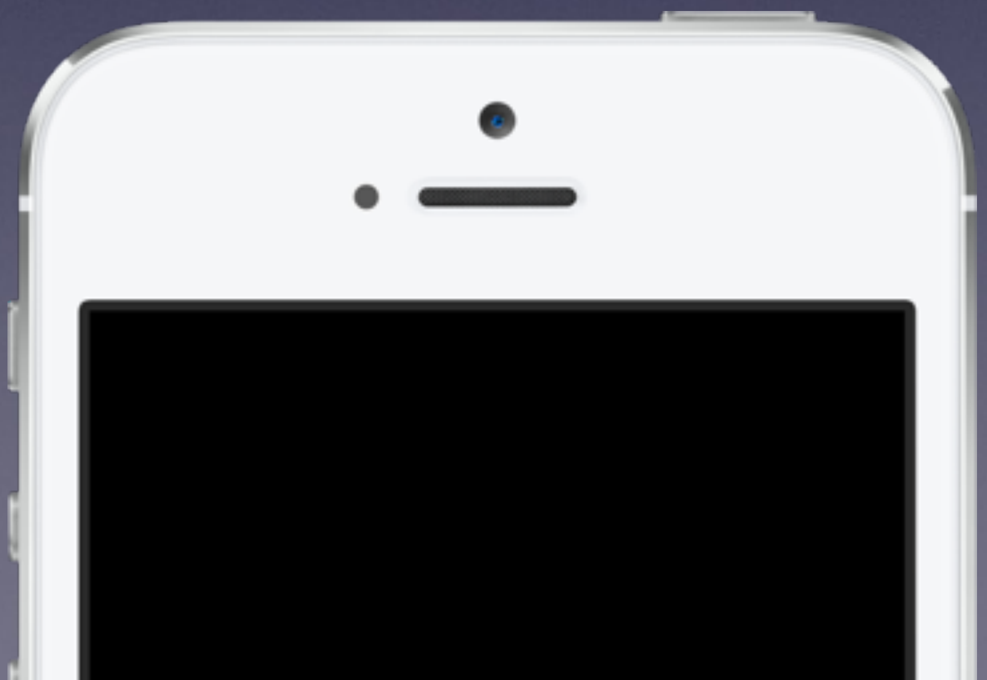
COMS W3101: Programming for iOS

Michael Vitrano



Persistence, Blocks, and Concurrency

- Survey of various persistence mechanisms
- NSCoding
- Working with the filesystem
- Blocks
- Concurrency with Grand Central Dispatch

App Life-cycle

- An app first launches, the first point to customize its behavior is in the app delegate's **-application:didFinishLaunchingWithOptions:** method
- At this point, the user is presented with your UI and can interact with the app
- When the app leaves the foreground, it does is not killed but suspended until the user reopens the app
  - When the user reopens the app from this state, the state of the app is exactly as it was before it was suspended
- The app will not live on forever though, eventually it will be killed to free up memory to be used by other apps

NSUserDefaults

- Provides a lightweight, key-value store to save user preferences across application launches
- You access the user defaults database through the **+standardUserDefaults** class method
- There are built in methods for storing integers, doubles, and objects
    - Objects must be one of the following types: **NSData**, **NSString**, **NSNumber**, **NSDate**, **NSArray**, or **NSDictionary**.
- The framework is responsible for managing the storage of the objects
- Not meant for storing large amounts of information, if you store too much, at some point it will crash
- Need to call **-synchronize** to force the values set at runtime to be saved to disk

NSUserDefaults

```
// Saving some basic player information into the user defaults
[[NSUserDefaults standardUserDefaults] setInteger:10
                                            forKey:@“playCount”];
[[NSUserDefaults standardUserDefaults] setObject:@“Michael”
                                           forKey:@“characterName”];
[[NSUserDefaults standardUserDefaults] synchronize];

//………… At some point in the future

NSInteger playCount = [[NSUserDefaults standardUserDefaults] integerForKey:@“playCount”];
NSString *name = [[NSUserDefaults standardUserDefaults] objectForKey:@“characterName”];
```

Core Data

- Core Data is a framework that “provides generalized and automated solutions to common tasks associated with object life-cycle and object graph management, including persistence.”
- Allows a you to create a schema that represents objects and relationships between them
- Provides mechanisms for migrating from old versions of your schemas to new ones
- Supports complex querying of your object graph to retrieve objects
- Complicated and has significant overhead to getting started with using it

NSCoding

- NSCoding is a protocol with two simple methods:

```
- (void)encodeWithCoder:(NSCoder *)coder;
- (id)initWithCoder:(NSCoder *)coder;
```

- In **-encodeWithCoder:** the object is responsible for serializing its state

- In **-initWithCoder:** the object is responsible for recreating its state from the serialization encapsulated in the **coder** object

NSCoding

```
@interface Player : NSObject <NSCoding>

@property (nonatomic) NSString *name;
@property (nonatomic) UIColor *avatarColor;
@property (nonatomic) NSInteger healthPoints;

@end

@implementation

- (void)encodeWithCoder:(NSCoder *)coder
{
     [coder encodeObject:self.name forKey:@“name”];
     [coder encodeObject:self.avatarColor forKey:@“avatarColor”];
     [coder encodeInteger:self.healthPoints forKey:@“healthPoints”];
}

- (id)initWithCoder:(NSCoder *)coder
{
     if (self = [self init]) {
          self.name = [coder decodeObjectForKey:@“name”];
          self.avatarColor = [coder decodeObjectForKey:@“avatarColor”];
          self.healthPoints = [coder decodeIntegerForKey:@“healthPoints”];

     }
     return self;
}

@end
```

NSCoding

- Using the **NSKeyedArchiver** and **NSKeyedUnarchiver** classes, we can turn NSCoding compliant objects into NSData and recreate them later:

```
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:codingObj];

id<NSCoding> obj = [NSKeyedUnarchiver unarchiveObjectWithData:data];
```

The Filesystem

- iOS is a Unix-based operating system and has a Unix-based filesystem as well
- Apps are only allowed read and write access to the data inside of their “sandbox”
    - This effectively prevents most types of inter-app communication
- There are three important directories in an app’s sandbox:
    - The application bundle: contains your nibs, images, etc. This is read-only
    - Documents directory: writeable permanent storage for saving user data
    - Caches directory: writable storage for storing temporary data
- We find these directories using **NSSearchPathForDirectoriesInDomains** function with the appropriate options

NSFileManager

- NSFileManager is the Cocoa abstraction for working with the filesystem

- We can access a shared instance using the **+sharedManager** class method

- NSFileManager provides a set of utility functions that allow apps to create directories, copy/move/delete files, query the contents of a directory

NSFileManager

```
NSString *documentsPath = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                               NSUserDomainMask,
                                                               YES) firstObject];
NSURL *documentsURL = [NSURL URLWithString:documentsPath];

NSError *error;
NSFileManager *fileManager = [NSFileManager defaultManager];
NSArray *documentsDirContents = [fileManager contentsOfDirectoryAtURL:documentsURL
                                             includingPropertiesForKeys:nil
                                             options:0
                                             error:&error];


for (NSURL *url in documentsDirContents) {
    NSLog(@“%@ is in the documents directory”, url);
}
```

NSData

- We use NSData instances to represent raw bytes of data in memory
- We use the methods on NSData instances to read binary data from disk into memory and from memory onto disk

```
- (id)initWithContentsOfURL:(NSURL *)url;
- (BOOL)writeToURL:(NSURL *)url atomically:(BOOL)atomically;
```

- These methods can be used in conjunction with the NSData instances returned using the NSCoding protocol

Blocks

- Similar to functions but are declared inline within other blocks of code

- Blocks have access to the variables defined with the scope in which it was defined

- Blocks begin their definitions with the **^** operator

```
[array enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop){
    if (idx < 3) {
        NSLog(@“found: %@”, obj);
    } else {
        *stop = YES;
    }
}];
```

Blocks

```
NSInteger maxValue = 3;
[array enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop)
{
    // We can access maxValue inside of the block
    // By default, variables defined outside of the block
    // are read only
    if (idx < maxValue) {
        NSLog(@“found: %@”, obj);
    } else {
        *stop = YES;
    }
}];
```

Blocks

```
// using the __block storage type, thirdObject becomes writable
__block id thirdObject;
[array enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop)
{
    if (idx == 3) {
        thirdObject = obj;
        *stop = YES;
    }
}];
```

Blocks

```
NSMutableDictionary *dictionary = [NSMutableDictionary dictionary];
[array enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop)
{
    if (idx == 3) {
        // we can message objects captured inside of the block
        // objects captured in a block are strongly referenced until
        // the block has been disposed of
        [dictionary setObject:obj forKey:@“third”];
        *stop = YES;
    }
}];
```

Blocks

```
// we can save blocks into variables for use later
void (^block)(id, NSUInteger, BOOL*) = ^(id obj, NSUInteger idx, BOOL *stop){
    if (idx == 3) {
        NSLog(@"the third object is: %@", obj);
        *stop = YES;
    }
};

[array enumerateObjectsUsingBlock:block];

// They can also be stored in collections
[mutableArray addObject:block]
```

Blocks

```
// When using a block with a specific signature, its useful to typedef
// a type for that block like so

typedef void (^array_iterator_t)(id, NSUInteger, BOOL*);

// The preceding slide then becomes:

array_iterator_t block = ^(id obj, NSUInteger idx, BOOL *stop){
    if (idx == 3) {
        NSLog(@"the third object is: %@", obj);
        *stop = YES;
    }
};

[array enumerateObjectsUsingBlock:block];
```

Blocks

```
// You can use the typedef to declare a property that saves a block of
// this type

@property (nonatomic, copy) array_iterator_t block;

// Similarly, it can be used as parameter in a method:

- (void)iterateOverInternalDataStructure:(array_iterator_t)block;
```

Blocks

```
// You can also to declare a property that saves a block of without
// at typedef


@property (nonatomic, copy) void (^block)(id, NSUInteger, BOOL*);

// Similarly, it can be used as parameter in a method:

- (void)iterateOverInternalDataStructure:(void(^)(id, NSUInteger,
BOOL*))block;

// Note that the parameter is not named with the carat character
```

Blocks

```
// You can be lazy about specifying return types

double (^block)(id) = ^(id obj){
    // the return type of the block is inferred from the return statement
    return [obj doubleValue];
};

// You can also be lazy when a block has no parameters

void (^block)(void) = ^{
    // We don't need to provide an empty set of ()
    NSLog(@“We’re just printing”);
};
```

Blocks

```
// Executing a block is no different than executing a function

void (^block)(void) = ^{
    NSLog(@“We’re just printing”);
};

// “We’re just printing” will now print to the console
block();

// You can assign the return value of a block as if calling a function
double doubleVal = blockThatReturnsDouble();

// If your block takes arguments, pass then as if it were a function
blockThatTakesAStringAndInteger(@“Hello World”, 3);
```

Grand Central Dispatch

- GCD is a C-based API for managing concurrent tasks
- The developer puts a task in the form of blocks onto a queue and the system will take those tasks off the queue and execute them
- There are two types of queues:
  - Serial: meaning that at most one block from that queue is executing at a given time and blocks are executed in the order in which they are enqueued
  - Concurrent: blocks are dequeued in the order in which they were enqueued, but multiple blocks can execute at one time and they may finish in any order

Grand Central Dispatch

- The system is responsible for creating threads on which to execute your blocks
  - The number of threads created will be optimized for the hardware on which your code is running
- GCD is crucial for moving work off of your application’s main thread so that your app’s UI remains responsive which executing expensive tasks

Grand Central Dispatch

- Creating queues:

```
dispatch_queue_t serialQ = dispatch_queue_create(“com.vitrano.serial”,
DISPATCH_QUEUE_SERIAL);

dispatch_queue_t concurrentQ =
dispatch_queue_create(“com.vitrano.concurrent”,
DISPATCH_QUEUE_CONCURRENT);
```

- Enqueuing a block:

```
dispatch_async(serialQ, ^{
     [self performExpensiveTask];
});
```

- Get built-in queues:

```
   dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_get_main_queue();
```

Grand Central Dispatch

- Asynchronously loading an image in the background:

```
- (void)loadImageFromURL:(NSURL *)url
{
    dispatch_queue_t q = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(q, ^{
        NSData *data = [NSData dataWithContentsOfURL:url];
        UIImage *image = [UIImage imageWithData:data];

        // All work with the UI must be done on the main thread.
        dispatch_async(dispatch_get_main_queue(), ^{
            self.imageView.image = image;
        });
    });
}
```