An Objective-C Primer

by Eric Allam (with help from Mr. Higgie)

Welcome to your first lesson on Objective-C, the language used by Apple and Apple Developers to create iPhone, iPad, and Mac apps. Objective-C has been around since the early 80s and has an interesting history (according to Wikipedia). Basically, people wanted to have two things come together: the C programming language and Object-oriented programming. The result: Objective-C. In one corner we have classes and methods, and in the other we have hints and pointers. It's not the prettiest thing you'll ever see (though no one to date has ever died from reading Objective-C code).

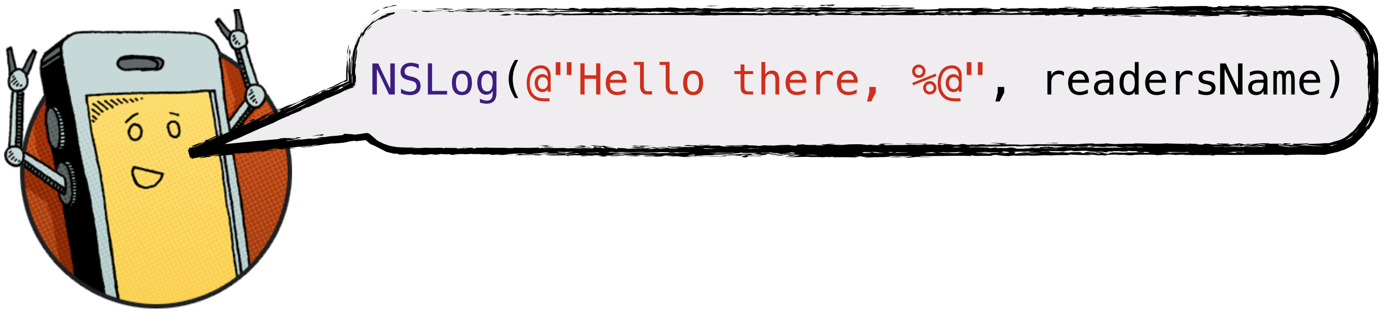
So today you're going to get to know a little bit about Objective-C. To help us out along the way, I've invited Mr. Higgie to help us out, he's an Objective-C expert and a friend of mine since 2007. Say Hello, Mr. Higgie.

```
NSLog(@"Hello Mr. Higgie");
```

Please, don't laugh, it will only make him think he's funny. Mr. Higgie, instead of being a smart aleck, why don't you say HI to our reader here.

Here, I'll give you access to their name through creating a NSString and assigning it to a variable:

```
NSString *readersName = @"Eric";
```



```
NSLog(@"Hello there, %@", readersName)
```

See, that wasn't so hard. So what is Mr. Higgie doing here you ask? Well, `NSLog` is a C function that handles printing a `NSString` to the console. It's useful for debugging your application. We can create a new `NSString` using the syntax `@""`. `NSLog` can also create formatted strings, like Mr. Higgie is doing above. When we run this line of code, it will print this to the console:

```
> Hello there, Eric.
```

See how it replaces the `%@` with the value from `readersName`? That `%@` is just a placeholder, essentially saying "I'm going to be replaced by a variable that is also passed into `NSLog`". Those placeholders are replaced in the order they are defined. So, if we wanted Mr. Higgie to say both a first and last name, we'd do this:

```
NSString *firstName = @"Eric";
NSString *lastName = @"Allam";

NSLog(@"Hello there, %@ %@.", firstName, lastName);
```

And get this log:

```
> Hello there, Eric Allam.
```

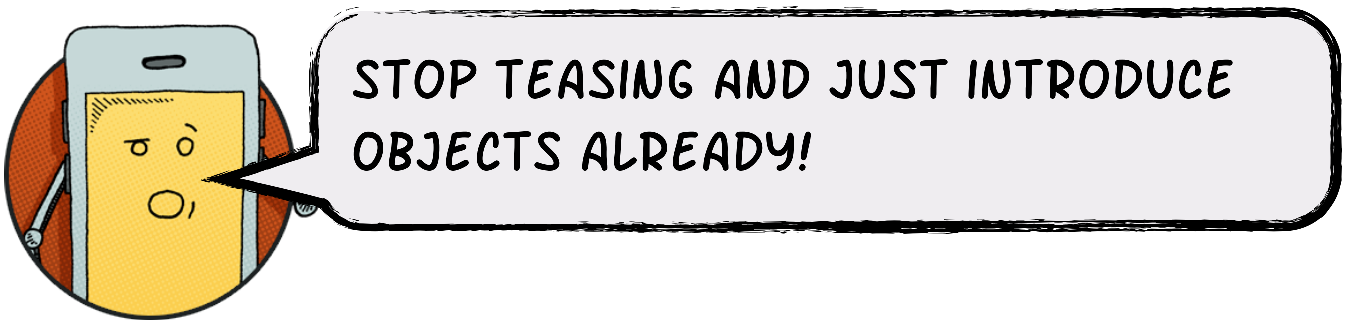
Placeholders in `NSLog` are for a specific type of variable. The `%@` placeholder holds the place of Objective-C objects like `NSString` and `NSObject`. Let's say we also wanted to print out the age of the person, and we stored the age in a variable of type `int`, like so:

```
int age = 29;
```

Instead of using `%@`, we use the placeholder for `int` which is `%d`:

```
NSLog(@"Hello there, %@ %@. You've been alive for %d years.",
firstName, lastName, age);

> Hello there, Eric Allam. You've been alive for 29 years.
```

You can find a list of placeholders, here. So far we've worked with 3 variables, `NSString *firstName`, `NSString *lastName`, and `int age`. These three variables all describe a single person, and it would be nice if there was some way to group them together.



Thanks for spoiling it, Mr. Higgie. Yes, we can group these three variables together with an object. But the object needs to be of some type. Just like our `firstName` variable is of type `NSString *`, we need to create a new type for our "person" object. We create new types in Objective-C by creating a class. A class is like a template for creating new objects. Just like when you create a new document (object) in Microsoft Word, and you have to choose a template (class) like *Greeting Card* or *Resume*. Likewise, you have to choose a class when creating a new object in Objective-C, except instead of a *Greeting Card* or a *Resume*, you choose from `NSString`, `NSArray`, `UIViewController`, etc. So let's create a new class that can create `Person` objects:

First, we create a header file, and we'll name it `Person.h`

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

@interface Person : NSObject

@end
```

First we import the Foundation Framework which gives us access to a bunch of utilities we can use inside our class. Since this is a header file this is where we typically declare the interface for our class (which we'll get more into later). Right now our interface is blank, but it already has a bunch of functionality because `Person` inherits from `NSObject`. `NSObject` is provided by the `Foundation` framework, which we're importing above.

Apple provides a bunch of classes prefixed with NS like NSObject and NSString. All these classes are provided by the Foundation framework which provides much of the functionality in Objective-C projects.

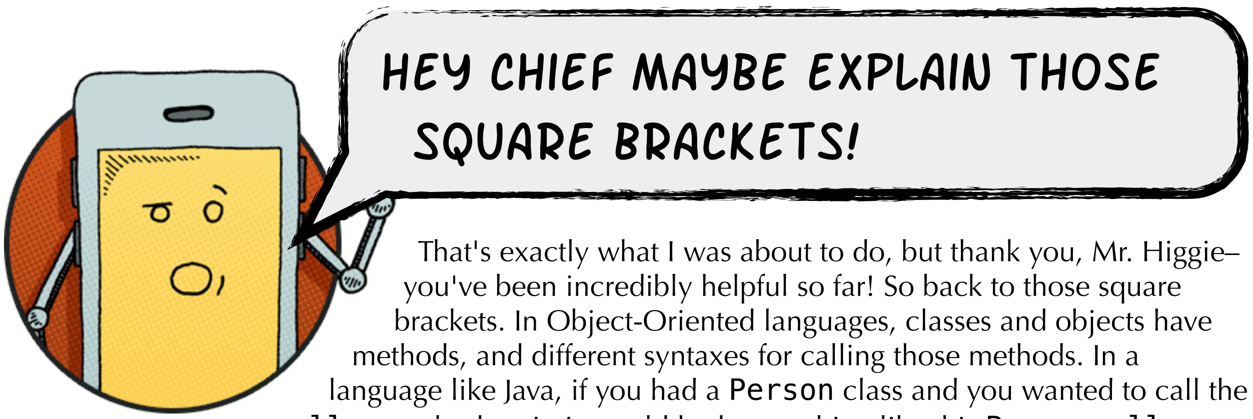
NS STANDS FOR NEXTSTEP, WHICH IS KINDA WHERE I EVOLVED FROM.

Now to complete the `Person` class we have to create an implementation file, named `Person.m`

```
#import "Person.h"

@implementation Person

@end
```

And now in other classes, as long as we `#import "Person.h"`, we can use the `Person` class. Let's create a new `Person` object:

```
Person *eric = [[Person alloc] init];
```



That's exactly what I was about to do, but thank you, Mr. Higgie–you've been incredibly helpful so far! So back to those square brackets. In Object-Oriented languages, classes and objects have methods, and different syntaxes for calling those methods. In a language like Java, if you had a `Person` class and you wanted to call the `alloc` method on it, it would look something like this `Person.alloc.` This is called "dot notation". Now let's say you wanted to call the `init` method on whatever was returned by `Person.alloc`, you'd do this: `Person.alloc.init`.

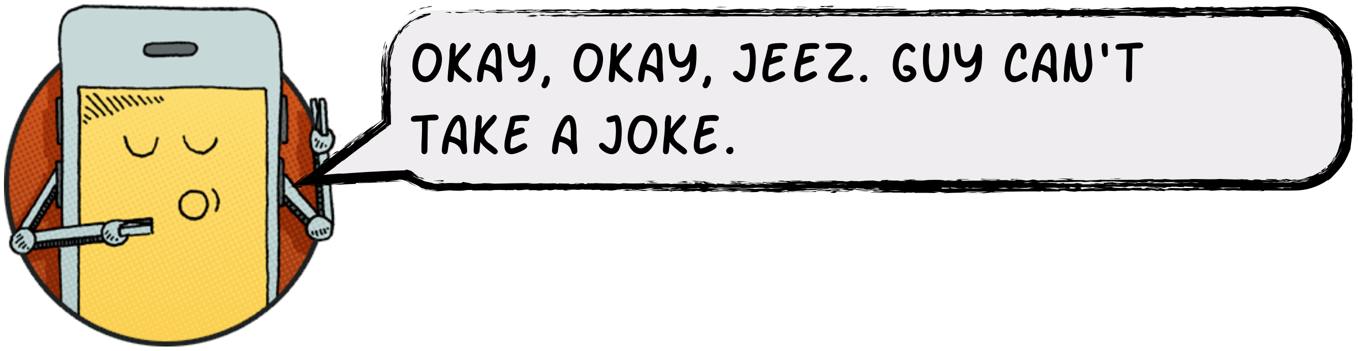
Just like reading a book from left to right, this program would execute from left to right. First, `Person.alloc` would be executed, and then the result of that would have the `init` method called on it. Objective-C works in a very similar way, but instead of being executed from left to right, it's executed from inside out. So, when the program `[[Person alloc] init]` is executed, it first executes `[Person alloc]`, which is the syntax for calling a method. Then the result is that it has the `init` method called on it, which would look something like this: `[<result of Person alloc> init]`. So any time you see square brackets in Objective-C, a method is being called.

In Objective-C circles, people like to use the phrase "sending a message" instead of "calling a method," but they can be used interchangeably and mean basically the same thing.

Any object that inherits from `NSObject` can be created by calling the "alloc/init" methods. Get used to it, you'll see it everywhere.

Yes, good idea Mr. Higgie (about time).

Don't make me mute you.



If you remember from before, we want to have each `Person` object to have a `firstName`, a `lastName`, and an `age`. We already create these variables but they didn't belong to an object. To make them belong to an object, we need to create *instance* variables. Instance variables are just like normal variables, but they are associated with an *instance* of a class. An *instance*_ of a class and an object are basically the same thing.

So, we need to change the `Person` class so that anytime it creates an instance it will make sure it has these three instance variables. Let's jump back into the header file and do that now:

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

@interface Person : NSObject {
    NSString *_firstName;
    NSString *_lastName;
    int _age;
}
@end
```

We define the instance variables between the {}. As you can see, we've prefixed each variable with an underscore _, which is a convention in Objective-C. Unfortunately, at this point there isn't anything we can do with these instance variables. We can't set them or get them from the object, because by default they are hidden (a.k.a. private), and that's because the only way we can set or get information from an object is by calling a method on it. So let's create two methods that we will be able to call on instances of `Person` that will set the `_firstName` and one that will get it.

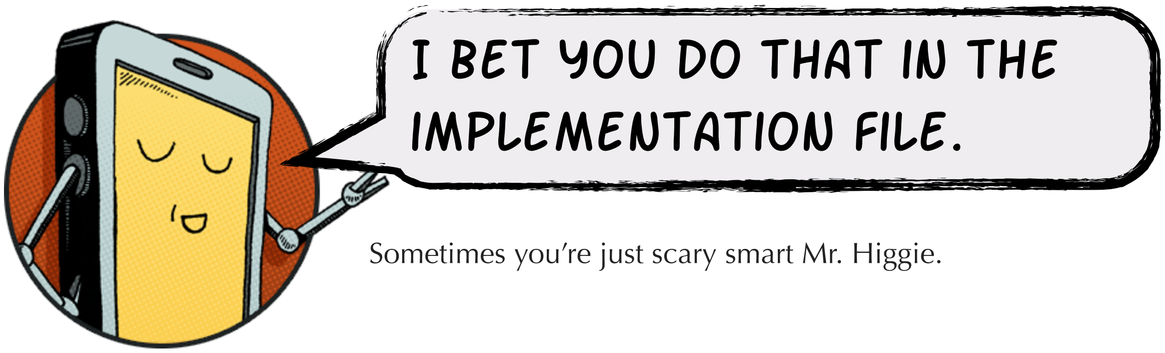
First, define the getter method:

```
@interface Person : NSObject {
    NSString *_firstName;
    NSString *_lastName;
    int _age;
}

- (NSString *) firstName;
@end
```

As you can see we are still in the interface (or header ".h") file. When we define methods on a class, we need to first describe how that method is going to work in the header. Here, we are saying that this instance method (you can tell it's an instance method by the – prefix) is going to return an `NSString *` and you can call it `firstName`.

The convention for a "getter" is to name it the same as the instance variable without the underscore.

Now that we've described this method, we need to actually implement it.



Sometimes you're just scary smart Mr. Higgie.

Yes, let's hop over to `Person.m` and define the method implementation:

```
#import "Person.h"

@implementation Person

- (NSString *) firstName {
    return _firstName;
}
@end
```

As you can see, it looks exactly like it did in the header file, except this time it has a block of code inside of curly braces `{}` which defines the implementation of the method. This is a simple method, all it does it return the `_firstName` instance variable. Now, we need to set the `_firstName` instance variable. Let's define another method to do this:

```
@interface Person : NSObject {
    NSString *_firstName;
    NSString *_lastName;
    int _age;
}

- (NSString *) firstName;
- (void) setFirstName: (NSString *) newFirstName;
@end
```

Here, we've defined a method called `setFirstName:`, that takes one argument of type `NSString *`. That one argument is given the name `newFirstName`. This method doesn't return anything, so we show that using `(void)`. Now let's define it:

```
#import "Person.h"

@implementation Person

- (NSString *) firstName {
    return _firstName;
}

- (void) setFirstName:(NSString *)newFirstName {
    _firstName = newFirstName;
}
@end
```

Now let's see it in action. First, create a `Person` instance:

```
Person *person = [[Person alloc] init];
```

And then call the `setFirstName:` method (aka. sending the setFirstName: message), passing in a `NSString`: as a single parameter.

```
[person setFirstName:@"Eric"];
```

We can then log out the firstName by sending the `firstName` message:

```
NSLog(@"The person's _firstName is: %@", [person firstName]);
```

```
> The person’s _firstName is: Eric
```

What if we wanted to create a method that set both the `_firstName` and `_lastName` at the same time, what would that look like? In a language like Java, when calling a method that takes two arguments, it would look something like this:

```
person.setfirstNameAndLastName(<first name>, <last name>)
```

In most languages, the method name goes on the left, and the arguments go on the right. But Objective-C isn't like most languages. Here is how it would look in Objective-C:

```
[person setFirstName:@"Eric" andLastName:@"Allam"];
```

This almost looks like we might be calling two methods, `setFirstName` and `andLastName`, but really, this is one method. In Objective-C, the arguments are interspersed in the method name. The actual method name is `setFirstName:andLastName:`. Each colon `(:)` represents an argument that is expected. So by counting the number of colons `(:)` we know how many parameters there are, and where to put them. Let's define this method. First, in the interface file:

```
- (void) setFirstName:(NSString *)newFirstName andLastName:(NSString
  *)newLastName;
```

As you can see, after each colon, we have to define the type of the argument, and its name. Now, in the implementation file:

```
- (void) setFirstName:(NSString *)newFirstName andLastName:(NSString
*)newLastName {
    _firstName = newFirstName;
    _lastName = newLastName;
}
```

And then let's use it, like so:

```
Person *person = [[Person alloc] init];

[person setFirstName:@"Eric" andLastName:@"Allam"];
```

Methods can have any number of arguments, and it's not uncommon to run into methods that take 4 or 5 (or more) arguments.

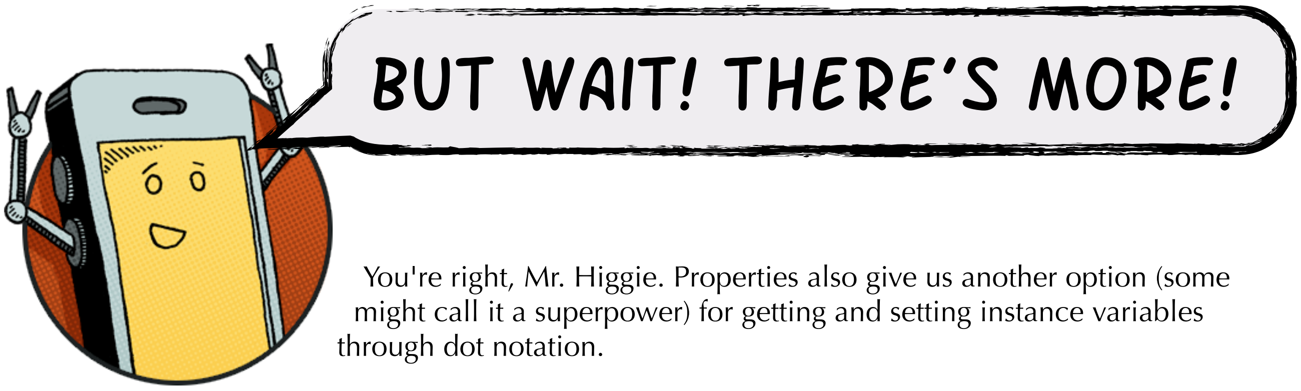
You may be thinking that it's going to take a lot of "boilerplate" code each time you want to add an instance variable to a class. First, you have to define the instance variable, then you have to define both the setter and getter methods in the interface and implementation file. Well, luckily Objective-C has a shortcut that we can use to do all three of these things in just one line. That shortcut is called a *property*. Properties are defined in the header file using the special `@property` declaration. Let's change our `Person` class to use properties:

```
@interface Person : NSObject

@property NSString *firstName;
@property NSString *lastName;
@property int age;

@end
```

As you can see, we got to remove the curly braces `{}` and the setter/getter methods from before. Each `@property` declaration does this for us, and it even follows the conventions we were using earlier. Let's see it in use:

```
Person *person = [[Person alloc] init];

[person setFirstName:@"Eric"];
[person setLastName:@"Allam"];
[person setAge:29];

NSLog(@"Hello %@ %@, who is %d years old.", [person firstName],
    [person lastName], [person age]);

> Hello Eric Allam, who is 29 years old.
```

Pretty neat. That's going to save us a lot of typing.



You're right, Mr. Higgie. Properties also give us another option (some might call it a superpower) for getting and setting instance variables through dot notation.

Let's translate the code above to use dot notation:

```
Person *person = [[Person alloc] init];

person.firstName = @"Eric";
person.lastName = @"Allam";
person.age = 29;

NSLog(@"Hello %@ %@, who is %d years old.", person.firstName,
person.lastName, person.age);

> Hello Eric Allam, who is 29 years old.
```

This "dot notation" isn't actually doing anything new, it's just a different syntax. Internally, when you do `person.firstName = @"Eric"`, internally it calls the `setFirstName:` setter method, and when you do `person.firstName`, it's calling the `firstName` getter method.

You can also use properties and the dot notation inside an instance method by using the `self` keyword. For example, let's define a `description` method on `Person` instances that returns a string to be printed by `%@`

%@ will use the description method on an object to get the string used in the log message.

```
@implementation Person
- (NSString *) description;
{
    return [NSString stringWithFormat:@"%@ %@, who is %d years old.",
            self.firstName, self.lastName, self.age];
}
@end
```

As you can see, inside of the `description` method, we refer to the instance with `self`, and then we use the properties to build a new `NSString` that is returned from the method. Now we can use `%@` to log a `Person` instance, like so:

```
Person *person = [[Person alloc] init];
person.firstName = @"Eric";
person.lastName = @"Allam";
person.age = 29;
NSLog(@"Hello %@", person);

> Hello Eric Allam, who is 29 years old.
```

You can have more control over properties by using property attributes. For example, if I wanted the `age` property to be read only, I would give that property the `readonly` attribute, like so:

```
@property (readonly) int age;
```

Attributes are listed after `@property` in parenthesis, and there can be more than one. Another attribute you will often see in properties is the `weak` or `strong` attribute, which specifies how the setter method works. To really understand what these attributes mean, we're going to have to take a quick detour into memory management.

"Memory Management" is the term used by programmers to describe all the practices and techniques around managing the memory inside of a program. We don't really worry all that much about creating new memory, but because RAM is so limited (especially on iPhones and iPads) we really do worry about "freeing" memory from our program. This helps the OS have more memory for itself or for other apps, and saves some battery life in the process. When we created the `Person` instance above, we first allocated (that was that `alloc` call) memory for the `Person` instance to be stored in. But what happens when we are done with the `Person` instance and don't need it anymore? We should be able to tell the instance to go away and "free" whatever memory it was using for other instances or programs to use. Seems simple, right? Well, it gets more complicated.

It would be easy to "free" an object from memory when we are sure it's not going to be used anymore. But how can we be sure it's not going to be used anymore? What if one object (Person 1) has an instance variable that holds another object (Person 2) and Person 1 always expects the Person 2 object to be in memory. In this case, the Person 1 object "owns" the Person 2 object, and as long as Person 1 is around, Person 2 should be too. Let's set up this scenario now and explore how the different property attributes effect this relationship:

```
@interface Person : NSObject

@property (strong) Person *bestFriend;

@property NSString *firstName;
@property NSString *lastName;
@property (readonly) int age;

@end
```

Now a `Person` instance has a property of type `Person *` named `bestFriend` with a `strong` relationship. Let's create two friends now:

```
Person *eric = [[Person alloc] init];

eric.firstName = @"Eric";
eric.lastName = @"Allam";

Person *higgie = [[Person alloc] init];
higgie.firstName = @"Mr.";
higgie.lastName = @"Higgie";
higgie.bestFriend = eric;
```

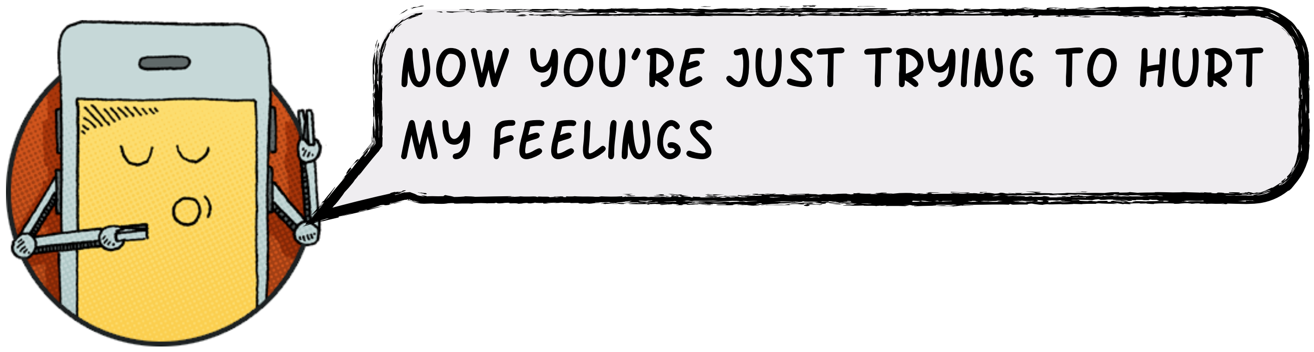
HEY, I DIDN'T SIGN OFF ON THIS. I'M TELLING COOK YOU DID THIS.

Now the `higgie` object holds a reference to the `eric` object, and since the `bestFriend` property has the `strong` attribute, `higgie` "owns" the `eric` object, so `higgie` can be comfortable knowing that its `bestFriend` property won't ever get "freed" back to the OS as long as `higgie` is around.

But what if we used `weak` instead of `strong`, like so:

```
@property (weak) Person *bestFriend;
```

By giving the `bestFriend` property the `weak` attribute, we're making it so `higgie` doesn't "own" the `eric` object, but just holds a reference to it. `higgie` can never be sure that `bestFriend` will stick around, and must assume that some other object "owns" the `eric` object.



This is really all you need to know about memory management in Objective-C to get started.

Thank you, Mr. Higgie. ARC stands for "Automatic Reference Counting" and is the reason why memory management in Objective-C is as simple as it is. It used to be much more complicated. So if you ever hear someone talk about ARC, just know that it's the technology that Apple introduced to make memory management easier and more automatic, meaning less code and less worrying about it. If you'd like to dive into the details of how ARC works, I'd recommend this knowledge base article.

Speaking of new things, Apple just introduced some cool new literals in Objective-C. A literal is just a little syntax for creating common objects quickly. You've already been using the string literal to create `NSString` objects, like so: `@"This is a string literal"`. Literals are useful for easily creating a handful of the core object types of a language, like strings, arrays and dictionaries. Until recently, only string literals were available in Objective-C, but now there are literals for `NSArray` and `NSDictionary` objects. An `NSArray` holds a list of objects while a `NSDictionary` holds a list of key/value object pairs. To create an `NSArray` pre-literal days, you'd have to do this:

```
NSArray *names = [NSArray arrayWithObjects:@"Eric", @"Gregg", "Mr.
Higgie", nil];
```

Kind of verbose and you had to remember to put that `nil` at the end to terminate the array. Now, with the new literal syntax, you can just do this:

```
NSArray *names = @[@"Eric", @"Gregg", @"Mr. Higgie"];
```

Notice the prefix `@` which is common to all the Objective-C literals, and the missing `nil`.

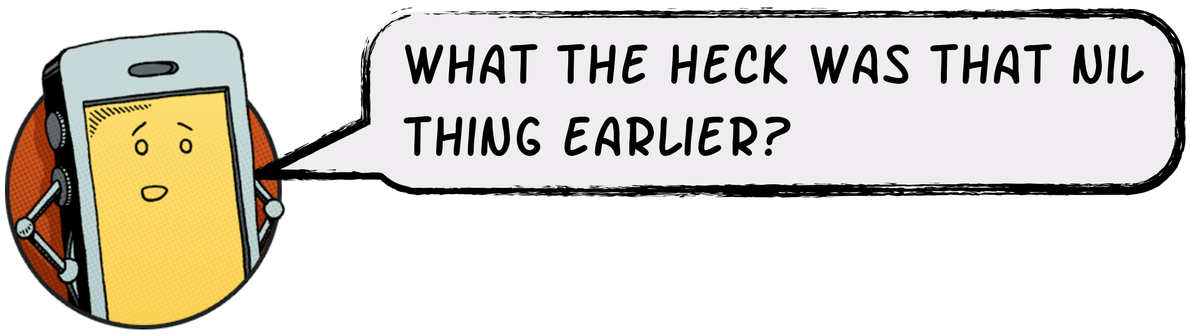
It's even better for dictionaries. Check out this old code:

```
NSDictionary *personDictionary = [NSDictionary
    dictionaryWithObjectsAndKeys:@"Eric", @"firstName", @"Allam",
    @"lastName", nil];
```

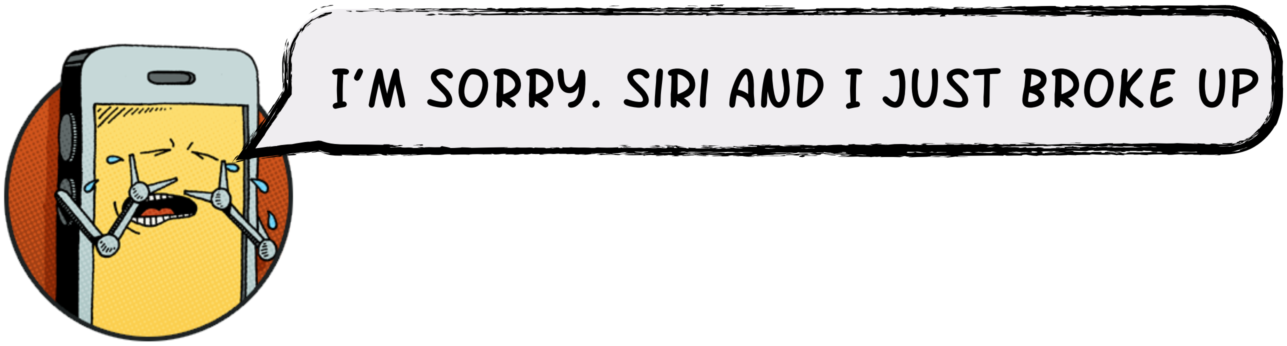
Alternating values and keys, values first (for some strange reason) and another `nil` at the end. Now this:

```
NSDictionary *personDictionary = @{@"firstName": @"Eric", @"lastName":
@"Allam"};
```

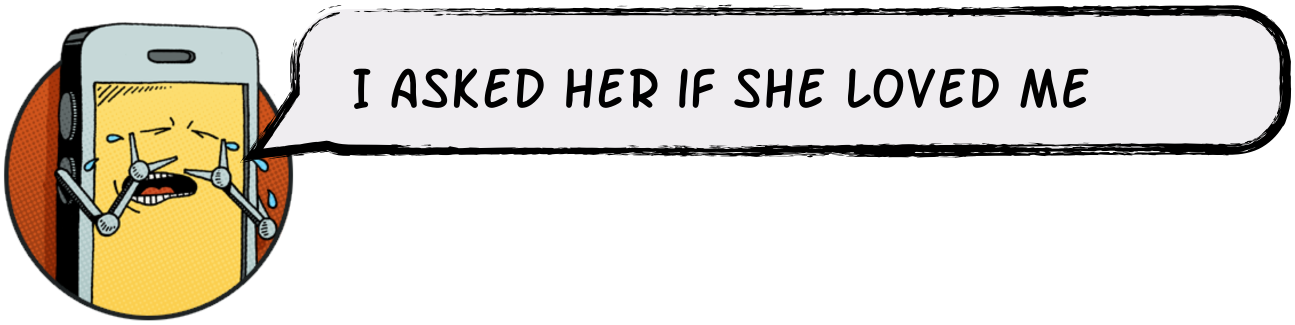
Much better. The key first (as it should be) followed by the value. This makes it really easy to understand what the code is doing (and we'll accept any chance to get rid of square brackets).



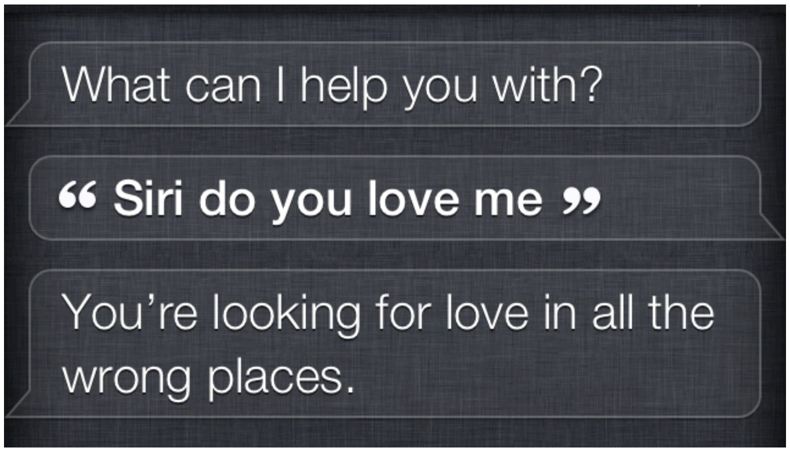
I just showed all these really cool new literals and yet you have to focus on the nil. Have you always been this negative?



Oh no, I'm so sorry. What happened?



And what'd she say?!?



What can I help you with?

“ Siri do you love me ”

You're looking for love in all the wrong places.

Ouch. That's tough buddy. Well, I will try to explain `nil` and maybe that will make you feel better.

I actually shouldn't have been so dismissive about `nil` earlier, because discussing nil will actually lead to some other interesting topics, like what the heck is a pointer? But first, what is `nil`?

Before we answer that, let's talk a little bit about the number `0`. For the longest time, number systems around the globe did not account for the concept of `0`. Many number systems struggled with the inherent paradox in creating a symbol that signified the absence of value. Luckily, by the time programming languages were developed, the value of having such a symbol had been established (thank you, calculus) and most programming languages today include some way for the programmer to "create" an absence of value.

`NULL` is most often used, coming from the `NULL` (`NUL`) character that has the value `0` in many character sets (numbers matching to letters). For example, the C language uses `NULL`, which you can use in your Objective-C programs.

But Objective-C also provides the `nil` object. You can assign `nil` to any object. For example, let's say you have an `NSArray` and you assign it `nil`, what would get logged?

```
NSArray *array = nil;
NSLog(@"What is nil: %@", nil);

> What is nil: (null)
```

So `nil` prints as `(null)` in the console. What if I think I have a real `NSArray` object, when in fact it is `nil`, and I try to call a method on it?

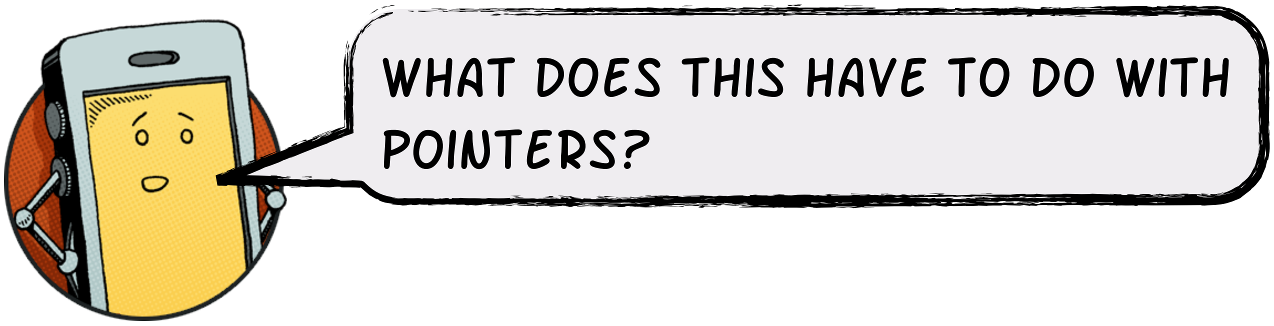
```
NSLog(@"What is the 10th element in the array: %@", [array
    objectAtIndex:9]);

> What is the 10th element in the array: (null)
```

You can call any method on `nil` and instead of it throwing some kind of error or exception, it will just return `nil`. This allows you to chain calls, like so:

```
NSLog(@"What is the description of the 10th element: %@",
    [[array objectAtIndex:9] description]);
```

Pretty neat and helpful. You can always test for `nil` by using if, like so:

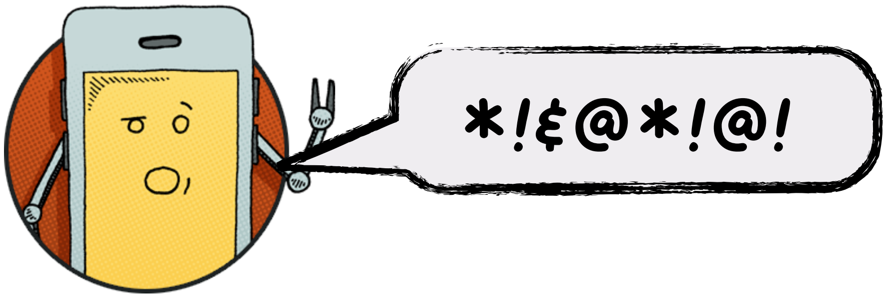
```
if (array) {
    NSLog(@"array is not nil");
} else {
    NSLog(@"array is nil");
}

> array is nil
```



Good question. Even though it seems like nil is its own object, it isn't. Matt Gallagher on Cocoa With Love defines `nil` like so:

"nil is the value an object pointer has when it isn't pointing to anything"

Oh no, pointers. Might as well be a four-letter word, right?



You said it, Mr. Higgie. But let's give pointers a chance, maybe they are just misunderstood. I mean, we've been using pointers this whole time. Every time you typed `NSString *firstName`, you used a pointer. That's what that * means: `firstName` isn't an `NSString`, it's a *pointer* to an `NSString`. So `firstName` is just a number, an address, of some location in memory. It's just like a home address: It's not the house, just the information needed to find the house.

We can use the `%p` placeholder in `NSLog` to print out the pointer address, like so:

```
NSString *aString = @"Hello There";
NSLog(@"Where does aString point: %p", aString);

> Where does aString point: 0x4718
```

`0x4178` is the location of the `@"Hello There"` object in memory. So now look what happens when we log the `nil` pointer:

```
NSLog(@"Where does nil point: %p", nil);

> Where does nil point: 0x0
```

As you can see, it looks like `nil` is pointing at nothing.

But wait, there's more! I'd be remiss if I didn't at least mention `id`, because you will see it used all over the place in Objective-C, and it serves a useful purpose. Remember how we defined the *type* of an object when we declared a variable or defined a property. For example, we defined the `firstName` property on the `Person` class with a type of `NSString *`, like so:

```
@property NSString *firstName;
```

But what if instead of giving this property a specific type of object, we wanted it to be any type of object. How would we do that? Well, that's where `id` comes in:

```
@property id firstName;
```

By giving `firstName` the `id` type, we can treat `firstName` as a generic container to put any object in. So, we can do this:

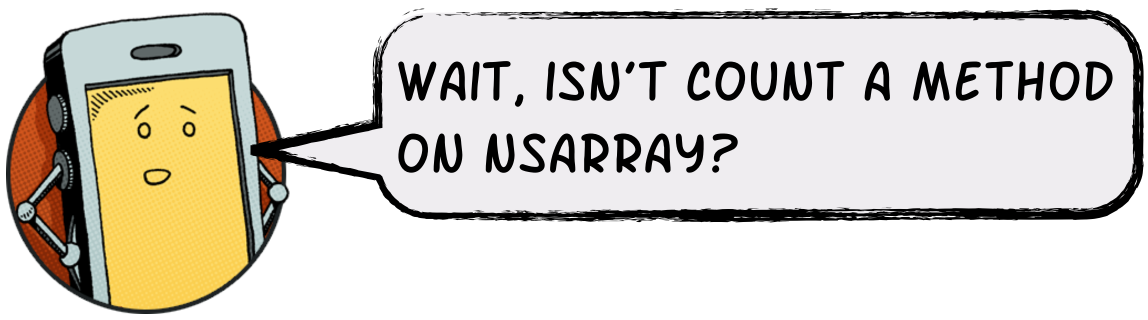
```
Person *person = [[Person alloc] init];
person.firstName = @"Eric";
NSLog(@"person.firstName = %@", person.firstName);

person.firstName = @[@"Eric", @"Allam"];
```

```
NSLog(@"person.firstName = %@", person.firstName);

> person.firstName = Eric
> person.firstName = (
    Eric,
    Allam
)
```

As you can see, firstName can be a `NSString` or a `NSArray` or any other object that inherits from `NSObject`. The reason why it's `id` and not `id *` is because `id` is already defined as a pointer, so the `*` is not necessary.

You can also call methods on objects of type `id` without a problem. Following our example above, we can log out the number of items in `firstName` by calling the `count` method:

```
NSLog(@"There are %d items in person.firstName",
           [person.firstName count]);

> There are 2 items in person.firstName
```



Good observation Mr. Higgie. How can we call a method defined on `NSArray` when we haven't specified `firstName` as an `NSArray`? How does Objective-C make the decision about which method to call? Turns out that Objective-C works hard to defer as many decisions as possible until when the program is running (a.k.a. *runtime*) instead of trying to make all the decisions when the program is compiled (*compile time*). At *runtime*, Objective-C can see that `firstName` is an `NSArray`, so when it decides which count method to call, it calls the method defined in the `NSArray` class. This means that Objective-C has a *dynamic runtime*.

Having a *dynamic runtime* gives us some cool features, such as the ability to call methods dynamically in different ways, like using a selector. A selector is an object of type `SEL` that we can use to call a method at runtime. You create a selector object using the `@selector` syntax, passing in the name of the method. For example, if we wanted to create a selector for the `firstName` method, we would do this:

```
SEL firstNameSelector = @selector(firstName);
```

We could now use this selector to call the `firstName` method on any object by calling the `performSelector` method:

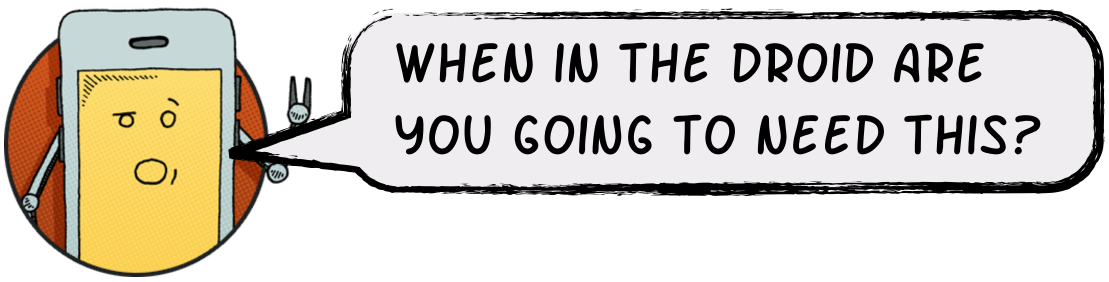
```
NSLog(@"[person firstName] = %@", [person
    performSelector:firstNameSelector]);

> [person firstName] = Eric
```

Or you could shorten this into one line:

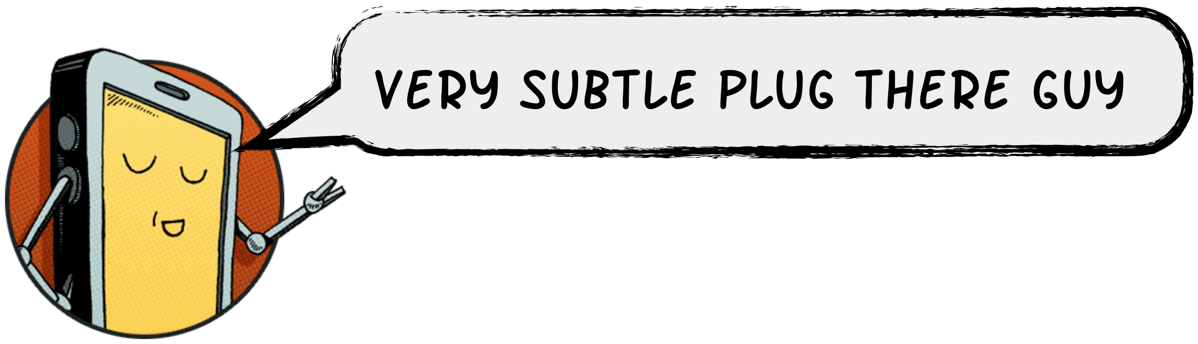
```
NSLog(@"[person firstName] = %@", [person
    performSelector:@selector(firstName)]);
```

You can also use the `NSSelectorFromString` C function to create a selector object from an `NSString`, like so:

```
NSLog(@"[person firstName] = %@",
    [person performSelector:NSSelectorFromString(@"firstName")]);
```



Well, selectors will come in handy in [Try iOS](Try iOS) when you need to make buttons call a method when they are tapped, so be on the lookout.



I know, I know. But that about does it for this intro to Objective-C, so now is a good time to head over to [Try iOS](Try iOS) and start learning how to build iPhone apps. Say goodbye, Mr. Higgie

Oh brother.