the inheritance, any method taking an immutable collection can take a mutable collection in its place**

- Immutable versions have a performance benefit

Basic Containers

- Ordered arrays (roughly like Java vectors):
  - NSArray, NSMutableArray
  - Can slice and enumerate. Mutable arrays can have objects removed and inserted
  - Key methods:
    - **filteredArrayUsingPredicate** -- returns array of elements where predicate is true
    - **objectEnumerator** -- returns an enumerator
    - **count** -- returns size of array
    - **objectAtIndex** -- gets a specific object
    - **reverseObjectEnumerator** -- reads the array backwards!
    - **indexOfObject** -- searches for an object and returns its index
    - **makeObjectsPerformSelector** -- applies a function to an array
    - **sortedArrayUsingFunction:context** -- sorts an array
    - **arrayWithObjects** -- creates a new array with a list of objects

Basic Containers

- Sets (unordered collections):
  - NSSet, NSMutableSet
  - Similar to arrays, but no indices or reverse enumerator

- Dictionaries (hash tables, associative arrays)
  - NSDictionary, NSMutableDictionary
    - keyEnumerator, objectEnumerator -- iterate over keys or values
    - setObject:forKey -- inserts/replaces an object
    - objectForKey -- gets an object given a key

Enumeration

- Generally, NSEnumerator used to iterate through objects
  - Idiom goes like this:

```
NSArray *array = [NSArray arrayWithObjects:first,second,nil];
NSEnumerator *arrayEnumerator = [array objectEnumerator];
id value;
while(value=[arrayEnumerator nextObject])
{
 // do something with value
}
```

- If an object implements the NSFastEnumerator protocol (the built in containers do), you can do the much more elegant:

```
for(id value in array)
{
 // do something with value
}
```

An aside

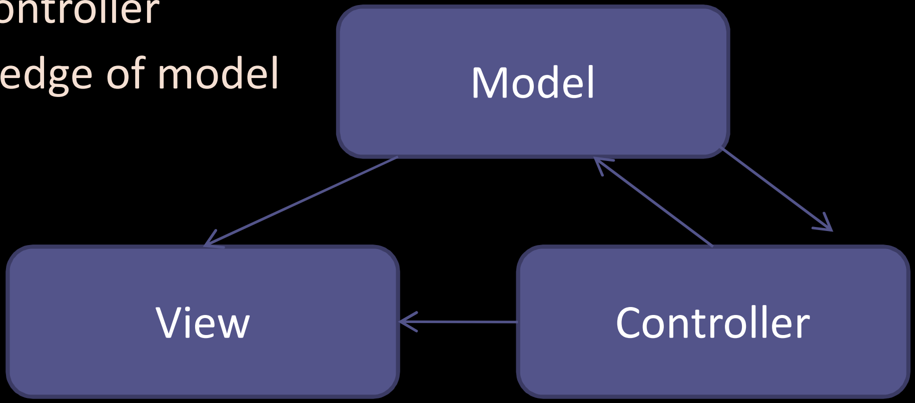
- Why isn't there syntactic sugar for containers, since there is for NSString?

```
//In an ideal world...
@[firstCar, secondCar, thirdCar]; // makes an NSMutableArray
@(firstCar, secondCar, thirdCar); // makes an NSArray
@{@"first"=firstCar, @"second"=secondCar}; // makes an NSMutableDictionary
@<firstCar, secondCar, thirdCar>; // makes an NSSet
```

  - Only Apple knows
    - But it's pretty inconvenient sometimes
    - Writing a few simple macros can help (despite the fact that C-style macros are generally evil)

Model-View-Controller pattern

- A key idea in Cocoa programming is the model-view-controller pattern
- Data (the **model**) is separated from how it is displayed (the **view**) and how it is interacted with (the **controller**)
- These components communicate by sending **messages**
  - Usually three separate classes
- Model has no knowledge of view or controller
- View and controller usually has knowledge of model



Boxing and Unboxing

- Raw C types can't go in containers
  - Can't put an int in an NSArray

- "Boxing" solves this problem
  - Creates a wrapper around raw types
  - NSNumber can convert to and from raw C number types
    NSValue converts to and from any C value (structs etc.)
  - NSNull represents a null value

```
int i=4;
double d=3.5;
NSNumber *numberI = [NSNumber numberWithInt:i]; // pack into an NSNumber
NSNumber *numberD = [NSNumber numberWithInt:d];
double k = [numberD doubleValue];  //take it back out
NSArray *array = [NSArray arrayWithObject:numberI]; // fine

CGPoint pt = CGPointMake(5,5);
NSValue *value = [NSValue value:&pt withObjCType:@encode(typeof(pt))];
CGPoint pt2;
[value getValue:&pt2]; // better hope that pt2 is of the right type!
```

Boxing and Unboxing (II)

- Works, but is **verbose**
- Java does this automatically, would be nice if Objective-C did it too...
- You can hack some macros to do this more simply
  - Not sure this is a good idea though...

- Note the use of **@encode** to convert a C type to a string representing it's type
  - This happens at compile time
  - In combination with GCC's typeof extension, can get type of expressions

Exceptions

- Objective-C has exceptions
  - Be aware that they have a huge performance penalty if the exception occurs
  - Not for flow control!
- **@try** -- begin an exception block
- **@catch** -- catch an exception of a given type
- **@finally** -- specify a block to executed whatever happens (optional)
- **@throw** -- throw an exception

```
@try
{
  [obj doSomething];
} @catch(NumberOverflowExecption *e) {
// catch a number overflow exception
}
@catch(NSException *e) {
//Catch a general exception
}
@finally{
// clean up...
}
```

Exceptions

- Multiple catch clauses possible
  - Must be ordered from most specific to least specific
  - the first @catch block which is of a compatible type with it's argument will get the exception
  - @catch(id e) catches **everything**
    - Exceptions don't have to be sublcasses of NSException, but they **should** be
    - The API always throws NSException exceptions

- @throw just takes an object to throw

```
NSException *exception = [NSException exceptionWithName:@"AudioUnavailable"
reason:@"Device is in use" userInfo:nil];
@throw exception;
```

- NSException has a handy class method **raise** which creates and raises and exception -- so you don't need to explicitly use @raise

Categories

- Categories are a unusual Objective-C feature
  - Allow classes to be extended without subclassing
  - Without even having the source code!

- You can add new methods to a class
  - All instances then respond to this new method
  - All instances which are subclasses will get the method too
  - **CANNOT** add new instance variables
- Just use **@interface** with an existing class name and **(category)**

```
@interface NSArray (random)
- (id) randomElement;
@end

@implementation NSArray(random)
- (id) randomElement {
  int i = rand() % [self count];
  return [self objectAtIndex:i];
}
@end
```

Categories (II)

- Every **NSArray** will now respond to **randomElement**!

- Can be used to spread a class definition over several source files
  - Define one main part
  - Then categories for each sub-section
  - Probably isn't a good idea to have such a big class in the first place though!

- It's conventional to name your source files **ClassName+CategoryName.m** / .h
  - **NSArray+random.m** and **NSArray+random.h** for example

- Remember, no variables can be added -- just methods

Message Forwarding

- Sometimes it's useful for objects to do something other than raise an error when sent a message that does not relate to one of their methods

- Most usefully, it can pass that message on to another object
  - If you override the **forwardInvocation:** method you can receive any messages which are not mapped to methods and handle them however you want

- For example, you could make a container that broadcasts messages to any of its elements...

```
- (void) forwardInvocation:(NSInvocation *)invocation
{
   for(id object in self) //assume we confrom to NSFastEnumeration
   {
     if([object respondsToSelector:[invocation selector]]) {
        [invocation invokeWithTarget:object];
     }
   }
}
```

Files and data

- Each application has it's own space it can read/write to
  - You can read from the bundle, but not write to it

- Use **NSHomeDirectory** to get the home directory of an application

```
//Get path of output.txt
NSString *outputPath = [NSHomeDirectory() stringByAppendingPathComponent:@"output.txt"];
```

- **NSData** manages blocks of raw data (just a chunk of bytes)
  - Can read and write from files
  - Convert to and from strings

```
NSString *filename = [[NSBundle mainBundle] pathForResource:@"data" ofType:@"raw"];
NSData *fileData = [NSData dataWithContentsOfFile:filename];
// do something with fileData
[fileData writeToFile:filename atomically:NO];

//Convert to and from ASCII string
NSString *dataString =[[NSString alloc] initWithData:fileData encoding:NSASCIIEncoding];
NSData *newData = [dataString dataUsingEncoding:NSASCIIEncoding];
```

NSFileHandle

- Low-level access to files with **NSFileHandle**

```
//Get path of output.txt
NSString *outputPath = [NSHomeDirectory() stringByAppendingPathComponent:@"output.txt"];
NSFileHandle *outputHandle = [NSFileHandle fileHandleForWritingAtPath:outputPath];
NSString *dataToWrite = @"This the data to write out!\n";
NSData *rawBytes = [dataToWrite dataUsingEncoding:NSASCIIStringEncoding];
[outputHandle writeData:rawBytes];
```

- Reads and writes using **NSData** (blocks of bytes)

- Can seek inside files for random access
- Also allows reading in the background
  - uses target-action to inform an object when the data is finished reading

Serialization: Archiving

- Cocoa supports object serialization under the name of *archiving*

- Allows Objective-C objects to be written or read from disk
  - Stores all dependencies so that entire object graph is regenerated

- **NSArchiver** and **NSUnarchiver** are sequential archivers (read objects in a big list)

- **NSKeyedArchiver** and **NSKeyedUnarchiver** allow access as if archives were hash tables (random access by name, for example)
  - In general, keyed archives should always be used

- Objects can only be archived if they conform to the **NSCoding** protocol

Serialization: Archiving

- It's easy to save an object using the archiveRootObject function

```
NSObject *object; // some object
[NSKeyedArchiver archiveRootObject:object toFile:@"object.archive"];
```

- And recover it with unarchiveRootObject

```
NSObject *object = [NSKeyedArchiver unarchiveObjectWithFile:"object.archive"];
```

- You can also save multiple objects and access them via keys
  - See the API docs for this

- Encoding and decoding of classes can be customized so that entire object graph does not have to be written out, or certain parts of the data can be excluded
  - See "Subclassing NSCoder"

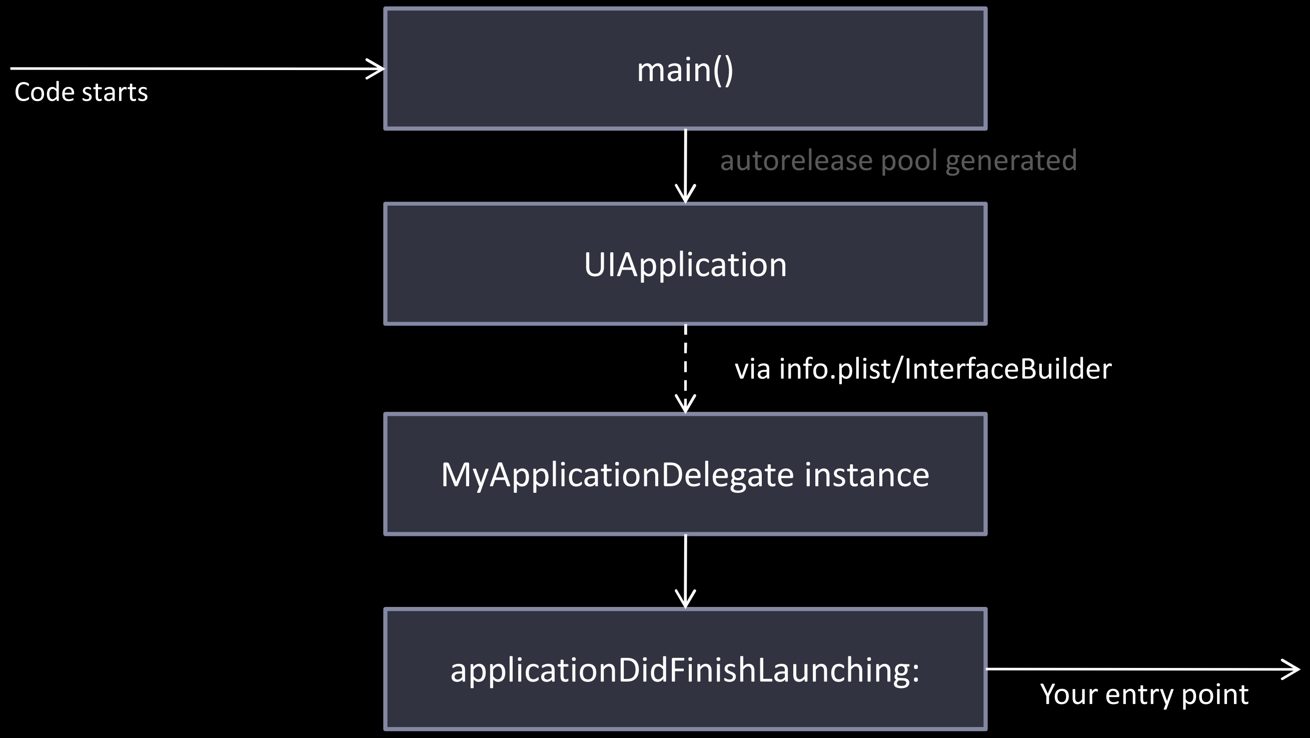
Startup

- **main**() is executed -- this is the entry point for all Objective-C applications
- An instance of **UIApplicationMain** is created
        - XCode inserts this code automatically

- The arguments to this specify a principal class (not really used much) and a delegate class (application delegate)
    - These are normally **nil!**
    - **info.plist specifies the nib file which specifies the delegate and principal**
    - Seems confusing, but you can view the connections in InterfaceBuilder
    - In general, the XCode app creation process will automatically create a skeleton delegate class and link it to UIApplicationMain

- Messages are then sent to the delegate, beginning with:
    - **applicationDidFinishLaunching:**
        - This is the entry point for user code -- it is called as soon as the application set up has been taken care of

Startup structure



Code starts

main()

autorelease pool generated

UIApplication

via info.plist/InterfaceBuilder

MyApplicationDelegate instance

applicationDidFinishLaunching:

Your entry point

Application Delegate

- There are a few really important things in the delegate
  - **applicationDidFinishLaunching**:
  - the **window** property -- this is the main window component
    - add subview(s) to this to make them visible
    - Usually just add the view of a UIViewController subclass

```
-(void) applicationDidFinishLaunching:(UIApplication *)application{
   //assume we have a viewController instance variable
   viewController = [[MyViewController alloc] init];
   [window addSubView:viewcontroller.view];
   [window makeKeyAndVisible];
}
```

  - **dealloc** -- called when memory is deallocated as the application shuts down

```
- (void) dealloc{
 [viewController release];
 [window release];
 [super dealloc];
}
```

Memory Warnings

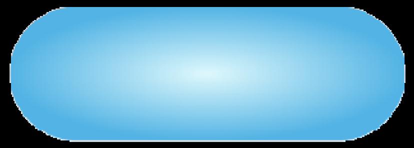
- OS warns apps if memory is about to run out
  - can happen because other services (like SMS or calls) have been allocated memory
- App will receive **didReceiveMemoryWarning:**
  - This message is sent to all active UIViewController subclasses in your app
- **You should respond to this**
  - If you don't, the app will be closed by the OS when memory runs out
- Quick memory management note:
  - you can get any object's retain count by sending it the **retainCount** message

Aside: Apple visual style

- Use **gradients**
- Use **transparency**
- Use **antialiasing**
- Use **animation**
- **Round corners**

Aside: the Aqua effect

Rounded box



Radial gradient, bright to saturated

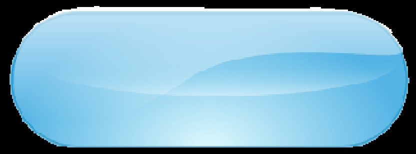
Move center of gradient to bottom

Smaller rounded box at the top

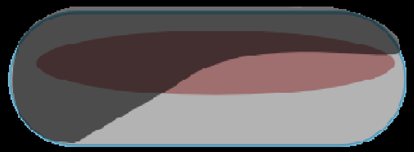
Vertical gradient, top white, bottom transparent

Adjust top box's size and transparency

Add an outline, slighty darker than the inside



Curved shapes can give fancier effects



Core Location

- **CoreLocation** allows you to find out where the phone is, and where it is pointing (compass)
- Uses GPS, cell location and WiFi positioning
    - Transparent interface -- all programmer gets is a position and an accuracy estimate

- Simple API using **CLLocationManager**
    - Delegate model -- you ask the manager to send messages when position or heading changes

```
locationManager = [[CLLocationManager alloc] init];
locationManager.delegate = self; // must conform to CLLocationManagerDelegate

[locationManager startUpdatingLocation];

//later...

[locationManager stopUpdatingLocation];
```

Core Location

- Note you can specify
  - desired accuracy (reduces effort taken to get fix)
  - distance filter (so that updates only occur after position changes by a certain amount)

```
// set accuracy to coarsest
locationManger.desiredAccuracy = kCLLocationAccuracyThreeKilomoters;

// only update if we move at least a kilometer
locationManager.distanceFilter = 2000
```

- The delegate gets **didUpdateToLocation** messages
  - give new position as latitude, longitude

```
- (void)locationManager:(CLLocationManager *)manager
didUpdateToLocation:(CLLocation *)newLocation fromLocation:(CLLocation
*)oldLocation
{
   double newLatitude = newLocation.latitude;
   double newLongitude = newLocation.longitude;
   double newAltitude = newLocation.altitude;

}
```

The Compass

- Request compass updates with **startUpdatingHeading**

```
[locationManager startUpdatingHeading];

// later...

[locationManager stopUpdatingHeading];
```

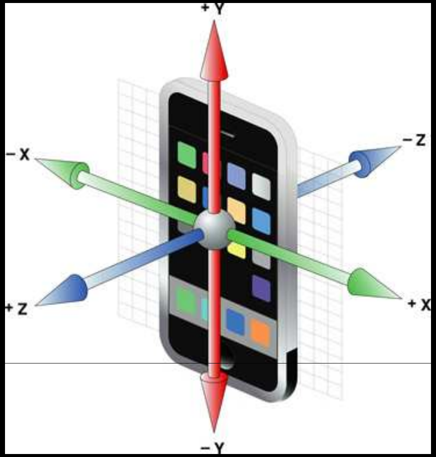
- and receive them with a call to **didUpdateHeading**

```
- (void) locationManager:(CLLocationManager *)manager didUpdateHeading:(CLHeading
*)newHeading
{
   //results in degrees
   double rawHeading = newHeading.magneticHeading; // raw magnetic heading
   double trueHeading = newHeading.trueHeading; // compensated (with location)


}
```

Accelerometer

- Reading the accelerometer is easy
    - iPhone acclerometer is 3 axis
- UIAccelerometer class used for access
    - get the shared object, pass it a delegate, set update rate
    - receive x,y,z accelerations...
    - we will cover doing interesting things with it later...



```
UIAccelerometer accelerometer = [UIAccelerometer sharedAccelerometer];
accelerometer.updateInterval = 0.05; // seconds!
accelerometer.delegate = self; // updates go to this object
// must implement UIAccelerometerDelegate protocol

// in the delegate class
- (void) accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration
{
  NSLog(@"%f %f %f\n", acceleration.x, acceleration.y, acceleration.z);
}
```

Magnetometer

- Raw magnetic readings can be obtained
  - These allow direct measurement of magnetic field
  - Uses: detecting disturbances, full device orientation...

- Simply part of the heading update data

```
- (void) locationManager:(CLLocationManager *)manager didUpdateHeading:(CLHeading
*)newHeading
{

   //normalized to -128 to +128
   NSLog(@"%f %f %f\n", newHeading.x, newHeading.y, newHeading.z);
}
```

User Inteface

- User interface components form part of the UI* class hierarchy

- User interface components inherit from **UIView**
  - Abstract class for drawing and handling events
  - **Can subclass it to make custom controls**

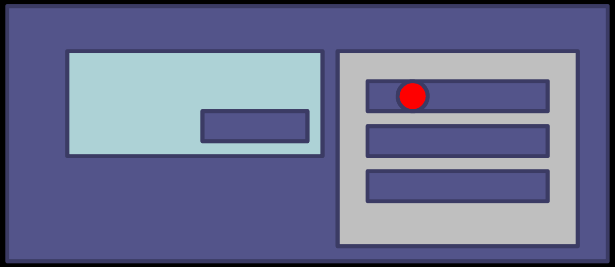
UIView

- Important methods:
  - **initWithFrame**:
    - creates a new view with a given frame
    - lots of controls are initialized this way
  - **addSubview**
    - Add another view to this one (i.e. draw it on top)
  - **removeSubview/bringSubviewToFront/sendSubviewToBack**
  - **drawRect**
    - override this to customize drawing!
  - **setNeedsDisplay**
    - call this to force redraw

UIView

- Touch handling
  - **hitTest:point withEvent:event**
  - sends messages to subviews to find deepest target that this point touches

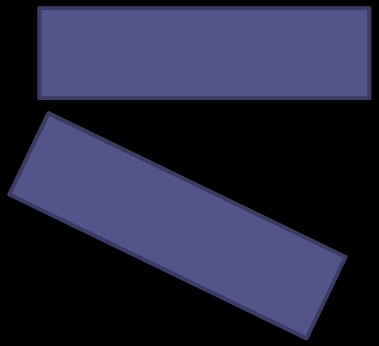
  - **pointInside**
    - returns true if the point is inside the control at all



  - co-ordinate conversion with **convertPoint** methods
    - screen to control co-ordinates (where am I clicking in this button?)

UIView

- Important properties
  - **frame**
    - rectangle control occupies
  - **transform**
    - specifies a transform applied before drawing
    - this can be used to rotate/scale/translate controls
      - just set the **transform** property
      - use **CGAffineTransform** to specify transform
  - **alpha**
    - specifies the control transparency
  - **backgroundColor**
    - background color of control (if applicable)
  - **hidden**
    - if you set it to YES, the control will disappear...



UIView

- Important properties
  - **multiTouchEnabled**
    - if **YES**, will receive multiple finger contacts.
    - NOTE: the default value is **NO**!
  - View hierarchy
    - **superview**
      - parent of this view
    - **subviews**
      - NSArray of immediate subviews