iOS Dev Crib Sheet

• As you dive into the deep end of the iOS development pool, the first thing to remember is that the “mother ship” holds the authoritative documentation for this endeavor — http://developer.apple.com/ios

• Some of this documentation is also available from within Xcode, so that is worth figuring out as well

• But, until then, here are some notes that may help get you started and acclimatized

In the Shadow of C

The iOS programming language is Objective-C: it is a superset of C, with object-oriented constructs

• This provenance drives virtually every idiosyncrasy in the language — the verbose syntax, lack of true garbage collection, occasional repetitiveness, and other things that quite frankly drive programmers who know other languages pretty batty

• But, it can call existing C code seamlessly; is this a worthwhile tradeoff? — Now there’s a fun language design discussion

• The question is academic for development though, as this is the single official iOS programming language
The Long Arm of Java

• As it turns out, “JavaScript” is not the only name that was motivated by the marketing mojo that Java used to have — in iOS, we have Cocoa (or, more precisely, Cocoa Touch)

• Where Objective-C is the language of iOS development, Cocoa Touch is its primary library

• To complete the triumvirate, we have Xcode, the primary development environment for the platform

A Bit of History

• All of these technologies trace their lineage, quite directly, to NeXTSTEP, the development platform of the NeXT computers from the late 1980s to 1990s

• This technology became the basis for Mac OS X, which in turn is the basis for what is now iOS

• This is why many of the classes you’ll see start with “NS” — yes, they go back that far

• Would you say that the technology is showing is age? — That is another fun discussion
Objective-C Tidbits

(This assumes that you are familiar with object-oriented programming concepts and terms)

- Objective-C classes are typically separated into .h files, which define what is publicly accessible about them, and .m files, which supply their implementation.

- A class’s definition is given in a block that starts with `@interface` and ends with (reasonably enough) `@end`.

- Its implementation, in turn, is delimited by (surprise) `@implementation` and `@end`.

- To distinguish Objective-C’s constructs from C, strikingly different syntax is used:
  - Methods are written in infix notation, preceded by – for instance methods and + for class methods, with colons : separating a method’s name segments from its parameters.
  - Method invocations are delimited by brackets [ ].
  - Certain declarations take the form of keywords preceded by the @ sign.

- Objective-C also has a form of closure, called blocks — they begin with a caret ^ followed by the block’s return type and parameters, both in parentheses.
• Constructors can actually be any class method that returns an instance of that class

• The typical “baseline” constructor separates object memory allocation (the alloc class method) from initialization (the init instance method):

  ```objc
  [[Classname alloc] init];
  ```

• Objective-C (in iOS) does not have automatic garbage collection — you generally invoke the retain method if you want to keep an object beyond the current scope then call release when you no longer need it

• If you want the gory details, look up “Objective-C memory management”

• Other concepts/keywords/types include:

  ◇ protocols — These are similar to Java’s interfaces; classes that satisfy a protocol specify it between angle brackets `< >`

  ◇ nil — This is Objective-C’s null object keyword

  ◇ self — This is Objective-C’s “this object” keyword

  ◇ id — This is Objective-C’s “generic object” data type

  ◇ "..." literals are standard C strings; to specify an actual string object — i.e., an instance of the NSString class — use @"..."

  ◇ IBOutlet and IBAction — These keywords mark properties and methods that are visible to user interface components (more on those later)

• Classes to learn about first: NSObject (duh), NSString, NSArray, NSDictionary, NSInteger, and NSNumber
iOS App Fundamentals

- Like all C programs, the entry point to an iOS app is the `main` function, typically in a file named `main.m`
- The `main` function eventually calls `UIApplicationMain`, which in turn instantiates your `app delegate` class — this is typically where your actual code starts
- The rest of your application is a collection of `model objects, views, and controllers` — Cocoa adheres to the MVC paradigm quite closely

Creating Views

- In Cocoa, views (user interface elements) may be created both programmatically (i.e., in Objective-C code) or visually, using an interface builder tool (separate in Xcode versions < 4, integrated after that)
- Visually created views are stored in separate files called NIBs (there’s that NeXTSTEP legacy again — “NIB” stood for “NeXT Interface Builder”)
- Special hooks associate the user interface in a NIB with a particular “owning” class (usually its controller)
Typical iOS Workflow

1. Initialize an iOS project using Xcode with one of its prebuilt templates — this creates a baseline app of a certain type (as determined by the template), with stubbed out methods and NIBs (sure, you can start up a project from absolutely nothing, but why bother when a lot of pieces are relatively standardized)

2. Start tweaking the NIBs to your liking (presumably you have a design already sketched out) — stay cognizant of iOS user interface guidelines while doing this

3. Modify the NIBs’ owning classes to correspond with the user interface elements that you have defined and the methods that they need to call:
   a. User interface elements that you will need to access programmatically should be defined as IBOutlet properties of the owning class
   b. Methods that your user interface elements will call (usually in response to user actions or events) should have the signature
      - (IBAction)methodName:(id)sender

4. Associate the NIBs’ user interface elements with their corresponding outlets and actions

5. Build, run, and test the app; rinse and repeat
Digging Deeper

• As with most object-oriented platforms, effective iOS development relies on familiarity with and proper use of its class library — get to know the structure of the available documentation so you can find the class that you need and the things that it can do

• Heed the MVC paradigm well: data and core computation should be separated into their own, “plain old” classes (model objects), and user actions (events) in views should call controller methods that then update or use these model objects

• Write unit tests — Objective-C/Cocoa’s testing framework can call unit tests in isolated functional parts (“logic tests”) or within the full context of the iOS app (“application tests”)

• Start putting your app on an actual iOS device — getting started here is a whole other set of hoops to jump, but is pretty straightforward once you get over the setup pains, not the least of which is interacting with the iOS Provisioning Portal web site at:


• Get to know the plethora of events, notifications, and transitions that an iOS app experiences, as well as what to do when these take place