HACKING WITH SWIFT

Objective-C fo
Swift Devel ers

FREE SAMPLE

Paul Hudson

Chapter 1
Overview

Concepts

We're going to jump into the deep end and look at the biggest ways Objective-C differs from Swift. I really want to dive into some code, but there a few things you need to understand first so please bear with me – the code is coming soon, I promise!

The most important thing to know about Objective-C is this: it's a strict superset of C, which is a language that's over 40 years old. This means that valid C code is also valid Objective-C code, and you can mix and match them freely. You can even use C++ with Objective-C, which usually has the moniker Objective-C++, but this is less common.

C and C++ bring with them a lot of baggage, and the places where C and Objective-C come together are a bit rough around the edges, but it does mean it's easy to make use of C- and C++-based code in Objective-C projects. The first game I ever wrote for iOS was written in Objective-C++, where a tiny amount of Objective-C acted as a shim to call my underlying C++ code.

One immediately obvious piece of baggage are header files: when you create a class in Objective-C, it's made up of YourClass.h (a header file) and YourClass.m (the implementation file). The "m" originally stood for "messages", but most people today consider it the "iMplementation" file. Your header file describes what the class exposes to the outside world: properties that can be accessed and methods that can be called. Your implementation file is where you write the actual code for those methods.

This split between H and M doesn't exist in Swift, where an entire class or struct is created inside a single file. But in Objective-C it's important: when you want to use another class, the compiler only needs to read the H file to understand how the class can be used. This lets you draw on closed-source components such as Google's analytics library: they give you the H file that describes how their components work, and a ".a" file that contains their compiled source code.

The second immediately obvious piece of baggage is the C preprocessor. This gets its own chapter later in the book for people who really want to dig in to how it works, but the concept is quite simple: the preprocessor is a compilation phase that takes place before the Objective-C code is built, which allows it to rewrite your source code before it's compiled. This doesn't exist in Swift, and with good reason: the main reasons for using it are header files (which Swift doesn't have) and creating macros, which are code definitions that get replaced when your source code is built. For example, rather than writing 3.14159265359 repeatedly, you can create a macro called `PI` and give it that value – it's a bit like a constant in Swift, but as you'll see in the Preprocessor chapter these macros can do a lot more, which is both good and bad.

Easy differences

Swift is a much more advanced language than Objective-C, and as such has some features that simply do not exist in Objective-C.

Specifically, Objective-C does not support the following:

- Type inference.
- Operator overloading.
- Protocol extensions.
- String interpolation.
- Namespaces.
- Tuples.
- Optionals.
- Playgrounds.
- `guard` and `defer`.
- Closed and half-open ranges.
- Enums with associated values.

Structs exist in Objective-C, but are used much less frequently than in Swift – Objective-C is, as you might imagine given its name, aggressively focused on objects!

Objective-C shares most operators with Swift, although it has retained the `++` and `--` operators that were deprecated in Swift 2.2. The nil coalescing operator is written as `?:` rather than `??`.

How things are named

Early versions of Swift – 1.0 to 2.2 – used almost identical naming conventions for methods and properties as Objective-C did. In Swift 3.0, Apple introduced the Great Cocoa Renamification, which involved renaming almost every method and property to be "more

Swifty", which caused pretty much every existing project to break until it was upgraded to use the new naming conventions.

As you're learning Objective-C now, you'll need to perform the Great Cocoa Renamification in reverse as you work, effectively figuring out the names of Objective-C methods by reversing Apple's process.

Here's what that means in detail:

1. In Objective-C, the first parameter to a method does not have a label, so the label for the first parameter is usually part of the method name. For example, `UIFont.preferredFont(forTextStyle:)` in Swift is written as `[UIFont preferredFontForTextStyle:]` in Objective-C.
2. Objective-C likes to repeat the names of things to make its methods clear. That means the `append()` method of `NSAttributedString` in Swift becomes `appendAttributedString` in Objective-C, and the `components(separatedBy:)` of strings becomes `componentsSeparatedByString:`.
3. Some things that are properties in Swift are methods in Objective-C. For example, `UIColor.blue` in Swift is `[UIColor blueColor]` in Objective-C.
4. Many class names need to have an "NS" prefix before them in Objective-C. For example, `NSUserDefaults`, `NSFileManager`, `NSNotificationCenter`, and `NSUUID`. Other than that they work identically to their Swift counterparts.
5. Objective-C uses a C-based API for Core Graphics and Grand Central Dispatch. This was also how Swift worked in Swift 2.2 and earlier, and I'm afraid it involves some fairly long function names such as `CGContextSetFillColorWithColor()`.
6. Some constants in Swift have been namespaced, whereas in Objective-C they are global. For example, `NSTextAlignment.left` in Swift is just `NSTextAlignmentLeft` in Objective-C.
7. If you are using a method that has a completion closure, it's required in Objective-C even when it's nil.

Realistically, it's the combination of 1 and 2 that cause the biggest difference. Here are some examples in Swift and Objective-C so you can see what you're working with – notice how

Objective-C adds lots of extra words into its method names so they are unmistakably clear:

```
// find the index of an item in an array
names.index(of: "Taylor")
[names indexOfObject: @"Taylor"];

// get the current UIDevice
UIDevice.current()
[UIDevice currentDevice];

// replace a string
"Hello, world".replacingOccurrences(of: "Hello", with:
"Goodbye")
[@"Hello, world" stringByReplacingOccurrencesOfString:@"Hello"
withString:@"Goodbye"];

// dismiss a view controller
dismiss(animated: true)
[self dismissViewControllerAnimated:true completion:nil];
```

Once you've had to mentally convert a dozen or so method names, you'll find it becomes second nature. Just ask yourself the question, "what repetitive words could I add to this method to make it longer?" Honestly, it's almost as if Apple made their money selling replacement keyboards…

Namespaces

The lack of namespaces in Objective-C might not make much sense at first, but it bears some explanation. A namespace is a way to group functionality together in discrete, re-usable chunks. When you namespace your code, it ensures the names you use for your classes don't overlap the names other people have used because you have additional context. For example, you could create a class called Person and not have to worry about Apple creating another class called Person, because the two wouldn't conflict. Swift automatically namespaces your code, so that your classes are automatically wrapped inside your module – something like

YourApp.YourClass.

Objective-C has no concept of namespaces, which means it requires that all class names be globally unique. This is easier said than done: if you use five libraries, those library might each use three other libraries, and each library might define lots of class names. It's possible that library A and library B might both include library C – and potentially even different versions of it.

This creates a lot of problems, and Apple's solution is simple and pervasive: use two-, three-, or four-letter prefixes to make each class name unique. Think about it: UITableView, SKSpriteNode, MKMapView, XCTestCase – you've been using this prefix approach all along, perhaps without even realizing it was designed to solve an Objective-C shortcoming!

What, no optionals?

Objective-C doesn't have any concept of optionals, which will make about half of you cheer and the other half worry. This caused all sorts of issues when Swift was first announced: all the Objective-C APIs had to be imported into Swift, and that meant trying to decide whether a `UIView` was actually a `UIView?` – was the view definitely there, or could it actually be nil?

In Swift this distinction really matters: if you try to use a value that is actually nil, your app crashes. But in Objective-C, working with nil values is perfectly OK: you can send a message to a nil object, and nothing happens.

Let me repeat that, perhaps with some bold text: **you can send a message to a nil object, and nothing happens.** This is unthinkable in Swift, because doing it is considered programmer error. If you run some code and nothing happens, your first port of call should be checking whether you messaged nil, because it's silently ignored.

To help bridge Objective-C and Swift, Apple introduced some new keywords that sort of help fill the hole left by optionals. It then audited its own APIs to use these new keywords, which is why over time some APIs in Swift have changed their optionality. These are covered in the Nullability chapter much later in the book – they bring helpful improvements to Objective-C, but adoption is extremely low so you might never see them in the wild.

Safety

Living without optionals and being able to message nil objects should give you some hints that Objective-C is less safe than Swift.

However, the truth is that this lack of safety goes much deeper: Objective-C lets you force one data type into another, it only recently introduced the concept of generics (e.g. arrays that can hold only strings), it has none of the advanced string functionality that lets Swift mix ASCII and emoji with ease, it lets you read array values that don't exist, your `switch` blocks don't need to be exhaustive, it makes almost everything a variable rather than a constant, and much more.

When you're coming from the Swift world, these features seem like bugs, and make Objective-C appear a bit like the wild west. To some extent, that's true: if you're writing natural Swift, then Objective-C is so laid back that you'll wonder how Apple managed to build their whole ecosystem on top of it. However, think for a moment what it's like for Objective-C developers coming to Swift: suddenly the compiler has become significantly more pedantic, and you need to spell out every last thing in order to make your code build. "Yes, I really do want to put that `UInt` into an `Int`" is quite tiresome at first, and simply wasn't necessary in Objective-C.

There is one small thing you can do to help make Objective-C a little bit less dangerous: whenever you create an Objective-C project, go to the Build Settings tab in Xcode and set "Treat Warnings as Errors" to Yes. That one change will stop you from making some terrible mistakes with the language, for example trying to squeeze a number into a string.

Basic syntax

Let's look at some basic Objective-C now: variables, conditional statements, **`switch`** blocks, and loops.

The best way to experiment with these is to create a new project in Xcode: go to File > New Project, then choose macOS > Application from the left-hand side, and Command Line Tool. Yes, I realize you probably haven't made many command-line apps before, but trust me on this: it's the closest thing Objective-C has to a playground, and helps keep things simple.

I know it's obvious, but: make sure you choose Objective-C for your project's language! As you're coming from the world of Swift, that will be the default language option, so please change it.

What's in the template?

When you create a project from this template, you'll be given only one file: main.m. Inside there are only a few lines of code, but even those few introduce several major concepts. You should see something like this:

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

int main(int argc, const char * argv[]) {
   @autoreleasepool {
      // insert code here...
      NSLog(@"Hello, World!");
   }
   return 0;
}
```

If you have some C experience, you'll recognize most of that already as just being the regular entry point for command-line applications. But there are two parts that are unique to Objective-C: **`@autoreleasepool`** and **`@"Hello, World!"`**.

The **`@`** symbol gets abused extensively in Objective-C, so you'd better get used to typing it a lot. What it means is "this next bit is Objective-C, and definitely not C." **`NSLog()`** is a function akin to **`print()`** in Swift, and the template code writes out a basic message. Without the **`@`** sign before "Hello, World!" that message would be interpreted as a C string: an array of characters in ASCII, ending with a 0. Objective-C, like Swift, has its own string data structure that offers Unicode compatibility, methods for manipulation, and more. **`NSLog()`** expects to receive one of these Objective-C strings, not a C string, which is why the **`@`** is required.

The **`@autoreleasepool`** means "I'm going to be allocating lots of memory; when I'm finished, please free it up." Everything inside the opening and closing braces forms part of this pool, which right now is the entire program.

It's worth me briefly mentioning the C code too, in particular the way the function is written. Here it is again:

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

And here's what each thing means:

- **`int`**: This function returns an integer.
- **`main`**: The function is named **`main()`**.
- **`int argc`**: The first parameter is an integer called **`argc`**.
- **`const char * argv[]`**: The second parameter is an array of strings called **`argv`**.

This **`main()`** function, with those parameter, is the standard way to create command-line programs, and it will automatically be called when the program is run.

A few other small things before we move on. First, note that **`return`** is used to return a value from a function, just as with Swift. Second, every statement must end with a semi-colon. In our code, that means **`NSLog()`** and **`return`** both end with a semi-colon. Third, **`//`** is a comment, like Swift.

Importing headers

There was one line I didn't discuss, which was the very first one: **`#import <Foundation/Foundation.h>`**. This is a preprocessor directive, which I mentioned briefly already. It means this code gets replaced by the preprocessor even before your code gets built. Any line of code that begins with `#` is a preprocessor directive, so watch out for it.

This particular directive means "find the header file for Foundation (Apple's fundamental Objective-C framework), and paste it here." The preprocessor literally takes the contents of Foundation.h – a header file that itself imports many other headers – and copies it in place of that `#import` line.

If you've written C or C++ code before, you'll be more familiar with the `#include` directive, which almost does the same thing. However, `#import` has a subtle extra feature that makes it much nicer to use: if you `#import` a header it will only ever be included once, whereas with `#include` it can potentially be included several times. C programmers often workaround the problems of `#include` by writing header guards, but Objective-C's `#import` does all that for you.

When you `#import` a system library, you place the library's name in angle brackets, aka Pulp Fiction brackets. For example, `#import <UIKit/UIKit.h>`. However, when you import your own header files, you use double quotes, like this: `#import "MyClass.h"`. This distinction is important: using angle brackets means "search for this header in the system libraries," and using double quotes means "search for this header in the system libraries, but also in my project."

Creating variables

Objective-C does not support type inference, and, unlike Swift, almost everything is created as a variable. In practice, this means you need to tell Xcode the type of every piece of data you want to work with.

To get started, replace the existing `// insert code here...` comment with this:

```
int i = 10;
```

That creates a new integer and assigns it the value 10. Notice there's no `let` or `var` in there – these things are variable by default. If you want to make it a constant, you should use this instead:

```
const int i = 10;
```

However, you're swimming against the current: few people bother to do this.

To create a string, you need to use the **`NSString`** class. Yes, it's a class rather than a struct; yes, the "NS" is another namespace prefix; and yes, you need to use the **`@`** symbol just like before.

Try writing this:

```
NSString str = @"Reject common sense to make the impossible
possible!";
```

When you try to build that, Xcode will give you an error message: "Interface type cannot be statically allocated." This is an example of a Really Bad Error Message, which ought to make Swift developers feel right at home.

What Xcode means is that any kind of object, like **`NSString`**, must be allocated using a special approach called *pointers*. If you think back to how hard optionals were to grasp in Swift, pointers are just as bad in Objective-C. I'm going to go into more detail on pointers later, but the least you need to know is this: a pointer is a reference to a location in memory where some specific data lives. If you imagine a photo that took up 30MB of RAM, you wouldn't want to copy all that data around each time you used the photo. Instead, you can pass around a pointer that specifies where in RAM the 30MB is, and that's good enough.

In Objective-C, all objects must be pointers, and **`NSStrings`** are objects. So, we need to write this instead:

```
NSString *str = @"Reject common sense to make the impossible
possible!";
```

Note the asterisk, which is what marks the pointer. So, **`str`** isn't an **`NSString`**, it's just a pointer to where an **`NSString`** exists in RAM.

Let's briefly look at one more data type: arrays. These are called **`NSArray`** in Objective-C, and you need **`@`** to start the array. I'll make an array that contains strings, so I'll need an **`@`** for each of the strings, like this:

```
NSArray *array = @[@"Hello", @"World"];
```

I'll be going into more detail about strings and other data types later on, but for now you know enough about creating variables to move on.

Conditions

Conditional statements mostly work the same as in Swift, although you must always type parentheses around your conditions. These parentheses, like the line-terminating semi-colons, are often accidentally missed off when you're coming from Swift, but Xcode will refuse to compile until you fix it. This says a lot about Objective-C: it won't bat an eyelid while you commit typecasting war crimes, but will throw a tantrum if you forget a semi-colon.

Here's a basic conditional statement:

```
int i = 10;

if (i == 10) {
   NSLog(@"Hello, World!");
}
```

However, Objective-C does introduce one twist, which simultaneously introduces a whole new class of bugs while saving you a couple of keystrokes: if the content of your conditional statement is just one statement, you can omit the braces. For example, these two pieces of code do exactly the same thing:

```
if (i == 10) {
   NSLog(@"Hello, World!");
} else {
   NSLog(@"Goodbye!");
}

if (i == 10)
   NSLog(@"Hello, World!");
```

```
else
   NSLog(@"Goodbye!");
```

As you're just at the stage of learning Objective-C, I would suggest you stay away from the latter option. If you desperately wish to avoid braces, at least write your **`if`** statement on a single line, like this:

```
if (i == 10) NSLog(@"Hello, World!");
```

The advantages of this syntax are dubious at best.

Switch/case

If there's one thing you're guaranteed to screw up, it's **`switch/case`** in Objective-C. The reason for this is two-fold: first, Objective-C is significantly less powerful than Swift here so you need to do more work yourself, and second case statements have implicit fallthrough. This is the *opposite* of Swift, and means you nearly always want to write **`break;`** at the end of **`case`** blocks to avoid fallthrough.

First, a basic example:

```
int i = 20;

switch (i) {
   case 20:
      NSLog(@"It's 20!");
      break;

   case 40:
      NSLog(@"It's 40!");
      break;

   case 60:
      NSLog(@"It's 60!");
      break;
```

```
    default:
        NSLog(@"It's something else.");
}
```

Notice again the parentheses for **`switch (i)`**. Run that now and you should see "It's 20!" printed out, but try removing the **`break;`** statements and you'll see what I mean about implicit fallthrough: it will print "It's 20!" then "It's 40!", "It's 60!" and "It's something else." one after the other. In Objective-C, not having **`break`** is equivalent to adding **`fallthrough`** in Swift.

Objective-C does have support for pattern matching, but it's limited to range: you write one number, then **`...`** with a space on either side, then another number, like this:

```
switch (i) {
    case 1 ... 30:
        NSLog(@"It's between 1 and 30!");
        break;

    default:
        NSLog(@"It's something else.");
}
```

There's one catch you need to be aware of, which is a language corner case that's guaranteed to bite you sooner or later. It might sound crazy, but here's the rule: you can't use the first line of a **`case`** block to declare a new variable unless you wrap the **`case`** block in braces.

To demonstrate this, consider the code below:

```
switch (i) {
    case 10:
        int foo = 1;
        NSLog(@"It's something else.");
}
```

Objective-C doesn't require that **`switch`** blocks be exhaustive, meaning that we don't need a **`default`** case – that code would be perfectly valid, if it were not for the fact that I try to declare a new variable straight after the **`case`**.

There are two ways to fix this problem: either place braces around the contents of the **`case`** block, or simply make the **`NSLog()`** line come first. Both of these two are legal:

```
switch (i) {
   case 10:
   {
      int foo = 1;
      NSLog(@"It's something else.");
   }
}

switch (i) {
   case 10:
      NSLog(@"It's something else.");
      int foo = 1;
}
```

Loops

Objective-C has the full set of loop options, including the C-style **`for`** loop that was deprecated in Swift 2.2.

Let's start off with the most common loop type, known as fast enumeration:

```
NSArray *names = @[@"Laura", @"Janet", @"Kim"];

for (NSString *name in names) {
   NSLog(@"Hello, %@", name);
}
```

Remember when I said that the **`@`** symbol was abused extensively in Objective-C? That one snippet of code has six of them – it's almost as if Apple makes most of its money shipping replacement keyboards to developers who have worn down their **`@`** keys.

That code snippet creates an array of names, then loops over each one and prints a greeting, The syntax for **`NSLog()`** might seem particularly bizarre at first, but it's a result of Objective-C not having string interpolation. **`NSLog()`** is a variadic function, and combines the string in its first parameter with the values of the second and subsequent parameters. The **`%@`** part is called a format specifier, and means "insert the contents of an object here" – which in our case is the **`name`** variable.

Use you can use C-style **`for`** loops like this:

```
for (int i = 1; i <= 5; ++i) {
   NSLog(@"%d * %d is %d", i, i, i * i);
}
```

`%d` is another format specifier, and means "int". Now that I've used three in one call, you can see more clearly how they are replaced in the string: the first **`%d`** matches the first **`i`**, the second **`%d`** matches the second **`i`**, and the third **`%d`** matches the **`i * i`**.

You can use **`while`** just like in Swift, and **`do/while`** is identical to Swift's **`repeat/while`** construct.

As with conditions, you can omit braces from loops if your loop body contains only one statement. This has the same dubious cost/benefits, so use it with caution.

Calling methods

There's a whole chapter on writing methods later on, and I'll be introducing methods on a case-by-case basis when looking at common data types, but before all of that I at least want to show you how methods look.

Consider the following Swift:

```
let myObject = new MyObject()
```

In Objective-C, several things change. First, **`new`** is a message you send to the **`MyObject`** class, like this:

```
MyObject *myObject = [MyObject new];
```

As you can see, you write an opening bracket, your object name, a space, then your message name, then a closing bracket. Note: although this is technically called "sending a message", I'll be referring to it as "calling a method" from now on because that's what everyone except Apple calls it.

Where things get tricksy is when you want to call two methods at once. In Swift you might write something like this:

```
myObject.method1().method2()
```

In Objective-C, you need to balance the brackets on the outermost left-hand side, like this:

```
[[myobject method1] method2]
```

When you're writing Objective-C, you often know exactly what line of code you want to write so you sometimes start by writing two, three or even four open brackets. Xcode does attempt to complete your brackets for you, but sometimes gets it wrong, which is thoroughly annoying.

Once particular place when you are likely to use two method calls on a single line is when creating objects. You've already seen the **`new`** method, which allocates memory for an object and initializes it with some default information. However, you can also run those two parts individually: allocate some memory with one method, then initialize a value with a second, like this:

```
MyObject *myObject = [[MyObject alloc] init];
```

The **`alloc`** is run first to set aside enough RAM to store the object, then **`init`** is run to place a default value into the object. The reason for this separation is down to helper functions, which in Swift get mapped to all the initializers you're used to: do you want to create a string from a file, from a URL, from a format, or something else?

Objective-C method calls look similar to Swift's, although brackets are used rather than parentheses, and you don't use commas between parameters. The lack of commas means that it is stylistically preferred not to place a space after the colon for named parameters. Here's an example in both Swift and Objective-C:

```
myObject.executeMethod(hello, param2: world)
[myObject executeMethod:hello param2:world];
```

We're going to go in to much more detail on methods later on, but for now you know enough to continue.

Nil coalescing

As with Swift, one useful way of ensuring a value exists is to use nil coalescing. Objective-C doesn't have a dedicated `??` operator for this, but instead allows you to hijack the ternary operator, `?:`.

For example, this will print a name or "Anonymous" depending on whether the `name` variable has a value:

```
NSString *name = nil;
NSLog(@"Hello, %@!", name ?: @"Anonymous");
```

Pointers

If pointers confuse you at first, don't worry about it: they confuse almost *everyone* at first. Worse, you even get pointer pointers, and even pointer pointer pointers – although to be fair anything beyond double pointers are considered bad style.

A regular variable contains a value, for example a house object. A pointer contains a pointer to the location of the house, like a signpost – it's much smaller than the real thing, and all it does it say "it's over there." Pointers allow objects to be passed around efficiently, because if you send the house object into a function all you're doing is sending a number in, which is the location of the house in RAM.

To continue the house metaphor, imagine a white house that had three signposts pointing to it. If you repaint the house so it's red, all three signposts are now pointing to the red house. You don't have a situation where one is pointing to a new red house and the other two are pointing to the old one. The same is true with pointers: if you have three pointers that point to the same object in memory, and you change that object, that change happens to all the pointers.

For now, you need to know that all Objective-C objects must be pointers, so you'll be using lots of asterisks. A bit later on you'll see an example of pointer pointers to handle errors.

Constant pointers

Objective-C developers create variables rather than constants by default, which is the opposite mindset of Swift developers. In fact, if you asked a random Objective-C developer "how do you make a constant string?" I'd say it's 50/50 whether they can give you the right answer first time.

To put this into context, the below code will not compile:

```
const i = 10;
i = 20;
```

That creates a constant integer then tries to change it, which is clearly bad. But this code works fine:

```
const NSString *first = @"Hello";
first = @"World";
```

You can even write this, which evaluates to the same thing:

```
NSString const *first = @"Hello";
first = @"World";
```

The reason for this is subtle but important: both of those lines mean "I want to ensure the string doesn't change, but I don't mind if the pointer does." Remember, all objects are pointers, so this is equivalent to saying, "I don't mind if you move your signpost to point somewhere else, as long as you don't change my house."

`NSString` is an immutable class, which means its value cannot be changed once it has been created. When you think you're changing its value, what's actually happening is that the old string is destroyed, a new one is created, and the pointer is updated to reflect the change.

We can demonstrate this by using the **`%p`** format specifier for **`NSLog()`**, which means "print the pointer of this object." This is useful for debugging purposes, because it allows you to track the specific value of an object in memory. In our case, we can see the pointer address change as a new object is produced. Try this code:

```
NSString *first = @"Hello";
NSLog(@"%p", first);
first = @"World";
NSLog(@"%p", first);
```

When I ran that, I got the following output:

```
2016-05-06 11:56:55.204 OCTest[57100:15178322] 0x100001038
2016-05-06 11:56:55.205 OCTest[57100:15178322] 0x100001078
```

So, at first the string lived at memory address 0x100001038, but after it was at 0x100001078 – the pointer has been moved.

If we want to create a string that *can't* be changed, what we need is a constant pointer. The **NSString** itself is effectively already **const** because it's immutable, so we now just need to make sure no one moves our signpost. To do this, you need to move the **const** keyword *after* the pointer's asterisk, like this:

```
NSString * const first = @"Hello";
```

The size of integers

I’ve been using the `int` data type in the examples so far, because it’s what you’re used to in Swift. But in practice, you’re far more likely to use a different data type called `NSInteger`. The different is subtle, and becoming less important as each year goes by, but for now it matters – and it matters in Swift too, so if you’re not sure about number sizes in Swift this will be good to read.

When the iPhone first launched, it had a 32-bit CPU, which meant numbers were stored in binary using 32 1s and 0s. From the iPhone 5s onwards, 64-bit has been the standard, which means that numbers are stored in binary using 64 1s and 0s.

Clearly you don’t want to write two different pieces of code to handle each architecture, so Apple’s solution is `NSInteger`: on 32-bit systems this holds a 32-bit number, and on 64-bit systems it holds a 64-bit number. `NSInteger` is used extensively across iOS, macOS, tvOS and watchOS, which means you need to use it to avoid causing problems.

The difference matters less and less each year as more people upgrade to 64-bit iOS devices, but right now it’s important. Consider, for example, if you saved an array of integers to iCloud, and a user accesses it from their 64-bit iPhone and also from their 32-bit iPad. If the iPhone wrote 64-bit integers, the iPad wouldn’t be able to read them correctly.

In this situation – where you need to work with both 64-bit and 32-bit devices – using `NSInteger` is the wrong choice. Instead, you should specify the exact size you need using `int32_t` or `int64_t`. That way, the integer size is preserved regardless of what CPU you’re running on.

This same problem (and the same solution) applies to floating-point numbers: like Swift, Objective-C has `float` and `double` types for holding single-precision and double-precision floating-point numbers, and `CGFloat` is designed to map to either `float` or `double` depending on the current CPU.

So, the short and simple version: you should be using `NSInteger` and `CGFloat` almost all the time, with exceptions being when you need to store data across platforms or when you have to work with an API that requires a specific size.

What is truth?

For historical reasons, Objective-C's integers are written as **`int`** and its booleans are written as **`BOOL`**. Most primitive data types (integers, floats, etc) are written using all lowercase letters, but **`BOOL`** is an exception and used to be one of the quirky edges of the languages.

Objective-C's quirkiness comes in two forms: first, you'll find there are two data types for booleans, **`bool`** and **`BOOL`**; second, you'll find that people usually write **`YES`** and **`NO`** in place of **`true`** and **`false`**, although both work.

Traditionally, C didn't have a dedicated boolean data type, it just used the data type "signed char" instead, then called that **`BOOL`**. This is actually an eight-bit value, but (with some small corner cases around the edges) this worked well enough. A dedicated **`bool`** was introduced in a C language updated called C99, and as of 64-bit iOS the **`BOOL`** pseudo-type is just an alias for the **`bool`** data type. **You will find both in use.** For example, Apple's C frameworks like Core Graphics usually use **`bool`**, whereas its Objective-C frameworks like UIKit usually use **`BOOL`**.

I don't mind whether you write **`YES`** and **`NO`**, **`true`** and **`false`**, or **`BOOL`** and **`bool`**; as usual, the most important thing is to following whatever coding convention you find when you open someone else's project. If you're starting a project fresh and don't have existing code to examine, go with **`BOOL`**, **`YES`**, and **`NO`** until you find a reason not to.