Programming with Objective-C

Developer

Contents

Figures

SwiftObjective-C

About Objective-C

Objective-C is the primary programming language you use when writing software for OS X and iOS. It's a superset of the C programming language and provides object-oriented capabilities and a dynamic runtime. Objective-C inherits the syntax, primitive types, and flow control statements of C and adds syntax for defining classes and methods. It also adds language-level support for object graph management and object literals while providing dynamic typing and binding, deferring many responsibilities until runtime.

At a Glance

This document introduces the Objective-C language and offers extensive examples of its use. You'll learn how to create your own classes describing custom objects and see how to work with some of the framework classes provided by Cocoa and Cocoa Touch. Although the framework classes are separate from the language, their use is tightly wound into coding with Objective-C and many language-level features rely on behavior offered by these classes.

An App Is Built from a Network of Objects

When building apps for OS X or iOS, you'll spend most of your time working with objects. Those objects are instances of Objective-C classes, some of which are provided for you by Cocoa or Cocoa Touch and some of which you'll write yourself.

If you're writing your own class, start by providing a description of the class that details the intended public interface to instances of the class. This interface includes the public properties to encapsulate relevant data, along with a list of methods. Method declarations indicate the messages that an object can receive, and include information about the parameters required whenever the method is called. You'll also provide a class implementation, which includes the executable code for each method declared in the interface.

Relevant Chapters: Defining Classes (page 12), Working with Objects (page 25), Encapsulating Data (page 43)

Categories Extend Existing Classes

Rather than creating an entirely new class to provide minor additional capabilities over an existing class, it's possible to define a category to add custom behavior to an existing class. You can use a category to add methods to any class, including classes for which you don't have the original implementation source code, such as framework classes like `NSString`.

If you do have the original source code for a class, you can use a class extension to add new properties, or modify the attributes of existing properties. Class extensions are commonly used to hide private behavior for use either within a single source code file, or within the private implementation of a custom framework.

Relevant Chapters: Customizing Existing Classes (page 68)

Protocols Define Messaging Contracts

The majority of work in an Objective-C app occurs as a result of objects sending messages to each other. Often, these messages are defined by the methods declared explicitly in a class interface. Sometimes, however, it is useful to be able to define a set of related methods that aren't tied directly to a specific class.

Objective-C uses protocols to define a group of related methods, such as the methods an object might call on its delegate, which are either optional or required. Any class can indicate that it adopts a protocol, which means that it must also provide implementations for all of the required methods in the protocol.

Relevant Chapters: Working with Protocols (page 77)

Values and Collections Are Often Represented as Objective-C Objects

It's common in Objective-C to use Cocoa or Cocoa Touch classes to represent values. The `NSString` class is used for strings of characters, the `NSNumber` class for different types of numbers such as integer or floating point, and the `NSValue` class for other values such as C structures. You can also use any of the primitive types defined by the C language, such as `int`, `float` or `char`.

Collections are usually represented as instances of one of the collection classes, such as `NSArray`, `NSSet`, or `NSDictionary`, which are each used to collect other Objective-C objects.

Relevant Chapters: Values and Collections (page 85)

Blocks Simplify Common Tasks

Blocks are a language feature introduced to C, Objective-C and C++ to represent a unit of work; they encapsulate a block of code along with captured state, which makes them similar to closures in other programming languages. Blocks are often used to simplify common tasks such as collection enumeration, sorting and testing. They also make it easy to schedule tasks for concurrent or asynchronous execution using technologies like Grand Central Dispatch (GCD).

Relevant Chapters: Working with Blocks (page 104)

Error Objects Are Used for Runtime Problems

Although Objective-C includes syntax for exception handling, Cocoa and Cocoa Touch use exceptions only for programming errors (such as out of bounds array access), which should be fixed before an app is shipped.

All other errors—including runtime problems such as running out of disk space or not being able to access a web service—are represented by instances of the `NSError` class. Your app should plan for errors and decide how best to handle them in order to present the best possible user experience when something goes wrong.

Relevant Chapters: Dealing with Errors (page 117)

Objective-C Code Follows Established Conventions

When writing Objective-C code, you should keep in mind a number of established coding conventions. Method names, for example, start with a lowercase letter and use camel case for multiple words; for example, `doSomething` or `doSomethingElse`. It's not just the capitalization that's important, though; you should also make sure that your code is as readable as possible, which means that method names should be expressive, but not too verbose.

In addition, there are a few conventions that are required if you wish to take advantage of language or framework features. Property accessor methods, for example, must follow strict naming conventions in order to work with technologies like Key-Value Coding (KVC) or Key-Value Observing (KVO).

Relevant Chapters: Conventions (page 122)

Prerequisites

If you are new to OS X or iOS development, you should read through *Start Developing iOS Apps Today* or *Start Developing Mac Apps Today* before reading this document, to get a general overview of the application development process for iOS and OS X. Additionally, you should become familiar with Xcode before trying to follow the exercises at the end of most chapters in this document. Xcode is the IDE used to build apps for iOS and OS X; you'll use it to write your code, design your app's user interface, test your application, and debug any problems.

Although it's preferable to have some familiarity with C or one of the C-based languages such as Java or C#, this document does include inline examples of basic C language features such as flow control statements. If you have knowledge of another higher-level programming language, such as Ruby or Python, you should be able to follow the content.

Reasonable coverage is given to general object-oriented programming principles, particularly as they apply in the context of Objective-C, but it is assumed that you have at least a minimal familiarity with basic object-oriented concepts. If you're not familiar with these concepts, you should read the relevant chapters in *Concepts in Objective-C Programming*.

See Also

The content in this document applies to Xcode 4.4 or later and assumes you are targeting either OS X v10.7 or later, or iOS 5 or later. For more information about Xcode, see *Xcode Overview*. For information on language feature availability, see *Objective-C Feature Availability Index*.

Objective-C apps use reference counting to determine the lifetime of objects. For the most part, the Automatic Reference Counting (ARC) feature of the compiler takes care of this for you. If you are unable to take advantage of ARC, or need to convert or maintain legacy code that manages an object's memory manually, you should read *Advanced Memory Management Programming Guide*.

In addition to the compiler, the Objective-C language uses a runtime system to enable its dynamic and object-oriented features. Although you don't usually need to worry about how Objective-C "works," it's possible to interact directly with this runtime system, as described by *Objective-C Runtime Programming Guide* and *Objective-C Runtime Reference*.

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.

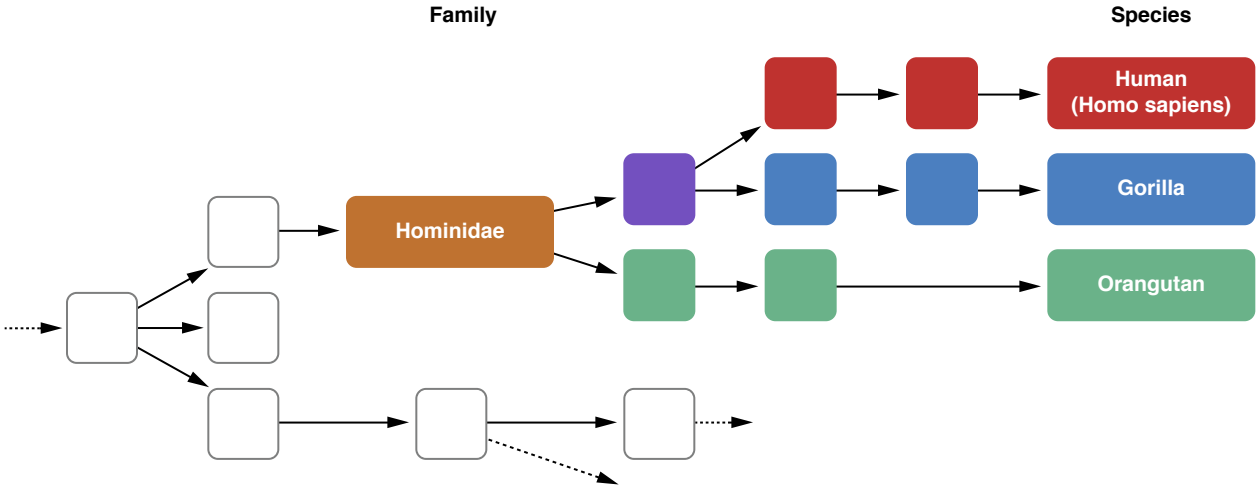
Although `NSString` and `NSMutableString` are different classes, they have many similarities. Rather than writing two completely separate classes from scratch that just happen to have some similar behavior, it makes sense to make use of inheritance.

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 (page 14).

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 (page 14). 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.

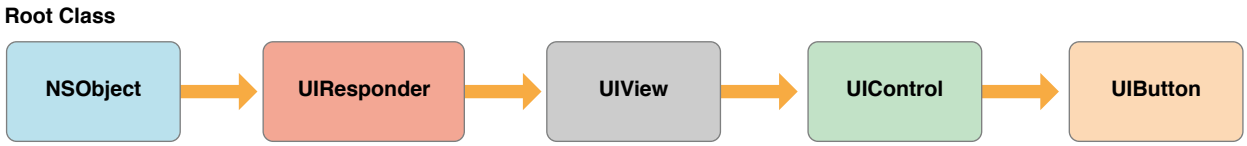
When an Objective-C object needs to work with an instance of another class, it is expected that the other class offers certain basic characteristics and behavior. For this reason, Objective-C defines a root class from which the vast majority of other classes inherit, called `NSObject`. When one object 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 (page 15).

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 (page 43).

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 (page 23).

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 (page 28).

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 (page 46)) 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 (page 122).

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 (page 25).

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 (page 34). 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.

Working with Objects

Objective-CSwift

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!*."

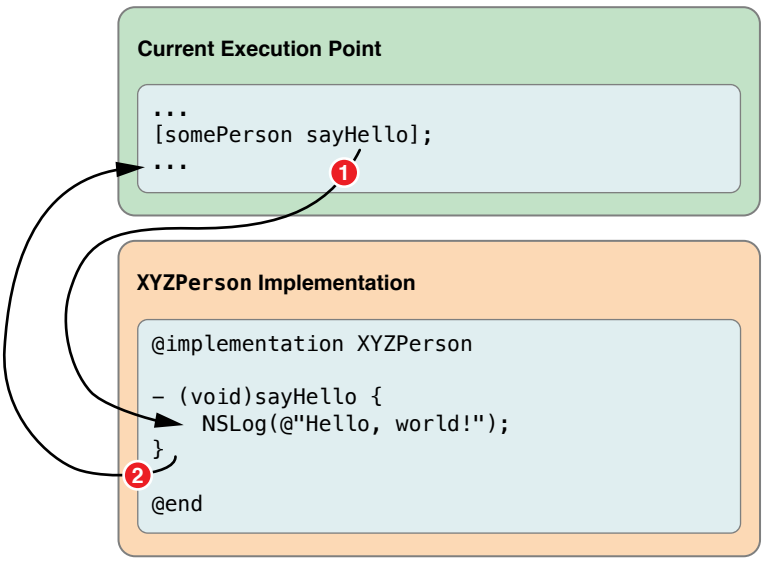
Literals and object creation are explained further in Objects Are Created Dynamically (page 34), later in this chapter.

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 (page 26) 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.

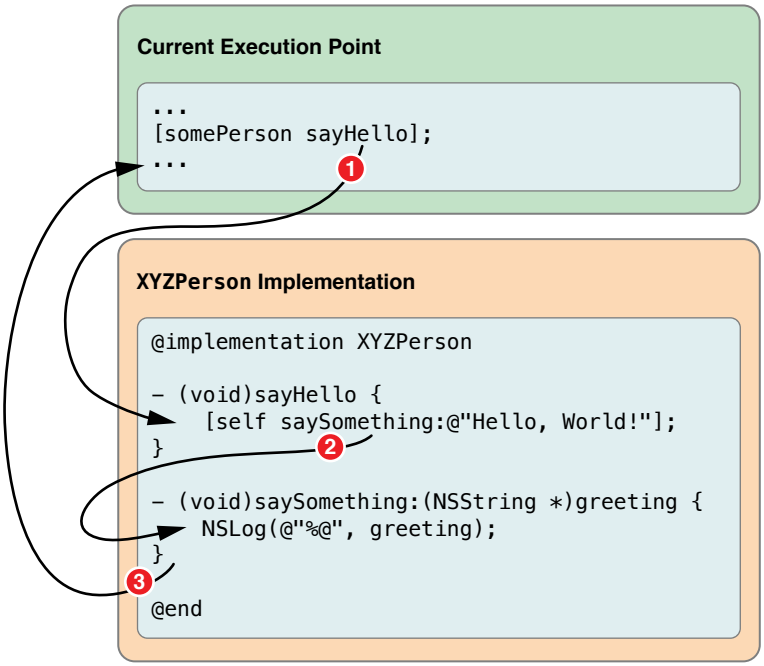
You might decide to refactor the `XYZPerson` implementation by modifying the `sayHello` method to use the `saySomething:` method shown above, thereby moving the `NSLog()` call to a separate method. This would mean you could add further methods, like `sayGoodbye`, that would each call through to the `saySomething:` 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 (page 31).

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:

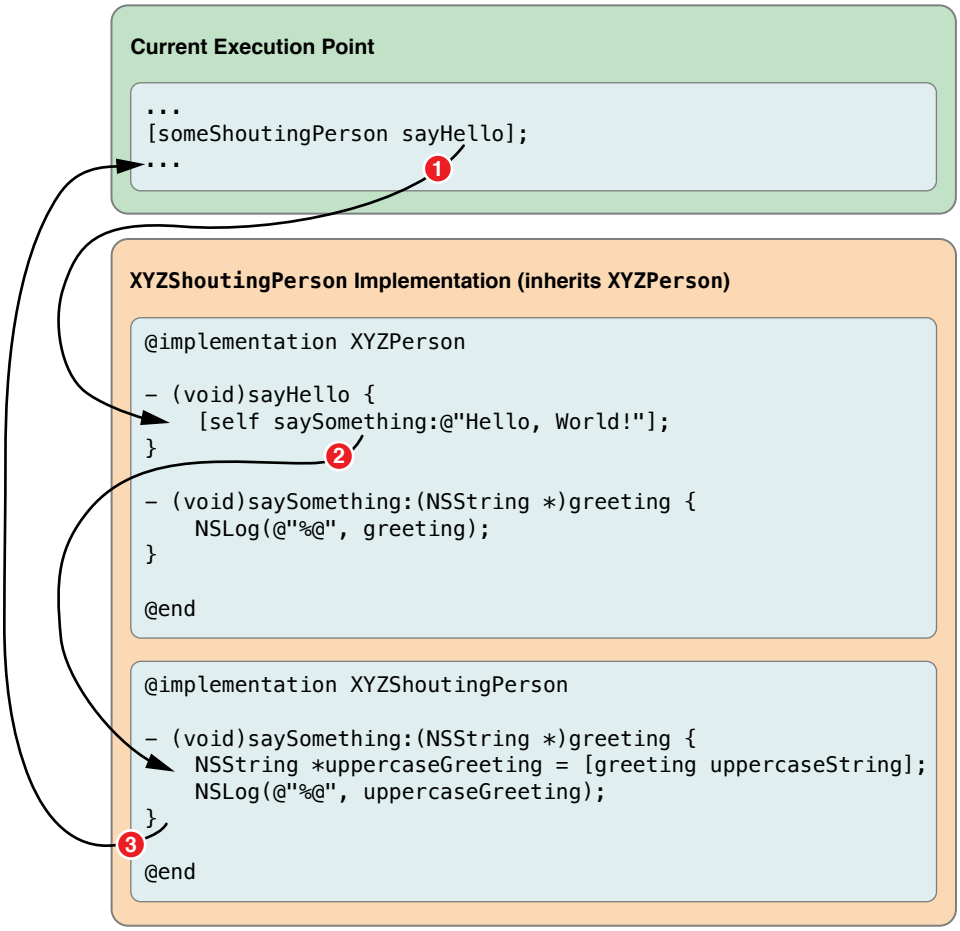
```
@interface XYZShoutingPerson : XYZPerson
@end
```

```
@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 (page 33).

Figure 2-3 Program flow for an overridden method



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.

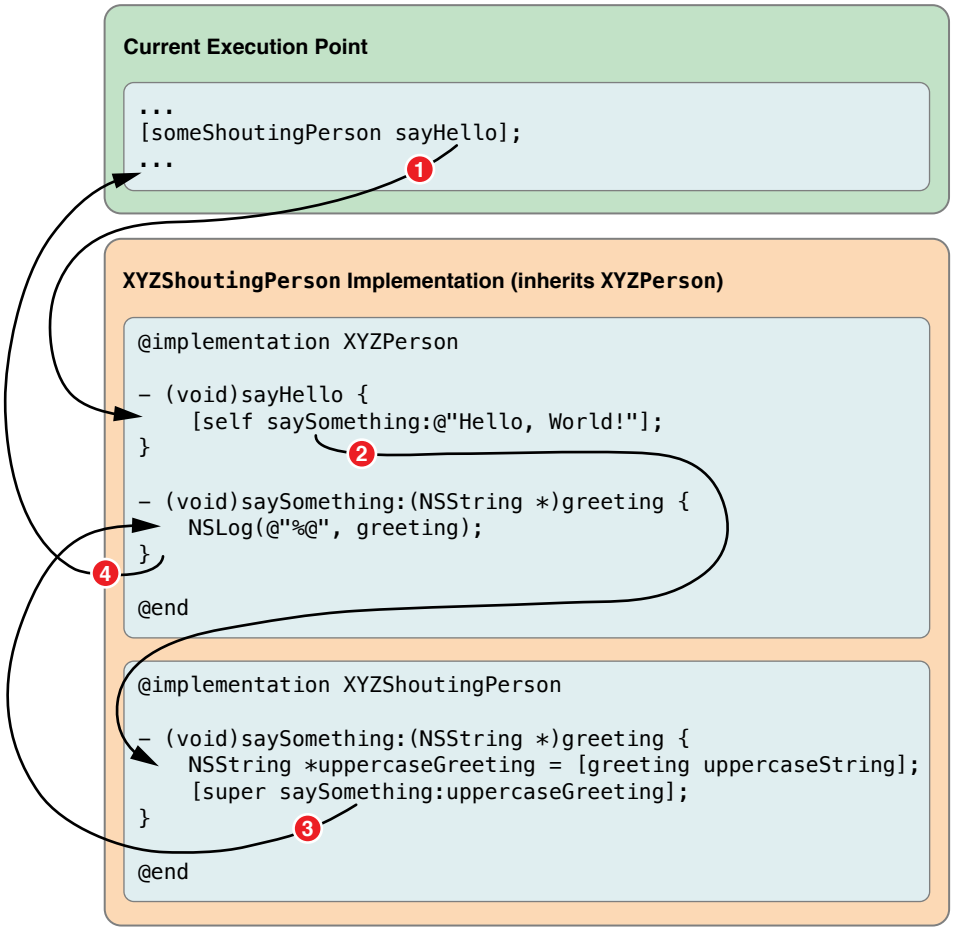
A better idea would be to change the `XYZShoutingPerson` version of `saySomething:` to call through to the superclass (`XYZPerson`) implementation to handle the actual greeting:

```
@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 (page 34).

Figure 2-4 Program flow when messaging super



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 (page 38).

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`.

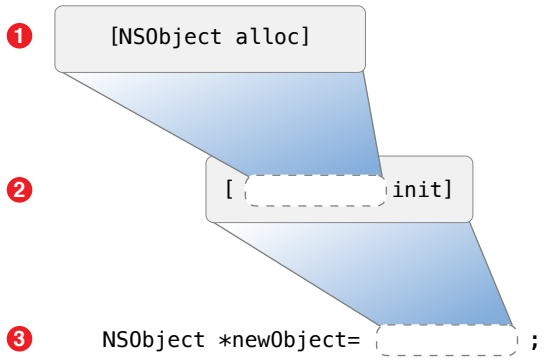
If one method returns an object pointer, it's possible to nest the call to that method as the receiver in a call to another method, thereby combining multiple message calls in one statement. The correct way to allocate and initialize an object is to nest the `alloc` call *inside* the call to `init`, 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 (page 36).

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 (page 85).

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`.

Encapsulating Data

SwiftObjective-C

In addition to the messaging behavior covered in the previous chapter, an object also encapsulates data through its properties.

This chapter describes the Objective-C syntax used to declare properties for an object and explains how those properties are implemented by default through synthesis of accessor methods and instance variables. If a property is backed by an instance variable, that variable must be set correctly in any initialization methods.

If an object needs to maintain a link to another object through a property, it's important to consider the nature of the relationship between the two objects. Although memory management for Objective-C objects is mostly handled for you through Automatic Reference Counting (ARC), it's important to know how to avoid problems like strong reference cycles, which lead to memory leaks. This chapter explains the lifecycle of an object, and describes how to think in terms of managing your graph of objects through relationships.

Properties Encapsulate an Object's Values

Most objects need to keep track of information in order to perform their tasks. Some objects are designed to model one or more values, such as a Cocoa `NSNumber` class to hold a numeric value or a custom `XYZPerson` class to model a person with a first and last name. Some objects are more general in scope, perhaps handling the interaction between a user interface and the information it displays, but even these objects need to keep track of user interface elements or the related model objects.

Declare Public Properties for Exposed Data

Objective-C properties offer a way to define the information that a class is intended to encapsulate. As you saw in Properties Control Access to an Object's Values (page 16), property declarations are included in the interface for a class, like this:

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

In this example, the `XYZPerson` class declares string properties to hold a person's first and last name.

Given that one of the primary principles in object-oriented programming is that an object should hide its internal workings behind its public interface, it's important to access an object's properties using behavior exposed by the object rather than trying to gain access to the internal values directly.

Use Accessor Methods to Get or Set Property Values

You access or set an object's properties via accessor methods:

```
NSString *firstName = [somePerson firstName];
[somePerson setFirstName:@"Johnny"];
```

By default, these accessor methods are synthesized automatically for you by the compiler, so you don't need to do anything other than declare the property using `@property` in the class interface.

The synthesized methods follow specific naming conventions:

- The method used to access the value (the *getter* method) has the same name as the property.

  The getter method for a property called `firstName` will also be called `firstName`.

- The method used to set the value (the *setter* method) starts with the word "set" and then uses the capitalized property name.

  The setter method for a property called `firstName` will be called `setFirstName:`.

If you don't want to allow a property to be changed via a setter method, you can add an attribute to a property declaration to specify that it should be `readonly`:

```
@property (readonly) NSString *fullName;
```

As well as showing other objects how they are supposed to interact with the property, attributes also tell the compiler how to synthesize the relevant accessor methods.

In this case, the compiler will synthesize a `fullName` getter method, but not a `setFullName:` method.

> **Note:** The opposite of `readonly` is `readwrite`. There's no need to specify the `readwrite` attribute explicitly, because it is the default.

If you want to use a different name for an accessor method, it's possible to specify a custom name by adding attributes to the property. In the case of Boolean properties (properties that have a `YES` or `NO` value), it's customary for the getter method to start with the word "is." The getter method for a property called `finished`, for example, should be called `isFinished`.

Again, it's possible to add an attribute on the property:

```
@property (getter=isFinished) BOOL finished;
```

If you need to specify multiple attributes, simply include them as a comma-separated list, like this:

```
@property (readonly, getter=isFinished) BOOL finished;
```

In this case, the compiler will synthesize only an `isFinished` method, but not a `setFinished:` method.

> **Note:** In general, property accessor methods should be Key-Value Coding (KVC) compliant, which means that they follow explicit naming conventions.
>
> See *Key-Value Coding Programming Guide* for more information.

Dot Syntax Is a Concise Alternative to Accessor Method Calls

As well as making explicit accessor method calls, Objective-C offers an alternative dot syntax to access an object's properties.

Dot syntax allows you to access properties like this:

```
NSString *firstName = somePerson.firstName;
somePerson.firstName = @"Johnny";
```

Dot syntax is purely a convenient wrapper around accessor method calls. When you use dot syntax, the property is still accessed or changed using the getter and setter methods mentioned above:

- Getting a value using `somePerson.firstName` is the same as using `[somePerson firstName]`
- Setting a value using `somePerson.firstName = @"Johnny"` is the same as using `[somePerson setFirstName:@"Johnny"]`

This means that property access via dot syntax is also controlled by the property attributes. If a property is marked `readonly`, you'll get a compiler error if you try to set it using dot syntax.

Most Properties Are Backed by Instance Variables

By default, a `readwrite` property will be backed by an instance variable, which will again be synthesized automatically by the compiler.

An instance variable is a variable that exists and holds its value for the life of the object. The memory used for instance variables is allocated when the object is first created (through `alloc`), and freed when the object is deallocated.

Unless you specify otherwise, the synthesized instance variable has the same name as the property, but with an underscore prefix. For a property called `firstName`, for example, the synthesized instance variable will be called `_firstName`.

Although it's best practice for an object to access its own properties using accessor methods or dot syntax, it's possible to access the instance variable directly from any of the instance methods in a class implementation. The underscore prefix makes it clear that you're accessing an instance variable rather than, for example, a local variable:

```
- (void)someMethod {
    NSString *myString = @"An interesting string";

    _someString = myString;
}
```

In this example, it's clear that `myString` is a local variable and `_someString` is an instance variable.

In general, you should use accessor methods or dot syntax for property access even if you're accessing an object's properties from within its own implementation, in which case you should use `self`:

```
- (void)someMethod {
    NSString *myString = @"An interesting string";

    self.someString = myString;
  // or
    [self setSomeString:myString];
}
```

The exception to this rule is when writing initialization, deallocation or custom accessor methods, as described later in this section.

You Can Customize Synthesized Instance Variable Names

As mentioned earlier, the default behavior for a writeable property is to use an instance variable called `_propertyName`.

If you wish to use a different name for the instance variable, you need to direct the compiler to synthesize the variable using the following syntax in your implementation:

```
@implementation YourClass
@synthesize propertyName = instanceVariableName;
...
@end
```

For example:

```
@synthesize firstName = ivar_firstName;
```

In this case, the property will still be called `firstName`, and be accessible through `firstName` and `setFirstName:` accessor methods or dot syntax, but it will be backed by an instance variable called `ivar_firstName`.

Important: If you use `@synthesize` without specifying an instance variable name, like this:

```
@synthesize firstName;
```

the instance variable will bear the same name as the property.

In this example, the instance variable will also be called `firstName`, without an underscore.

You Can Define Instance Variables without Properties

It's best practice to use a property on an object any time you need to keep track of a value or another object.

If you do need to define your own instance variables without declaring a property, you can add them inside braces at the top of the class interface or implementation, like this:

```
@interface SomeClass : NSObject {
```

```
    NSString *_myNonPropertyInstanceVariable;
}
...
@end

@implementation SomeClass {
    NSString *_anotherCustomInstanceVariable;
}
...
@end
```

> **Note:** You can also add instance variables at the top of a class extension, as described in Class Extensions Extend the Internal Implementation (page 72).

Access Instance Variables Directly from Initializer Methods

Setter methods can have additional side-effects. They may trigger KVC notifications, or perform further tasks if you write your own custom methods.

You should always access the instance variables directly from within an initialization method because at the time a property is set, the rest of the object may not yet be completely initialized. Even if you don't provide custom accessor methods or know of any side effects from within your own class, a future subclass may very well override the behavior.

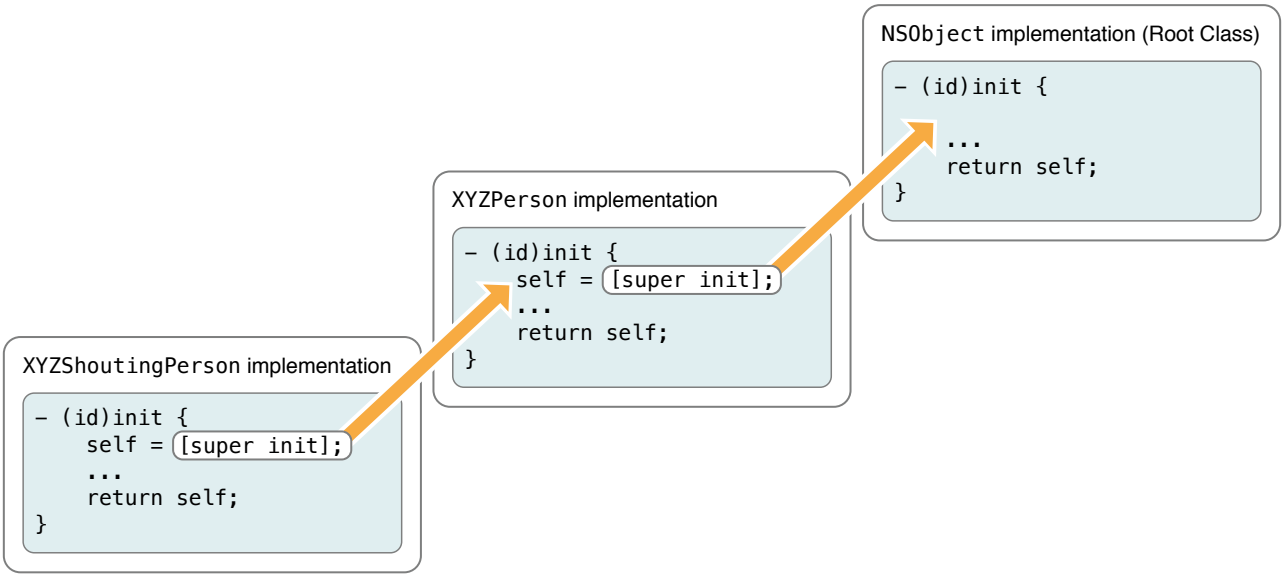
A typical `init` method looks like this:

```
- (id)init {
    self = [super init];

    if (self) {
        // initialize instance variables here
    }

    return self;
}
```

An `init` method should assign `self` to the result of calling the superclass's initialization method before doing its own initialization. A superclass may fail to initialize the object correctly and return `nil` so you should always check to make sure `self` is not `nil` before performing your own initialization.

By calling `[super init]` as the first line in the method, an object is initialized from its root class down through each subclass `init` implementation in order. Figure 3-1 (page 49) shows the process for initializing an `XYZShoutingPerson` object.

Figure 3-1 The initialization process



As you saw in the previous chapter, an object is initialized either by calling `init`, or by calling a method that initializes the object with specific values.

In the case of the `XYZPerson` class, it would make sense to offer an initialization method that set the person's initial first and last names:

```
- (id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName;
```

You'd implement the method like this:

```
- (id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName {
    self = [super init];

    if (self) {
        _firstName = aFirstName;
```

```
        _lastName = aLastName;
    }

    return self;
}
```

The Designated Initializer is the Primary Initialization Method

If an object declares one or more initialization methods, you should decide which method is the *designated initializer*. This is often the method that offers the most options for initialization (such as the method with the most arguments), and is called by other methods you write for convenience. You should also typically override `init` to call your designated initializer with suitable default values.

If an `XYZPerson` also had a property for a date of birth, the designated initializer might be:

```
- (id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName
                                            dateOfBirth:(NSDate *)aDOB;
```

This method would set the relevant instance variables, as shown above. If you still wished to provide a convenience initializer for just first and last names, you would implement the method to call the designated initializer, like this:

```
- (id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName {
    return [self initWithFirstName:aFirstName lastName:aLastName dateOfBirth:nil];
}
```

You might also implement a standard `init` method to provide suitable defaults:

```
- (id)init {
    return [self initWithFirstName:@"John" lastName:@"Doe" dateOfBirth:nil];
}
```

If you need to write an initialization method when subclassing a class that uses multiple `init` methods, you should either override the superclass's designated initializer to perform your own initialization, or add your own additional initializer. Either way, you should call the superclass's designated initializer (in place of `[super init];`) before doing any of your own initialization.

You Can Implement Custom Accessor Methods

Properties don't always have to be backed by their own instance variables.

As an example, the `XYZPerson` class might define a read-only property for a person's full name:

```
@property (readonly) NSString *fullName;
```

Rather than having to update the `fullName` property every time the first or last name changed, it would be easier just to write a custom accessor method to build the full name string on request:

```
- (NSString *)fullName {
    return [NSString stringWithFormat:@"%@ %@", self.firstName, self.lastName];
}
```

This simple example uses a format string and specifiers (as described in the previous chapter) to build a string containing a person's first and last names separated by a space.

> **Note:** Although this is a convenient example, it's important to realize that it's locale-specific, and is only suitable for use in countries that put a person's given name before the family name.

If you need to write a custom accessor method for a property that does use an instance variable, you must access that instance variable directly from within the method. For example, it's common to delay the initialization of a property until it's first requested, using a "lazy accessor," like this:

```
- (XYZObject *)someImportantObject {
    if (!_someImportantObject) {
        _someImportantObject = [[XYZObject alloc] init];
    }

    return _someImportantObject;
}
```

Before returning the value, this method first checks whether the `_someImportantObject` instance variable is `nil`; if it is, it allocates an object.

Note: The compiler will automatically synthesize an instance variable in all situations where it's also synthesizing at least one accessor method. If you implement both a getter and a setter for a `readwrite` property, or a getter for a `readonly` property, the compiler will assume that you are taking control over the property implementation and won't synthesize an instance variable automatically.

If you still need an instance variable, you'll need to request that one be synthesized:

```
@synthesize property = _property;
```

Properties Are Atomic by Default

By default, an Objective-C property is *atomic*:

```
@interface XYZObject : NSObject
@property NSObject *implicitAtomicObject;          // atomic by default
@property (atomic) NSObject *explicitAtomicObject; // explicitly marked atomic
@end
```

This means that the synthesized accessors ensure that a value is always fully retrieved by the getter method or fully set via the setter method, even if the accessors are called simultaneously from different threads.

Because the internal implementation and synchronization of atomic accessor methods is private, it's not possible to combine a synthesized accessor with an accessor method that you implement yourself. You'll get a compiler warning if you try, for example, to provide a custom setter for an `atomic`, `readwrite` property but leave the compiler to synthesize the getter.

You can use the `nonatomic` property attribute to specify that synthesized accessors simply set or return a value directly, with no guarantees about what happens if that same value is accessed simultaneously from different threads. For this reason, it's faster to access a `nonatomic` property than an `atomic` one, and it's fine to combine a synthesized setter, for example, with your own getter implementation:

```
@interface XYZObject : NSObject
@property (nonatomic) NSObject *nonatomicObject;
@end
```

```
@implementation XYZObject
```

```
- (NSObject *)nonatomicObject {
    return _nonatomicObject;
}
// setter will be synthesized automatically
@end
```

Note: Property atomicity is not synonymous with an object's *thread safety*.

Consider an XYZPerson object in which both a person's first and last names are changed using atomic accessors from one thread. If another thread accesses both names at the same time, the atomic getter methods will return complete strings (without crashing), but there's no guarantee that those values will be the right names relative to each other. If the first name is accessed before the change, but the last name is accessed after the change, you'll end up with an inconsistent, mismatched pair of names.

This example is quite simple, but the problem of thread safety becomes much more complex when considered across a network of related objects. Thread safety is covered in more detail in *Concurrency Programming Guide*.

Manage the Object Graph through Ownership and Responsibility

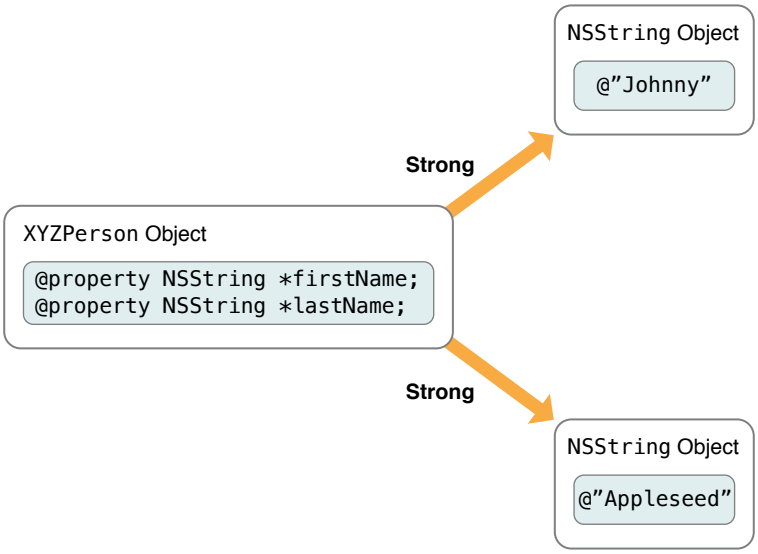
As you've already seen, memory for Objective-C objects is allocated dynamically (on the heap), which means you need to use pointers to keep track of an object's address. Unlike scalar values, it's not always possible to determine an object's lifetime by the scope of one pointer variable. Instead, an object must be kept active in memory for as long as it is needed by other objects.

Rather than trying to worry about managing the lifecycle of each object manually, you should instead think about the relationships between objects.

In the case of an `XYZPerson` object, for example, the two string properties for `firstName` and `lastName` are effectively "owned" by the `XYZPerson` instance. This means they should stay in memory as long as the `XYZPerson` object stays in memory.

When one object relies on other objects in this way, effectively taking ownership of those other objects, the first object is said to have *strong references* to the other objects. In Objective-C, an object is kept alive as long as it has at least one strong reference to it from another object. The relationships between the `XYZPerson` instance and the two `NSString` objects is shown in Figure 3-2 (page 54).

Figure 3-2 Strong Relationships



When an `XYZPerson` object is deallocated from memory, the two string objects will also be deallocated, assuming there aren't any other strong references left to them.

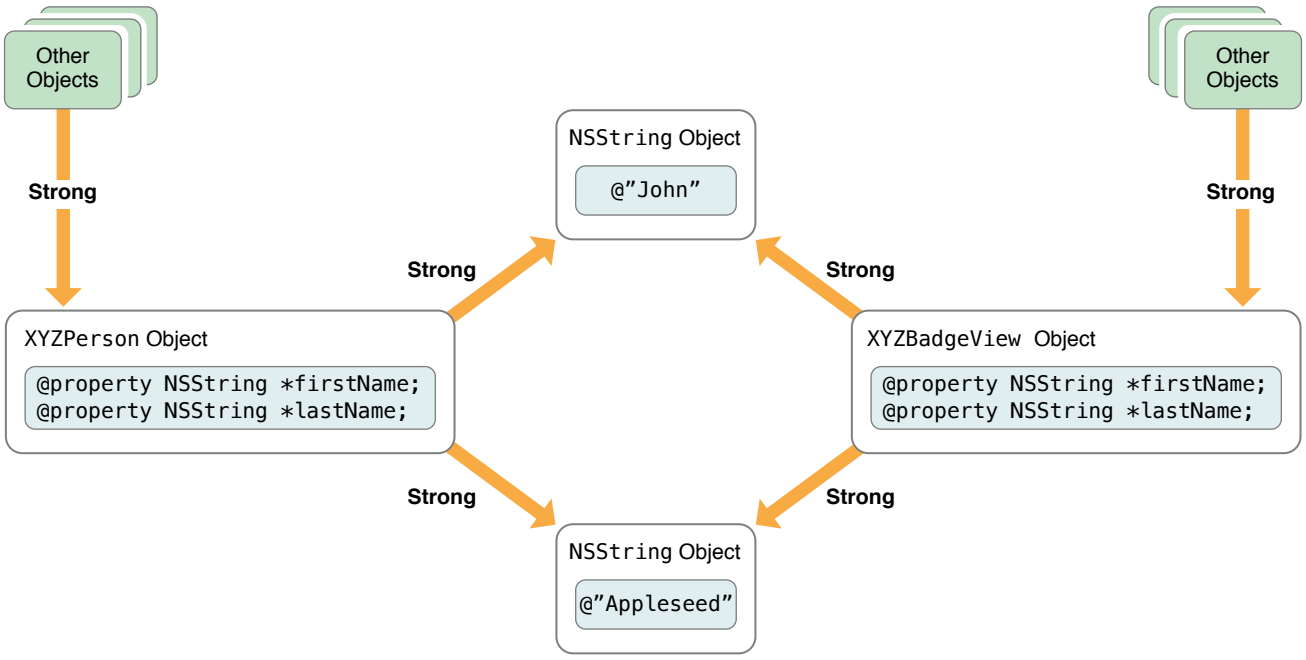
To add a little more complexity to this example, consider the object graph for an application like that shown in Figure 3-3 (page 55).

Figure 3-3 The Name Badge Maker application

When the user clicks the Update button, the badge preview is updated with the relevant name information.

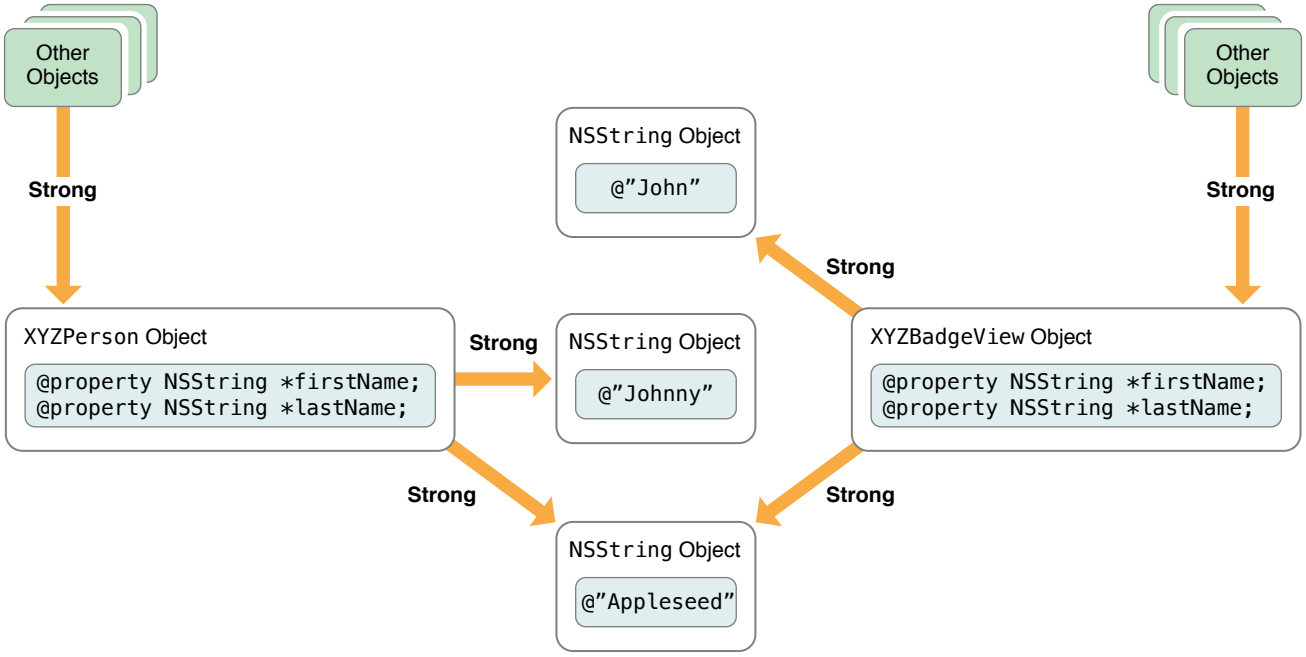
The first time a person's details are entered and the update button clicked, the simplified object graph might look like Figure 3-4 (page 56).

Figure 3-4 Simplified object graph for initial XYZPerson creation



When the user modifies the person's first name, the object graph changes to look like Figure 3-5 (page 57).

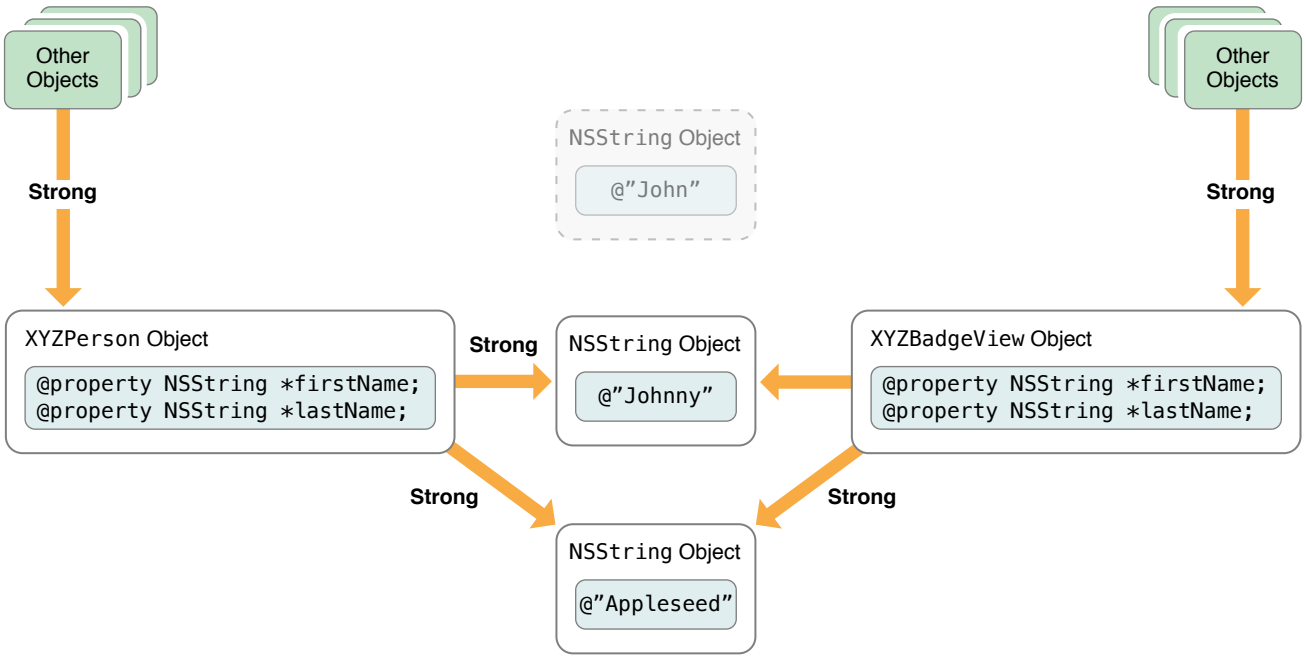
Figure 3-5 Simplified object graph while changing the person's first name



The badge display view maintains a strong relationship to the original `@"John"` string object, even though the `XYZPerson` object now has a different `firstName`. This means the `@"John"` object stays in memory, used by the badge view to print the name.

Once the user clicks the Update button a second time, the badge view is told to update its internal properties to match the person object, so the object graph looks like Figure 3-6 (page 58).

Figure 3-6 Simplified object graph after updating the badge view



At this point, the original `@"John"` object no longer has any strong references to it, so it is removed from memory.

By default, both Objective-C properties and variables maintain strong references to their objects. This is fine for many situations, but it does cause a potential problem with strong reference cycles.

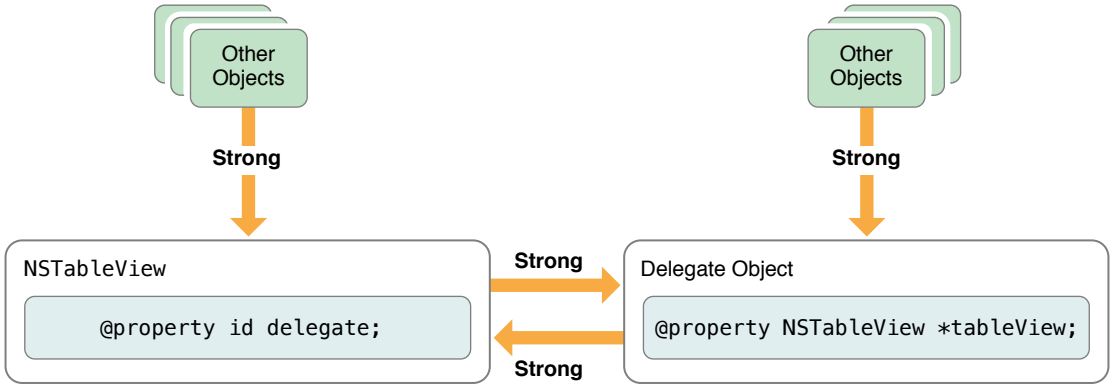
Avoid Strong Reference Cycles

Although strong references work well for one-way relationships between objects, you need to be careful when working with groups of interconnected objects. If a group of objects is connected by a circle of strong relationships, they keep each other alive even if there are no strong references from outside the group.

One obvious example of a potential reference cycle exists between a table view object (`UITableView` for iOS and `NSTableView` for OS X) and its delegate. In order for a generic table view class to be useful in multiple situations, it delegates some decisions to external objects. This means it relies on another object to decide what content it displays, or what to do if the user interacts with a specific entry in the table view.

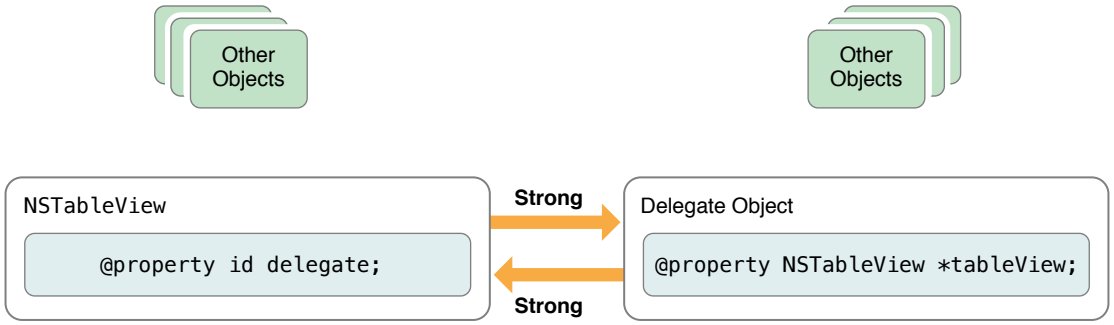
A common scenario is that the table view has a reference to its delegate and the delegate has a reference back to the table view, as shown in Figure 3-7 (page 59).

Figure 3-7 Strong references between a table view and its delegate



A problem occurs if the other objects give up their strong relationships to the table view and delegate, as shown in Figure 3-8 (page 59).

Figure 3-8 A strong reference cycle

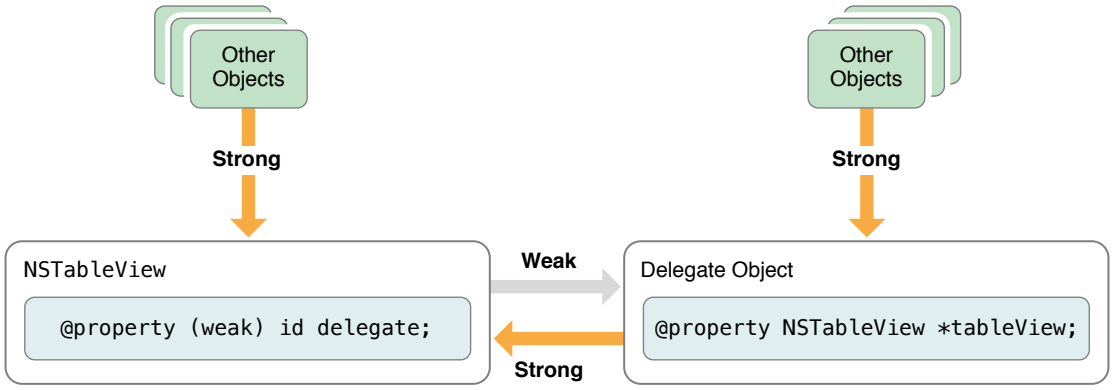


Even though there is no need for the objects to be kept in memory—there are no strong relationships to the table view or delegate other than the relationships between the two objects—the two remaining strong relationships keep the two objects alive. This is known as a *strong reference cycle*.

The way to solve this problem is to substitute one of the strong references for a *weak reference*. A weak reference does not imply ownership or responsibility between two objects, and does not keep an object alive.

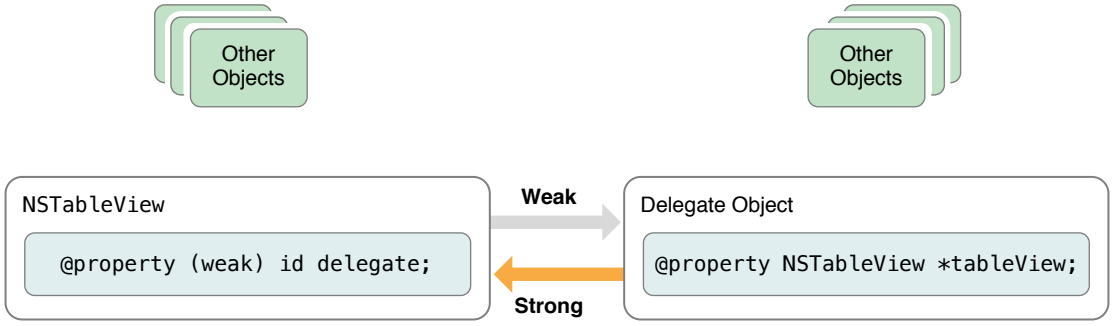
If the table view is modified to use a weak relationship to its delegate (which is how `UITableView` and `NSTableView` solve this problem), the initial object graph now looks like Figure 3-9 (page 60).

Figure 3-9 The correct relationship between a table view and its delegate



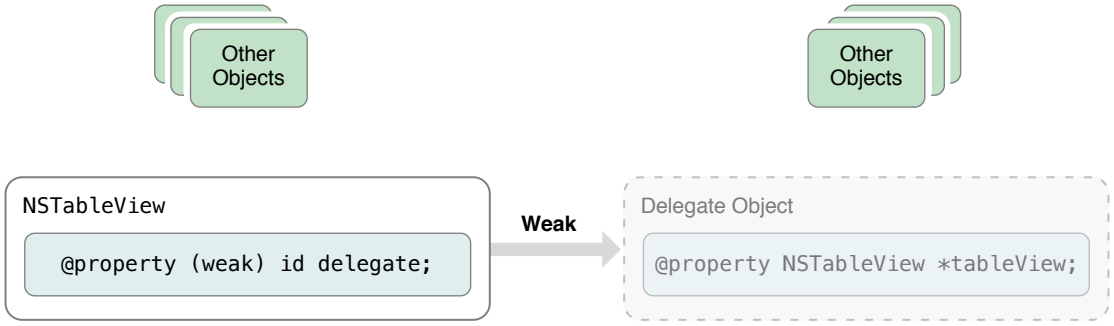
When the other objects in the graph give up their strong relationships to the table view and delegate this time, there are no strong references left to the delegate object, as shown in Figure 3-10 (page 60).

Figure 3-10 Avoiding a strong reference cycle



This means that the delegate object will be deallocated, thereby releasing the strong reference on the table view, as shown in Figure 3-11 (page 61).

Figure 3-11 Deallocating the delegate



Once the delegate is deallocated, there are no longer any strong references to the table view, so it too is deallocated.

Use Strong and Weak Declarations to Manage Ownership

By default, object properties declared like this:

```
@property id delegate;
```

use strong references for their synthesized instance variables. To declare a weak reference, add an attribute to the property, like this:

```
@property (weak) id delegate;
```

> **Note:** The opposite to `weak` is `strong`. There's no need to specify the `strong` attribute explicitly, because it is the default.

Local variables (and non-property instance variables) also maintain strong references to objects by default. This means that the following code will work exactly as you expect:

```
NSDate *originalDate = self.lastModificationDate;
self.lastModificationDate = [NSDate date];
NSLog(@"Last modification date changed from %@ to %@",
                    originalDate, self.lastModificationDate);
```

In this example, the local variable `originalDate` maintains a strong reference to the initial `lastModificationDate` object. When the `lastModificationDate` property is changed, the property no longer keeps a strong reference to the original date, but that date is still kept alive by the `originalDate` strong variable.

> **Note:** A variable maintains a strong reference to an object only as long as that variable is in scope, or until it is reassigned to another object or `nil`.

If you don't want a variable to maintain a strong reference, you can declare it as `__weak`, like this:

```
NSObject * __weak weakVariable;
```

Because a weak reference doesn't keep an object alive, it's possible for the referenced object to be deallocated while the reference is still in use. To avoid a dangerous dangling pointer to the memory originally occupied by the now deallocated object, a weak reference is automatically set to `nil` when its object is deallocated.

This means that if you use a weak variable in the previous date example:

```
NSDate * __weak originalDate = self.lastModificationDate;
self.lastModificationDate = [NSDate date];
```

the `originalDate` variable may potentially be set to `nil`. When `self.lastModificationDate` is reassigned, the property no longer maintains a strong reference to the original date. If there are no other strong references to it, the original date will be deallocated and `originalDate` set to `nil`.

Weak variables can be a source of confusion, particularly in code like this:

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

In this example, the newly allocated object has no strong references to it, so it is immediately deallocated and `someObject` is set to `nil`.

> **Note:** The opposite to `__weak` is `__strong`. Again, you don't need to specify `__strong` explicitly, because it is the default.

It's also important to consider the implications of a method that needs to access a weak property several times, like this:

```
- (void)someMethod {
    [self.weakProperty doSomething];
    ...
    [self.weakProperty doSomethingElse];
}
```

In situations like this, you might want to cache the weak property in a strong variable to ensure that it is kept in memory as long as you need to use it:

```
- (void)someMethod {
    NSObject *cachedObject = self.weakProperty;
    [cachedObject doSomething];
    ...
    [cachedObject doSomethingElse];
}
```

In this example, the `cachedObject` variable maintains a strong reference to the original weak property value so that it can't be deallocated as long as `cachedObject` is still in scope (and hasn't been reassigned another value).

It's particularly important to keep this in mind if you need to make sure a weak property is not `nil` before using it. It's not enough just to test it, like this:

```
    if (self.someWeakProperty) {
        [someObject doSomethingImportantWith:self.someWeakProperty];
    }
```

because in a multi-threaded application, the property may be deallocated between the test and the method call, rendering the test useless. Instead, you need to declare a strong local variable to cache the value, like this:

```
    NSObject *cachedObject = self.someWeakProperty;          // 1
    if (cachedObject) {                                      // 2
        [someObject doSomethingImportantWith:cachedObject];  // 3
    }                                                        // 4
    cachedObject = nil;                                      // 5
```

In this example, the strong reference is created in line 1, meaning that the object is guaranteed to be alive for the test and method call. In line 5, `cachedObject` is set to `nil`, thereby giving up the strong reference. If the original object has no other strong references to it at this point, it will be deallocated and `someWeakProperty` will be set to `nil`.

Use Unsafe Unretained References for Some Classes

There are a few classes in Cocoa and Cocoa Touch that don't yet support weak references, which means you can't declare a weak property or weak local variable to keep track of them. These classes include `NSTextView`, `NSFont` and `NSColorSpace`; for the full list, see *Transitioning to ARC Release Notes*.

If you need to use a weak reference to one of these classes, you must use an unsafe reference. For a property, this means using the `unsafe_unretained` attribute:

```
@property (unsafe_unretained) NSObject *unsafeProperty;
```

For variables, you need to use `__unsafe_unretained`:

```
    NSObject * __unsafe_unretained unsafeReference;
```

An unsafe reference is similar to a weak reference in that it doesn't keep its related object alive, but it won't be set to `nil` if the destination object is deallocated. This means that you'll be left with a dangling pointer to the memory originally occupied by the now deallocated object, hence the term "unsafe." Sending a message to a dangling pointer will result in a crash.

Copy Properties Maintain Their Own Copies

In some circumstances, an object may wish to keep its own copy of any objects that are set for its properties.

As an example, the class interface for the `XYZBadgeView` class shown earlier in Figure 3-4 (page 56) might look like this:

```
@interface XYZBadgeView : NSView
@property NSString *firstName;
@property NSString *lastName;
@end
```

Two `NSString` properties are declared, which both maintain implicit strong references to their objects.

Consider what happens if another object creates a string to set as one of the badge view's properties, like this:

```
NSMutableString *nameString = [NSMutableString stringWithString:@"John"];
self.badgeView.firstName = nameString;
```

This is perfectly valid, because `NSMutableString` is a subclass of `NSString`. Although the badge view thinks it's dealing with an `NSString` instance, it's actually dealing with an `NSMutableString`.

This means that the string can change:

```
[nameString appendString:@"ny"];
```

In this case, although the name was "John" at the time it was originally set for the badge view's `firstName` property, it's now "Johnny" because the mutable string was changed.

You might choose that the badge view should maintain its own copies of any strings set for its `firstName` and `lastName` properties, so that it effectively captures the strings at the time that the properties are set. By adding a `copy` attribute to the two property declarations:

```
@interface XYZBadgeView : NSView
@property (copy) NSString *firstName;
@property (copy) NSString *lastName;
@end
```

the view now maintains its own copies of the two strings. Even if a mutable string is set and subsequently changed, the badge view captures whatever value it has at the time it is set. For example:

```
NSMutableString *nameString = [NSMutableString stringWithString:@"John"];
self.badgeView.firstName = nameString;
[nameString appendString:@"ny"];
```

This time, the `firstName` held by the badge view will be an unaffected copy of the original "John" string.

The `copy` attribute means that the property will use a strong reference, because it must hold on to the new object it creates.

> **Note:** Any object that you wish to set for a `copy` property must support `NSCopying`, which means that it should conform to the `NSCopying` protocol.
>
> Protocols are described in Protocols Define Messaging Contracts (page 77). For more information on `NSCopying`, see `NSCopying` or the *Advanced Memory Management Programming Guide*.

If you need to set a `copy` property's instance variable directly, for example in an initializer method, don't forget to set a copy of the original object:

```
- (id)initWithSomeOriginalString:(NSString *)aString {
    self = [super init];
    if (self) {
        _instanceVariableForCopyProperty = [aString copy];
    }
    return self;
}
```

Exercises

1. Modify the `sayHello` method from the `XYZPerson` class to log a greeting using the person's first name and last name.
2. Declare and implement a new designated initializer used to create an `XYZPerson` using a specified first name, last name and date of birth, along with a suitable class factory method.

   Don't forget to override `init` to call the designated initializer.
3. Test what happens if you set a mutable string as the person's first name, then mutate that string before calling your modified `sayHello` method. Change the `NSString` property declarations by adding the `copy` attribute and test again.
4. Try creating `XYZPerson` objects using a variety of strong and weak variables in the `main()` function. Verify that the strong variables keep the `XYZPerson` objects alive at least as long as you expect.

   In order to help verify when an `XYZPerson` object is deallocated, you might want to tie into the object lifecycle by providing a `dealloc` method in the `XYZPerson` implementation. This method is called automatically when an Objective-C object is deallocated from memory, and is normally used to release any memory you allocated manually, such as through the C `malloc()` function, as described in *Advanced Memory Management Programming Guide*.

   For the purposes of this exercise, override the `dealloc` method in `XYZPerson` to log a message, like this:

```
- (void)dealloc {
    NSLog(@"XYZPerson is being deallocated");
}
```

Try setting each `XYZPerson` pointer variable to `nil` to verify that the objects are deallocated when you expect them to be.

> **Note:** The Xcode project template for a Command Line Tool use an `@autoreleasepool { }` block inside the `main()` function. In order to use the Automatic Retain Count feature of the compiler to handle memory management for you, it's important that any code you write in `main()` goes inside this autorelease pool block.
>
> Autorelease pools are outside the scope of this document, but are covered in detail in *Advanced Memory Management Programming Guide*.
>
> When you're writing a Cocoa or Cocoa Touch application rather than a command line tool, you won't usually need to worry about creating your own autorelease pools, because you're tying into a framework of objects that will ensure one is already in place.

5. Modify the `XYZPerson` class description so that you can keep track of a spouse or partner.

   You'll need to decide how best to model the relationship, thinking carefully about object graph management.

Customizing Existing Classes

Objects should have clearly-defined tasks, such as modeling specific information, displaying visual content or controlling the flow of information. As you've already seen, a class interface defines the ways in which others are expected to interact with an object to help it accomplish those tasks.

Sometimes, you may find that you wish to extend an existing class by adding behavior that is useful only in certain situations. As an example, you might find that your application often needs to display a string of characters in a visual interface. Rather than creating some string-drawing object to use every time you need to display a string, it would make more sense if it was possible to give the `NSString` class itself the ability to draw its own characters on screen.

In situations like this, it doesn't always make sense to add the utility behavior to the original, primary class interface. Drawing abilities are unlikely to be needed the majority of times any string object is used in an application, for example, and in the case of `NSString`, you can't modify the original interface or implementation because it's a framework class.

Furthermore, it might not make sense to subclass the existing class, because you may want your drawing behavior available not only to the original `NSString` class but also any subclasses of that class, like `NSMutableString`. And, although `NSString` is available on both OS X and iOS, the drawing code would need to be different for each platform, so you'd need to use a different subclass on each platform.

Instead, Objective-C allows you to add your own methods to existing classes through categories and class extensions.

Categories Add Methods to Existing Classes

If you need to add a method to an existing class, perhaps to add functionality to make it easier to do something in your own application, the easiest way is to use a category.

The syntax to declare a *category* uses the `@interface` keyword, just like a standard Objective-C class description, but does not indicate any inheritance from a subclass. Instead, it specifies the name of the category in parentheses, like this:

```
@interface ClassName (CategoryName)
```

```
@end
```

A category can be declared for any class, even if you don't have the original implementation source code (such as for standard Cocoa or Cocoa Touch classes). Any methods that you declare in a category will be available to all instances of the original class, as well as any subclasses of the original class. At runtime, there's no difference between a method added by a category and one that is implemented by the original class.

Consider the `XYZPerson` class from the previous chapters, which has properties for a person's first and last name. If you're writing a record-keeping application, you may find that you frequently need to display a list of people by last name, like this:

```
Appleseed, John
Doe, Jane
Smith, Bob
Warwick, Kate
```

Rather than having to write code to generate a suitable `lastName, firstName` string each time you wanted to display it, you could add a category to the `XYZPerson` class, like this:

```
#import "XYZPerson.h"

@interface XYZPerson (XYZPersonNameDisplayAdditions)
- (NSString *)lastNameFirstNameString;
@end
```

In this example, the `XYZPersonNameDisplayAdditions` category declares one additional method to return the necessary string.

A category is usually declared in a separate header file and implemented in a separate source code file. In the case of `XYZPerson`, you might declare the category in a header file called `XYZPerson+XYZPersonNameDisplayAdditions.h`.

Even though any methods added by a category are available to all instances of the class and its subclasses, you'll need to import the category header file in any source code file where you wish to use the additional methods, otherwise you'll run into compiler warnings and errors.

The category implementation might look like this:

```
#import "XYZPerson+XYZPersonNameDisplayAdditions.h"
```

```
@implementation XYZPerson (XYZPersonNameDisplayAdditions)
- (NSString *)lastNameFirstNameString {
    return [NSString stringWithFormat:@"%@, %@", self.lastName, self.firstName];
}
@end
```

Once you've declared a category and implemented the methods, you can use those methods from any instance of the class, as if they were part of the original class interface:

```
#import "XYZPerson+XYZPersonNameDisplayAdditions.h"
@implementation SomeObject
- (void)someMethod {
    XYZPerson *person = [[XYZPerson alloc] initWithFirstName:@"John"
                                                    lastName:@"Doe"];
    XYZShoutingPerson *shoutingPerson =
                        [[XYZShoutingPerson alloc] initWithFirstName:@"Monica"
                                                            lastName:@"Robinson"];

    NSLog(@"The two people are %@ and %@",
        [person lastNameFirstNameString], [shoutingPerson lastNameFirstNameString]);
}
@end
```

As well as just adding methods to existing classes, you can also use categories to split the implementation of a complex class across multiple source code files. You might, for example, put the drawing code for a custom user interface element in a separate file to the rest of the implementation if the geometrical calculations, colors, and gradients, etc, are particularly complicated. Alternatively, you could provide different implementations for the category methods, depending on whether you were writing an app for OS X or iOS.

Categories can be used to declare either instance methods or class methods but are not usually suitable for declaring additional properties. It's valid syntax to include a property declaration in a category interface, but it's not possible to declare an additional instance variable in a category. This means the compiler won't synthesize any instance variable, nor will it synthesize any property accessor methods. You can write your own accessor methods in the category implementation, but you won't be able to keep track of a value for that property unless it's already stored by the original class.

The only way to add a traditional property—backed by a new instance variable—to an existing class is to use a class extension, as described in Class Extensions Extend the Internal Implementation (page 72).

> **Note:** Cocoa and Cocoa Touch include a variety of categories for some of the primary framework classes.
>
> The string-drawing functionality mentioned in the introduction to this chapter is in fact already provided for `NSString` by the `NSStringDrawing` category for OS X, which includes the `drawAtPoint:withAttributes:` and `drawInRect:withAttributes:` methods. For iOS, the `UIStringDrawing` category includes methods such as `drawAtPoint:withFont:` and `drawInRect:withFont:`.

Avoid Category Method Name Clashes

Because the methods declared in a category are added to an existing class, you need to be very careful about method names.

If the name of a method declared in a category is the same as a method in the original class, or a method in another category on the same class (or even a superclass), the behavior is undefined as to which method implementation is used at runtime. This is less likely to be an issue if you're using categories with your own classes, but can cause problems when using categories to add methods to standard Cocoa or Cocoa Touch classes.

An application that works with a remote web service, for example, might need an easy way to encode a string of characters using Base64 encoding. It would make sense to define a category on `NSString` to add an instance method to return a Base64-encoded version of a string, so you might add a convenience method called `base64EncodedString`.

A problem arises if you link to another framework that also happens to define its own category on `NSString`, including its own method called `base64EncodedString`. At runtime, only one of the method implementations will "win" and be added to `NSString`, but which one is undefined.

Another problem can arise if you add convenience methods to Cocoa or Cocoa Touch classes that are then added to the original classes in later releases. The `NSSortDescriptor` class, for example, which describes how a collection of objects should be ordered, has always had an `initWithKey:ascending:` initialization method, but didn't offer a corresponding class factory method under early OS X and iOS versions.

By convention, the class factory method should be called `sortDescriptorWithKey:ascending:`, so you might have chosen to add a category on `NSSortDescriptor` to provide this method for convenience. This would have worked as you'd expect under older versions of OS X and iOS, but with the release of Mac OS X

version 10.6 and iOS 4.0, a `sortDescriptorWithKey:ascending:` method was added to the original `NSSortDescriptor` class, meaning you'd now end up with a naming clash when your application was run on these or later platforms.

In order to avoid undefined behavior, it's best practice to add a prefix to method names in categories on framework classes, just like you should add a prefix to the names of your own *classes*. You might choose to use the same three letters you use for your class prefixes, but lowercase to follow the usual convention for method names, then an underscore, before the rest of the method name. For the `NSSortDescriptor` example, your own category might look like this:

```
@interface NSSortDescriptor (XYZAdditions)
+ (id)xyz_sortDescriptorWithKey:(NSString *)key ascending:(BOOL)ascending;
@end
```

This means you can be sure that your method will be used at runtime. The ambiguity is removed because your code now looks like this:

```
    NSSortDescriptor *descriptor =
              [NSSortDescriptor xyz_sortDescriptorWithKey:@"name" ascending:YES];
```

Class Extensions Extend the Internal Implementation

A class extension bears some similarity to a category, but it can only be added to a class for which you have the source code at compile time (the class is compiled at the same time as the class extension). The methods declared by a class extension are implemented in the `@implementation` block for the original class so you can't, for example, declare a class extension on a framework class, such as a Cocoa or Cocoa Touch class like `NSString`.

The syntax to declare a class extension is similar to the syntax for a category, and looks like this:

```
@interface ClassName ()


@end
```

Because no name is given in the parentheses, class extensions are often referred to as *anonymous categories*.

Unlike regular categories, a class extension can add its own properties and instance variables to a class. If you declare a property in a class extension, like this:

```
@interface XYZPerson ()
@property NSObject *extraProperty;
@end
```

the compiler will automatically synthesize the relevant accessor methods, as well as an instance variable, inside the primary class implementation.

If you add any *methods* in a class extension, these must be implemented in the primary implementation for the class.

It's also possible to use a class extension to add custom instance variables. These are declared inside braces in the class extension interface:

```
@interface XYZPerson () {
    id _someCustomInstanceVariable;
}
...
@end
```

Use Class Extensions to Hide Private Information

The primary interface for a class is used to define the way that other classes are expected to interact with it. In other words, it's the *public* interface to the class.

Class extensions are often used to extend the public interface with additional *private* methods or properties for use within the implementation of the class itself. It's common, for example, to define a property as `readonly` in the interface, but as `readwrite` in a class extension declared above the implementation, in order that the internal methods of the class can change the property value directly.

As an example, the `XYZPerson` class might add a property called `uniqueIdentifier`, designed to keep track of information like a Social Security Number in the US.

It usually requires a large amount of paperwork to have a unique identifier assigned to an individual in the real world, so the `XYZPerson` class interface might declare this property as `readonly`, and provide some method that requests an identifier be assigned, like this:

```
@interface XYZPerson : NSObject
...
@property (readonly) NSString *uniqueIdentifier;
```

```
- (void)assignUniqueIdentifier;
@end
```

This means that it's not possible for the `uniqueIdentifier` to be set directly by another object. If a person doesn't already have one, a request must be made to assign an identifier by calling the `assignUniqueIdentifier` method.

In order for the `XYZPerson` class to be able to change the property internally, it makes sense to redeclare the property in a class extension that's defined at the top of the implementation file for the class:

```
@interface XYZPerson ()
@property (readwrite) NSString *uniqueIdentifier;
@end

@implementation XYZPerson
...
@end
```

> **Note:** The `readwrite` attribute is optional, because it's the default. You may like to use it when redeclaring a property, for clarity.

This means that the compiler will now also synthesize a setter method, so any method inside the `XYZPerson` implementation will be able to set the property directly using either the setter or dot syntax.

By declaring the class extension inside the source code file for the `XYZPerson` implementation, the information stays private to the `XYZPerson` class. If another type of object tries to set the property, the compiler will generate an error.

Note: By adding the class extension shown above, redeclaring the `uniqueIdentifier` property as a `readwrite` property, a `setUniqueIdentifier:` method will exist at runtime on every `XYZPerson` object, regardless of whether other source code files were aware of the class extension or not.

The compiler will complain if code in one of those other source code files attempts to call a private method or set a `readonly` property, but it's possible to avoid compiler errors and leverage dynamic runtime features to call those methods in other ways, such as by using one of the `performSelector:...` methods offered by `NSObject`. You should avoid a class hierarchy or design where this is necessary; instead, the primary class interface should always define the correct "public" interactions.

If you intend to make "private" methods or properties available to select other classes, such as related classes within a framework, you can declare the class extension in a separate header file and import it in the source files that need it. It's not uncommon to have two header files for a class, for example, such as `XYZPerson.h` and `XYZPersonPrivate.h`. When you release the framework, you only release the public `XYZPerson.h` header file.

Consider Other Alternatives for Class Customization

Categories and class extensions make it easy to add behavior directly to an existing class, but sometimes this isn't the best option.

One of the primary goals of object-oriented programming is to write reusable code, which means that classes should be reusable in a variety of situations, wherever possible. If you're creating a view class to describe an object that displays information on screen, for example, it's a good idea to think whether the class could be usable in multiple situations.

Rather than hard-coding decisions about layout or content, one alternative is to leverage inheritance and leave those decisions in methods specifically designed to be overridden by subclasses. Although this does make it relatively easy to reuse the class, you still need to create a new subclass every time you want to make use of that original class.

Another alternative is for a class to use a *delegate* object. Any decisions that might limit reusability can be delegated to another object, which is left to make those decisions at runtime. One common example is a standard table view class (`NSTableView` for OS X and `UITableView` for iOS). In order for a generic table view (an object that displays information using one or more columns and rows) to be useful, it leaves decisions about its content to be decided by another object at runtime. Delegation is covered in detail in the next chapter, Working with Protocols (page 77).

Interact Directly with the Objective-C Runtime

Objective-C offers its dynamic behavior through the Objective-C runtime system.

Many decisions, such as which methods are called when messages are sent, aren't made at compile-time but are instead determined when the application is run. Objective-C is more than just a language that is compiled down to machine code. Instead, it requires a runtime system in place to execute that code.

It's possible to interact directly with this runtime system, such as by adding *associative references* to an object. Unlike class extensions, associated references do not affect the original class declaration and implementation, which means you can use them with framework classes for which you don't have access to the original source code.

An associative reference links one object with another, in a similar way to a property or instance variable. For more information, see Associative References. To learn more about the Objective-C Runtime in general, see *Objective-C Runtime Programming Guide*.

Exercises

1. Add a category to the `XYZPerson` class to declare and implement additional behavior, such as displaying a person's name in different ways.
2. Add a category to `NSString` in order to add a method to draw the uppercase version of a string at a given point, calling through to one of the existing `NSStringDrawing` category methods to perform the actual drawing. These methods are documented in *NSString UIKit Additions Reference* for iOS and *NSString Application Kit Additions Reference* for OS X.
3. Add two `readonly` properties to the original `XYZPerson` class implementation to represent a person's height and weight, along with methods to `measureWeight` and `measureHeight`.

   Use a class extension to redeclare the properties as `readwrite`, and implement the methods to set the properties to suitable values.

Working with Protocols

In the real world, people on official business are often required to follow strict procedures when dealing with certain situations. Law enforcement officials, for example, are required to "follow protocol" when making enquiries or collecting evidence.

In the world of object-oriented programming, it's important to be able to define a set of behavior that is expected of an object in a given situation. As an example, a table view expects to be able to communicate with a data source object in order to find out what it is required to display. This means that the data source must respond to a specific set of messages that the table view might send.

The data source could be an instance of any class, such as a view controller (a subclass of `NSViewController` on OS X or `UIViewController` on iOS) or a dedicated data source class that perhaps just inherits from `NSObject`. In order for the table view to know whether an object is suitable as a data source, it's important to be able to declare that the object implements the necessary methods.

Objective-C allows you to define *protocols*, which declare the methods expected to be used for a particular situation. This chapter describes the syntax to define a formal protocol, and explains how to mark a class interface as *conforming* to a protocol, which means that the class must implement the required methods.

Protocols Define Messaging Contracts

A class interface declares the methods and properties associated with that class. A protocol, by contrast, is used to declare methods and properties that are independent of any specific class.

The basic syntax to define a protocol looks like this:

```
@protocol ProtocolName
// list of methods and properties
@end
```

Protocols can include declarations for both instance methods and class methods, as well as properties.

As an example, consider a custom view class that is used to display a pie chart, as shown in Figure 5-1 (page 78).

Figure 5-1 A Custom Pie Chart View

To make the view as reusable as possible, all decisions about the information should be left to another object, a data source. This means that multiple instances of the same view class could display different information just by communicating with different sources.

The minimum information needed by the pie chart view includes the number of segments, the relative size of each segment, and the title of each segment. The pie chart's data source protocol, therefore, might look like this:

```
@protocol XYZPieChartViewDataSource
- (NSUInteger)numberOfSegments;
- (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
```

```
- (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
@end
```

> **Note:** This protocol uses the `NSUInteger` value for unsigned integer scalar values. This type is discussed in more detail in the next chapter.

The pie chart view class interface would need a property to keep track of the data source object. This object could be of any class, so the basic property type will be `id`. The only thing that is known about the object is that it conforms to the relevant protocol.

The syntax to declare the data source property for the view would look like this:

```
@interface XYZPieChartView : UIView
@property (weak) id <XYZPieChartViewDataSource> dataSource;
...
@end
```

Objective-C uses angle brackets to indicate conformance to a protocol. This example declares a weak property for a generic object pointer that conforms to the `XYZPieChartViewDataSource` protocol.

> **Note:** Delegate and data source properties are usually marked as `weak` for the object graph management reasons described earlier, in Avoid Strong Reference Cycles (page 58).

By specifying the required protocol conformance on the property, you'll get a compiler warning if you attempt to set the property to an object that doesn't conform to the protocol, even though the basic property class type is generic. It doesn't matter whether the object is an instance of `UIViewController` or `NSObject`. All that matters is that it conforms to the protocol, which means the pie chart view knows it can request the information it needs.

Protocols Can Have Optional Methods

By default, all methods declared in a protocol are required methods. This means that any class that conforms to the protocol must implement those methods.

It's also possible to specify *optional* methods in a protocol. These are methods that a class can implement only if it needs to.

As an example, you might decide that the titles on the pie chart should be optional. If the data source object doesn't implement the `titleForSegmentAtIndex:`, no titles should be shown in the view.

You can mark protocol methods as optional using the `@optional` directive, like this:

```
@protocol XYZPieChartViewDataSource
- (NSUInteger)numberOfSegments;
- (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
@optional
- (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
@end
```

In this case, only the `titleForSegmentAtIndex:` method is marked optional. The previous methods have no directive, so are assumed to be required.

The `@optional` directive applies to any methods that follow it, either until the end of the protocol definition, or until another directive is encountered, such as `@required`. You might add further methods to the protocol like this:

```
@protocol XYZPieChartViewDataSource
- (NSUInteger)numberOfSegments;
- (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
@optional
- (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
- (BOOL)shouldExplodeSegmentAtIndex:(NSUInteger)segmentIndex;
@required
- (UIColor *)colorForSegmentAtIndex:(NSUInteger)segmentIndex;
@end
```

This example defines a protocol with three required methods and two optional methods.

Check that Optional Methods Are Implemented at Runtime

If a method in a protocol is marked as optional, you must check whether an object implements that method before attempting to call it.

As an example, the pie chart view might test for the segment title method like this:

```
    NSString *thisSegmentTitle;
    if ([self.dataSource respondsToSelector:@selector(titleForSegmentAtIndex:)])
{
        thisSegmentTitle = [self.dataSource titleForSegmentAtIndex:index];
    }
```

The `respondsToSelector:` method uses a selector, which refers to the identifier for a method after compilation. You can provide the correct identifier by using the `@selector()` directive and specifying the name of the method.

If the data source in this example implements the method, the title is used; otherwise, the title remains `nil`.

> **Remember:** Local *object* variables are automatically initialized to `nil`.

If you attempt to call the `respondsToSelector:` method on an `id` conforming to the protocol as it's defined above, you'll get a compiler error that there's no known instance method for it. Once you qualify an `id` with a protocol, all static type-checking comes back; you'll get an error if you try to call any method that isn't defined in the specified protocol. One way to avoid the compiler error is to set the custom protocol to adopt the `NSObject` protocol.

Protocols Inherit from Other Protocols

In the same way that an Objective-C class can inherit from a superclass, you can also specify that one protocol conforms to another.

As an example, it's best practice to define your protocols to conform to the `NSObject` protocol (some of the `NSObject` behavior is split from its class interface into a separate protocol; the `NSObject` class adopts the `NSObject` protocol).

By indicating that your own protocol conforms to the `NSObject` protocol, you're indicating that any object that adopts the custom protocol will also provide implementations for each of the `NSObject` protocol methods. Because you're presumably using some subclass of `NSObject`, you don't need to worry about providing your own implementations for these `NSObject` methods. The protocol adoption is useful, however, for situations like that described above.

To specify that one protocol conforms to another, you provide the name of the other protocol in angle brackets, like this:

```
@protocol MyProtocol <NSObject>
```

```
...
@end
```

In this example, any object that adopts `MyProtocol` also effectively adopts all the methods declared in the `NSObject` protocol.

Conforming to Protocols

The syntax to indicate that a class adopts a protocol again uses angle brackets, like this

```
@interface MyClass : NSObject <MyProtocol>
...
@end
```

This means that any instance of `MyClass` will respond not only to the methods declared specifically in the interface, but that `MyClass` also provides implementations for the required methods in `MyProtocol`. There's no need to redeclare the protocol methods in the class interface—the adoption of the protocol is sufficient.

> **Note:** The compiler does not automatically synthesize properties declared in adopted protocols.

If you need a class to adopt multiple protocols, you can specify them as a comma-separated list, like this:

```
@interface MyClass : NSObject <MyProtocol, AnotherProtocol, YetAnotherProtocol>
...
@end
```

> **Tip:** If you find yourself adopting a large number of protocols in a class, it may be a sign that you need to refactor an overly-complex class by splitting the necessary behavior across multiple smaller classes, each with clearly-defined responsibilities.
>
> One relatively common pitfall for new OS X and iOS developers is to use a single application delegate class to contain the majority of an application's functionality (managing underlying data structures, serving the data to multiple user interface elements, as well as responding to gestures and other user interaction). As complexity increases, the class becomes more difficult to maintain.

Once you've indicated conformance to a protocol, the class must at least provide method implementations for each of the required protocol methods, as well as any optional methods you choose. The compiler will warn you if you fail to implement any of the required methods.

> **Note:** The method declaration in a protocol is just like any other declaration. The method name and argument types in the implementation must match the declaration in the protocol.

Cocoa and Cocoa Touch Define a Large Number of Protocols

Protocols are used by Cocoa and Cocoa Touch objects for a variety of different situations. For example, the table view classes (`NSTableView` for OS X and `UITableView` for iOS) both use a data source object to supply them with the necessary information. Both define their own data source protocol, which is used in much the same way as the `XYZPieChartViewDataSource` protocol example above. Both table view classes also allow you to set a delegate object, which again must conform to the relevant `NSTableViewDelegate` or `UITableViewDelegate` protocol. The delegate is responsible for dealing with user interactions, or customizing the display of certain entries.

Some protocols are used to indicate *non-hierarchical similarities* between classes. Rather than being linked to specific class requirements, some protocols instead relate to more general Cocoa or Cocoa Touch communication mechanisms that may be adopted by multiple, unrelated classes.

For example, many framework model objects (such as the collection classes like `NSArray` and `NSDictionary`) support the `NSCoding` protocol, which means they can encode and decode their properties for archival or distribution as raw data. `NSCoding` makes it relatively easy to write entire object graphs to disk, provided every object within the graph adopts the protocol.

A few Objective-C language-level features also rely on protocols. In order to use fast enumeration, for example, a collection must adopt the `NSFastEnumeration` protocol, as described in Fast Enumeration Makes It Easy to Enumerate a Collection (page 101). Additionally, some objects can be copied, such as when using a property with a `copy` attribute as described in Copy Properties Maintain Their Own Copies (page 64). Any object you try to copy must adopt the `NSCopying` protocol, otherwise you'll get a runtime exception.

Protocols Are Used for Anonymity

Protocols are also useful in situations where the class of an object isn't known, or needs to stay hidden.

As an example, the developer of a framework may choose not to publish the interface for one of the classes within the framework. Because the class name isn't known, it's not possible for a *user* of the framework to create an instance of that class directly. Instead, some other object in the framework would typically be designated to return a ready-made instance, like this:

```
id utility = [frameworkObject anonymousUtility];
```

In order for this `anonymousUtility` object to be useful, the developer of the framework can publish a protocol that reveals some of its methods. Even though the original class interface isn't provided, which means the class stays anonymous, the object can still be used in a limited way:

```
id <XYZFrameworkUtility> utility = [frameworkObject anonymousUtility];
```

If you're writing an iOS app that uses the Core Data framework, for example, you'll likely run into the `NSFetchedResultsController` class. This class is designed to help a data source object supply stored data to an iOS `UITableView`, making it easy to provide information like the number of rows.

If you're working with a table view whose content is split into multiple sections, you can also ask a fetched results controller for the relevant section information. Rather than returning a specific class containing this section information, the `NSFetchedResultsController` class instead returns an anonymous object, which conforms to the `NSFetchedResultsSectionInfo` protocol. This means it's still possible to query the object for the information you need, such as the number of rows in a section:

```
NSInteger sectionNumber = ...
id <NSFetchedResultsSectionInfo> sectionInfo =
        [self.fetchedResultsController.sections objectAtIndex:sectionNumber];
NSInteger numberOfRowsInSection = [sectionInfo numberOfObjects];
```

Even though you don't know the class of the `sectionInfo` object, the `NSFetchedResultsSectionInfo` protocol dictates that it can respond to the `numberOfObjects` message.

Values and Collections

Objective-CSwift

Although Objective-C is an object-oriented programming language, it is a superset of C, which means you can use any of the standard C *scalar* (non-object) types like `int`, `float` and `char` in Objective-C code. There are also additional scalar types available in Cocoa and Cocoa Touch applications, such as `NSInteger`, `NSUInteger` and `CGFloat`, which have different definitions depending on the target architecture.

Scalar types are used in situations where you just don't need the benefits (or associated overheads) of using an object to represent a value. While strings of characters are usually represented as instances of the `NSString` class, numeric values are often stored in scalar local variables or properties.

It's possible to declare a C-style array in Objective-C, but you'll find that collections in Cocoa and Cocoa Touch applications are usually represented using instances of classes like `NSArray` or `NSDictionary`. These classes can only be used to collect Objective-C objects, which means that you'll need to create instances of classes like `NSValue`, `NSNumber` or `NSString` in order to represent values before you can add them to a collection.

The previous chapters in this guide make frequent use of the `NSString` class and its initialization and class factory methods, as well as the Objective-C `@"string"` literal, which offers a concise syntax to create an `NSString` instance. This chapter explains how to create `NSValue` and `NSNumber` objects, using either method calls or through Objective-C value literal syntax.

Basic C Primitive Types Are Available in Objective-C

Each of the standard C scalar variable types is available in Objective-C:

```
int someInteger = 42;
float someFloatingPointNumber = 3.1415;
double someDoublePrecisionFloatingPointNumber = 6.02214199e23;
```

as well as the standard C operators:

```
int someInteger = 42;
someInteger++;            // someInteger == 43
```

```
    int anotherInteger = 64;
    anotherInteger--;         // anotherInteger == 63

    anotherInteger *= 2;      // anotherInteger == 126
```

If you use a scalar type for an Objective-C property, like this:

```
@interface XYZCalculator : NSObject
@property double currentValue;
@end
```

it's also possible to use C operators on the property when accessing the value via dot syntax, like this:

```
@implementation XYZCalculator
- (void)increment {
    self.currentValue++;
}
- (void)decrement {
    self.currentValue--;
}
- (void)multiplyBy:(double)factor {
    self.currentValue *= factor;
}
@end
```

Dot syntax is purely a syntactic wrapper around accessor method calls, so each of the operations in this example is equivalent to first using the get accessor method to get the value, then performing the operation, then using the set accessor method to set the value to the result.

Objective-C Defines Additional Primitive Types

The `BOOL` scalar type is defined in Objective-C to hold a Boolean value, which is either `YES` or `NO`. As you might expect, `YES` is logically equivalent to `true` and `1`, while `NO` is equivalent to `false` and `0`.

Many parameters to methods on Cocoa and Cocoa Touch objects also use special scalar numeric types, such as `NSInteger` or `CGFloat`.

For example, the `NSTableViewDataSource` and `UITableViewDataSource` protocols (described in the previous chapter) both have methods requesting the number of rows to display:

```
@protocol NSTableViewDataSource <NSObject>
- (NSInteger)numberOfRowsInTableView:(NSTableView *)tableView;
...
@end
```

These types, like `NSInteger` and `NSUInteger`, are defined differently depending on the target architecture. When building for a 32-bit environment (such as for iOS), they are 32-bit signed and unsigned integers respectively; when building for a 64-bit environment (such as for the modern OS X runtime) they are 64-bit signed and unsigned integers respectively.

It's best practice to use these platform-specific types if you might be passing values across API boundaries (both internal and exported APIs), such as arguments or return values in method or function calls between your application code and a framework.

For local variables, such as a counter in a loop, it's fine to use the basic C types if you know that the value is within the standard limits.

C Structures Can Hold Primitive Values

Some Cocoa and Cocoa Touch API use C structures to hold their values. As an example, it's possible to ask a string object for the range of a substring, like this:

```
    NSString *mainString = @"This is a long string";
    NSRange substringRange = [mainString rangeOfString:@"long"];
```

An `NSRange` structure holds a location and a length. In this case, `substringRange` will hold a range of `{10,4}`—the "`l`" at the start of `@"long"` is the character at zero-based index `10` in `mainString`, and `@"long"` is `4` characters in length.

Similarly, if you need to write custom drawing code, you'll need to interact with Quartz, which requires structures based around the `CGFloat` data type, like `NSPoint` and `NSSize` on OS X and `CGPoint` and `CGSize` on iOS. Again, `CGFloat` is defined differently depending on the target architecture.

For more information on the Quartz 2D drawing engine, see *Quartz 2D Programming Guide*.

Objects Can Represent Primitive Values

If you need to represent a scalar value as an object, such as when working with the collection classes described in the next section, you can use one of the basic value classes provided by Cocoa and Cocoa Touch.

Strings Are Represented by Instances of the NSString Class

As you've seen in the previous chapters, `NSString` is used to represent a string of characters, like `Hello World`. There are various ways to create `NSString` objects, including standard allocation and initialization, class factory methods or literal syntax:

```
NSString *firstString = [[NSString alloc] initWithCString:"Hello World!"
                                                  encoding:NSUTF8StringEncoding];
NSString *secondString = [NSString stringWithCString:"Hello World!"
                                              encoding:NSUTF8StringEncoding];
NSString *thirdString = @"Hello World!";
```

Each of these examples effectively accomplishes the same thing—creating a string object that represents the provided characters.

The basic `NSString` class is immutable, which means its contents are set at creation and cannot later be changed. If you need to represent a different string, you must create a new string object, like this:

```
NSString *name = @"John";
name = [name stringByAppendingString:@"ny"];    // returns a new string object
```

The `NSMutableString` class is the mutable subclass of `NSString`, and allows you to change its character contents at runtime using methods like `appendString:` or `appendFormat:`, like this:

```
NSMutableString *name = [NSMutableString stringWithString:@"John"];
[name appendString:@"ny"];   // same object, but now represents "Johnny"
```

Format Strings Are Used to Build Strings from Other Objects or Values

If you need to build a string containing variable values, you need to work with a **format string**. This allows you to use format specifiers to indicate how the values are inserted:

```
int magicNumber = ...
```

```
    NSString *magicString = [NSString stringWithFormat:@"The magic number is %i",
magicNumber];
```

The available format specifiers are described in String Format Specifiers. For more information about strings in general, see the *String Programming Guide*.

Numbers Are Represented by Instances of the NSNumber Class

The `NSNumber` class is used to represent any of the basic C scalar types, including `char`, `double`, `float`, `int`, `long`, `short`, and the `unsigned` variants of each, as well as the Objective-C Boolean type, `BOOL`.

As with `NSString`, you have a variety of options to create `NSNumber` instances, including allocation and initialization or the class factory methods:

```
    NSNumber *magicNumber = [[NSNumber alloc] initWithInt:42];
    NSNumber *unsignedNumber = [[NSNumber alloc] initWithUnsignedInt:42u];
    NSNumber *longNumber = [[NSNumber alloc] initWithLong:42l];

    NSNumber *boolNumber = [[NSNumber alloc] initWithBOOL:YES];

    NSNumber *simpleFloat = [NSNumber numberWithFloat:3.14f];
    NSNumber *betterDouble = [NSNumber numberWithDouble:3.1415926535];

    NSNumber *someChar = [NSNumber numberWithChar:'T'];
```

It's also possible to create `NSNumber` instances using Objective-C literal syntax:

```
    NSNumber *magicNumber = @42;
    NSNumber *unsignedNumber = @42u;
    NSNumber *longNumber = @42l;

    NSNumber *boolNumber = @YES;

    NSNumber *simpleFloat = @3.14f;
    NSNumber *betterDouble = @3.1415926535;

    NSNumber *someChar = @'T';
```

These examples are equivalent to using the NSNumber class factory methods.

Once you've created an NSNumber instance it's possible to request the scalar value using one of the accessor methods:

```
int scalarMagic = [magicNumber intValue];
unsigned int scalarUnsigned = [unsignedNumber unsignedIntValue];
long scalarLong = [longNumber longValue];

BOOL scalarBool = [boolNumber boolValue];

float scalarSimpleFloat = [simpleFloat floatValue];
double scalarBetterDouble = [betterDouble doubleValue];

char scalarChar = [someChar charValue];
```

The NSNumber class also offers methods to work with the additional Objective-C primitive types. If you need to create an object representation of the scalar NSInteger and NSUInteger types, for example, make sure you use the correct methods:

```
NSInteger anInteger = 64;
NSUInteger anUnsignedInteger = 100;

NSNumber *firstInteger = [[NSNumber alloc] initWithInteger:anInteger];
NSNumber *secondInteger = [NSNumber numberWithUnsignedInteger:anUnsignedInteger];

NSInteger integerCheck = [firstInteger integerValue];
NSUInteger unsignedCheck = [secondInteger unsignedIntegerValue];
```

All NSNumber instances are immutable, and there is no mutable subclass; if you need a different number, simply use another NSNumber instance.

> **Note:** `NSNumber` is actually a class cluster. This means that when you create an instance at runtime, you'll get a suitable concrete subclass to hold the provided value. Just treat the created object as an instance of `NSNumber`.

Represent Other Values Using Instances of the NSValue Class

The `NSNumber` class is itself a subclass of the basic `NSValue` class, which provides an object wrapper around a single value or data item. In addition to the basic C scalar types, `NSValue` can also be used to represent pointers and structures.

The `NSValue` class offers various factory methods to create a value with a given standard structure, which makes it easy to create an instance to represent, for example, an `NSRange`, like the example from earlier in the chapter:

```
    NSString *mainString = @"This is a long string";
    NSRange substringRange = [mainString rangeOfString:@"long"];
    NSValue *rangeValue = [NSValue valueWithRange:substringRange];
```

It's also possible to create `NSValue` objects to represent custom structures. If you have a particular need to use a C structure (rather than an Objective-C object) to store information, like this:

```
typedef struct {
    int i;
    float f;
} MyIntegerFloatStruct;
```

you can create an `NSValue` instance by providing a pointer to the structure as well as an encoded Objective-C type. The `@encode()` compiler directive is used to create the correct Objective-C type, like this:

```
    struct MyIntegerFloatStruct aStruct;
    aStruct.i = 42;
    aStruct.f = 3.14;

    NSValue *structValue = [NSValue value:&aStruct
                             withObjCType:@encode(MyIntegerFloatStruct)];
```

The standard C reference operator (`&`) is used to provide the address of `aStruct` for the `value` parameter.

Most Collections Are Objects

Although it's possible to use a C array to hold a collection of scalar values, or even object pointers, most collections in Objective-C code are instances of one of the Cocoa and Cocoa Touch collection classes, like `NSArray`, `NSSet` and `NSDictionary`.

These classes are used to manage groups of objects, which means any item you wish to add to a collection must be an instance of an Objective-C class. If you need to add a scalar value, you must first create a suitable `NSNumber` or `NSValue` instance to represent it.

Rather than somehow maintaining a separate copy of each collected object, the collection classes use strong references to keep track of their contents. This means that any object that you add to a collection will be kept alive at least as long as the collection is kept alive, as described in Manage the Object Graph through Ownership and Responsibility (page 53).

In addition to keeping track of their contents, each of the Cocoa and Cocoa Touch collection classes make it easy to perform certain tasks, such as enumeration, accessing specific items, or finding out whether a particular object is part of the collection.

The basic `NSArray`, `NSSet` and `NSDictionary` classes are immutable, which means their contents are set at creation. Each also has a mutable subclass to allow you to add or remove objects at will.

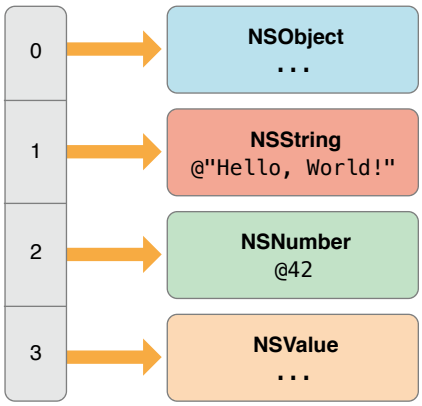
For more information on the different collection classes available in Cocoa and Cocoa Touch, see *Collections Programming Topics*.

Arrays Are Ordered Collections

An `NSArray` is used to represent an *ordered collection* of objects. The only requirement is that each item is an Objective-C object— there's no requirement for each object to be an instance of the same class.

To maintain order in the array, each element is stored at a zero-based index, as shown in Figure 6-1 (page 93).

Figure 6-1 An Array of Objective-C Objects



Creating Arrays

As with the value classes described earlier in this chapter, you can create an array through allocation and initialization, class factory methods, or literal syntax.

There are a variety of different initialization and factory methods available, depending on the number of objects:

```
+ (id)arrayWithObject:(id)anObject;
+ (id)arrayWithObjects:(id)firstObject, ...;
- (id)initWithObjects:(id)firstObject, ...;
```

The `arrayWithObjects:` and `initWithObjects:` methods both take a nil-terminated, variable number of arguments, which means that you must include `nil` as the last value, like this:

```
  NSArray *someArray =
[NSArray arrayWithObjects:someObject, someString, someNumber, someValue, nil];
```

This example creates an array like the one shown earlier, in Figure 6-1 (page 93). The first object, `someObject`, will have an array index of `0`; the last object, `someValue`, will have an index of `3`.

It's possible to truncate the list of items unintentionally if one of the provided values is `nil`, like this:

```
id firstObject = @"someString";
id secondObject = nil;
id thirdObject = @"anotherString";
```

```
    NSArray *someArray =
[NSArray arrayWithObjects:firstObject, secondObject, thirdObject, nil];
```

In this case, `someArray` will contain only `firstObject`, because the `nil secondObject` would be interpreted as the end of the list of items.

Literal Syntax

It's also possible to create an array using an Objective-C literal, like this:

```
NSArray *someArray = @[firstObject, secondObject, thirdObject];
```

You should not terminate the list of objects with `nil` when using this literal syntax, and in fact `nil` is an invalid value. You'll get an exception at runtime if you try to execute the following code, for example:

```
id firstObject = @"someString";
id secondObject = nil;
NSArray *someArray = @[firstObject, secondObject];
// exception: "attempt to insert nil object"
```

If you do need to represent a `nil` value in one of the collection classes, you should use the `NSNull` singleton class, as described in Represent nil with NSNull (page 99).

Querying Array Objects

Once you've created an array, you can query it for information like the number of objects, or whether it contains a given item:

```
NSUInteger numberOfItems = [someArray count];

if ([someArray containsObject:someString]) {
    ...
}
```

You can also query the array for an item at a given index. You'll get an out-of-bounds exception at runtime if you attempt to request an invalid index, so you should always check the number of items first:

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", [someArray objectAtIndex:0]);
}
```

This example checks whether the number of items is greater than zero. If so, it logs a description of the first item, which has an index of zero.

Subscripting

There's also a subscript syntax alternative to using `objectAtIndex:`, which is just like accessing a value in a standard C array. The previous example could be re-written like this:

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", someArray[0]);
}
```

Sorting Array Objects

The `NSArray` class also offers a variety of methods to sort its collected objects. Because `NSArray` is immutable, each of these methods returns a new array containing the items in the sorted order.

As an example, you can sort an array of strings by the result of calling `compare:` on each string, like this:

```
NSArray *unsortedStrings = @[@"gammaString", @"alphaString", @"betaString"];
NSArray *sortedStrings =
            [unsortedStrings sortedArrayUsingSelector:@selector(compare:)];
```

Mutability

Although the `NSArray` class itself is immutable, this has no bearing on any collected objects. If you add a mutable string to an immutable array, for example, like this:

```
NSMutableString *mutableString = [NSMutableString stringWithString:@"Hello"];
NSArray *immutableArray = @[mutableString];
```

there's nothing to stop you from mutating the string:

```
if ([immutableArray count] > 0) {
    id string = immutableArray[0];
```

```
    if ([string isKindOfClass:[NSMutableString class]]) {
        [string appendString:@" World!"];
    }
}
```

If you need to be able to add or remove objects from an array after initial creation, you'll need to use `NSMutableArray`, which adds a variety of methods to add , remove or replace one or more objects:

```
NSMutableArray *mutableArray = [NSMutableArray array];
[mutableArray addObject:@"gamma"];
[mutableArray addObject:@"alpha"];
[mutableArray addObject:@"beta"];

[mutableArray replaceObjectAtIndex:0 withObject:@"epsilon"];
```

This example creates an array that ends up with the objects `@"epsilon"`, `@"alpha"`, `@"beta"`.

It's also possible to sort a mutable array in place, without creating a secondary array:

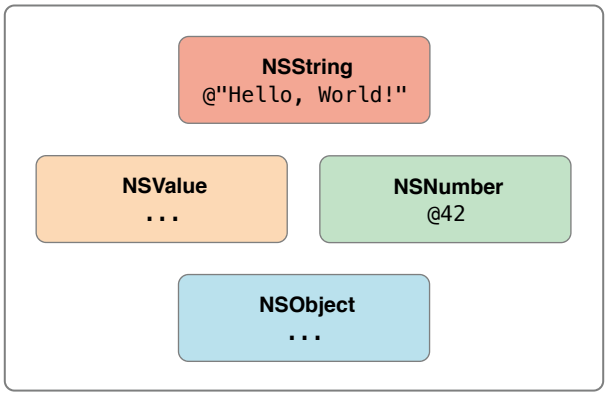
```
[mutableArray sortUsingSelector:@selector(caseInsensitiveCompare:)];
```

In this case the contained items will be sorted into the ascending, case insensitive order of `@"alpha"`, `@"beta"`, `@"epsilon"`.

Sets Are Unordered Collections

An `NSSet` is similar to an array, but maintains an unordered group of **distinct** objects, as shown in Figure 6-2 (page 97).

Figure 6-2 A Set of Objects



Because sets don't maintain order, they offer a performance improvement over arrays when it comes to testing for membership.

The basic `NSSet` class is again immutable, so its contents must be specified at creation, using either allocation and initialization or a class factory method, like this:

```
NSSet *simpleSet =
  [NSSet setWithObjects:@"Hello, World!", @42, aValue, anObject, nil];
```

As with `NSArray`, the `initWithObjects:` and `setWithObjects:` methods both take a nil-terminated, variable number of arguments. The mutable `NSSet` subclass is `NSMutableSet`.

Sets only store one reference to an individual object, even if you try and add an object more than once:

```
NSNumber *number = @42;
NSSet *numberSet =
  [NSSet setWithObjects:number, number, number, number, nil];
// numberSet only contains one object
```

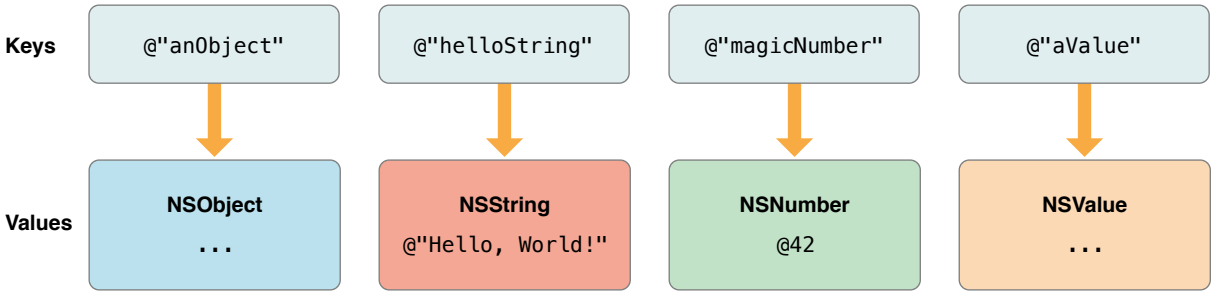
For more information on sets, see Sets: Unordered Collections of Objects.

Dictionaries Collect Key-Value Pairs

Rather than simply maintaining an ordered or unordered collection of objects, an `NSDictionary` stores objects against given keys, which can then be used for retrieval.

It's best practice to use string objects as dictionary keys, as shown in Figure 6-3 (page 98).

Figure 6-3 A Dictionary of Objects



Note: It's possible to use other objects as keys, but it's important to note that each key is copied for use by a dictionary and so must support `NSCopying`.

If you wish to be able to use Key-Value Coding, however, as described in *Key-Value Coding Programming Guide*, you must use string keys for dictionary objects.

Creating Dictionaries

You can create dictionaries using either allocation and initialization, or class factory methods, like this:

```
NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
                anObject, @"anObject",
          @"Hello, World!", @"helloString",
                      @42, @"magicNumber",
                someValue, @"aValue",
                           nil];
```

Note that for the `dictionaryWithObjectsAndKeys:` and `initWithObjectsAndKeys:` methods, each object is specified before its key, and again, the list of objects and keys must be nil-terminated.

Literal Syntax

Objective-C also offers a literal syntax for dictionary creation, like this:

```
NSDictionary *dictionary = @{
                @"anObject" : someObject,
            @"helloString" : @"Hello, World!",
            @"magicNumber" : @42,
                 @"aValue" : someValue
};
```

Note that for dictionary literals, the key is specified before its object and is not nil-terminated.

Querying Dictionaries

Once you've created a dictionary, you can ask it for the object stored against a given key, like this:

```
NSNumber *storedNumber = [dictionary objectForKey:@"magicNumber"];
```

If the object isn't found, the `objectForKey:` method will return `nil`.

There's also a subscript syntax alternative to using `objectForKey:`, which looks like this:

```
NSNumber *storedNumber = dictionary[@"magicNumber"];
```

Mutability

If you need to add or remove objects from a dictionary after creation, you need to use the `NSMutableDictionary` subclass, like this:

```
[dictionary setObject:@"another string" forKey:@"secondString"];
[dictionary removeObjectForKey:@"anObject"];
```

Represent nil with NSNull

It's not possible to add `nil` to the collection classes described in this section because `nil` in Objective-C means "no object." If you need to represent "no object" in a collection, you can use the `NSNull` class:

```
NSArray *array = @[ @"string", @42, [NSNull null] ];
```

`NSNull` is a singleton class, which means that the `null` method will always return the same instance. This means that you can check whether an object in an array is equal to the shared `NSNull` instance:

```
    for (id object in array) {
        if (object == [NSNull null]) {
            NSLog(@"Found a null object");
        }
    }
```

Use Collections to Persist Your Object Graph

The `NSArray` and `NSDictionary` classes make it easy to write their contents directly to disk, like this:

```
    NSURL *fileURL = ...
    NSArray *array = @[@"first", @"second", @"third"];

    BOOL success = [array writeToURL:fileURL atomically:YES];
    if (!success) {
        // an error occured...
    }
```

If every contained object is one of the *property list* types (`NSArray`, `NSDictionary`, `NSString`, `NSData`, `NSDate` and `NSNumber`), it's possible to recreate the entire hierarchy from disk, like this:

```
    NSURL *fileURL = ...
    NSArray *array = [NSArray arrayWithContentsOfURL:fileURL];
    if (!array) {
        // an error occurred...
    }
```

For more information on property lists, see *Property List Programming Guide*.

If you need to persist other types of objects than just the standard property list classes shown above, you can use an archiver object, such as `NSKeyedArchiver`, to create an archive of the collected objects.

The only requirement to create an archive is that each object must support the `NSCoding` protocol. This means that each object must know how to *encode* itself to an archive (by implementing the `encodeWithCoder:` method) and *decode* itself when read from an existing archive (the `initWithCoder:` method).

The `NSArray`, `NSSet` and `NSDictionary` classes, and their mutable subclasses, all support `NSCoding`, which means you can persist complex hierarchies of objects using an archiver. If you use Interface Builder to lay out windows and views, for example, the resulting nib file is just an archive of the object hierarchy that you've created visually. At runtime, the nib file is unarchived to a hierarchy of objects using the relevant classes.

For more information on Archives, see *Archives and Serializations Programming Guide*.

Use the Most Efficient Collection Enumeration Techniques

Objective-C and Cocoa or Cocoa Touch offer a variety of ways to enumerate the contents of a collection. Although it's possible to use a traditional C `for` loop to iterate over the contents, like this:

```
int count = [array count];
for (int index = 0; index < count; index++) {
    id eachObject = [array objectAtIndex:index];
    ...
}
```

it's best practice to use one of the other techniques described in this section.

Fast Enumeration Makes It Easy to Enumerate a Collection

Many collection classes conform to the `NSFastEnumeration` protocol, including `NSArray`, `NSSet` and `NSDictionary`. This means that you can use fast enumeration, an Objective-C language-level feature.

The fast enumeration syntax to enumerate the contents of an array or set looks like this:

```
for (<Type> <variable> in <collection>) {
    ...
}
```

As an example, you might use fast enumeration to log a description of each object in an array, like this:

```
for (id eachObject in array) {
    NSLog(@"Object: %@", eachObject);
}
```

The `eachObject` variable is set automatically to the current object for each pass through the loop, so one log statement appears per object.

If you use fast enumeration with a dictionary, you iterate over the dictionary *keys*, like this:

```
for (NSString *eachKey in dictionary) {
    id object = dictionary[eachKey];
    NSLog(@"Object: %@ for key: %@", object, eachKey);
}
```

Fast enumeration behaves much like a standard C `for` loop, so you can use the `break` keyword to interrupt the iteration, or `continue` to advance to the next element.

If you are enumerating an ordered collection, the enumeration proceeds in that order. For an `NSArray`, this means the first pass will be for the object at index `0`, the second for object at index `1`, etc. If you need to keep track of the current index, simply count the iterations as they occur:

```
int index = 0;
for (id eachObject in array) {
    NSLog(@"Object at index %i is: %@", index, eachObject);
    index++;
}
```

You cannot mutate a collection during fast enumeration, even if the collection is mutable. If you attempt to add or remove a collected object from within the loop, you'll generate a runtime exception.

Most Collections Also Support Enumerator Objects

It's also possible to enumerate many Cocoa and Cocoa Touch collections by using an `NSEnumerator` object.

You can ask an `NSArray`, for example, for an `objectEnumerator` or a `reverseObjectEnumerator`. It's possible to use these objects with fast enumeration, like this:

```
for (id eachObject in [array reverseObjectEnumerator]) {
    ...
}
```

In this example, the loop will iterate over the collected objects in reverse order, so the last object will be first, and so on.

It's also possible to iterate through the contents by calling the enumerator's `nextObject` method repeatedly, like this:

```
id eachObject;
while ( (eachObject = [enumerator nextObject]) ) {
    NSLog(@"Current object is: %@", eachObject);
}
```

In this example, a `while` loop is used to set the `eachObject` variable to the next object for each pass through the loop. When there are no more objects left, the `nextObject` method will return `nil`, which evaluates as a logical value of false so the loop stops.

Note: Because it's a common programmer error to use the C assignment operator (=) when you mean the equality operator (==), the compiler will warn you if you set a variable in a conditional branch or loop, like this:

```
if (someVariable = YES) {
    ...
}
```

If you really do mean to reassign a variable (the logical value of the overall assignment is the final value of the left hand side), you can indicate this by placing the assignment in parentheses, like this:

```
if ( (someVariable = YES) ) {
    ...
}
```

As with fast enumeration, you cannot mutate a collection while enumerating. And, as you might gather from the name, it's faster to use fast enumeration than to use an enumerator object manually.

Many Collections Support Block-Based Enumeration

It's also possible to enumerate `NSArray`, `NSSet` and `NSDictionary` using blocks. Blocks are covered in detail in the next chapter.

Working with Blocks

An Objective-C class defines an object that combines data with related behavior. Sometimes, it makes sense just to represent a single task or unit of behavior, rather than a collection of methods.

Blocks are a language-level feature added to C, Objective-C and C++, which allow you to create distinct segments of code that can be passed around to methods or functions as if they were values. Blocks are Objective-C objects, which means they can be added to collections like `NSArray` or `NSDictionary`. They also have the ability to capture values from the enclosing scope, making them similar to *closures* or *lambdas* in other programming languages.

This chapter explains the syntax to declare and refer to blocks, and shows how to use blocks to simplify common tasks such as collection enumeration. For further information, see *Blocks Programming Topics*.

Block Syntax

The syntax to define a block literal uses the caret symbol (^), like this:

```
^{
     NSLog(@"This is a block");
}
```

As with function and method definitions, the braces indicate the start and end of the block. In this example, the block doesn't return any value, and doesn't take any arguments.

In the same way that you can use a function pointer to refer to a C function, you can declare a variable to keep track of a block, like this:

```
void (^simpleBlock)(void);
```

If you're not used to dealing with C function pointers, the syntax may seem a little unusual. This example declares a variable called `simpleBlock` to refer to a block that takes no arguments and doesn't return a value, which means the variable can be assigned the block literal shown above, like this:

```
simpleBlock = ^{
    NSLog(@"This is a block");
};
```

This is just like any other variable assignment, so the statement must be terminated by a semi-colon after the closing brace. You can also combine the variable declaration and assignment:

```
void (^simpleBlock)(void) = ^{
    NSLog(@"This is a block");
};
```

Once you've declared and assigned a block variable, you can use it to invoke the block:

```
simpleBlock();
```

> **Note:** If you attempt to invoke a block using an unassigned variable (a `nil` block variable), your app will crash.

Blocks Take Arguments and Return Values

Blocks can also take arguments and return values just like methods and functions.

As an example, consider a variable to refer to a block that returns the result of multiplying two values:

```
double (^multiplyTwoValues)(double, double);
```

The corresponding block literal might look like this:

```
^ (double firstValue, double secondValue) {
    return firstValue * secondValue;
}
```

The `firstValue` and `secondValue` are used to refer to the values supplied when the block is invoked, just like any function definition. In this example, the return type is inferred from the return statement inside the block.

If you prefer, you can make the return type explicit by specifying it between the caret and the argument list:

```
    ^ double (double firstValue, double secondValue) {
        return firstValue * secondValue;
    }
```

Once you've declared and defined the block, you can invoke it just like you would a function:

```
    double (^multiplyTwoValues)(double, double) =
                              ^(double firstValue, double secondValue) {
                                  return firstValue * secondValue;
                              };

    double result = multiplyTwoValues(2,4);

    NSLog(@"The result is %f", result);
```

Blocks Can Capture Values from the Enclosing Scope

As well as containing executable code, a block also has the ability to capture state from its enclosing scope.

If you declare a block literal from within a method, for example, it's possible to capture any of the values accessible within the scope of that method, like this:

```
- (void)testMethod {
    int anInteger = 42;

    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
    };

    testBlock();
}
```

In this example, `anInteger` is declared outside of the block, but the value is captured when the block is defined.

Only the value is captured, unless you specify otherwise. This means that if you change the external value of the variable between the time you define the block and the time it's invoked, like this:

```
int anInteger = 42;

void (^testBlock)(void) = ^{
    NSLog(@"Integer is: %i", anInteger);
};

anInteger = 84;

testBlock();
```

the value captured by the block is unaffected. This means that the log output would still show:

```
Integer is: 42
```

It also means that the block cannot change the value of the original variable, or even the captured value (it's captured as a `const` variable).

Use __block Variables to Share Storage

If you need to be able to change the value of a captured variable from within a block, you can use the `__block` storage type modifier on the original variable declaration. This means that the variable lives in storage that is shared between the lexical scope of the original variable and any blocks declared within that scope.

As an example, you might rewrite the previous example like this:

```
__block int anInteger = 42;

void (^testBlock)(void) = ^{
    NSLog(@"Integer is: %i", anInteger);
};

anInteger = 84;

testBlock();
```

Because `anInteger` is declared as a `__block` variable, its storage is shared with the block declaration. This means that the log output would now show:

```
Integer is: 84
```

It also means that the block can modify the original value, like this:

```
    __block int anInteger = 42;

    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
        anInteger = 100;
    };

    testBlock();
    NSLog(@"Value of original variable is now: %i", anInteger);
```

This time, the output would show:

```
Integer is: 42
Value of original variable is now: 100
```

You Can Pass Blocks as Arguments to Methods or Functions

Each of the previous examples in this chapter invokes the block immediately after it's defined. In practice, it's common to pass blocks to functions or methods for invocation elsewhere. You might use Grand Central Dispatch to invoke a block in the background, for example, or define a block to represent a task to be invoked repeatedly, such as when enumerating a collection. Concurrency and enumeration are covered later in this chapter.

Blocks are also used for callbacks, defining the code to be executed when a task completes. As an example, your app might need to respond to a user action by creating an object that performs a complicated task, such as requesting information from a web service. Because the task might take a long time, you should display some kind of progress indicator while the task is occurring, then hide that indicator once the task is complete.

It would be possible to accomplish this using delegation: You'd need to create a suitable delegate protocol, implement the required method, set your object as the delegate of the task, then wait for it to call a delegate method on your object once the task finished.

Blocks make this much easier, however, because you can define the callback behavior at the time you initiate the task, like this:

```
- (IBAction)fetchRemoteInformation:(id)sender {
    [self showProgressIndicator];

    XYZWebTask *task = ...

    [task beginTaskWithCallbackBlock:^{
        [self hideProgressIndicator];
    }];
}
```

This example calls a method to display the progress indicator, then creates the task and tells it to start. The callback block specifies the code to be executed once the task completes; in this case, it simply calls a method to hide the progress indicator. Note that this callback block captures `self` in order to be able to call the `hideProgressIndicator` method when invoked. It's important to take care when capturing `self` because it's easy to create a strong reference cycle, as described later in Avoid Strong Reference Cycles when Capturing self (page 112).

In terms of code readability, the block makes it easy to see in one place exactly what will happen before and after the task completes, avoiding the need to trace through delegate methods to find out what's going to happen.

The declaration for the `beginTaskWithCallbackBlock:` method shown in this example would look like this:

```
- (void)beginTaskWithCallbackBlock:(void (^)(void))callbackBlock;
```

The `(void (^)(void))` specifies that the parameter is a block that doesn't take any arguments or return any values. The implementation of the method can invoke the block in the usual way:

```
- (void)beginTaskWithCallbackBlock:(void (^)(void))callbackBlock {
    ...
    callbackBlock();
}
```

Method parameters that expect a block with one or more arguments are specified in the same way as with a block variable:

```
- (void)doSomethingWithBlock:(void (^)(double, double))block {
    ...
    block(21.0, 2.0);
}
```

A Block Should Always Be the Last Argument to a Method

It's best practice to use only one block argument to a method. If the method also needs other non-block arguments, the block should come last:

```
- (void)beginTaskWithName:(NSString *)name completion:(void(^)(void))callback;
```

This makes the method call easier to read when specifying the block inline, like this:

```
    [self beginTaskWithName:@"MyTask" completion:^{
        NSLog(@"The task is complete");
    }];
```

Use Type Definitions to Simplify Block Syntax

If you need to define more than one block with the same signature, you might like to define your own type for that signature.

As an example, you can define a type for a simple block with no arguments or return value, like this:

```
typedef void (^XYZSimpleBlock)(void);
```

You can then use your custom type for method parameters or when creating block variables:

```
    XYZSimpleBlock anotherBlock = ^{
        ...
    };
```

```
- (void)beginFetchWithCallbackBlock:(XYZSimpleBlock)callbackBlock {
    ...
    callbackBlock();
```

```
}
```

Custom type definitions are particularly useful when dealing with blocks that return blocks or take other blocks as arguments. Consider the following example:

```
void (^(^complexBlock)(void (^)(void)))(void) = ^ (void (^aBlock)(void)) {
    ...
    return ^{
        ...
    };
};
```

The `complexBlock` variable refers to a block that takes another block as an argument (`aBlock`) and returns yet another block.

Rewriting the code to use a type definition makes this much more readable:

```
XYZSimpleBlock (^betterBlock)(XYZSimpleBlock) = ^ (XYZSimpleBlock aBlock) {
    ...
    return ^{
        ...
    };
};
```

Objects Use Properties to Keep Track of Blocks

The syntax to define a property to keep track of a block is similar to a block variable:

```
@interface XYZObject : NSObject
@property (copy) void (^blockProperty)(void);
@end
```

> **Note:** You should specify `copy` as the property attribute, because a block needs to be copied to keep track of its captured state outside of the original scope. This isn't something you need to worry about when using Automatic Reference Counting, as it will happen automatically, but it's best practice for the property attribute to show the resultant behavior. For more information, see *Blocks Programming Topics*.

A block property is set or invoked like any other block variable:

```
self.blockProperty = ^{
    ...
};
self.blockProperty();
```

It's also possible to use type definitions for block property declarations, like this:

```
typedef void (^XYZSimpleBlock)(void);

@interface XYZObject : NSObject
@property (copy) XYZSimpleBlock blockProperty;
@end
```

Avoid Strong Reference Cycles when Capturing self

If you need to capture `self` in a block, such as when defining a callback block, it's important to consider the memory management implications.

Blocks maintain strong references to any captured objects, including `self`, which means that it's easy to end up with a strong reference cycle if, for example, an object maintains a `copy` property for a block that captures `self`:

```
@interface XYZBlockKeeper : NSObject
@property (copy) void (^block)(void);
@end
```

```
@implementation XYZBlockKeeper
- (void)configureBlock {
```

```
    self.block = ^{
        [self doSomething];    // capturing a strong reference to self
                               // creates a strong reference cycle
    };
}
...
@end
```

The compiler will warn you for a simple example like this, but a more complex example might involve multiple strong references between objects to create the cycle, making it more difficult to diagnose.

To avoid this problem, it's best practice to capture a weak reference to `self`, like this:

```
- (void)configureBlock {
    XYZBlockKeeper * __weak weakSelf = self;
    self.block = ^{
        [weakSelf doSomething];   // capture the weak reference
                                  // to avoid the reference cycle
    }
}
```

By capturing the weak pointer to `self`, the block won't maintain a strong relationship back to the `XYZBlockKeeper` object. If that object is deallocated before the block is called, the `weakSelf` pointer will simply be set to `nil`.

Blocks Can Simplify Enumeration

In addition to general completion handlers, many Cocoa and Cocoa Touch API use blocks to simplify common tasks, such as collection enumeration. The `NSArray` class, for example, offers three block-based methods, including:

```
- (void)enumerateObjectsUsingBlock:(void (^)(id obj, NSUInteger idx, BOOL
*stop))block;
```

This method takes a single argument, which is a block to be invoked once for each item in the array:

```
NSArray *array = ...
[array enumerateObjectsUsingBlock:^ (id obj, NSUInteger idx, BOOL *stop) {
    NSLog(@"Object at index %lu is %@", idx, obj);
}];
```

The block itself takes three arguments, the first two of which refer to the current object and its index in the array. The third argument is a pointer to a Boolean variable that you can use to stop the enumeration, like this:

```
[array enumerateObjectsUsingBlock:^ (id obj, NSUInteger idx, BOOL *stop) {
    if (...) {
        *stop = YES;
    }
}];
```

It's also possible to customize the enumeration by using the `enumerateObjectsWithOptions:usingBlock:` method. Specifying the `NSEnumerationReverse` option, for example, will iterate through the collection in reverse order.

If the code in the enumeration block is processor-intensive—and safe for concurrent execution—you can use the `NSEnumerationConcurrent` option:

```
[array enumerateObjectsWithOptions:NSEnumerationConcurrent
                        usingBlock:^ (id obj, NSUInteger idx, BOOL *stop) {
    ...
}];
```

This flag indicates that the enumeration block invocations may be distributed across multiple threads, offering a potential performance increase if the block code is particularly processor intensive. Note that the enumeration order is undefined when using this option.

The `NSDictionary` class also offers block-based methods, including:

```
NSDictionary *dictionary = ...
[dictionary enumerateKeysAndObjectsUsingBlock:^ (id key, id obj, BOOL *stop)
{
    NSLog(@"key: %@, value: %@", key, obj);
}];
```

This makes it more convenient to enumerate each key-value pair than when using a traditional loop, for example.

Blocks Can Simplify Concurrent Tasks

A block represents a distinct unit of work, combining executable code with optional state captured from the surrounding scope. This makes it ideal for asynchronous invocation using one of the concurrency options available for OS X and iOS. Rather than having to figure out how to work with low-level mechanisms like threads, you can simply define your tasks using blocks and then let the system perform those tasks as processor resources become available.

OS X and iOS offer a variety of technologies for concurrency, including two task-scheduling mechanisms: Operation queues and Grand Central Dispatch. These mechanisms revolve around the idea of a queue of tasks waiting to be invoked. You add your blocks to a queue in the order you need them to be invoked, and the system dequeues them for invocation when processor time and resources become available.

A *serial queue* only allows one task to execute at a time—the next task in the queue won't be dequeued and invoked until the previous task has finished. A *concurrent queue* invokes as many tasks as it can, without waiting for previous tasks to finish.

Use Block Operations with Operation Queues

An operation queue is the Cocoa and Cocoa Touch approach to task scheduling. You create an `NSOperation` instance to encapsulate a unit of work along with any necessary data, then add that operation to an `NSOperationQueue` for execution.

Although you can create your own custom `NSOperation` subclass to implement complex tasks, it's also possible to use the `NSBlockOperation` to create an operation using a block, like this:

```
NSBlockOperation *operation = [NSBlockOperation blockOperationWithBlock:^{
    ...
}];
```

It's possible to execute an operation manually but operations are usually added either to an existing operation queue or a queue you create yourself, ready for execution:

```
// schedule task on main queue:
NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];
[mainQueue addOperation:operation];
```

```
// schedule task on background queue:
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[queue addOperation:operation];
```

If you use an operation queue, you can configure priorities or dependencies between operations, such as specifying that one operation should not be executed until a group of other operations has completed. You can also monitor changes to the state of your operations through key-value observing, which makes it easy to update a progress indicator, for example, when a task completes.

For more information on operations and operation queues, see Operation Queues.

Schedule Blocks on Dispatch Queues with Grand Central Dispatch

If you need to schedule an arbitrary block of code for execution, you can work directly with *dispatch queues* controlled by Grand Central Dispatch (GCD). Dispatch queues make it easy to perform tasks either synchronously or asynchronously with respect to the caller, and execute their tasks in a first-in, first-out order.

You can either create your own dispatch queue or use one of the queues provided automatically by GCD. If you need to schedule a task for concurrent execution, for example, you can get a reference to an existing queue by using the `dispatch_get_global_queue()` function and specifying a queue priority, like this:

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0);
```

To dispatch the block to the queue, you use either the `dispatch_async()` or `dispatch_sync()` functions. The `dispatch_async()` function returns immediately, without waiting for the block to be invoked:

```
dispatch_async(queue, ^{
    NSLog(@"Block for asynchronous execution");
});
```

The `dispatch_sync()` function doesn't return until the block has completed execution; you might use it in a situation where a concurrent block needs to wait for another task to complete on the main thread before continuing, for example.

For more information on dispatch queues and GCD, see Dispatch Queues.

Dealing with Errors

SwiftObjective-C

Almost every app encounters errors. Some of these errors will be outside of your control, such as running out of disk space or losing network connectivity. Some of these errors will be recoverable, such as invalid user input. And, while all developers strive for perfection, the occasional programmer error may also occur.

If you're coming from other platforms and languages, you may be used to working with exceptions for the majority of error handling. When you're writing code with Objective-C, exceptions are used solely for programmer errors, like out-of-bounds array access or invalid method arguments. These are the problems that you should find and fix during testing before you ship your app.

All other errors are represented by instances of the `NSError` class. This chapter gives a brief introduction to using `NSError` objects, including how to work with framework methods that may fail and return errors. For further information, see *Error Handling Programming Guide*.

Use NSError for Most Errors

Errors are an unavoidable part of any app's lifecycle. If you need to request data from a remote web service, for example, there are a variety of potential problems that may arise, including:

- No network connectivity
- The remote web service may be inaccessible
- The remote web service may not be able to serve the information you request
- The data you receive may not match what you were expecting

Sadly, it's not possible to build contingency plans and solutions for every conceivable problem. Instead, you must plan for errors and know how to deal with them to give the best possible user experience.

Some Delegate Methods Alert You to Errors

If you're implementing a delegate object for use with a framework class that performs a certain task, like downloading information from a remote web service, you'll typically find that you need to implement at least one error-related method. The `NSURLConnectionDelegate` protocol, for example, includes a `connection:didFailWithError:` method:

```
- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error;
```

If an error occurs, this delegate method will be called to provide you with an `NSError` object to describe the problem.

An `NSError` object contains a numeric error code, domain and description, as well as other relevant information packaged in a user info dictionary.

Rather than making the requirement that every possible error have a unique numeric code, Cocoa and Cocoa Touch errors are divided into domains. If an error occurs in an `NSURLConnection`, for example, the `connection:didFailWithError:` method above will provide an error from `NSURLErrorDomain`.

The error object also includes a localized description, such as "A server with the specified hostname could not be found."

Some Methods Pass Errors by Reference

Some Cocoa and Cocoa Touch API pass back errors by reference. As an example, you might decide to store the data that you receive from a web service by writing it to disk, using the `NSData` method `writeToURL:options:error:`. The last parameter to this method is a reference to an `NSError` pointer:

```
- (BOOL)writeToURL:(NSURL *)aURL
           options:(NSDataWritingOptions)mask
             error:(NSError **)errorPtr;
```

Before you call this method, you'll need to create a suitable pointer so that you can pass its address:

```
    NSError *anyError;
    BOOL success = [receivedData writeToURL:someLocalFileURL
                                    options:0
                                      error:&anyError];
    if (!success) {
        NSLog(@"Write failed with error: %@", anyError);
        // present error to user
    }
```

If an error occurs, the `writeToURL:...` method will return `NO`, and update your `anyError` pointer to point to an error object describing the problem.

When dealing with errors passed by reference, it's important to test the return value of the method to see whether an error occurred, as shown above. Don't just test to see whether the error pointer was set to point to an error.

> **Tip:** If you're not interested in the error object, just pass `NULL` for the `error:` parameter.

Recover if Possible or Display the Error to the User

The best user experience is for your app to recover transparently from an error. If you're making a remote web request, for example, you might try making the request again with a different server. Alternatively, you might need to request additional information from the user such as valid username or password credentials before trying again.

If it's not possible to recover from an error, you should alert the user. If you're developing with Cocoa Touch for iOS, you'll need to create and configure a `UIAlertView` to display the error. If you're developing with Cocoa for OS X, you can call `presentError:` on any `NSResponder` object (like a view, window or even the application object itself) and the error will propagate up the responder chain for further configuration or recovery. When it reaches the application object, the application presents the error to the user through an alert panel.

For more information on presenting errors to the user, see Displaying Information From Error Objects.

Generating Your Own Errors

In order to create your own `NSError` objects you'll need to define your own error domain, which should be of the form:

```
com.companyName.appOrFrameworkName.ErrorDomain
```

You'll also need to pick a unique error code for each error that may occur in your domain, along with a suitable description, which is stored in the user info dictionary for the error, like this:

```
NSString *domain = @"com.MyCompany.MyApplication.ErrorDomain";
NSString *desc = NSLocalizedString(@"Unable to…", @"");
NSDictionary *userInfo = @{ NSLocalizedDescriptionKey : desc };

NSError *error = [NSError errorWithDomain:domain
                                     code:-101
                                 userInfo:userInfo];
```

This example uses the `NSLocalizedString` function to look up a localized version of the error description from a `Localizable.strings` file, as described in Localizing String Resources.

If you need to pass back an error by reference as described earlier, your method signature should include a parameter for a pointer to a pointer to an `NSError` object. You should also use the return value to indicate success or failure, like this:

```
- (BOOL)doSomethingThatMayGenerateAnError:(NSError **)errorPtr;
```

If an error occurs, you should start by checking whether a non-`NULL` pointer was provided for the error parameter before you attempt to dereference it to set the error, before returning `NO` to indicate failure, like this:

```
- (BOOL)doSomethingThatMayGenerateAnError:(NSError **)errorPtr {
    ...
    // error occurred
    if (errorPtr) {
        *errorPtr = [NSError errorWithDomain:...
                                        code:...
                                    userInfo:...];
    }
    return NO;
}
```

Exceptions Are Used for Programmer Errors

Objective-C supports exceptions in much the same way as other programming languages, with a similar syntax to Java or C++. As with `NSError`, exceptions in Cocoa and Cocoa Touch are objects, represented by instances of the `NSException` class

If you need to write code that might cause an exception to be thrown, you can enclose that code inside a try-catch block:

```
    @try {
        // do something that might throw an exception
    }
    @catch (NSException *exception) {
```

```
        // deal with the exception
    }
    @finally {
        // optional block of clean-up code
        // executed whether or not an exception occurred
    }
```

If an exception is thrown by the code inside the `@try` block, it will be caught by the `@catch` block so that you can deal with it. If you're working with a low-level C++ library that uses exceptions for error handling, for example, you might catch its exceptions and generate suitable `NSError` objects to display to the user.

If an exception is thrown and not caught, the default uncaught exception handler logs a message to the console and terminates the application.

You should not use a try-catch block in place of standard programming checks for Objective-C methods. In the case of an `NSArray`, for example, you should always check the array's `count` to determine the number of items before trying to access an object at a given index. The `objectAtIndex:` method throws an exception if you make an out-of-bounds request so that you can find the bug in your code early in the development cycle—you should avoid throwing exceptions in an app that you ship to users.

For more information on exceptions in Objective-C applications, see *Exception Programming Topics*.

Conventions

When you're working with the framework classes, you'll notice that Objective-C code is very easy to read. Class and method names are much more descriptive than you might find with general C code functions or the C Standard Library, and camel case is used for names with multiple words. You should follow the same conventions used by Cocoa and Cocoa Touch when you're writing your own classes to make your code more readable, both for you and for other Objective-C developers that may need to work with your projects, and to keep your codebase consistent.

In addition, many Objective-C and framework features require you to follow strict naming conventions in order for various mechanisms to work correctly. Accessor method names, for example, must follow the conventions in order to work with techniques such as Key-Value Coding (KVC) or Key-Value Observing (KVO).

This chapter covers some of the most common conventions used in Cocoa and Cocoa Touch code, and explains the situations when it's necessary for names to be unique across an entire app project, including its linked frameworks.

Some Names Must Be Unique Across Your App

Each time you create a new type, symbol, or identifier, you should first consider the scope in which the name must be unique. Sometimes this scope might be the entire application, including its linked frameworks; sometimes the scope is limited just to an enclosing class or even just a block of code.

Class Names Must Be Unique Across an Entire App

Objective-C classes must be named uniquely not only within the code that you're writing in a project, but also across any frameworks or bundles you might be including. As an example, you should avoid using generic class names like `ViewController` or `TextParser` because it's possible a framework you include in your app may fail to follow conventions and create classes with the same names.

In order to keep class names unique, the convention is to use prefixes on all classes. You'll have noticed that Cocoa and Cocoa Touch class names typically start either with `NS` or `UI`. Two-letter prefixes like these are reserved by Apple for use in framework classes. As you learn more about Cocoa and Cocoa Touch, you'll encounter a variety of other prefixes that relate to specific frameworks:

Prefix	Framework
NS	Foundation (OS X and iOS) and Application Kit (OS X)
UI	UIKit (iOS)
AB	Address Book
CA	Core Animation
CI	Core Image

Your own classes should use three letter prefixes. These might relate to a combination of your company name and your app name, or even a specific component within your app. As an example, if your company were called Whispering Oak, and you were developing a game called Zebra Surprise, you might choose `WZS` or `WOZ` as your class prefix.

You should also name your classes using a noun that makes it clear what the class represents, like these examples from Cocoa and Cocoa Touch:

`NSWindow`	`CAAnimation`	`NSWindowController`	`NSManagedObjectContext`

If multiple words are needed in a class name, you should use *camel case* by capitalizing the first letter of each subsequent word.

Method Names Should Be Expressive and Unique Within a Class

Once you've chosen a unique name for a class, the methods that you declare need only be unique within that class. It's common to use the same name as a method in another class, for example, either to override a superclass method, or take advantage of polymorphism. Methods that perform the same task in multiple classes should have the same name, return type and parameter types.

Method names do not have a prefix, and should start with a lowercase letter; camel case is used again for multiple words, like these examples from the `NSString` class:

`length`	`characterAtIndex:`	`lengthOfBytesUsingEncoding:`

If a method takes one or more arguments, the name of the method should indicate each parameter:

`substringFromIndex:`	`writeToURL:`
	`atomically:encoding:`
	`error:`
		`enumerateSubstringsInRange:`
		`options:usingBlock:`

The first portion of the method name should indicate the primary intent or result of calling the method. If a method returns a value, for example, the first word normally indicates what will be returned, like the `length`, `character...` and `substring...` methods shown above. Multiple words are used if you need to indicate something important about the return value, as with the `mutableCopy`, `capitalizedString` or `lastPathComponent` methods from the `NSString` class. If a method performs an action, such as writing to disk or enumerating the contents, the first word should indicate that action, as shown by the `write...` and `enumerate...` methods.

If a method includes an *error* pointer parameter to be set if an error occurred, this should be the last parameter to the method. If a method takes a *block*, the block parameter should be the last parameter in order to make any method invocations as readable as possible when specifying a block inline. For the same reason, it's best to avoid methods that take multiple block arguments, wherever possible.

It's also important to aim for clear but concise method names. Clarity doesn't necessarily mean verbosity but brevity doesn't necessarily result in clarity, so it's best to aim for a happy medium:

`stringAfterFindingAndReplacingAllOccurrencesOfThisString: withThisString:`	Too verbose
`strReplacingStr:str:`	Too concise
`stringByReplacingOccurrencesOfString: withString:`	Just right

You should avoid abbreviating words in method names unless you are sure that the abbreviation is well known across multiple languages and cultures. A list of common abbreviations is given in Acceptable Abbreviations and Acronyms.

Always Use a Prefix for Method Names in Categories on Framework Classes

When using a category to add methods to an existing framework class, you should include a prefix on the method name to avoid clashes, as described in Avoid Category Method Name Clashes (page 71).

Local Variables Must Be Unique Within The Same Scope

Because Objective-C is a superset of the C language, the C variable scope rules also apply to Objective-C. A local variable name must not clash with any other variables declared within the same scope:

```
- (void)someMethod {
    int interestingNumber = 42;
    ...
    int interestingNumber = 44; // not allowed
```

```
}
```

Although the C language does allow you to declare a new local variable with the same name as one declared in the *enclosing* scope, like this:

```
- (void)someMethod {
    int interestingNumber = 42;
    ...
    for (NSNumber *eachNumber in array) {
        int interestingNumber = [eachNumber intValue]; // not advisable
        ...
    }
}
```

it makes the code confusing and less readable, so it's best practice to avoid this wherever possible.

Some Method Names Must Follow Conventions

In addition to considering uniqueness, it's also essential for a few important method types to follow strict conventions. These conventions are used by some of the underlying mechanisms of Objective-C, the compiler and runtime, in addition to behavior that is required by classes in Cocoa and Cocoa Touch.

Accessor Method Names Must Follow Conventions

When you use the `@property` syntax to declare properties on an object, as described in Encapsulating Data (page 43), the compiler automatically synthesizes the relevant getter and setter methods (unless you indicate otherwise). If you need to provide your own accessor method implementations for any reason, it's important to make sure that you use the right method names for a property in order for your methods to be called through dot syntax, for example.

Unless specified otherwise, a getter method should use the same name as the property. For a property called `firstName`, the accessor method should also be called `firstName`. The exception to this rule is for Boolean properties, for which the getter method should start with `is`. For a property called `paused`, for example, the getter method should be called `isPaused`.

The setter method for a property should use the form `setPropertyName:`. For a property called `firstName`, the setter method should be called `setFirstName:`; for a Boolean property called `paused`, the setter method should be called `setPaused:`.

Although the `@property` syntax allows you to specify different accessor method names, you should only do so for situations like a Boolean property. It's essential to follow the conventions described here, otherwise techniques like Key Value Coding (the ability to get or set a property using `valueForKey:` and `setValue:forKey:`) won't work. For more information on KVC, see *Key-Value Coding Programming Guide*.

Object Creation Method Names Must Follow Conventions

As you've seen in earlier chapters, there are often multiple ways to create instances of a class. You might use a combination of allocation and initialization, like this:

```
NSMutableArray *array = [[NSMutableArray alloc] init];
```

or use the `new` convenience method as an alternative to calling `alloc` and `init` explicitly:

```
NSMutableArray *array = [NSMutableArray new];
```

Some classes also offer class factory methods:

```
NSMutableArray *array = [NSMutableArray array];
```

Class factory methods should always start with the name of the class (without the prefix) that they create, with the exception of subclasses of classes with existing factory methods. In the case of the `NSArray` class, for example, the factory methods start with `array`. The `NSMutableArray` class doesn't define any of its own class-specific factory methods, so the factory methods for a mutable array still begin with `array`.

There are various memory management rules underpinning Objective-C, which the compiler uses to ensure objects are kept alive as long as necessary. Although you typically don't need to worry too much about these rules, the compiler judges which rule it should follow based on the name of the creation method. Objects created via factory methods are managed slightly differently from objects that are created through traditional allocation and initialization or `new` because of the use of autorelease pool blocks. For more information on autorelease pool blocks and memory management in general, see *Advanced Memory Management Programming Guide*.

Document Revision History

This table describes the changes to *Programming with Objective-C*.

Date	Notes
2014-09-17	Noted that required properties in protocols are not auto–synthesized.
2012-12-13	Added a missing word and corrected a code display issue.
2012-07-20	New document that describes elements of best practice when writing code with Objective-C using ARC.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010