Life as a Dev – Welcome!

There are two parts to this afternoon class. Afternoon breakouts/lectures are generally 1 hour long, however given today is a bit of a unique day, this afternoon class may run a bit longer (1.5h max).

iOS Curriculum Specifics

This is your first iOS-specific lecture. For that reason, we'll spend 5-10 minutes at the beginning talking about the iOS curriculum, including a week-by-week outline.

Git and Github

- Understanding the importance of version control - why we use it.
- Understanding the basic terminology of Git: add, commit, push, pull, branch, merge checkout.
- What are branches? and why do we use them?

Objective-C

- Understanding what is a variable and what it is made up of
- Understanding the difference between values and references
- Understanding nil and its importance in Objective C
- Understanding local scope and its importance

Foundation Class of the day: NSString & NSMutableString

iOS Curriculum Specifics

Week 1: C, Source Control, OOP Fundamentals	*Basics*
Week 2: iOS & MVC	*iOS fundamentals and the most commonly used design pattern.*
Week 3: View Controllers, Networking, Maps + EXAM! **	*More fundamentals, Core Location. Progress review.*
Week 4: Midterm Project + Swift	*Your first app. Swift.*
Week 5: CS Week	*Self-study and development*
Week 6: Databases and Testing	*Persisting your data – Core Data. Testing your applications to improve them.*
Week 7: Media + Animations	*Camera. Photography, AV recording. Animation as a design imperative to engage users. Games.*
Week 8: Advanced Topics, Final Project & Advanced lectures	*Sample topics include ARKit, HomeKit, ResearchKit, design patterns, reactive programming.*
Week 9: Final Project & Advanced lectures	
Week 10: Final Project & No Lectures	

Documentation. https://developer.apple.com/documentation/
Bookmark this link immediately. Learning to read documentation .

Lectures and Breakouts – Why and what they are for

Lighthouse lectures and breakouts are designed to cover the fundamentals required for you to understand iOS development. They are not designed to cover everything, time won't allow for that.

Each lecture and breakout is structured around a core component that you will need to build your first iOS apps.

Taken as a whole, the lectures and breakouts are designed to give you a thorough introduction to the tools and design philosophy used in developing iOS apps. More importantly, we will show you how to continue your development career as a professional iOS developer, and how to learn with your own resources and skills.

It's important to note that your assignments following lectures will not necessarily be related to their subjects. This is by design. You have a large array of tools to get to grips with in the next few weeks, and it's important to gain exposure to as many as you can here.

Impostor syndrome. A positive mindset vs a negative mindset.

Researching a problem.
Breaking down a problem into smaller tasks and then smaller tasks.
Debugging.
Researching other people's solutions.

Asking for help. Knowing when you need to ask for help.
Trying things out. Getting feedback. Mentors. Testing yourself.

Post-instruction notes.

Version control. Git. GitHub

Git provides us with the ability to recall any code in any project at any time. Git at its core is essentially a CHANGELOG. Changed files are recorded with a hash and these are grouped together and written to the log.

Basic Git usage:

Initialize a new repo	`git init` Creates a hidden `.git` folder in the repo.
See current state of repo	`git status` Altered but unadded files will be in red, altered and added files in green. It will also list the local repo's state against any remote directories you set up below.
Add files to staging area	`git add file1 file2 file3…` `git add -A` Adds all changed files Officially, `add` creates a snapshot (`commit` 'records' it)
Commit files from staging area to local repo log	`git commit -m "This was a hotfix"`

Show each file's changes from one commit to the next &
Show each commit's changes vs other commits in history.
	`git diff file1 file2`
	
	`git diff 78fae34 dd2e34a`

Add a remote folder storage location (ie GitHub, Bitbucket), called `origin` by convention	`git remote add origin https://yourRepoHere.git`

Push latest commit to remote* repo**	`git push -u origin master`
	`remote` = `origin`
	`repo` = `master` (default convention)

Checkout a specific commit to look at, or work on.
Create a new branch and checkout at the same time
	`git checkout 12345abc`
	`git checkout -b routerXPMNT`

Show all branches	`git branch` (More options later)

Merge two branches	[Developer is on branch `develop` and wants to update with the latest updates from master] `git merge master`

Resolving Git merge conflicts
When git finds two commits with different line contents, its default behaviour is to not fix any resulting conflicts, but rather to just flag them. Git does not have any semantic capabilities to understand the code it manages, merely the ability to show any differences it encounters.

When conflicts are found, the two different sources will be indicated directly in code through the use of multiple greater-than (>>>>) or less-than (<<<<) symbols.

It's up to you to fix these errors but this is a valuable skill to learn.

Xcode and Objective-C

Xcode is the default Apple integrated development environment (IDE), and includes a powerful, open source, freely-available compiler under the hood, LLVM, that sets the standard for dynamic, fast, and solid compilation. It acts as a window into the Cocoa Touch SDK and UI libraries, used extensively in iOS app development. It includes a fast code editor, a view debugging mode allowing for exploded view examination of UI code during runtime, and costs nothing.

Objective-C is the original systems and applications programming language used in developing first macOS applications, then iOS apps and other architectures. Currently it has been somewhat deprecated owing to the popularity of its replacement, Swift, which boasts advantages for the newly-minted developer by being much easier to parse and more useful for extension than Objective-C.

Learning Objective-C first before starting on Swift will help you in numerous ways. The most important of these is that a lot of legacy code out there is still around, and the vast majority of that legacy code is in Objective-C. Being familiar with ObjC will demonstrate greater knowledge, understanding more languages gives you more flexibility and perspective in understanding a given problem.

Variables in Objective-C
A variable is a location in memory, paired with a symbolic shortcut name or alias.

Primitive variables, or scalar variables, are simple, and consist of either character strings (ASCII, alphabets, etc.) or numbers of one sort or another (integers, floats, doubles – all rational numbers).

C and Objective-C variables are declared like this:

```
type variableName = someValue;
```

i.e.,
```
char inputText = 'Enter your name below:\n';
int curriculumWeeks = 10;
char ab, cd;                    // undefined
```

Value types and reference types

Value types

Scalar variables like these examples above are simply memory locations storing the literal data within.

So at memory location 0xfae3122ab, you'll find the text 'Enter your name below:\n' and at memory location 0xfae3122ac, the integer 10.

Reference types

C and Objective-C both use another way to list memory locations, through the use of pointers.

Pointers are references to memory locations, not to their data contents.

So a pointer to inputText is rather a reference or pointer to inputText's memory location, and not to 'Enter your name below:\n'.

What that means in practical terms is this:

If I have a value type – an integer variable, a character string variable, a float variable – and I later change the value, I've changed the data itself at that same memory location.

If I have instead a pointer to that same value type, I have a reference to its location in memory. That reference is NOT a reference to its data, which means that *data value can change* *but the pointer will not*. This has implications for keeping track of these data changes and memory, which will be discussed later.

You can declare a new pointer variable with an asterisk before the variable name:

```
type *variableName;
float *xCoordinate;
double *molesOfOxygen;
char *inputData;
```

As with scalars any further listing of the variables can omit the asterisk, it's only necessary to write in explicitly at declaration.

Nil, Null, 0 ...

'In Objective-C, there are several different varieties of *nothing*.'
--- Matt Thompson, NSHipster

In C's primitive variables, we can only represent nothing as 0.

For C reference types ie pointers, we say that nothing is represented as NULL, which is effectively equal to 0 as far as pointers themselves are concerned.

In other words, when we have no pointers, we have no references to any memory locations.

Objective-C extends this idea by introducing the concept of `nil`.

nil is an object pointer to nothing.

Let's unpack this. `nil` is an object pointer. This is already different from C: `NULL` represents no pointer. `nil` on the other hand **is** a pointer, a pointer to an object. Which object? Nothing.

In practical terms, here's where this is important. You will often need to check that you have something to refer to with variables, and by implication, ensure that there ***isn't nothing***. Remember this as it will be key to understanding the **object lifecycle** later on.

You might want to check that you have data to use, for example:

```
if data != nil {
    NSLog(@“I have a data object, as I should do”);
} else {
    NSLog(@“Where is my data??”);
}
```

Many programming languages have different methods to handle freeing up memory.
Expiring and clearing out invalid or obsolete references to memory that are no longer required.

What do we need to accomplish this?

Basically, we need two facets of the same thing:

1. We need to be able to track all memory allocations – how many RAM locations did we use? Which?
2. We need to be able to track all references to those allocations (ie reference counting)

The first in C is handled by malloc : The C memory allocator. However, reference counting in C (ie reference tracking) doesn't really exist.

Objective-C focuses on the second of these by implementing Automatic Reference Counting or ARC. ARC is not the same thing as garbage collection, like you would see in Java, where garbage collection is essentially scheduled cache and memory cleanup.

Objective-C Object Lifecycle
Objective-C accomplishes this cleanup through the use of NSObject. Every NSObject is allocated memory to start with, but none have any references yet: `nil` or 0. References are added as needed, and ARC keeps count. When the object has 0 references, the object is destroyed in memory and freed up for reuse.

The key difference between this and garbage collection is that freeing up memory in the Objective-C runtime happens automatically and not at predetermined times and states, as with garbage collection.

Unlike other programming languages (NB: Pay attention to this, you will need this when you learn about Swift optionals) Objective-C does not treat `nil` as a capital offense. You can happily pass messages to `nil`; your programs will not crash, because the messages that are passed are still going to an object. It's just a `nil` object.

The Foundation framework (think of it as the Objective-C Standard Library, if you know C++) adds yet another definition of nothing, `NSNull`. Unlike everything above, `NSNull` is an actual Foundation object, and it's designed for those classes or objects where `nil` entries cannot be used, namely `NSArray` and `NSDictionary`.

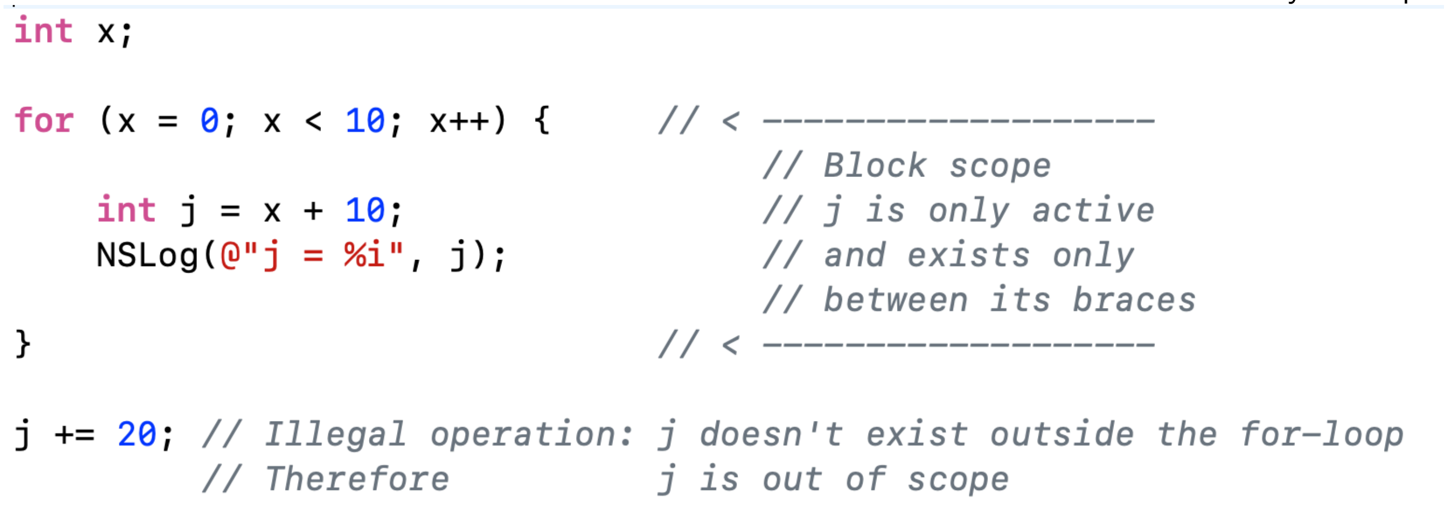
Variables and Scope
Objective-C programs have code split up between functions, classes, structs, and scalar types as discussed. These will have variables in order to store and manipulate data.

When you declare a variable in Objective-C you will decide whether or not if it is available to other parts of your program. This availability will be determined by where in the code it is declared, how it is declared, where its accessing code is located in your program.

This availability is officially called variable scope

Here is an example of block scope.

Courtesy technopia

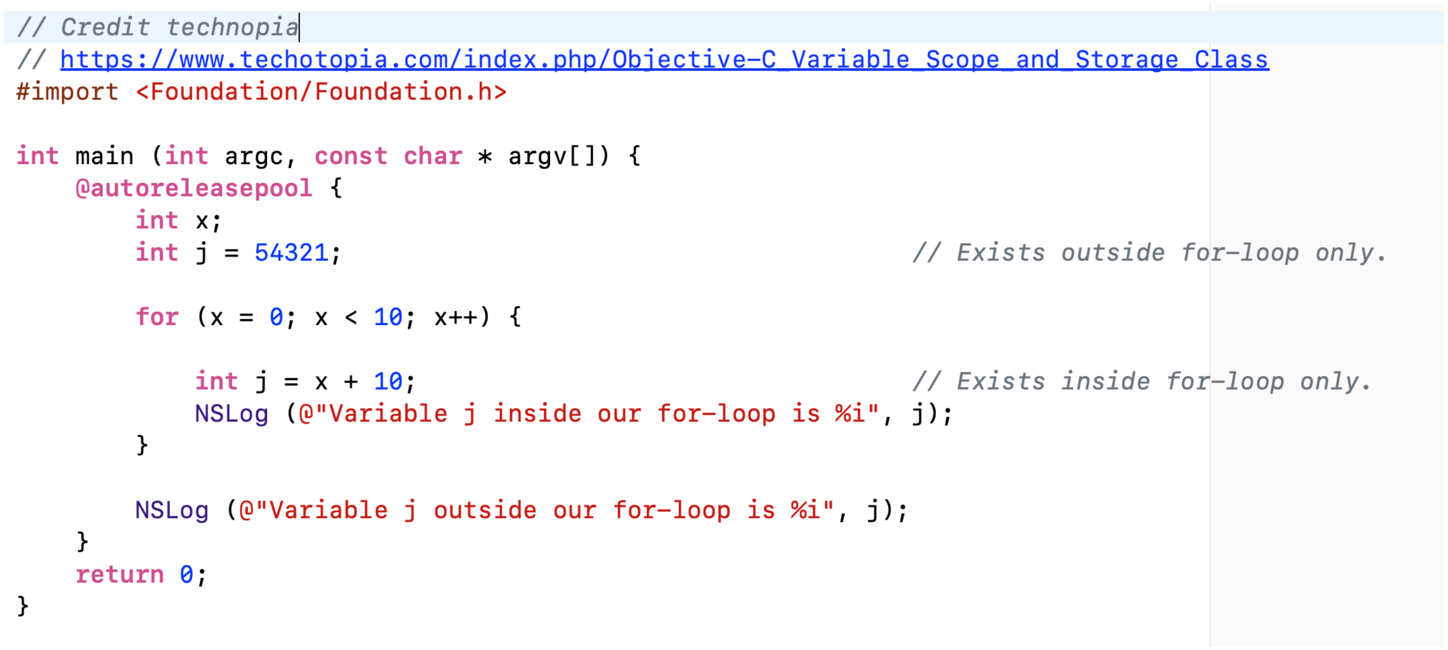


```
int x;

for (x = 0; x < 10; x++) {      // < ------------------
                                   // Block scope
    int j = x + 10;                // j is only active
    NSLog(@"j = %i", j);           // and exists only
                                   // between its braces
}                               // < ------------------

j += 20; // Illegal operation: j doesn't exist outside the for-loop
         // Therefore          j is out of scope
```

One side-effect of this is that you can have, and confuse, two j variables:



```
// Credit technopia
// https://www.techotopia.com/index.php/Objective-C_Variable_Scope_and_Storage_Class
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    @autoreleasepool {
        int x;
        int j = 54321;                              // Exists outside for-loop only.

        for (x = 0; x < 10; x++) {

            int j = x + 10;                         // Exists inside for-loop only.
            NSLog (@"Variable j inside our for-loop is %i", j);
        }

        NSLog (@"Variable j outside our for-loop is %i", j);
    }
    return 0;
}
```

What's wrong with the following code?

```
// Broken scope example
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]){

    @autoreleasepool {
        int j = 10;
        int k = 20;
        int result;

        result = multiply();

    }
    return 0;
}

int multiply() {
    return j * k;
}
```

Here is some sample output as a clue.

```
In function 'multiply':
error: 'j' undeclared (first use in this function)
error: (Each undeclared identifier is reported only once
error: for each function it appears in.)
error: 'k' undeclared (first use in this function)
```

Global scope variables are declared outside of any statement blocks and typically placed near the top of a source file. However, at this stage, these variables by default are NOT global; to make them actively global you need to take another step.

```
#import <Foundation/Foundation.h>

int myVar = 321;

int main (int argc, const char * argv[])
{
   @autoreleasepool {

        NSLog (@"myVar = %i", myVar);
   }
   return 0;
}
```

Let's include another file, `displayit.m` and function to use it.
Remember, we haven't 'activated' the `myVar` variable as truly global yet.

```
#import <Foundation/Foundation.h>

void displayit() {
    NSLog (@"MyVar from different source file is %i", myVar);
}
```

Here’s what we see when we try to compile it.

```
clang -framework Foundation main.m displayit.m -o main
displayit.m:7:59: error: use of undeclared identifier 'myVar'
NSLog (@"MyVar from different source file is %i", myVar);
```

The reason why this fails is because we have to confirm the myVar variable from whatever source file we use to get at it.

In this example, we have to tell the `displayIt.m` file that the `myVar` variable it’s looking for is outside itself.

We tell the compiler to hunt outside the caller:

```
#import <Foundation/Foundation.h>

extern int myVar;

void displayit() {
    NSLog (@"MyVar from different source file is %i", myVar);
}
```

File scope. We’ve seen how we can use global scope variables to spread access to data values across the whole of an app. But what if you want to restrict variables to just their own source file? This would make these variables, and more importantly, their data values, inaccessible to other parts of the program. This is a good thing, because it cuts down on one possible security risk, by making that data specific only to the file that has a justified use for it.

To do this, declare your variable as `static`, as follows, in main.m:

```
#import <Foundation/Foundation.h>

static int myVar = 543;
void displayIt(void);

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        NSLog (@"myVar = %i", myVar);
        displayIt();

    }
    return 0;
}
```

Remove extern from your `displayIt.m` file.

```
#import <Foundation/Foundation.h>

int myVar;

void displayIt() {
    NSLog (@"MyVar from different source file is %i", myVar);
}
```

A full list of variable storage class specifiers supported by Objective-C is as follows:
`extern` - Specifies that the variable name is referencing a global variable specified in a different source file to the current file.
`static` - Specifies that the variable is to be accessible only within the scope of the current source file.
`auto` - The default value for variable declarations. Specifies the variable is to be local or global depending on where the declaration is made within the code. Since this is the default setting this specifier is rarely, if ever, used.
`const` - Declares a variable as being read-only. In other words, specifies that once the variable has been assigned a value, that value will not be subsequently changed.
`volatile` - Specifies that the value assigned to a variable will be changed in subsequent code. The default behavior for variable declarations.

Foundation Class of the Day: NSString and NSMutableString

Consult the Apple developer documentation site first to get an overview:

`NSString` Class Reference
https://developer.apple.com/documentation/foundation/nsstring?language=objc

`NSMutableString` Class Reference
https://developer.apple.com/documentation/foundation/nsmutablestring?language=objc

See TutorialsPoint tutorial and Ry's Objective-C Tutorial, NSString chapter.

https://www.codzify.com/ObjectiveC/objectivecStringObjects

https://www.ios-blog.com/tutorials/objective-c-strings-a-guide-for-beginners/

https://www.techotopia.com/index.php/Working_with_String_Objects_in_Objective-C