Objective C is a language based on C with a lot of objective-oriented features added. Since it is a superset of C you can mix C and Objective C code freely.

Syntax Overview

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

Any line of code that begins with # symbol is a preprocessor directive. Line above basically means - find Foundation.h file and paste it here. Foundation is Apple's Objective C framework that contains fundamental classes we use in iOS and OS X development.

Creating Variables

```
int i = 7;
```

Line above declares an integer variable *i* and assigns it a value 7.

To create a string you would use a NSString class.

```
NSString *greeting = @"Hello World";
```

Note the asterisk (*) before "greeting". In Objective C all objects must be pointers and NSStrings are objects. All this means is that "greeting" is not a NSString, it is just a pointer to a place in memory where NSString exists.

Conditions

Basic conditional statement looks like this:

```
int numberOfCharacters = 280;

if (numberOfCharacters == 280) {
      NSLog(@"I could tweet this.");
}
```

Syntax for switch statement is as follows:

```
int i = 2;

switch (i) {
    case 1:
      NSLog(@"First Case");
      break;
    case 2:
      NSLog(@"Second Case");
      break;
    case 3:
      NSLog(@"Third Case");
      break;
    default:
      NSLog(@"Default Case");
}
```

This will print out “Second Case”. Note that all cases have implicit fallthrough and that is why we use break statement.

Loops

Most widely used form of loop in Objective c is a fast enumeration loop, written as follows:

```
NSArray *flavors = @[@"chocolate", @"vanilla", @"strawberry"];

for (NSString *flavor in flavors) {
    NSLog(@"Ice Cream with: %@", flavor);
}
```

This code creates an array of flavors and then loops through it to print out all the flavors. %@ is a format specifier and means - insert the contents of an object here. In our case the object is “flavor”.

If you prefer you can also use a C-styled for loop:

```
for (int i = 0; i < 10; i++) {
      NSLog(@"%d", i);
}
```

%d is a format specifier for int.

While loops are just as simple:

```
int number = 10;

while (number < 100) {
      NSLog(@"value of number: %d\n", number);
      number++;
}
```

Data Types

Strings

NSString is a class, which means that it is a reference type. A reference type is a type that, when passed to a function, returns a pointer to the same instance. In contrast, a value type would pass a copy of the value itself, not a pointer to the value.

You have already seen a basic way to create a string. They can also be initialized in different ways, including but not limited to:

```
const *char cString = "Some text.";
NSString *input = [NSString stringWithCString:cString
encoding:NSUTF8StringEncoding];

NSString *path = [[NSBundle mainBundle] pathForResource:@"somefile"
ofType:@"txt"];
NSString *text = [NSString stringWithContentsOfFile:path
encoding:NSUTF8StringEncoding error:nil];
```

NSString comes with a number of methods for manipulating strings. You can see some of them below:

```
stringByReplacingOccurrencesofString //replaces one string with another
stringByAppendingString //adds a new string to existing one
componentsSeparatedByString //splits a string and creates an array
integerValue //converts a string into integer
substringFromIndex //creates new string from a part of existing one
```

NSString is immutable, meaning it cannot be modified. To modify a string you would use NSMutableString class. You can create mutable strings in two ways:

```
// Create a mutable copy of an existing string
NSMutableString *greeting = [@"Good Morning" mutableCopy];

// Use one of the NSMutableString initializers
NSMutableString *exampleString = [NSMutableString stringWithCapacity:2048];
```

Note that assigning a NSMutableString to a NSString is not a good idea. The following code demonstrates why:

```
NSMutableString *first = [@"Some text." mutableCopy];
NSString *second = first;
[first setString:@"Some other text."];
NSLog(@"%@", second);
```

This code will print out “Some other text.”, even though you might have thought you had a immutable string with value “Some text”. Better way to write this would be to take copy of value during assignment:

```
NSString *second = [first copy];
```

Numbers

Since arrays and dictionaries in Objective C can only hold objects (and not primitive types), there was a need for object that would act as wrapper for numbers. This object is called NSNumber.

You can create NSNumber with initializers and you can even read number with different type. Example:

```
NSNumber *seven = [NSNumber numberWithInteger:7];
float floatSeven = [seven floatValue];
```

Primitive types for integers include NSInteger, NSUInteger, int64_t...

Arrays

You can create and loop over an array in the following way:

```
NSArray *colors = @[@"Red", @"Green", @"Blue"];

for (NSString *color in colors) {
    NSLog(@"Color %@? is a part of the RGB color model.", color);
}
```

Finding item in an array is done by indexing into it:

```
NSLog(@"G in RGB stands for: %@", colors[1]);
```

Older code used to index into arrays with objectAtIndex method:

```
NSLog(@"R in RGB stands for: %@", [colors objectAtIndex:0]);
```

To create mutable arrays you would use one of the initializers from the NSMutableArray or by mutableCopy method on an existing NSArray:

```
NSMutableArray *colors = [@[@"Red", @"Green", @"Blue"] mutableCopy];
```

Examples of working with arrays:

```
[colors insertObject:@"Yellow" atIndex:1];
[colors removeAllObjects];
```

Dictionaries

For dictionaries we use NSDictionary class. In objective C dictionaries are unordered.

```
NSDictionary *landmarks = @{
    @"Angra": @"Taj Mahal",
    @"Barcelona": @"Basilica of the Sagrada Familia",
    @"San Francisco": @"Alcatraz"
};

for (NSString *key in landmarks) {
    NSLog(@"%@ is located in %@", landmarks[key], key);
}
```

NSData

NSData and it's mutable counterpart NSMutableData are wrappers for byte buffers. Example of creating data from contents of a file:

```
NSData *data = [NSData dataWithContentsOfFile:filePath];
```

NSObject

NSObject is a universal base class of Objective C. Almost all other classes inherit from this class. NSObject provides a number of useful methods such as copy, mutableCopy, respondsToSelector, conformsToProtocol and so on.

NSError

NSError class is used to report errors from failed function calls.

```
NSError *error;
NSString *stringFromFile = [NSString
stringWithContentsOfFile:@"somefile.txt" usedEncoding:NSUTF8StringEncoding
error:&error];
```

Classes

When creating new class you will have to deal with two files: YourClassName.h and YourClassName.m. These are called header file and implementation file, respectively. In header file you define methods and properties of your class, and in .m file you implement those methods.

Methods

Syntax for writing methods in Objective C is as follows:

```
- (void)printName:(NSString*)name {
    NSLog(@"%@", name);
}
```

The “-” marks the start of a regular method. If we had “+” instead, that would mean we are creating static method.

We place return type in parentheses. This method’s return type is “void” which means that nothing is returned. That is followed by method name and parameters. In this case we only have one parameter - name of type NSString.

Properties

To understand properties we need to understand instance variables. Instance variable or ivar for

short is a simple pointer to an object. They should only be accessed from implementation and not from outside. They are declared in either .h or .m file:

```
NSNumber *someNumber;
NSString *someString;
```

Property is a method that gets and sets the value of an instance variable. To define a property you write:

```
@property NSNumber *someNumber;
@property NSString *someString;
```

When you create a property it automatically generates accessor methods (getter and setter).

Properties can have attributes attached to them. Property attributes change a way in which the property works. Property can be atomic or nonatomic, readonly or readwrite, strong or weak.

Atomic is default attribute. If you don't write anything your property will be atomic. Atomic property is guaranteed, if you read from it, to return a value. Basically, it means, that reading a value at the same time as it's being written to on another thread won't produce garbage data.

Nonatomic is the opposite of atomic, meaning that you have to worry about race conditions.

Readonly properties don't have setters since you can only read from them.

Readwrite is a default attribute and it means - generate both getter and setter for this property.

Default attribute strong means that you have a reference to an object and as long as you keep it, that object will not be deallocated.

Weak attribute creates weak reference to an object meaning that you don't hold it alive. As soon as all other strong references to the same object are released the object is deallocated.

```
// Both of these mean the same since strong, atomic and readwrite are all
default attributes
@property NSString *someString;
@property (strong, atomic, readwrite) NSString *someString;
```

Categories

You can use categories to add new methods to a class, without the need for subclassing. This is useful if we want to add new methods to classes for which we don't have access to source code or we are not allowed to change.

When creating a category we also have two files, .h and .m. Most people name category files in the following fashion: ClassToExtend+CategoryName.h and ClassToExtend+CategoryName.m.

Header file would look like this:

```
#import "ClassToExtend.h"

@interface ClassToExtend (CategoryName)

// define new methods

@end
```

Implementation file:

```
#import "ClassToExtend+CategoryName.h"

@implementation ClassToExtend (CategoryName)

// implement new methods

@end
```