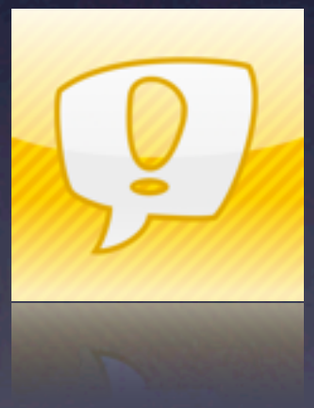
CS 47

Beginning iPhone Application Development

Introductions

Who, why, which?

Shameless Plug: LoudTap



Wifi Access

(If it works..)

- SSID: Stanford
- Username/password: csp47guest

Expectations

- This is a programming class
- Prerequisites are important:
  - C/C++/Java and OO proficiency
  - Intel Mac running OSX 10.5+

We're Going to Learn

- How write iPhone applications
- How to run them in the simulator
- How to run them on the iPhone
- How to publish to the App Store

No Class March 4

- We'll probably extend the class an extra week; Need to verify with CSP

Grading

- Your grade will be based on your class project
  - A = Wow!
  - B = Ok, not bad
  - C = You wrote two lines of code
  - D = Did you even come to class?

Homework

- There is no mandatory homework other than the class project
- Highly recommended that you spend time reviewing sample code and documentation on your own.

Online Resources

- Class website: http://cs47.fieldman.org
- Discussion Group: http://groups.google.com/group/stanfordcs47
- Developer Site: http://developer.apple.com
- My email: jason@fieldman.org

Office Hours

- Saturdays from 11am - noon
- Come and talk about whatever
- Red Rock Cafe (2nd floor)

Objective-C

Buckle Up

You will probably need to review this later, on your own.

Objective-C Overview

- All iPhone SDK programming is done in Objective-C
- Looks a lot like C/C++
  - Object-oriented
  - Header files (.h), source files (.m)

Mixing in C/C++

- You must use Objective-C to interact with general iPhone SDK elements
- But, you can use C and C++ code in an iPhone application
  - Data structures
  - OpenGL applications
- C/C++ code executes faster (less overhead), but generally not enough to avoid Objective-C

What's New in Obj-C?

Categories	Used to add to classes without subclassing
Classes	Define objects
Messages	Send commands to objects
Properties	Easy definition of accessors and mutators
Protocols	Define methods that classes promise to respond to
@	Compiler directives

What Does it Look Like?

```
- (void) updateLastAccessTimeForGroupId:(NSNumber*)groupId {

  NSDictionary *accessDic = [NSDictionary dictionary];
  NSString     *groupKey  = [groupId stringvalue];
  ISPOGroup    *group     = [self groupWithId:groupId];
  NSNumber     *curTime   =
      [NSNumber numberWithLongLong:group.lastMessage.timestamp];

  [accessDic setObject:curTime forKey:groupKey];
  [[GroupsViewController sharedInstance] notifyGroupsUpdate];

}
```

Messages

- a.k.a. Selectors, Methods, Functions
- Messages are how you interact with objects
- Once you understand how messages are defined and called, you can read most Obj-C code

Messages

- Message calls are defined by square braces:

```
[receiver message];
```

- Real-life example:

```
[window makeKeyAndVisible];
```

- Equivalent to:

C++:
```
window->makeKeyAndVisible();
```

Java:
```
window.makeKeyAndVisible();
```

Messages

- Message calls can take arguments:

```
[receiver message:argument];
```

- Real-life example:

```
[textView setText:@”Hello World”];
```

- Equivalent to:

```
textView.setText(“Hello World”);
```

Messages

- Message calls can take multiple arguments:

```
[receiver message:arg1 label2:arg2];
```

- Real-life example:

```
[button setTitle:@”Press Me”
       forState:UIControlStateNormal];
```

- Equivalent to:

```
button.setTitle(“Press Me”, UIControlStateNormal);
```

Messages

- Message calls can return values:

```
int output = [receiver message];
int output = [receiver messageWithInput:arg1];
```

- Equivalent to:

```
int output = receiver.messageWithInput(arg1);
```

Messages

- Messages can nest

```
[receiver message:[receiver2 message2]];
```

- Equivalent to:

```
receiver.message(receiver2.message2());
```

Messages

- Recap

```
[receiver message];
[receiver message:arg1];
[receiver message:arg1 label2:arg2];

int output = [receiver message:arg1];

[receiver message:[receiver2 msg2]];
```

Messages: Calling on nil

- Calling a message on a nil object just returns nil (0).

```
MyClass *myObject = nil;
[myObject callSomeMessage];
int result = [myObject getSomeValue];
```

- No compiler or run-time errors. Nothing happens for the first message, and the second message returns nil
- You will see: this is awesome. Makes dealing with model data much less complicated

Messages: Accessors

- All class member variables are private
- They must be set/get using accessor messages:

```
[myObject setVariable:value];
value = [myObject variable];
```

- Or, more commonly, using dot syntax:

```
myObject.variable = value;
value = myObject.variable;
```

- Accessors must be explicitly defined or synthesized

Message Declarations

- C/C++ Example:
  ```
  int funcName(int arg1, char *arg2);
  ```
- In Objective-C:
  ```
  - (int) funcName:(int)arg1 withArg:(char*)arg2;
  ```
- Would be called like this:
  ```
  int arg1 = 6;
  char *arg2 = “Hello World”;
  int v = [myObject funcName:arg1 withArg:arg2];
  ```

Messages: Class vs. Instance

- C++
```
MyClass::classFunction();
myObject->instanceFunction();
```
- Java
```
MyClass.classFunction();
myObject.instanceFunction();
```
- Objective-C
```
[MyClass classMessage];
[myObject instanceMessage];
```

Messages: Class vs. Instance

- Declaring a class message:
  `+ (int) classMessage:(int)arg1;`
- Declaring an instance message:
  `- (int) instanceMessage:(int)arg1;`
- The only syntax difference is the +/-!
  - + for class; - for instance

Instantiating Objects

- Equivalent to C++:

```
MyClass *myObject = new MyClass();
```

- Equivalent to Java:

```
MyClass myObject = new MyClass();
```

Instantiating Objects

Two general types of object instantiation

i) Create an autoreleased object:

```
NSArray *array = [NSArray array];
```

ii) Create a retained object:

```
NSArray *array = [[NSArray alloc] init];
```

Either style can take optional arguments:

```
NSNumber *n = [NSNumber numberWithInt:3];
NSNumber *n = [[NSNumber alloc] initWithInt:3];
```

Memory Management

- C:

```
char *str = malloc(10);
free(str);
```

- C++:

```
MyClass *myObject = new MyClass();
delete myObject;
```

- Java:

```
MyClass myObject = new MyClass();
```

Memory Management

- Objective-C uses a hybrid system. No automatic garbage collection, but we get a reference counter API.
- When an object's reference counter hits 0, its memory is released
- The reference counter API is provided through the NSObject and NSAutoreleasePool classes.

Memory Management

Autoreleased Objects:

```
NSArray *array = [NSArray array];

---- SAME AS ----

NSArray *array = [NSArray alloc]; // ref +1
[array init];
[array autorelease]; // ref -1 at end of event

---- SAME AS ----

NSArray *array = [[[NSArray alloc] init] autorelease];
```

Memory Management

```
NSArray *array = [NSArray array];
NSArray *array = [NSArray arrayWithCapacity:10];

NSString *string = [NSString string];
NSString *string = [NSString stringWithFormat:@”%d”, 4];

NSNumber *number = [NSNumber numberWithInt:4];
NSNumber *number = [NSNumber numberWithBool:YES];
NSNumber *number = [NSNumber numberWithFloat:1.1];

MyClass *myObject = [MyClass myAutoreleaseMessage:arg];
```

Memory Management

Manually Managed Objects:

```
{
  // alloc: ref +1
  NSArray *array = [[NSArray alloc] init];
  ...
  ...
  [array release]; // release: ref -1 immediately
}
```

(Memory leak if not released!)

Memory Management

- What if you want to keep an object persistent?

```
[myObject retain]; // ref +1
```

- What happens to this object?

```
{
  NSArray *array = [NSArray array];
  [array retain];
}
```

Don’t Panic

Classes

- Classes are defined by:
  - Interface (.h)
  - Implementation (.m)
- Similar to C++, with separate class declaration (.h) and implementation (.cpp) files.

Classes

- Example class interface

```
@interface MyClass : NSObject {
   NSString *myString;
   int       myInt;
}
@end
```

Classes

- Let's add getters to our class

```
@interface MyClass : NSObject {
  NSString *myString;
  int       myInt;
}
- (NSString*) myString;
- (int) myInt;
@end
```

Classes

- Let's add setters

```
@interface MyClass : NSObject {
    NSString *mystring;
    int        myInt;
}
- (NSString*) myString;
- (int) myInt;

- (void) setMyString:(NSString*)newString;
- (void) setMyInt:(int)newInt;
@end
```

Classes

- Now let's implement the getters

```
@implementation MyClass

- (NSString*) myString {
  return myString;
}

- (int) myInt {
  return myInt;
}

@end
```

Classes

- Now let's implement the setters

```
@implementation MyClass

- (void) setMyString:(NSString*)newString {
  [myString autorelease];
  myString = [newString retain];
}

- (void) setMyInt:(int)newInt {
  myInt = newInt;
}

@end
```

Classes

- What if our setMyString implementation looked like this?

```
- (void) setMyString:(NSString*)newString {
  myString = newString;
}
```

Classes

- Classes can implement any message, not just accessors

```
@interface MyClass : NSObject {
}
- (void) doSomethingCrazy:(int)arg1;
- (NSString*) getSomeString;
@end
```

- You just need to add their implementations to the @implementation section

Classes

```
- (id) init {
  if (self = [super init]) {
    [self setMyString:@”Hello World”];
    [self setMyInt:3];
  }
  return self;
}

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

Classes

- Using properties in an interface

```
@interface MyClass : NSObject {
  NSString *mystring;
  int       myInt;
}

@property (nonatomic, retain) NSString *myString;
@property (nonatomic, assign) int       myInt;

@end
```

- Activates the “dot syntax” accessors for that member

Classes

- Implementating properties

```
@implementation MyClass

@synthesize myString;
@synthesize myInt;

@end
```

- Generates setters/getters (but you can override!)
- Synthesize *DOES NOT* implement the destructor in dealloc

Classes

- Example of properties being accessed through dot syntax

```
...
MyClass *myObject = [[[MyClass alloc] init] autorelease];
myObject.myString = @”Test”;
myObject.myInt    = 10;
...
```

Classes

- Setters aren't restricted to just setting values. Often you will want to do more!

```
- (void) setMyString:(NSString*)newString {
  [myString release];
  myString = [newString retain];
  myInt    = [myString intValue];
}
...
MyClass *myObject = [[[MyClass alloc] init] autorelease];
myObject.myString = @"10"; //also sets myInt!
```

Delegates/Protocols

- Similar to interfaces in Java (i.e. "X implements Y").
- Allow you to encapsulate a complex class that you do not want to subclass.
- Allows a class to interact with other objects by querying them with messages, rather than requiring messages to being queried on them.
- Example: HTTP connection. Set the delegate of the connection object to receive a notice when data is complete, rather than having to poll the connection

Protocols

- Protocols look like this

```
@protocol MyClassDelegate <NSObject>
@required
- (void) myClass:(MyClass*)myObj sawEvent:(int)eId;
@optional
- (NSString*) tagForMyClass:(MyClass*)myObj;
@end
```

- Anything that implements this protocol is contractually bound to implement the required messages!

Protocols

- How do you indicate that you implement a protocol?

```
@interface MyOtherClass : NSObject <MyClassDelegate> {
}
@end
```

- You do not need to add prototypes for the delegate messages in the interface, since it is implied
- You must now implement the MyClassDelegate messages in the implementation section of this class

Delegates

- How to add a delegate to a class

```
@interface MyClass : NSObject {
  id<MyClassDelegate> myDelegate;
}

@property (retain) id<MyClassDelegate> myDelegate;

@end
```

Delegates

- You can now pass delegate messages to your delegate member variable

```
...
[myDelegate myClass:self sawEvent:SoAndSoEventId];
NSString *myTag = [myDelegate tagForMyClass:self];
...
```

- If myDelegate isn't set yet, it's just nil. The messages have no effect and return nil

Phew.