Defining Classes

When you write software for OS X or iOS, most of your time is spent working with objects. Objects in Objective-C are just like objects in other object-oriented programming languages: they package data with related behavior.

An app is built as a large ecosystem of interconnected objects that communicate with each other to solve specific problems, such as displaying a visual interface, responding to user input, or storing information. For OS X or iOS development, you don't need to create objects from scratch to solve every conceivable problem; instead you have a large library of existing objects available for your use, provided by Cocoa (for OS X) and Cocoa Touch (for iOS).

Some of these objects are immediately usable, such as basic data types like strings and numbers, or user interface elements like buttons and table views. Some are designed for you to customize with your own code to behave in the way you require. The app development process involves deciding how best to customize and combine the objects provided by the underlying frameworks with your own objects to give your app its unique set of features and functionality.

In object-oriented programming terms, an object is an instance of a class. This chapter demonstrates how to define classes in Objective-C by declaring an interface, which describes the way you intend the class and its instances to be used. This interface includes the list of messages that the class can receive, so you also need to provide the class implementation, which contains the code to be executed in response to each message.

Classes Are Blueprints for Objects

A class describes the behavior and properties common to any particular type of object. For a string object (in Objective-C, this is an instance of the class `NSString`), the class offers various ways to examine and convert the internal characters that it represents. Similarly, the class used to describe a number object (`NSNumber`) offers functionality around an internal numeric value, such as converting that value to a different numeric type.

In the same way that multiple buildings constructed from the same blueprint are identical in structure, every instance of a class shares the same properties and behavior as all other instances of that class. Every `NSString` instance behaves in the same way, regardless of the internal string of characters it holds.

Any particular object is designed to be used in specific ways. You might know that a string object represents some string of characters, but you don't need to know the exact internal mechanisms used to store those characters. You don't know anything about the internal behavior used by the object itself to work directly with its characters, but you do need to know how you are expected to interact with the object, perhaps to ask it for specific characters or request a new object in which all the original characters are converted to uppercase.

In Objective-C, the *class interface* specifies exactly how a given type of object is intended to be used by other objects. In other words, it defines the public interface between instances of the class and the outside world.

Mutability Determines Whether a Represented Value Can Be Changed

Some classes define objects that are *immutable*. This means that the internal contents must be set when an object is created, and cannot subsequently be changed by other objects. In Objective-C, all basic `NSString` and `NSNumber` objects are immutable. If you need to represent a different number, you must use a new `NSNumber` instance.

Some immutable classes also offer a *mutable* version. If you specifically need to change the contents of a string at runtime, for example by appending characters as they are received over a network connection, you can use an instance of the `NSMutableString` class. Instances of this class behave just like `NSString` objects, except that they also offer functionality to change the characters that the object represents.

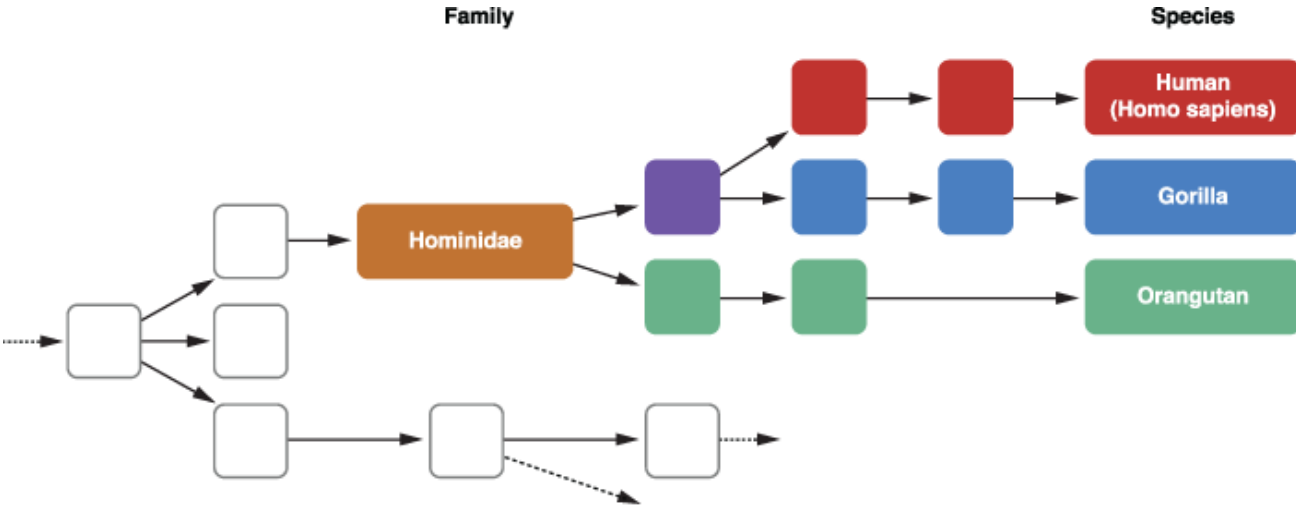
Although `NSString` and `NSMutableString` are different classes, they have many similarities. Rather than writing two completely separate classes from scratch that just happen to have some similar behavior, it makes sense to make use of inheritance.

Classes Inherit from Other Classes

In the natural world, taxonomy classifies animals into groups with terms like species, genus, and family. These groups are hierarchical, such that multiple species may belong to one genus, and multiple genera to one family.

Gorillas, humans, and orangutans, for example, have a number of obvious similarities. Although they each belong to different species, and even different genera, tribes, and subfamilies, they are taxonomically related since they all belong to the same family (called “Hominidae”), as shown in Figure 1-1.

Figure 1-1 Taxonomic relationships between species



In the world of object-oriented programming, objects are also categorized into hierarchical groups. Rather than using distinct terms for the different hierarchical levels such as genus or species, objects are simply organized into classes. In the same way that humans inherit certain characteristics as members of the Hominidae family, a class can be set to inherit functionality from a parent class.

When one class inherits from another, the child inherits all the behavior and properties defined by the parent. It also has the opportunity either to define its own additional behavior and properties, or override the behavior of the parent.

In the case of Objective-C string classes, the class description for `NSMutableString` specifies that the class inherits from `NSString`, as shown in Figure 1-2. All of the functionality provided by `NSString` is available in `NSMutableString`, such as querying specific characters or requesting new uppercase strings, but `NSMutableString` adds methods that allow you to append, insert, replace or delete substrings and individual characters.

Figure 1-2 NSMutableString class inheritance

The Root Class Provides Base Functionality

In the same way that all living organisms share some basic “life” characteristics, some functionality is common across all objects in Objective-C.

When an Objective-C object needs to work with an instance of another class, it is expected that the other class offers certain basic characteristics and behavior. For this reason, Objective-C defines a root class from which the vast majority of other classes inherit, called `NSObject`. When one object encounters another object, it expects to be able to interact using at least the basic behavior defined by the `NSObject` class description.

When you’re defining your own classes, you should at a minimum inherit from `NSObject`. In general, you should find a Cocoa or Cocoa Touch object that offers the closest functionality to what you need and inherit from that.

If you want to define a custom button for use in an iOS app, for example, and the provided `UIButton` class doesn’t offer enough customizable attributes to satisfy your needs, it makes more sense to create a new class inheriting from `UIButton` than from `NSObject`. If you simply inherited from `NSObject`, you’d need to duplicate all the complex visual interactions and communication defined by the `UIButton` class just to make your button behave in the way expected by the user. Furthermore, by inheriting from `UIButton`, your subclass automatically gains any future enhancements or bug fixes that might be applied to the internal `UIButton` behavior.

The `UIButton` class itself is defined to inherit from `UIControl`, which describes basic behavior common to all user interface controls on iOS. The `UIControl` class in turn inherits from `UIView`, giving it functionality common to objects that are displayed on screen. `UIView` inherits from `UIResponder`, allowing it to respond to user input such as taps, gestures or shakes. Finally, at the root of the tree, `UIResponder` inherits from `NSObject`, as shown in Figure 1-3.

Figure 1-3 UIButton class inheritance

This chain of inheritance means that any custom subclass of `UIButton` would inherit not only the functionality declared by `UIButton` itself, but also the functionality inherited from each superclass in turn. You'd end up with a class for an object that behaved like a button, could display itself on screen, respond to user input, and communicate with any other basic Cocoa Touch object.

It's important to keep the inheritance chain in mind for any class you need to use, in order to work out exactly what it can do. The class reference documentation provided for Cocoa and Cocoa Touch, for example, allows easy navigation from any class to each of its superclasses. If you can't find what you're looking for in one class interface or reference, it may very well be defined or documented in a superclass further up the chain.

The Interface for a Class Defines Expected Interactions

One of the many benefits of object-oriented programming is the idea mentioned earlier—all you need to know in order to use a class is how to interact with its instances. More specifically, an object should be designed to hide the details of its internal implementation.

If you use a standard `UIButton` in an iOS app, for example, you don't need to worry about how pixels are manipulated so that the button appears on screen. All you need to know is that you can change certain attributes, such as the button's title and color, and trust that when you add it to your visual interface, it will be displayed correctly and behave in the way you expect.

When you're defining your own class, you need to start by figuring out these public attributes and behaviors. What attributes do you want to be accessible publicly? Should you allow those attributes to be changed? How do other objects communicate with instances of your class?

This information goes into the interface for your class—it defines the way you intend other objects to interact with instances of your class. The public interface is described separately from the *internal* behavior of your class, which makes up the class implementation. In Objective-C, the interface and implementation are usually placed in separate files so that you only need to make the interface public.

Basic Syntax

The Objective-C syntax used to declare a class interface looks like this:

```
@interface SimpleClass : NSObject


@end
```

This example declares a class named `SimpleClass`, which inherits from `NSObject`.

The public properties and behavior are defined inside the `@interface` declaration. In this example, nothing is specified beyond the superclass, so the only functionality expected to be available on instances of `SimpleClass` is the functionality inherited from `NSObject`.

Properties Control Access to an Object's Values

Objects often have properties intended for public access. If you define a class to represent a human being in a record-keeping app, for example, you might decide you need properties for strings representing a person's first and last names.

Declarations for these properties should be added inside the interface, like this:

```
@interface Person : NSObject


@property NSString *firstName;
@property NSString *lastName;


@end
```

In this example, the `Person` class declares two public properties, both of which are instances of the `NSString` class.

Both these properties are for Objective-C objects, so they use an *asterisk* to indicate that they are C pointers. They are also statements just like any other variable declaration in C, and therefore require a semi-colon at the end.

You might decide to add a property to represent a person's year of birth to allow you to sort people in year groups rather than just by name. You could use a property for a number *object*:

```
@property NSNumber *yearOfBirth;
```

but this might be considered overkill just to store a simple numeric value. One alternative would be to use one of the primitive types provided by C, which hold scalar values, such as an integer:

```
@property int yearOfBirth;
```

Property Attributes Indicate Data Accessibility and Storage Considerations

The examples shown so far all declare properties that are intended for complete public access. This means that other objects can both read and change the values of the properties.

In some cases, you might decide to declare that a property is not intended to be changed. In the real world, a person must fill out a large amount of paperwork to change their documented first or last name. If you were writing an official record-keeping app, you might choose that the public properties for a person's name be specified as read-only, requiring that any changes be requested through an intermediary object responsible for validating the request and approving or denying it.

Objective-C property declarations can include *property attributes*, which are used to indicate, among other things, whether a property is intended to be read-only. In an official record-keeping app, the Person class interface might look like this:

```
@interface Person : NSObject
@property (readonly) NSString *firstName;
@property (readonly) NSString *lastName;
@end
```

Property attributes are specified inside parentheses after the `@property` keyword, and are described fully in Declare Public Properties for Exposed Data.

Method Declarations Indicate the Messages an Object Can Receive

The examples so far have involved a class describing a typical model object, or an object designed primarily to encapsulate data. In the case of a `Person` class, it's possible that there wouldn't need to be any functionality beyond being able to access the two declared properties. The majority of classes, however, do include behavior in addition to any declared properties.

Given that Objective-C software is built from a large network of objects, it's important to note that those objects can interact with each other by sending messages. In Objective-C terms, one object sends a message to another object by calling a method on that object.

Objective-C methods are conceptually similar to standard functions in C and other programming languages, though the syntax is quite different. A C function declaration looks like this:

```
void SomeFunction();
```

The equivalent Objective-C method declaration looks like this:

```
- (void)someMethod;
```

In this case, the method has no parameters. The C `void` keyword is used inside parentheses at the beginning of the declaration to indicate that the method doesn't return any value once it's finished.

The minus sign (`-`) at the front of the method name indicates that it is an instance method, which can be called on any instance of the class. This differentiates it from class methods, which can be called on the class itself, as described in Objective-C Classes Are also Objects.

As with C function prototypes, a method declaration inside an Objective-C class interface is just like any other C statement and requires a terminating semi-colon.

Methods Can Take Parameters

If you need to declare a method to take one or more parameters, the syntax is very different to a typical C function.

For a C function, the parameters are specified inside parentheses, like this:

```
void SomeFunction(SomeType value);
```

An Objective-C method declaration includes the parameters as part of its name, using colons, like this:

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

As with the return type, the parameter type is specified in parentheses, just like a standard C type-cast.

If you need to supply multiple parameters, the syntax is again quite different from C. Multiple parameters to a C function are specified inside the parentheses, separated by commas; in Objective-C, the declaration for a method taking two parameters looks like this:

```
- (void)someMethodWithFirstValue:(SomeType)value1 secondValue:(AnotherType)value2;
```

In this example, `value1` and `value2` are the names used in the implementation to access the values supplied when the method is called, as if they were variables.

Some programming languages allow function definitions with so-called *named arguments*; it's important to note that this is not the case in Objective-C. The order of the parameters in a method call must match the method declaration, and in fact the `secondValue:` portion of the method declaration is part of the name of the method:

```
someMethodWithFirstValue:secondValue:
```

This is one of the features that helps make Objective-C such a readable language, because the values passed by a method call are specified *inline*, next to the relevant portion of the method name, as described in You Can Pass Objects for Method Parameters.

Note: The `value1` and `value2` value names used above aren't strictly part of the method declaration, which means it's not necessary to use exactly the same value names in the declaration as you do in the implementation. The only requirement is that the signature matches, which means you must keep the name of the method as well as the parameter and return types exactly the same.

As an example, this method has the same signature as the one shown above:

```
- (void)someMethodWithFirstValue:(SomeType)info1 secondValue:(AnotherType)info2;
```

These methods have different signatures to the one above:

```
- (void)someMethodWithFirstValue:(SomeType)info1 anotherValue:(AnotherType)info2;
- (void)someMethodWithFirstValue:(SomeType)info1 secondValue:(YetAnotherType)info2;
```

Class Names Must Be Unique

It's important to note that the name of each class must be unique within an app, even across included libraries or frameworks. If you attempt to create a new class with the same name as an existing class in a project, you'll receive a compiler error.

For this reason, it's advisable to prefix the names of any classes you define, using three or more letters. These letters might relate to the app you're currently writing, or to the name of a framework of reusable code, or perhaps just your initials.

All examples given in the rest of this document use class name prefixes, like this:

```
@interface XYZPerson : NSObject
@property (readonly) NSString *firstName;
@property (readonly) NSString *lastName;
@end
```

Historical Note: If you're wondering why so many of the classes you encounter have an `NS` prefix, it's because of the past history of Cocoa and Cocoa Touch. Cocoa began life as the collected frameworks used to build apps for the NeXTStep operating system. When Apple purchased NeXT back in 1996, much of NeXTStep was incorporated into OS X, including the existing class names. Cocoa *Touch* was introduced as the iOS equivalent of Cocoa; some classes are available in both Cocoa and Cocoa Touch, though there are also a large number of classes unique to each platform.

Two-letter prefixes like `NS` and `UI` (for User Interface elements on iOS) are reserved for use by Apple.

Method and property names, by contrast, need only be unique within the class in which they are defined. Although every C function in an app must have a unique name, it's perfectly acceptable (and often desirable) for multiple Objective-C classes to define methods with the same name. You can't define a method more than once within the same class declaration, however, though if you wish to override a method inherited from a parent class, you must use the exact name used in the original declaration.

As with methods, an object's properties and instance variables (described in Most Properties Are Backed by Instance Variables) need to be unique only within the class in which they are defined. If you make use of global variables, however, these must be named uniquely within an app or project.

Further naming conventions and suggestions are given in Conventions.

The Implementation of a Class Provides Its Internal Behavior

Once you've defined the interface for a class, including the properties and methods intended for public access, you need to write the code to implement the class behavior.

As stated earlier, the interface for a class is usually placed inside a dedicated file, often referred to as a *header file*, which generally has the filename extension `.h`. You write the implementation for an Objective-C class inside a source code file with the extension `.m`.

Whenever the interface is defined in a header file, you'll need to tell the compiler to read it before trying to compile the implementation in the source code file. Objective-C provides a preprocessor directive, `#import`, for this purpose. It's similar to the C `#include` directive, but makes sure that a file is only included once during compilation.

Note that preprocessor directives are different from traditional C statements and do not use a terminating semi-colon.

Basic Syntax

The basic syntax to provide the implementation for a class looks like this:

```
#import "XYZPerson.h"
```

```
@implementation XYZPerson


@end
```

If you declare any methods in the class interface, you'll need to implement them inside this file.

Implementing Methods

For a simple class interface with one method, like this:

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

the implementation might look like this:

```
#import "XYZPerson.h"


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

This example uses the `NSLog()` function to log a message to the console. It's similar to the standard C library `printf()` function, and takes a variable number of parameters, the first of which must be an Objective-C string.

Method implementations are similar to C function definitions in that they use braces to contain the relevant code. Furthermore, the name of the method must be identical to its prototype, and the parameter and return types must match exactly.

Objective-C inherits case sensitivity from C, so this method:

```
- (void)sayhello {
}
```

would be treated by the compiler as completely different to the `sayHello` method shown earlier.

In general, method names should begin with a lowercase letter. The Objective-C convention is to use more descriptive names for methods than you might see used for typical C functions. If a method name involves multiple words, use camel case (capitalizing the first letter of each new word) to make them easy to read.

Note also that whitespace is flexible in Objective-C. It's customary to indent each line inside any block of code using either tabs or spaces, and you'll often see the opening left brace on a separate line, like this:

```
- (void)sayHello
{
    NSLog(@"Hello, World!");
}
```

Xcode, Apple's integrated development environment (IDE) for creating OS X and iOS software, will automatically indent your code based on a set of customizable user preferences. See Changing the Indent and Tab Width in *Xcode Workspace Guide* for more information.

You'll see many more examples of method implementations in the next chapter, Working with Objects.

Objective-C Classes Are also Objects

In Objective-C, a class is itself an object with an opaque type called `Class`. Classes can't have properties defined using the declaration syntax shown earlier for instances, but they can receive messages.

The typical use for a class method is as a *factory method*, which is an alternative to the object allocation and initialization procedure described in Objects Are Created Dynamically. The `NSString` class, for example, has a variety of factory methods available to create either an empty string object, or a string object initialized with specific characters, including:

```
+ (id)string;
+ (id)stringWithString:(NSString *)aString;
+ (id)stringWithFormat:(NSString *)format, …;
+ (id)stringWithContentsOfFile:(NSString *)path encoding:(NSStringEncoding)enc error:
(NSError **)error;
+ (id)stringWithCString:(const char *)cString encoding:(NSStringEncoding)enc;
```

As shown in these examples, class methods are denoted by the use of a `+` sign, which differentiates them from instance methods using a `-` sign.

Class method prototypes may be included in a class interface, just like instance method prototypes. Class methods are implemented in the same way as instance methods, inside the `@implementation` block for the class.

Exercises

Note: In order to follow the exercises given at the end of each chapter, you may wish to create an Xcode project. This will allow you to make sure that your code compiles without errors.

Use Xcode's New Project template window to create a *Command Line Tool* from the available OS X Application project templates. When prompted, specify the project's Type as *Foundation*.

1. Use Xcode's New File template window to create the interface and implementation files for an Objective-C class called `XYZPerson`, which inherits from `NSObject`.
2. Add properties for a person's first name, last name and date of birth (dates are represented by the `NSDate` class) to the `XYZPerson` class interface.
3. Declare the `sayHello` method and implement it as shown earlier in the chapter.
4. Add a declaration for a class factory method, called "`person`". Don't worry about implementing this method until you've read the next chapter.

   Note: If you're compiling the code, you'll get a warning about an "Incomplete implementation" due to this missing implementation.