ES3 Lecture 3

iPhone development: XCode and Objective-C

Resources

- **developer.apple.com** -- sign up for a free online account to access the APIs!
- Stanford has an online course at **http://www.stanford.edu/class/cs193p/cgi-bin/index.php** the slides and lecture here are useful.
- OpenGLES stuff: **http://iphonedevelopment.blogspot.com/2009/04/opengl-es-from-ground-up-part-1-basic.html**
- **switchonthecode.com** some useful introductory tutorials here

iPhone Development

- The iPhone is based around the Cocoa Touch API
  - Variation on the Cocoa API used in OS X
  - Provides many basic objects (data structures) and specialized UI components
- Development in Objective-C
  - Apple does not permit the use of interpreted languages
- Development is exclusively with Apple's own XCode IDE
  - Provides IDE, editor, debugger

Objective-C

- It's C
  - Well, a superset of it
- Everything in C works exactly the same in Objective-C
  - Great for porting existing code

- But Objective-C has some very lightweight and clean object-oriented extensions
  - Much cleaner and simpler than C++

- Introduces an object type, and a **message based** OO model
  - You send messages to objects and they take actions
  - A more "loosely-coupled" approach to OO programming

Objective-C

- Dynamically typed (unlike C++)
  - The compiler can help check types for you, but binding is at runtime
  - This makes it very flexible
- Lots of Objective-C is based around **conventions**
  - Naming conventions especially
  - You should stick to these **rigorously!**
- Objective-C itself is very simple
  - Just adds the ability to use objects in C code with a few extra keywords and syntax
  - It's Cocoa -- the libraries used by OS X and the Cocoa Touch variant for the iPhone -- which provides most of the power
    - API comparable to the Java SDK in scope and power

Some Syntax

- Calling a method of an object uses square bracket notation

  ```
  [vehicle startMoving]
  ```

  - Sends the message `startMoving` to the object `vehicle`
  - This is like calling the `startMoving` method of `vehicle`

- Parameters are passed using colons

  ```
  [vehicle startMoving:YES]
  ```

Some Syntax

- Multiple parameters are of course possible, and they must have **names**
  - This does **NOT** mean they are optional or the order can be changed!
  - The parameter names are part of the name the compiler uses for the message

```
[vehicle startMoving:YES withSpeed:50.0 atAngle:30.0]
```

- This is like `vehicle.start_moving(true, speed=50.0, angle=30.0)` in Python

- Note that it is possible to take a variable number of arguments
  - It's a bit more complicated and not commonly done
    - Actually, each named parameter can have variable number of following arguments
    - Normally better to pack things into a container and send them that way

- NB: the compiler doesn't force type checking
  - It is possible to send a message to an object that doesn't know how to respond!
    - This will cause a runtime exception

Syntax: declaring classes

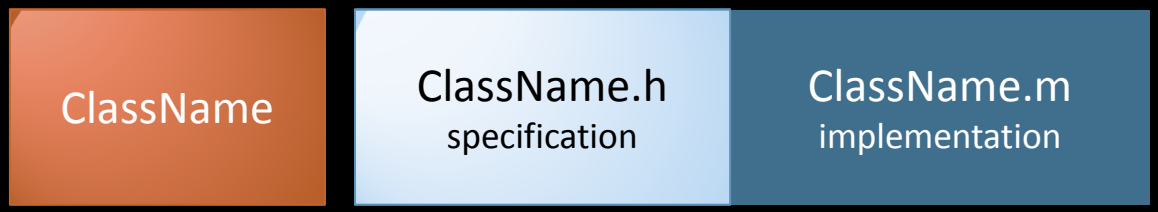
- Classes are declared with **@interface** (one of the few new keywords in Objective-C)

```
@interface Track : WorldObject {
// instance variables here
int times;
Vehicle *racer;
}
// methods
- (void) setRacer:(Vehicle*) racer;
@end
```

- Inheritance is specified with the name : superclass notation
- Classes are followed by a **block** where instance variables are listed
- Then the methods are listed
- Then **@end** (you can't miss that out!)

Basics

- Objective-C generally uses a file-per-class approach (as in Java)
  - Like many other things, it's not enforced!
- Each class has a specification (held in a .h file) and an implementation (held in a .m) file



- The specification lists all member variables and the prototypes of the methods
  - The implementation just has all the method implementation
- You should stick to this structure
  - XCode will help you -- it can automatically generate a pair of barebones .m and .h files for you

Basics (II)

- You specify the implementation of a class using @implementation

```
// in the .h file
@interface Vehicle {
...
}
...
- (void) start;
@end

// in the .m file
// NOTE: no following block!
@implementation Vehicle

- (void) start {
          // do some start stuff
}
...
@end
```

- Methods are listed in the implementation block
- Still need the leading + or -
- Identical to the signature in the .h file (you can just copy and paste)

Syntax: methods

- Methods are defined beginning with either - for instance methods, or + for class methods

```
- (void) goFaster;
+ (void) createVehicle;
```

- The first parameter is specified with (type)varname:

```
- (void) setSpeed:(double)newSpeed;
```

- Subsequent parameters need **names** as well as types!

```
- (void) setVelocity:(double)angle speed:(double)speed
- (BOOL) hasCollided:(double)time with:(Vehicle *)otherVehicle;
```

- Note that the name and the parameter name can be unrelated
  - The name is just there for the compiler to tell the method apart and to make it easier to read method calls with multiple parameters

Overloading and naming

- You can overload methods, but the overall method name must be different:

```
- (void) setSpeed:(double)newSpeed;

// This is ok
- (void) setSpeed:(double)newSpeed multiplier:(double)multiplier;

// This won't compile (you can't overload on type alone)
- (void) setSpeed:(NSString*)newSpeed;
```

- The compiler treats the name as if it were `setSpeed:multiplier:`
  - Types are not part of the name, so they will clash!
- It's a good idea to use parameter names so that the call reads well:

```
- (void) setPositionX:(double)x Y:(double)y;

//call it like
[setPositionX:40.0 Y:30.0];
```

Instantiating and using objects

- You create classes by calling **class methods**
- The method **alloc** allocates the memory for a class (but does nothing else)

```
Vehicle *racer = [racer alloc];
```

- You generally need to have some kind of init function to set up member variables
  - By convention, this should return itself

```
// in the class definition
+ (Vehicle *) initWithName:(NSString*)name {
  vehicleName = name;
  return self;
}
...
// in the body
Vehicle *racer = [[racer alloc] initWithName:"Monocycle"];
```

- NB: the method `+ (void) initialize` is called for all classes at start up
  - Useful for setting up class data
  - Classes have no storage -- have to use static global variables

Instantiating and using objects

- Many classes have factory methods
  - Generate new objects and return them to you
  - Uses autorelease pools (more later) for easier memory management

```
Vehicle *racer = [racer createRacerWithName:@"Monocycle"];
```

- In general, use these methods if they're available rather than alloc/init*: it'll simplify memory management

id

- In Objective-C, all **objects** are of type **id**
- **Any object can be stored in a variable of type id**

```
Vehicle *racer = [racer createWithName:@"Monocycle"];
NSString *name = @"John";

id obj;
obj = name; // fine
obj = racer; // fine
name = racer; // compiler will complain!
name = obj; // this is fine at compile time, but not at run time!

//Wrong! just type id, not id*
id *obj2 = name;
```

- (almost) everything in objective-C is either of type id, or a basic C type
  - int, double, float, char *, struct, enum, union, etc.
    - A few special cases, like type SEL for selectors

BOOL

- Objective-C introduces a boolean type BOOL
  - typedef for char
  - values are TRUE or FALSE, but more commonly YES or NO

```
BOOL flagSet = YES;
if(flagSet)
{
  [racer crashNow];
}
```

Self

- Note that an instance can always get a pointer to itself from the **self** variable

```
//This is an instance method
- (void) changeDirection:(double)newAngle speed:(double)speed
{
          //Just calls these methods on the current instance
          [self setSpeed:newSpeed];
          [self setAngle:newAngle];
}
```

- This is often essential when telling other objects where to send messages in response to events (the target-action model)

Protocols

- Like interfaces in Java
- Allow specification of methods a class must have
  - Compiler checks for you
- Classes can (and often do) implement multiple protocols

- Definition using @protocol instead of @class
  - no instance variables!

```
@protocol Steerable
- (void) setHeading:(double) newAngle;
- (void) setSpeed:(double) newSpeed;
@end
```

Protocols (II)

- A class declares that it uses a protocol by including in a <> bracketed list after the class name:object

```
@class Vehicle : MovingObject <Steerable, Drawable>
...
```

- Remember: Objective-C is dynamically typed
  - Compiler will **warn** you if you try and use an object which does not conform to a protocol
  - Error if you don't implement all the methods of the protocol

Properties

- Setting and getting instance variables with methods gets tedious

```
...
int type = [[racer engine] engineType];
[[rootMenu currentMenu] setType:type];
```

- Properties introduce new syntax which wraps up setting and getting instance variables
  - x.y notation
  - Still allows the encapsulation and separation that messaging encourages

```
int type = racer.engine.engineType;
rootMenu.currentMenu.type = type;
```

- t = x.y.z is converted by the compiler to t = [[x y] z]
- x.y.z = t is converted by the compiler to [[x y] setz:t]
  - last element in the dot list becomes setXXX when it is used as an lvalue in assignment

Properties (II)

- To use properties, you must declare them in the class definition

```
@class Racer {
int type;
Engine *engine;
NSString *name;
}
//methods
- (void) setName:(NSString *)newName;
- (NSString *)name;
- (void) setEngine:(Engine*)newEngine;
- (Engine *) engine;

@property NSString *name;
// Note the qualifiers in brackets!
@property (nonatomic, retain) NSString *Engine;
@end
```

- Now a x = racer.engine is equivalent to x = [racer engine]
  - It calls the method you define!
  - **Properties work by naming convention alone**

Properties (III)

- Sometimes it's useful to have automatic properties
  - just sets/reads the instance variables

- Objective-C can do this automatically using **@synthesize**
- This must go in the **implementation** of the class (not the definition!)

```
@synthesize name, engine;
```

- Note: no types given, just the name
- This automatically creates methods name, setName, engine and setEngine
  - just read and write instance variables

- You can make properties read only with the **readonly** attribute:

```
@property (assign, readonly) Vehicle* racer;
```

Types: inheritance

- Objective-C supports inheritance
- Methods and instance variables are inherited
- The superclass of an object can be accessed with **super**, exactly like **self**

- Two things:
  - in init functions you need to call [super init] to initialize the super class
  - in dealloc, you need to call [super dealloc] to deallocate storage of the super class

- Unlike C++ this is not automatic

Selectors, delegates and target-action

- One very common pattern is the target-action pattern
  - Send a message to an object, specifying a target (object) and an action (a message)
  - The receiving object sends that message to that object when some event occurs

```
//so vehicle will call the carCrashed method of self if it crashes
[vehicle setCrashTarget:self action:@selector(carCrashed:)
```

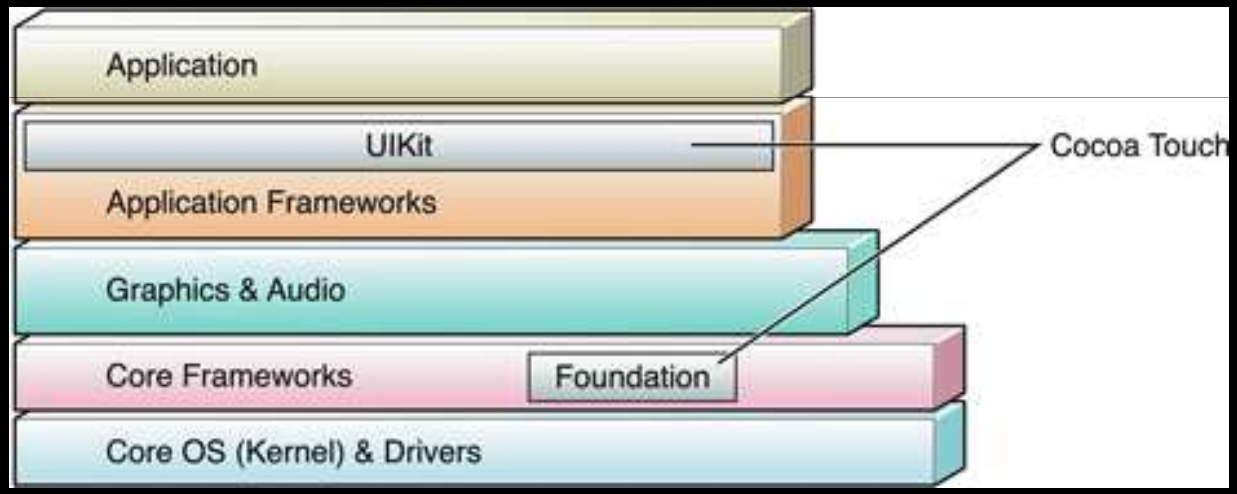
- Note the use of **@selector**
  - this returns a value of type SEL which identifies the message
  - rather than inefficiently using strings

- In the delegate model, an object is sent (normally conforming to some protocol) and the receiving class sends messages to it

```
[vehicle setActionHandler:self]
// vehicle now sends messages to this instance
// when actions happen
```

Structure of Cocoa

- UIKit: user interface components
- Foundation: everything else (containers, files, data, etc.)
- Core Frameworks (location, graphics etc.)



- see http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html

NSObject

- **Every class in Cocoa inherits from NSObject**
- Provides basic functionality (alloc, release, dealloc, etc.)
- Also provides useful reflection methods
  - instancesRespondToSelector -- test if an object will respond to a given message
  - class/superclass -- get the class/superclass of this object
  - isSubclassOfClass /conformsToProtocol -- checks if this object is a subclass of a class / conforms to a protocol
  - resolveClassMethod/resolveInstanceMethod -- look up a selector by name (i.e. with a string, at runtime)
  - performSelector -- call a method of this object (at runtime, so you can call methods without knowing types)
  - methodSignatureForSelector -- get the types for arguments and return values of a method
- These allow you to interrogate and perform actions on objects without knowing their type at compile time
  - In particular, performSelector: is often used to send messages to objects

NSString

- Replaces char* in all Cocoa libraries
  - Much, much better! Fully object-oriented, and supports virtually any encoding for different languages

- Special syntax for generating literals: @"text" creates an instance of NSString

```
NSString *myName = @"John";
```

- Can convert to and from C-style char* strings if you need to use C-libraries
- NSStrings are **immutable. You cannot change them after creation.**
  - You can use **NSMutableString** if you want to change after creation

- Methods for comparing, slicing, line splitting, delimiter separating, joining and converting to numerical values all provided as part of the NSString class.

Other useful Foundation classes

- **NSDate** : store and manipulate dates and time at high resolution (ms level).

```
NSDate *startTime = [NSDate date];
...
double secondsElapsed  = -[startTime timeIntervalSinceNow];
```

- **NSURL**: work with URLs
- **NSTimer**: set up timers to send messages at given intervals
- **NSException**: exception classes
- **NSDateFormatter**, NSNumberFormatter: stringify dates and numbers
- **NSNotification / NSNotificationCenter:** broadcast messages to listeners

Memory management (I)

- In Cocoa, **you** are responsible for memory management
  - no garbage collector (at least not on the iPhone)
- **Memory management is a really important concept to get your head around!**
- Manual reference counting system
  - Space is allocated with **alloc** and deallocated with **dealloc**
  - EXCEPT you NEVER call **dealloc** manually
    - (well except when you call [super dealloc] in the dealloc method of your own class)
  - Instead, you call **release**
  - This decreases the "retain count " of the object
  - When it goes to zero, the object is automatically deallocated
  - Calling **retain** on an object increases it's retain count (i.e. it will need another release before it can be deallocated)

Memory management (II)

Code	Retain Count
`Vehicle *car = [[Vehicle alloc] initWithName:@"car"];`	1 (alloc makes it 1)
`[car retain];`	2 (retain increased it)
`[car release];`	1
`[car release];`	0 (car is now automatically freed!)

- Often you will have init/dealloc methods in your classes which look like this:

```
- (void) initWithName:(NSString *)name
{
 Engine *engine = [[Engine alloc] init];
 ...
}

- (void) dealloc
{
  [engine release];
  [wheels release];
  [super dealloc]; // the ONLY time you ever manually call dealloc!
}
```

Autorelease Pools

- Autorelease pools make it (a bit) easier to manage memory
- Put objects in an autorelease pool by sending it the **autorelease** message

```
Vehicle *car = [Vehicle alloc];
[car autorelease];
```

- Basically, all API calls (which don't begin alloc, copy or new) will return autoreleased objects
- These objects will be released automatically "later"
  - This "later" is guaranteed to be at least until after the current function returns
  - But no more than that is guaranteed!
  - If you need to keep it longer, you must **retain** it by sending it a retain message

Conventions

- When you add an object to a container, it calls retain on it
  - So that it will remain allocated until it is removed from the container or the container itself is released
  - Releasing a container calls release on all its elements
- If an API call begins **alloc, copy** or **new** it will allocate memory
  - You must release it explicitly
- If it begins with something else (e.g. stringWithCString) it will return an object in an autorelease pool
  - DO NOT RELEASE IT YOURSELF UNLESS YOU RETAIN IT!
  - You MUST call retain on it if it will leave the current call
    - For example if you are caching it in an instance variable
  - Match each **retain** with a **release**

Conventions (II)

```
//Object created with alloc, so must be explicitly released
Vehicle *vehicle = [[Vehicle alloc] initWithName:@"Car"];
...
[vehicle release]


// This will be autoreleased
NSString *name = [NSString stringWithCString:"Hello, World!"];
myName = name;
// if this is an instance variable, watch out -- name will be autoreleased after
this function returns

[myName retain];
// now it's safe, but must call release on it in dealloc or whenever you're
finished with it
```

- Conventions are **very important**
- Like everything in Objective-C they are not enforced but should be obeyed
  - **All API calls obey these conventions**

Memory management: using properties

- This can get pretty tedious
  - Properties to the rescue!

- Properties can use the **retain** attribute

```
@class Racer {
Engine *engine;
}

// Note the qualifiers in brackets!
@property (nonatomic, retain) NSString *engine;
@end
...
//In the implementation
@synthesize engine;
```

- When you assign to the engine variable, the runtime will automatically send a retain message to it
- When the object is deallocated, it will automatically send release

Memory management: using properties (II)

```
@property (nonatomic, retain) NSString *engine;
@end
...
@synthesize engine;

//becomes:
- (void) setEngine:(Engine *)newEngine {
  engine = newEngine;
  [engine retain];
}

- (void) dealloc {
...
[engine release]; // adds this in invisibly
...
}
```

- One thing to note -- you must use property syntax to get the behaviour
  - If you are setting instance variables, you must use self.x = y
  - Not just x = y -- this sets the variable without calling the property
  - [self setx:y] works as well

```
engine = [engine createWithName:@"car"]; // won't retain it
self.engine = [engine createWithName:@"car"]; // WILL retain it
[self setEngine:[engine createWithName:@"car"]]; // WILL retain it
```

NS*, UI* vs CF*, CG*

- API sections beginning NS or UI are Objective-C
  - They use objects and inherit from NSObject
  - These are mainly the higher level parts of the API

- Parts of the API beginning C are pure C APIs (logically enough)
  - e.g. Core Graphics is CG (CGPoint, CGContext)
  - Doesn't use Objective-C objects, just plain C
    - structs, char*, void * etc.

```
//These convenience functions return C structs
CGPoint pt = CGPointMake(5,8);
CGRectMake rect = CGRectMake(20,20,5,5);
```

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

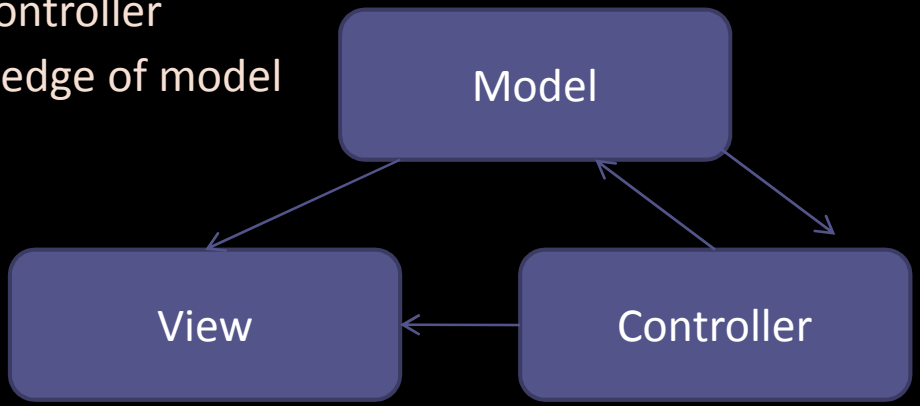
- Why isn't there syntactic sugar for containers, since there is for NSString?

```
//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



Bundles

- You will often need to access external data files in your application (graphics, sounds, text data,...)
  - Bundles provide a convenient way of storing and accessing such files
- NSBundle handles bundle access
  - [NSBundle mainBundle] returns the main bundle (which is all we'll cover)
  - Add and remove files to the main bundle via XCode
- Most common idiom is to get an path from a bundle:

```
//Get path of texture.png and load it into an image
NSString *texturePath = [[NSBundle mainBundle] pathForResource:@"texture" ofType:@"png"]
UIImage *textureImage = [UIImage imageWithContentsOfFile:texturePath];
```

- Bundles are **read-only**
  - cannot be modified after application is built/signed!

Good Practice

- Use Objective-C naming convention
  - Verbose, no abbreviations, camelCase names
  - Adhere and observe the memory management naming convention
- Types
  - Use container types, not C arrays
  - Use classes, not C structs
  - Use properties
  - Allow compile-time checking where possible
- Messaging
  - Don't break the message passing model
  - Avoid invoking directly methods where possible -- instead notify objects that things need done
    - **The lazy, loosely-coupled approach is strongly encouraged in Cocoa programming**