Working with Objects

The majority of work in an Objective-C application happens as a result of messages being sent back and forth across an ecosystem of objects. Some of these objects are instances of classes provided by Cocoa or Cocoa Touch, some are instances of your own classes.

The previous chapter described the syntax to define the interface and implementation for a class, including the syntax to implement methods containing the code to be executed in response to a message. This chapter explains how to send such a message to an object, and includes coverage of some of Objective-C's dynamic features, including dynamic typing and the ability to determine which method should be invoked at runtime.

Before an object can be used, it must be created properly using a combination of memory allocation for its properties and any necessary initialization of its internal values. This chapter describes how to nest the method calls to allocate and initialize an object in order to ensure that it is configured correctly.

Objects Send and Receive Messages

Although there are several different ways to send messages between objects in Objective-C, by far the most common is the basic syntax that uses square brackets, like this:

```
    [someObject doSomething];
```

The reference on the left, `someObject` in this case, is the receiver of the message. The message on the right, `doSomething`, is the name of the method to call on that receiver. In other words, when the above line of code is executed, `someObject` will be sent the `doSomething` message.

The previous chapter described how to create the interface for a class, like this:

```
@interface XYZPerson : NSObject
- (void)sayHello;
@end
```

and how to create the implementation of that class, like this:

```
@implementation XYZPerson
- (void)sayHello {
    NSLog(@"Hello, world!");
}
@end
```

Note: This example uses an Objective-C string literal, `@"Hello, world!"`. Strings are one of several class types in Objective-C that allow a shorthand literal syntax for their creation. Specifying `@"Hello, world!"` is conceptually equivalent to saying "An Objective-C string object that represents the string *Hello, world!*."

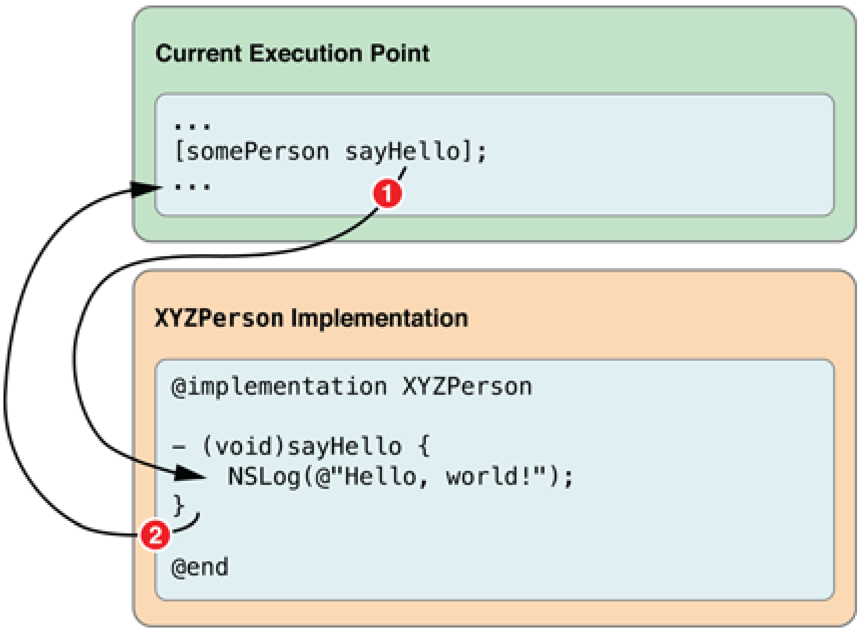
Literals and object creation are explained further in Objects Are Created Dynamically, later in this chapter.

Assuming you've got hold of an `XYZPerson` object, you could send it the `sayHello` message like this:

```
    [somePerson sayHello];
```

Sending an Objective-C message is conceptually very much like calling a C function. Figure 2-1 shows the effective program flow for the `sayHello` message.

Figure 2-1 Basic messaging program flow



In order to specify the receiver of a message, it's important to understand how pointers are used to refer to objects in Objective-C.

Use Pointers to Keep Track of Objects

C and Objective-C use variables to keep track of values, just like most other programming languages.

There are a number of basic scalar variable types defined in standard C, including integers, floating-point numbers and characters, which are declared and assigned values like this:

```
    int someInteger = 42;
    float someFloatingPointNumber = 3.14f;
```

Local variables, which are variables declared within a method or function, like this:

```
- (void)myMethod {
    int someInteger = 42;
}
```

are limited in scope to the method in which they are defined.

In this example, `someInteger` is declared as a local variable inside `myMethod`; once execution reaches the closing brace of the method, `someInteger` will no longer be accessible. When a local scalar variable (like an `int` or a `float`) goes away, the value disappears too.

Objective-C objects, by contrast, are allocated slightly differently. Objects normally have a longer life than the simple scope of a method call. In particular, an object often needs to stay alive longer than the original variable that was created to keep track of it, so an object's memory is allocated and deallocated dynamically.

Note: If you're used to using terms like the *stack* and the *heap*, a local variable is allocated on the stack, while objects are allocated on the heap.

This requires you to use C pointers (which hold memory addresses) to keep track of their location in memory, like this:

```
- (void)myMethod {
    NSString *myString = // get a string from somewhere...
    [...]
}
```

Although the scope of the pointer variable `myString` (the asterisk indicates it's a pointer) is limited to the scope of `myMethod`, the actual string object that it points to in memory may have a longer life outside that scope. It might already exist, or you might need to pass the object around in additional method calls, for example.

You Can Pass Objects for Method Parameters

If you need to pass along an object when sending a message, you supply an object pointer for one of the method parameters. The previous chapter described the syntax to declare a method with a single parameter:

```
- (void)someMethodWithValue:(SomeType)value;
```

The syntax to declare a method that takes a string object, therefore, looks like this:

```
- (void)saySomething:(NSString *)greeting;
```

You might implement the `saySomething:` method like this:

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

The `greeting` pointer behaves like a local variable and is limited in scope just to the `saySomething:` method, even though the actual string object that it points to existed prior to the method being called, and will continue to exist after the method completes.

Note: `NSLog()` uses format specifiers to indicate substitution tokens, just like the C standard library `printf()` function. The string logged to the console is the result of modifying the format string (the first argument) by inserting the provided values (the remaining arguments).

There is one additional substitution token available in Objective-C, `%@`, used to denote an object. At runtime, this specifier will be substituted with the result of calling either the `descriptionWithLocale:` method (if it exists) or the `description` method on the provided object. The `description` method is implemented by `NSObject` to return the class and memory address of the object, but many Cocoa and Cocoa Touch classes override it to provide more useful information. In the case of `NSString`, the `description` method simply returns the string of characters that it represents.

For more information about the available format specifiers for use with `NSLog()` and the `NSString` class, see String Format Specifiers.

Methods Can Return Values

As well as passing values through method parameters, it's possible for a method to return a value. Each method shown in this chapter so far has a return type of `void`. The C `void` keyword means a method doesn't return anything.

Specifying a return type of `int` means that the method returns a scalar integer value:

```
- (int)magicNumber;
```

The implementation of the method uses a C `return` statement to indicate the value that should be passed back after the method has finished executing, like this:

```
- (int)magicNumber {
    return 42;
}
```

It's perfectly acceptable to ignore the fact that a method returns a value. In this case the `magicNumber` method doesn't do anything useful other than return a value, but there's nothing wrong with calling the method like this:

```
    [someObject magicNumber];
```

If you do need to keep track of the returned value, you can declare a variable and assign it to the result of the method call, like this:

```
    int interestingNumber = [someObject magicNumber];
```

You can return objects from methods in just the same way. The `NSString` class, for example, offers an `uppercaseString` method:

```
- (NSString *)uppercaseString;
```

It's used in the same way as a method returning a scalar value, although you need to use a pointer to keep track of the result:

```
    NSString *testString = @"Hello, world!";
    NSString *revisedString = [testString uppercaseString];
```

When this method call returns, `revisedString` will point to an `NSString` object representing the characters `HELLO WORLD!`.

Remember that when implementing a method to return an object, like this:

```
- (NSString *)magicString {
    NSString *stringToReturn = // create an interesting string...

    return stringToReturn;
}
```

the string object continues to exist when it is passed as a return value even though the `stringToReturn` pointer goes out of scope.

There are some memory management considerations in this situation: a returned object (created on the heap) needs to exist long enough for it to be used by the original caller of the method, but not in perpetuity because that would create a memory leak. For the most part, the Automatic Reference Counting (ARC) feature of the Objective-C compiler takes care of these considerations for you.

Objects Can Send Messages to Themselves

Whenever you're writing a method implementation, you have access to an important hidden value, `self`. Conceptually, `self` is a way to refer to "the object that's received this message." It's a pointer, just like the `greeting` value above, and can be used to call a method on the current receiving object.

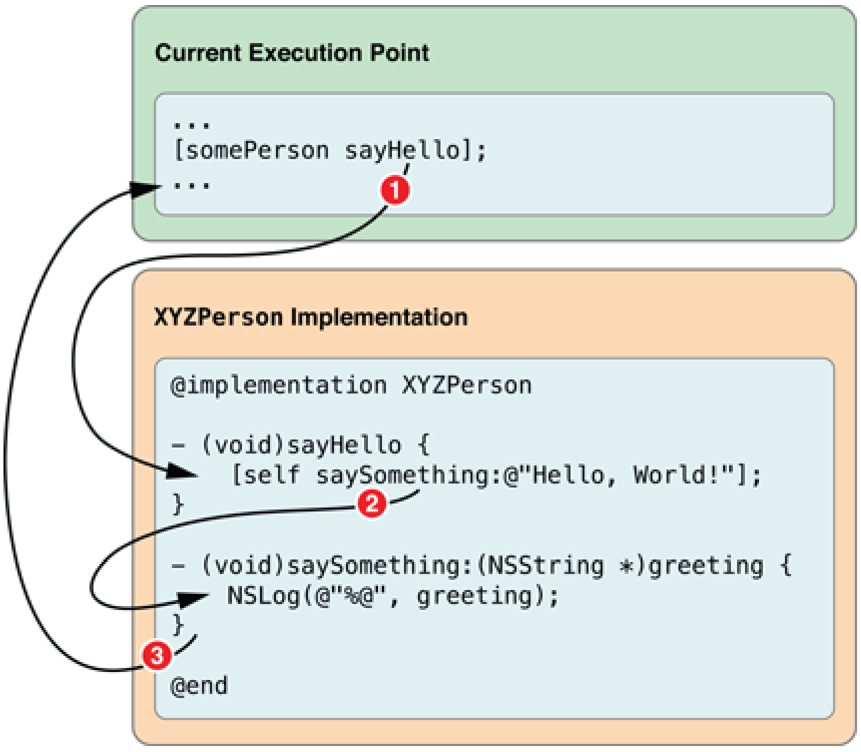
You might decide to refactor the XYZPerson implementation by modifying the sayHello method to use the saySomething: method shown above, thereby moving the NSLog() call to a separate method. This would mean you could add further methods, like sayGoodbye, that would each call through to the saySomething: method to handle the actual greeting process. If you later wanted to display each greeting in a text field in the user interface, you'd only need to modify the saySomething: method rather than having to go through and adjust each greeting method individually.

The new implementation using self to call a message on the current object would look like this:

```
@implementation XYZPerson
- (void)sayHello {
    [self saySomething:@"Hello, world!"];
}
- (void)saySomething:(NSString *)greeting {
    NSLog(@"%@", greeting);
}
@end
```

If you sent an XYZPerson object the sayHello message for this updated implementation, the effective program flow would be as shown in Figure 2-2.

Figure 2-2 Program flow when messaging self



Objects Can Call Methods Implemented by Their Superclasses

There's another important keyword available to you in Objective-C, called super. Sending a message to super is a way to call through to a method implementation defined by a superclass further up the inheritance chain. The most common use of super is when overriding a method.

Let's say you want to create a new type of person class, a "shouting person" class, where every greeting is displayed using capital letters. You could duplicate the entire XYZPerson class and modify each string in each method to be uppercase, but the simplest way would be to create a new class that inherits from XYZPerson, and just override the saySomething: method so that it displays the greeting in uppercase, like this:

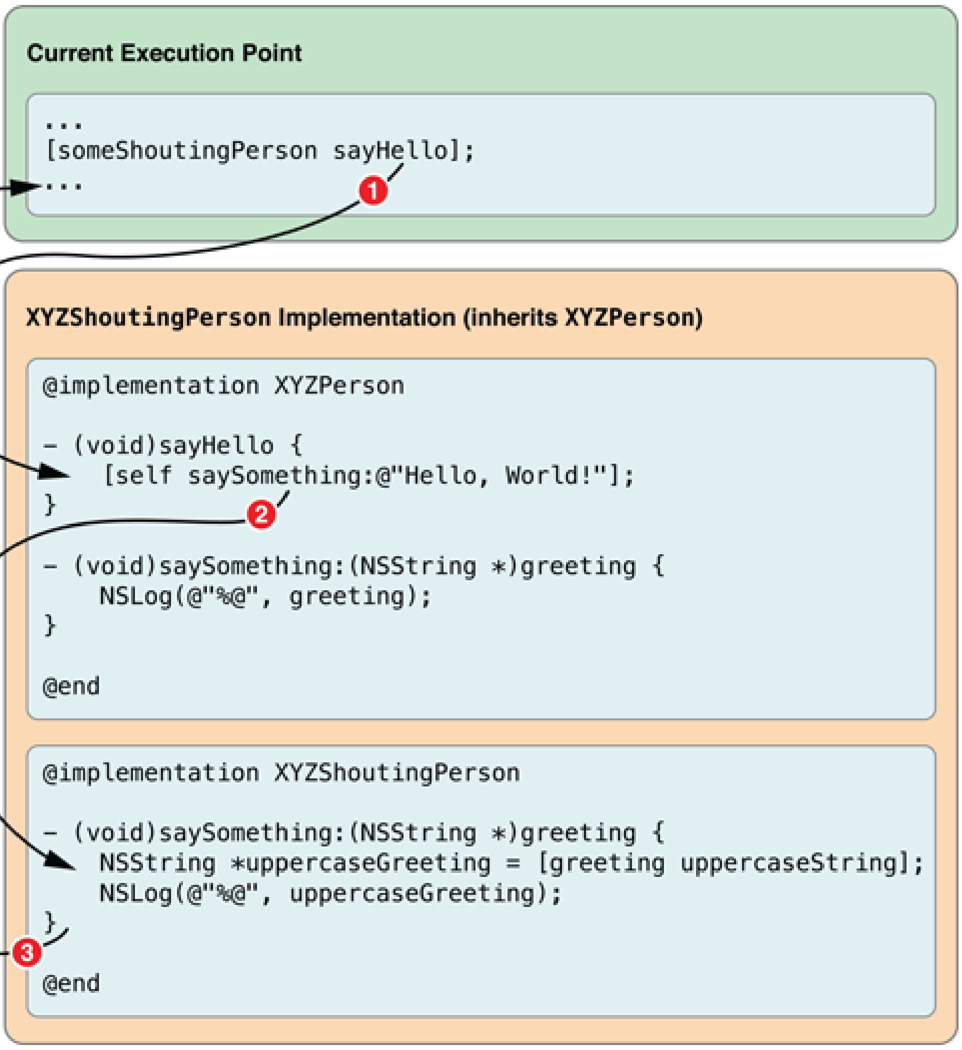
```
@interface XYZShoutingPerson : XYZPerson
@end
```

```
@implementation XYZShoutingPerson
- (void)saySomething:(NSString *)greeting {
    NSString *uppercaseGreeting = [greeting uppercaseString];
    NSLog(@"%@", uppercaseGreeting);
}
@end
```

This example declares an extra string pointer, `uppercaseGreeting` and assigns it the value returned from sending the original `greeting` object the `uppercaseString` message. As you saw earlier, this will be a new string object built by converting each character in the original string to uppercase.

Because `sayHello` is implemented by `XYZPerson`, and `XYZShoutingPerson` is set to inherit from `XYZPerson`, you can call `sayHello` on an `XYZShoutingPerson` object as well. When you call `sayHello` on an `XYZShoutingPerson`, the call to `[self saySomething:...]` will use the *overridden* implementation and display the greeting as uppercase, resulting in the effective program flow shown in Figure 2-3.

Figure 2-3 Program flow for an overridden method



```
...
[someShoutingPerson sayHello];
...
```

```
@implementation XYZPerson

- (void)sayHello {
    [self saySomething:@"Hello, World!"];
}

- (void)saySomething:(NSString *)greeting {
    NSLog(@"%@", greeting);
}

@end
```

```
@implementation XYZShoutingPerson

- (void)saySomething:(NSString *)greeting {
    NSString *uppercaseGreeting = [greeting uppercaseString];
    NSLog(@"%@", uppercaseGreeting);
}

@end
```

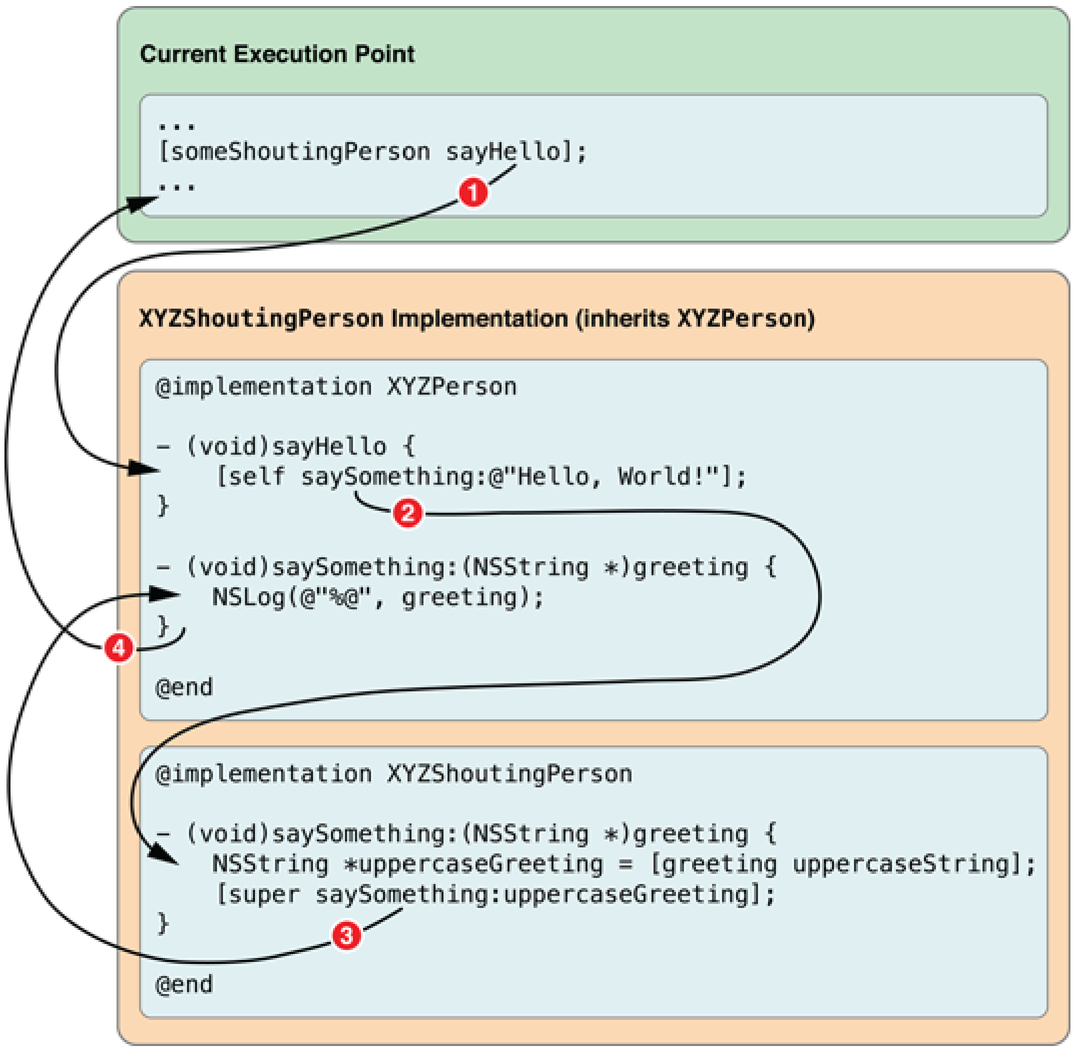
The new implementation isn't ideal, however, because if you did decide later to modify the `XYZPerson` implementation of `saySomething:` to display the greeting in a user interface element rather than through `NSLog()`, you'd need to modify the `XYZShoutingPerson` implementation as well.

A better idea would be to change the `XYZShoutingPerson` version of `saySomething:` to call through to the superclass (`XYZPerson`) implementation to handle the actual greeting:

```
@implementation XYZShoutingPerson
- (void)saySomething:(NSString *)greeting {
    NSString *uppercaseGreeting = [greeting uppercaseString];
    [super saySomething:uppercaseGreeting];
}
@end
```

The effective program flow that now results from sending an `XYZShoutingPerson` object the `sayHello` message is shown in Figure 2-4.

Figure 2-4 Program flow when messaging super



```
...
[someShoutingPerson sayHello];
...
```

```
@implementation XYZPerson

- (void)sayHello {
    [self saySomething:@"Hello, World!"];
}

- (void)saySomething:(NSString *)greeting {
    NSLog(@"%@", greeting);
}

@end
```

```
@implementation XYZShoutingPerson

- (void)saySomething:(NSString *)greeting {
    NSString *uppercaseGreeting = [greeting uppercaseString];
    [super saySomething:uppercaseGreeting];
}

@end
```

Objects Are Created Dynamically

As described earlier in this chapter, memory is allocated dynamically for an Objective-C object. The first step in creating an object is to make sure enough memory is allocated not only for the properties defined by an object's class, but also the properties defined on each of the superclasses in its inheritance chain.

The `NSObject` root class provides a class method, `alloc`, which handles this process for you:

```
+ (id)alloc;
```

Notice that the return type of this method is `id`. This is a special keyword used in Objective-C to mean "some kind of object." It is a pointer to an object, like `(NSObject *)`, but is special in that it doesn't use an asterisk. It's described in more detail later in this chapter, in Objective-C Is a Dynamic Language.

The `alloc` method has one other important task, which is to clear out the memory allocated for the object's properties by setting them to zero. This avoids the usual problem of memory containing garbage from whatever was stored before, but is not enough to initialize an object completely.

You need to combine a call to `alloc` with a call to `init`, another `NSObject` method:

```
- (id)init;
```

The `init` method is used by a class to make sure its properties have suitable initial values at creation, and is covered in more detail in the next chapter.

Note that `init` also returns an `id`.

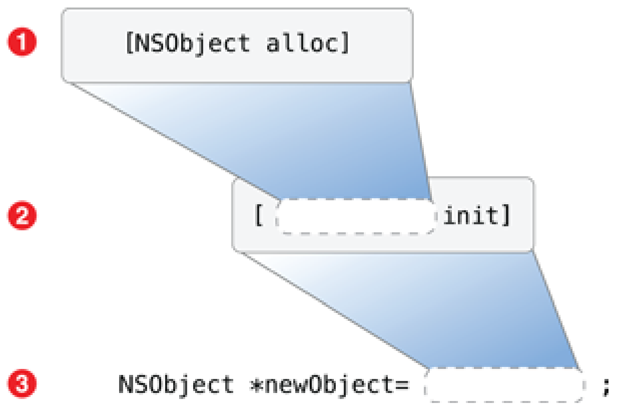
If one method returns an object pointer, it's possible to nest the call to that method as the receiver in a call to another method, thereby combining multiple message calls in one statement. The correct way to allocate and initialize an object is to nest the `alloc` call *inside* the call to `init`, like this:

```
    NSObject *newObject = [[NSObject alloc] init];
```

This example sets the `newObject` variable to point to a newly created `NSObject` instance.

The innermost call is carried out first, so the `NSObject` class is sent the `alloc` method, which returns a newly allocated `NSObject` instance. This returned object is then used as the receiver of the `init` message, which itself returns the object back to be assigned to the `newObject` pointer, as shown in Figure 2-5.

Figure 2-5 Nesting the alloc and init message



Note: It's possible for `init` to return a different object than was created by `alloc`, so it's best practice to nest the calls as shown.

Never initialize an object without reassigning any pointer to that object. As an example, don't do this:

```
    NSObject *someObject = [NSObject alloc];
    [someObject init];
```

If the call to `init` returns some other object, you'll be left with a pointer to the object that was originally allocated but never initialized.

Initializer Methods Can Take Arguments

Some objects need to be initialized with required values. An NSNumber object, for example, must be created with the numeric value it needs to represent.

The NSNumber class defines several initializers, including:

```
- (id)initWithBool:(BOOL)value;
- (id)initWithFloat:(float)value;
- (id)initWithInt:(int)value;
- (id)initWithLong:(long)value;
```

Initialization methods with arguments are called in just the same way as plain init methods—an NSNumber object is allocated and initialized like this:

```
    NSNumber *magicNumber = [[NSNumber alloc] initWithInt:42];
```

Class Factory Methods Are an Alternative to Allocation and Initialization

As mentioned in the previous chapter, a class can also define factory methods. Factory methods offer an alternative to the traditional alloc] init] process, without the need to nest two methods.

The NSNumber class defines several class factory methods to match its initializers, including:

```
+ (NSNumber *)numberWithBool:(BOOL)value;
+ (NSNumber *)numberWithFloat:(float)value;
+ (NSNumber *)numberWithInt:(int)value;
+ (NSNumber *)numberWithLong:(long)value;
```

A factory method is used like this:

```
    NSNumber *magicNumber = [NSNumber numberWithInt:42];
```

This is effectively the same as the previous example using alloc] initWithInt:]. Class factory methods usually just call straight through to alloc and the relevant init method, and are provided for convenience.

Use new to Create an Object If No Arguments Are Needed for Initialization

It’s also possible to create an instance of a class using the new class method. This method is provided by NSObject and doesn’t need to be overridden in your own subclasses.

It’s effectively the same as calling alloc and init with no arguments:

```
    XYZObject *object = [XYZObject new];
    // is effectively the same as:
    XYZObject *object = [[XYZObject alloc] init];
```

Literals Offer a Concise Object-Creation Syntax

Some classes allow you to use a more concise, *literal* syntax to create instances.

You can create an NSString instance, for example, using a special literal notation, like this:

```
NSString *someString = @"Hello, World!";
```

This is effectively the same as allocating and initializing an `NSString` or using one of its class factory methods:

```
NSString *someString = [NSString stringWithCString:"Hello, World!"
                                          encoding:NSUTF8StringEncoding];
```

The `NSNumber` class also allows a variety of literals:

```
NSNumber *myBOOL = @YES;
NSNumber *myFloat = @3.14f;
NSNumber *myInt = @42;
NSNumber *myLong = @42L;
```

Again, each of these examples is effectively the same as using the relevant initializer or a class factory method.

You can also create an `NSNumber` using a boxed expression, like this:

```
NSNumber *myInt = @(84 / 2);
```

In this case, the expression is evaluated, and an `NSNumber` instance created with the result.

Objective-C also supports literals to create immutable `NSArray` and `NSDictionary` objects; these are discussed further in Values and Collections.

Objective-C Is a Dynamic Language

As mentioned earlier, you need to use a pointer to keep track of an object in memory. Because of Objective-C's dynamic nature, it doesn't matter what specific class type you use for that pointer—the correct method will always be called on the relevant object when you send it a message.

The `id` type defines a generic object pointer. It's possible to use `id` when declaring a variable, but you lose *compile*-time information about the object.

Consider the following code:

```
id someObject = @"Hello, World!";
[someObject removeAllObjects];
```

In this case, `someObject` will point to an `NSString` instance, but the compiler knows nothing about that instance beyond the fact that it's some kind of object. The `removeAllObjects` message is defined by some Cocoa or Cocoa Touch objects (such as `NSMutableArray`) so the compiler doesn't complain, even though this code would generate an exception at runtime because an `NSString` object can't respond to `removeAllObjects`.

Rewriting the code to use a static type:

```
NSString *someObject = @"Hello, World!";
[someObject removeAllObjects];
```

means that the compiler will now generate an error because `removeAllObjects` is not declared in any public `NSString` interface that it knows about.

Because the class of an object is determined at runtime, it makes no difference what type you assign a variable when creating or working with an instance. To use the `XYZPerson` and

XYZShoutingPerson classes described earlier in this chapter, you might use the following code:

```
XYZPerson *firstPerson = [[XYZPerson alloc] init];
XYZPerson *secondPerson = [[XYZShoutingPerson alloc] init];
[firstPerson sayHello];
[secondPerson sayHello];
```

Although both firstPerson and secondPerson are statically typed as XYZPerson objects, secondPerson will point, at *runtime*, to an XYZShoutingPerson object. When the sayHello method is called on each object, the correct implementations will be used; for secondPerson, this means the XYZShoutingPerson version.

Determining Equality of Objects

If you need to determine whether one object is the same as another object, it's important to remember that you're working with pointers.

The standard C equality operator == is used to test equality between the values of two variables, like this:

```
if (someInteger == 42) {
    // someInteger has the value 42
}
```

When dealing with objects, the == operator is used to test whether two separate pointers are pointing to the same object:

```
if (firstPerson == secondPerson) {
    // firstPerson is the same object as secondPerson
}
```

If you need to test whether two objects represent the same data, you need to call a method like isEqual:, available from NSObject:

```
if ([firstPerson isEqual:secondPerson]) {
    // firstPerson is identical to secondPerson
}
```

If you need to compare whether one object represents a greater or lesser value than another object, you can't use the standard C comparison operators > and <. Instead, the basic Foundation types, like NSNumber, NSString and NSDate, provide a compare: method:

```
if ([someDate compare:anotherDate] == NSOrderedAscending) {
    // someDate is earlier than anotherDate
}
```

Working with nil

It's always a good idea to initialize scalar variables at the time you declare them, otherwise their initial values will contain garbage from the previous stack contents:

```
BOOL success = NO;
int magicNumber = 42;
```

This isn't necessary for object pointers, because the compiler will automatically set the variable to `nil` if you don't specify any other initial value:

```
    XYZPerson *somePerson;
    // somePerson is automatically set to nil
```

A `nil` value is the safest way to initialize an object pointer if you don't have another value to use, because it's perfectly acceptable in Objective-C to send a message to `nil`. If you do send a message to `nil`, obviously nothing happens.

> **Note:** If you expect a return value from a message sent to `nil`, the return value will be `nil` for object return types, `0` for numeric types, and `NO` for `BOOL` types. Returned structures have all members initialized to zero.

If you need to check to make sure an object is not `nil` (that a variable points to an object in memory), you can either use the standard C inequality operator:

```
    if (somePerson != nil) {
        // somePerson points to an object
    }
```

or simply supply the variable:

```
    if (somePerson) {
        // somePerson points to an object
    }
```

If the `somePerson` variable is `nil`, its logical value is `0` (false). If it has an address, it's not zero, so evaluates as true.

Similarly, if you need to check for a `nil` variable, you can either use the equality operator:

```
    if (somePerson == nil) {
        // somePerson does not point to an object
    }
```

or just use the C logical negation operator:

```
    if (!somePerson) {
        // somePerson does not point to an object
    }
```

Exercises

1. Open the `main.m` file in your project from the exercises at the end of the last chapter and find the `main()` function. As with any executable written in C, this function represents the starting point for your application.

   Create a new `XYZPerson` instance using `alloc` and `init`, and then call the `sayHello` method.

   > **Note:** If the compiler doesn't prompt you automatically, you will need to import the header file (containing the `XYZPerson` interface) at the top of `main.m`.

2. Implement the `saySomething:` method shown earlier in this chapter, and rewrite the `sayHello` method to use it. Add a variety of other greetings and call each of them on the instance you created above.
3. Create new class files for the `XYZShoutingPerson` class, set to inherit from `XYZPerson`.

   Override the `saySomething:` method to display the uppercase greeting, and test the behavior on an `XYZShoutingPerson` instance.
4. Implement the `XYZPerson` class `person` factory method you declared in the previous chapter, to return a correctly allocated and initialized instance of the `XYZPerson` class, then use the method in `main()` instead of your nested `alloc` and `init`.

Tip: Rather than using `[[XYZPerson alloc] init]` in the class factory method, instead try using `[[self alloc] init]`.

Using `self` in a class factory method means that you're referring to the class itself.

This means that you don't have to override the `person` method in the `XYZShoutingPerson` implementation to create the correct instance. Test this by checking that:

```
    XYZShoutingPerson *shoutingPerson = [XYZShoutingPerson person];
```

creates the correct type of object.

5. Create a new local `XYZPerson` pointer, but don't include any value assignment.

   Use a branch (`if` statement) to check whether the variable is automatically assigned as `nil`.