D

Crash Course in Objective-C

Objective-C is an object-oriented programming language used by Apple primarily for programming Mac OS X and iPhone/iPad applications. It is an extension to the standard ANSI C language and hence it should be an easy language to pick up if you are already familiar with the C programming language. This appendix assumes that you already have some background in C programming and focuses on the object-oriented aspects of the language. If you are coming from a Java or .NET background, many of the concepts should be familiar to you; you just have to understand the syntax of Objective-C and, in particular, pay attention to the section on memory management.

Objective-C source code files are contained in two types of files:

- `.h` — header files
- `.m` — implementation files

For the discussions that follow, assume that you have created a View-based Application project using Xcode and added an empty class named `SomeClass` to your project.

DIRECTIVES

If you observe the content of the `SomeClass.h` file, you will notice that at the top of the file is an `#import` statement:

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

@interface SomeClass : NSObject {

}

@end
```

The `#import` statement is known as a *preprocessor directive*. In C and C++, you use the `#include` preprocessor directive to include a file's content with the current source. In

Objective-C, you use the `#import` statement to do the same, except that the compiler ensures that the file is included at most only once. To import a header file from one of the frameworks, you specify the header filename using angle brackets (<>) in the `#import` statement. To import a header file from within your project, you use the `" and "` characters, as in the case of the `SomeClass.m` file:

```
#import "SomeClass.h"

@implementation SomeClass

@end
```

CLASSES

In Objective-C, you will spend a lot of time dealing with classes and objects. Hence it is important that you understand how classes are declared and defined in Objective-C.

@interface

To declare a class, you use the `@interface` compiler directive, like this:

```
@interface SomeClass : NSObject {

}
```

This is done in the header file (`.h`) and the class declaration contains no implementation. The preceding code declares a class named `SomeClass`, and this class inherits from the base class named `NSObject`.

NOTE *While you typically put your code declaration in an* `.h` *file, you can also put it inside an* `.m` *if need be. This is usually done for small projects.*

NOTE `NSObject` *is the root class of most Objective-C classes. It defines the basic interface of a class and contains methods common to all classes that inherit from it.* `NSObject` *also provides the standard memory management and initialization framework used by most objects in Objective-C as well as reflection and type operations.*

In a typical View Controller class, the class inherits from the `UIViewController` class, such as in the following:

```
@interface HelloWorldViewController : UIViewController {

}
```

@implementation

To implement a class declared in the header file, you use the `@implementation` compiler directive, like this:

```
#import "SomeClass.h"

@implementation SomeClass

@end
```

This is done in a separate file from the header file. In Objective-C, you define your class in an `.m` file. Note that the class definition ends with the `@end` compiler directive.

NOTE *As mentioned earlier, you can also put your declaration inside an* `.m` *file. Hence, in your* `.m` *file you would then have both the* `@interface` *and* `@implementation` *directives.*

@class

If your class references another class defined in another file, you need to import the header file of that file before you can use it. Consider the following example where you have defined two classes — `SomeClass` and `AnotherClass`. If you are using an instance of `AnotherClass` from within `SomeClass`, you need to import the `AnotherClass.h` file, as in the following code snippet:

```
//--SomeClass.h--
#import <Foundation/Foundation.h>
#import "AnotherClass.h"

@interface SomeClass : NSObject {
    //---an object from AnotherClass---
    AnotherClass *anotherClass;
}

@end

//---AnotherClass.h---
#import <Foundation/Foundation.h>

@interface AnotherClass : NSObject {

}

@end
```

However, if within `AnotherClass` you want to create an instance of `SomeClass`, you will not be able to simply import `SomeClass.h` in `AnotherClass`, like this:

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

```
#import "AnotherClass.h"

@interface SomeClass : NSObject {
    //---an object from AnotherClass---
    AnotherClass *anotherClass;
}

@end


//---AnotherClass.h---
#import <Foundation/Foundation.h>
#import "SomeClass.h"      //---cannot simply import here---

@interface AnotherClass : NSObject {
    SomeClass *someClass;  //---using an instance of SomeClass---
}

@end
```

Doing so results in circular inclusion. To prevent that, Objective-C uses the `@class` compiler directive as a forward declaration to inform the compiler that the class you specified is a valid class. You usually use the `@class` compiler directive in the header file, and in the implementation file you can use the `@import` compiler directive to tell the compiler more about the content of the class that you are using.

Using the `@class` compiler directive, the program now looks like this:

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

@class AnotherClass;   //---forward declaration---

@interface SomeClass : NSObject {
    //---an object from AnotherClass---
    AnotherClass *anotherClass;
}

@end


//---AnotherClass.h---
#import <Foundation/Foundation.h>

@class SomeClass;     //---forward declaration---

@interface AnotherClass : NSObject {
    SomeClass *someClass;  //---using an instance of SomeClass---
}

@end
```

NOTE *Another notable reason to use forward declaration where possible is that it will reduce your compile times because the compiler does not need to traverse as many included header files and their includes, etc.*

Class Instantiation

To create an instance of a class, you typically use the `alloc` keyword (more on this in the Memory Management section) to allocate memory for the object and then return it to a variable of the class type:

```
SomeClass *someClass = [SomeClass alloc];
```

In Objective-C, you need to prefix an object name with the `*` character when you declare an object. If you are declaring a variable of primitive type (such as `float`, `int`, `CGRect`, `NSInteger`, and so on), the `*` character is not required. Here are some examples:

```
CGRect frame;    //---CGRect is a structure---
int number;      //---int is a primitive type---
NSString *str;   //---NSString is a class
```

Besides specifying the returning class type, you can also use the `id` type, like this:

```
id someClass = [SomeClass alloc];
id str;
```

The `id` type means that the variable can refer to any type of object and hence the `*` is implicitly implied.

Fields

Fields are the data members of objects. For example, the following code shows that `SomeClass` has three fields — `anotherClass`, `rate`, and `name`:

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

@class AnotherClass;   //---forward declaration---

@interface SomeClass : NSObject {
    //---an object from AnotherClass---
    AnotherClass *anotherClass;
    float rate;
    NSString *name;
}

@end

@end
```

Access Privileges

By default, the access privilege of all fields is `@protected`. However, the access privilege can also be `@public` or `@private`. The following list shows the various access privileges:

- `@private` — visible only to the class that declares it
- `@public` — visible to all classes
- `@protected` — visible to the class that declares it as well as to inheriting classes

Using the example shown in the previous section, if you now try to access the fields in `SomeClass` from another class, such as a View Controller, you will not be able to see them:

```
SomeClass *someClass = [SomeClass alloc];
someClass->rate = 5;                //---rate is declared protected---
someClass->name = @"Wei-Meng Lee";  //---name is declared protected---
```

NOTE *Observe that to access the fields in a class directly, you use the* `->` *operator.*

To make the `rate` and `name` visible outside the class, modify the `SomeClass.h` file by adding the `@public` compiler directive:

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

@class AnotherClass;   //---forward declaration---

@interface SomeClass : NSObject {
    //---an object from AnotherClass---
    AnotherClass *anotherClass;

@public
    float rate;

@public
    NSString *name;
}

@end
```

The following two statements would now be valid:

```
someClass->rate = 5;                //---rate is now declared public---
someClass->name = @"Wei-Meng Lee";  //---name is now declared public---
```

Although you can access the fields directly, doing so goes against the design principles of object-oriented programming's rule of encapsulation. A better way is to encapsulate the two fields you want to expose in properties. Refer to the "Properties" section later in this appendix.

Methods

Methods are functions that are defined in a class. Objective-C supports two types of methods — instance methods and class methods.

Instance methods can be called only using an instance of the class. Instance methods are prefixed with the minus sign (-) character.

Class methods can be invoked directly using the class name and do not need an instance of the class in order to work. Class methods are prefixed with the plus sign (+) character.

NOTE *In some programming languages, such as C# and Java, class methods are known as static methods.*

The following code sample shows `SomeClass` with three instance methods and one class method declared:

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

@class AnotherClass;   //---forward declaration---

@interface SomeClass : NSObject {
    //---an object from AnotherClass---
    AnotherClass *anotherClass;
    float rate;
    NSString *name;
}

//---instance methods---
-(void) doSomething;
-(void) doSomething:(NSString *) str;
-(void) doSomething:(NSString *) str withAnotherPara:(float) value;

//---class method---
+(void) alsoDoSomething;

@end
```

The following shows the implementation of the methods that were declared in the header file:

```
#import "SomeClass.h"

@implementation SomeClass

-(void) doSomething {
    //---implementation here---
}

-(void) doSomething:(NSString *) str {
    //---implementation here---
```

```
}

-(void) doSomething:(NSString *) str withAnotherPara:(float) value {
    //---implementation here---
}

+(void) alsoDoSomething {
    //---implementation here---
}

@end
```

To invoke the three instance methods, you first need to create an instance of the class and then call them using the instance created:

```
SomeClass *someClass = [SomeClass alloc];
[someClass doSomething];
[someClass doSomething:@"some text"];
[someClass doSomething:@"some text" withAnotherPara:9.0f];
```

Class methods can be called directly using the class name, as the following shows:

```
[SomeClass alsoDoSomething];
```

In general, you create instance methods when you need to perform some actions that are related to the particular instance of the class (that is, the object). For example, suppose you defined a class that represents the information of an employee. You may expose an instance method that allows you to calculate the overtime wage of an employee. In this case, you use an instance method because the calculation involves data specific to a particular employee object.

Class methods, on the other hand, are commonly used for defining helper methods. For example, you might have a class method called `GetOvertimeRate:` that returns the rates for working overtime. As all employees get the same rate for working overtime (assuming this is the case for your company), then there is no need to create instance methods, and thus a class method will suffice.

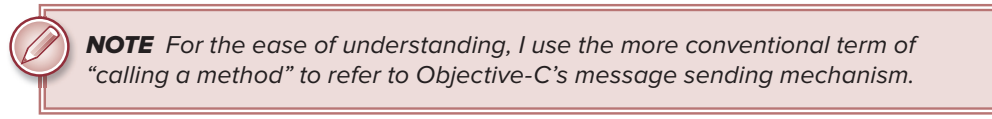
The next section shows how to call methods with a varying number of parameters.

Message Sending (Calling Methods)

In Objective-C, you use the following syntax to call a method:

```
[object method];
```

Strictly speaking, in Objective-C you do not call a method; you send a message to an object. The message to be passed to an object is resolved during runtime and is not enforced at compile time. This is why the compiler does not stop you from running your program even though you may have misspelled the name of a method. It does try to warn you that the target object may not respond to your message, though, because the target object will simply ignore the message. Figure D-1 shows the warning by the compiler when one of the parameters for the `UIAlertView`'s initializer is misspelled (the `cancelButtonsTitle:` should be `cancelButtonTitle:`).



NOTE *For the ease of understanding, I use the more conventional term of "calling a method" to refer to Objective-C's message sending mechanism.*

FIGURE D-1

Using the example from the previous section, the `doSomething` method has no parameter:

```
-(void) doSomething {
   //---implementation here---
}
```

Therefore, you can call it like this:

```
[someClass doSomething];
```

If a method has one or more inputs, you call it using the following syntax:

```
[object method:input1];                          //---one input---
[object method:input1 andSecondInput:input2];    //---two inputs---
```

The interesting thing about Objective-C is the way you call a method with multiple inputs. Using the earlier example:

```
-(void) doSomething:(NSString *) str withAnotherPara:(float) value {
    //---implementation here---
}
```

The name of the preceding method is `doSomething:withAnotherPara:`.

It is important to note the names of methods and to differentiate those that have parameters from those that do not. For example, `doSomething` refers to a method with no parameter, whereas `doSomething:` refers to a method with one parameter, and `doSomething:withAnotherPara:` refers to a method with two parameters. The presence or absence of colons in a method name dictates which method is invoked during runtime. This is important when passing method names as arguments, particularly when using the `@selector` (discussed in the Selectors section) notation to pass them to a delegate or notification event.

Method calls can also be nested, as the following example shows:

```
NSString *str = [[NSString alloc] initWithString:@"Hello World"];
```

Here, you first call the `alloc` class method of the `NSString` class and then call the `initWithString:` method of the returning result from the `alloc` method, which is of type `id`, a generic C type that Objective-C uses for an arbitrary object.

In general, you should not nest more than three levels because anything more than that makes the code difficult to read.

Properties

Properties allow you to expose your fields in your class so that you can control how values are set or returned. In the earlier example (in the Access Privileges section), you saw that you can directly access the fields of a class using the -> operator. However, this is not the ideal way and you should ideally expose your fields as properties.

Prior to Objective-C 2.0, programmers had to declare methods to make the fields accessible to other classes, like this:

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

@class AnotherClass;  //---forward declaration---

@interface SomeClass : NSObject {
    //---an object from AnotherClass---
    AnotherClass *anotherClass;
    float rate;
    NSString *name;
}

//---expose the rate field---
-(float) rate;                          //---get the value of rate---
-(void) setRate:(float) value;          //---set the value of rate

//---expose the name field---
-(NSString *) name;                    //---get the value of name---
-(void) setName:(NSString *) value;    //---set the value of name---

//---instance methods---
-(void) doSomething;
-(void) doSomething:(NSString *) str;
-(void) doSomething:(NSString *) str withAnotherPara:(float) value;

//---class method---
+(void) alsoDoSomething;

@end
```

These methods are known as *getters* and *setters* (or sometimes better known as *accessors* and *mutators*). The implementation of these methods may look like this:

```
#import "SomeClass.h"

@implementation SomeClass

-(float) rate {
    return rate;
}

-(void) setRate:(float) value {
```

```
    rate = value;
}

-(NSString *) name {
    return name;
}

-(void) setName:(NSString *) value {
    [value retain];
    [name release];
    name = value;
}

-(void) doSomething {
   //---implementation here---
}

-(void) doSomething:(NSString *) str {
    //---implementation here---
    NSLog(str);
}

-(void) doSomething:(NSString *) str withAnotherPara:(float) value {
    //---implementation here---
}

+(void) alsoDoSomething {
    //---implementation here---
}

@end
```

To set the value of these properties, you need to call the methods prefixed with the `set` keyword:

```
SomeClass *sc = [[SomeClass alloc] init];
[sc setRate:5.0f];
[sc setName:@"Wei-Meng Lee"];
```

Alternatively, you can use the dot notation introduced in Objective-C 2.0:

```
SomeClass *sc = [[SomeClass alloc] init];
sc.rate = 5;
sc.name = @"Wei-Meng Lee";
```

To obtain the values of properties, you can either call the methods directly or use the dot notation in Objective-C 2.0:

```
NSLog([sc name]); //---call the method---
NSLog(sc.name);   //---dot notation
```

To make a property read only, simply remove the method prefixed with the `set` keyword.

Notice that within the `setName:` method, you have various statements using the `retain` and `release` keywords. These keywords relate to memory management in Objective-C; you learn more about them in the "Memory Management" section, later in this appendix.

In Objective-C 2.0, you don't need to define getters and setters in order to expose fields as properties. You can do so via the `@property` and `@synthesize` compiler directives. Using the same example, you can use the `@property` to expose the `rate` and `name` fields as properties, like this:

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

@class AnotherClass;   //---forward declaration---

@interface SomeClass : NSObject {
    //---an object from AnotherClass---
    AnotherClass *anotherClass;
    float rate;
    NSString *name;
}

@property float rate;
@property (retain, nonatomic) NSString *name;

//---instance methods---
-(void) doSomething;
-(void) doSomething:(NSString *) str;
-(void) doSomething:(NSString *) str withAnotherPara:(float) value;

//---class method---
+(void) alsoDoSomething;

@end
```

The first `@property` statement defines `rate` to be a property. The second statement defines `name` as a property as well, but it also specifies the behavior of this property. In this case, it indicates the behavior as `retain` and `nonatomic`, which you learn more about in the section on memory management later in this appendix. In particular, `nonatomic` means that the property is not accessed in a thread-safe manner. This is alright if you are not writing multi-threaded applications. Most of the time, you will use the `retain` and `nonatomic` combination when declaring properties.

In the implementation file, rather than define the getter and setter methods, you can simply use the `@synthesize` keyword to get the compiler to automatically generate the getters and setters for you:

```
#import "SomeClass.h"

@implementation SomeClass

@synthesize rate, name;
```

As shown, you can combine several properties using a single `@synthesize` keyword. However, you can also separate them into individual statements:

```
@synthesize rate;
@synthesize name;
```

You can now use your properties as usual:

```
//---setting using setRate---
[sc setRate:5.0f];
[sc setName:@"Wei-Meng Lee"];

//---setting using dot notation---
sc.rate = 5;
sc.name = @"Wei-Meng Lee";

//---getting---
NSLog([sc name]); //---using the name method
NSLog(sc.name);   //---dot notation
```

To make a property read only, use the `readonly` keyword. The following statement makes the `name` property read only:

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

Initializers

When you create an instance of a class, you often initialize it at the same time. For example, in the earlier example (in the Class Instantiation section), you had this statement:

```
SomeClass *sc = [[SomeClass alloc] init];
```

The `alloc` keyword allocates memory for the object, and when an object is returned, the `init` method is called on the object to initialize the object. Recall that in `SomeClass`, you do not define a method named `init`. So where does the `init` method come from? It is actually defined in the `NSObject` class, which is the base class of most classes in Objective-C. The `init` method is known as an initializer.

If you want to create additional initializers, you can define methods that begin with the `init` word. (The use of the `init` word is more of a norm than a hard-and-fast rule.)

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

@class AnotherClass;   //---forward declaration---

@interface SomeClass : NSObject {
    //---an object from AnotherClass---
    AnotherClass *anotherClass;
    float rate;
    NSString *name;
}

@property float rate;
@property (retain, nonatomic) NSString *name;

//---instance methods---
-(void) doSomething;
```

```
-(void) doSomething:(NSString *) str;
-(void) doSomething:(NSString *) str withAnotherPara:(float) value;

//---class method---
+(void) alsoDoSomething;

- (id)initWithName:(NSString *) n;
- (id)initWithName:(NSString *) n andRate:(float) r;

@end
```

The preceding example contains two additional initializers: `initWithName:` and `initWithName:andRate:`. You can provide the implementations for the two initializers as follows:

```
#import "SomeClass.h"

@implementation SomeClass

@synthesize rate, name;

- (id)initWithName:(NSString *) n {
   return [self initWithName:n andRate:0.0f];
}

- (id)initWithName:(NSString *) n andRate:(float) r {
    if (self = [super init]) {
        self.name = n;
        self.rate = r;
    }
    return self;
}

//...
//...
```

Note that in the `initWithName:andRate:` initializer implementation, you first call the `init` initializer of the `super` (base) class so that its base class is properly initialized, which is necessary before you can initialize the current class:

```
- (id)initWithName:(NSString *) n andRate:(float) r {
    if (self = [super init]) {
        self.name = n;
        self.rate = r;
    }
    return self;
}
```

The rule for defining an initializer is simple: If a class is initialized properly, it should return a reference to `self` (hence the `id` type). If it fails, it should return `nil`.

For the `initWithName:` initializer implementation, notice that it calls the `initWithName:andRate:` initializer:

```
- (id)initWithName:(NSString *) n {
   return [self initWithName:n andRate:0.0f];
}
```

In general, if you have multiple initializers, each with different parameters, you should chain them by ensuring that they all call a single initializer that performs the call to the super class's `init` initializer. In Objective-C, the initializer that performs the call to the super class's `init` initializer is called the *designated initializer*.

NOTE *As a general guide, the designated initializer should be the one with the greatest number of parameters.*

To use the initializers, you can now call them during instantiation time:

```
SomeClass *sc1 = [[SomeClass alloc] initWithName:@"Wei-Meng Lee"
                                         andRate:35];
SomeClass *sc2 = [[SomeClass alloc] initWithName:@"Wei-Meng Lee"];
```

MEMORY MANAGEMENT

Memory management in Objective-C programming (especially for iPhone) is a very important topic that every iPhone developer needs to be aware of. As do all other popular languages, Objective-C supports garbage collection, which helps to remove unused objects when they go out of scope and hence releases memory that can be reused. However, because of the severe overhead involved in implementing garbage collection, the iPhone does not support garbage collection. This leaves you, the developer, to manually allocate and de-allocate the memory of objects when they are no longer needed.

This section discusses the various aspects of memory management on the iPhone.

Reference Counting

To help you allocate and de-allocate memory for objects, the iPhone OS uses a scheme known as *reference counting* to keep track of objects to determine whether they are still needed or can be disposed of. Reference counting basically uses a counter for each object, and as each object is created, the count increases by 1. When an object is released, the count decreases by 1. When the count reaches 0, the memory associated with the object is reclaimed by the OS.

In Objective-C, a few important keywords are associated with memory management. The following sections take a look at each of them.

alloc

The `alloc` keyword allocates memory for an object that you are creating. You have seen it in almost all exercises in this book. An example is as follows:

```
NSString *str = [[NSString alloc] initWithString:@"Hello"];
```

In this example, you are creating an `NSString` object and instantiating it with a default string. When the object is created, the reference count of that object is 1. Because you are the one creating it, the object belongs to you, and it is your responsibility to release the memory when you are done with it.

NOTE *See the "release" section for information on how to release an object.*

So how do you know when an object is owned, and by whom? Consider the following example:

```
NSString *str = [[NSString alloc] initWithString:@"Hello"];
NSString *str2 = str;
```

In this example, you use the `alloc` keyword for `str`, so you own `str`. Therefore, you need to release it when you no longer need it. However, `str2` is simply pointing to `str`, so you do not own `str2`, meaning that you need not release `str2` when you are done using it.

new

Besides using the `alloc` keyword to allocate memory for an object, you can also use the `new` keyword, like this:

```
NSString *str = [NSString new];
```

The `new` keyword is functionally equivalent to

```
NSString *str = [[NSString alloc] init];
```

As with the `alloc` keyword, using the `new` keyword makes you the owner of the object, so you need to release it when you are done with it.

retain

The `retain` keyword increases the reference count of an object by 1. Consider the previous example:

```
NSString *str = [[NSString alloc] initWithString:@"Hello"];
NSString *str2 = str;
```

In that example, you do not own `str2` because you do not use the `alloc` keyword on the object. When `str` is released, the `str2` will no longer be valid.

NOTE *How do you release* `str2`*, then? Well, it is autoreleased. See the "Convenience Method and Autorelease" section for more information.*

If you want to make sure that `str2` is available even if `str` is released, you need to use the `retain` keyword:

```
NSString *str = [[NSString alloc] initWithString:@"Hello"];
NSString *str2 = str;
[str2 retain];
[str release];
```

In the preceding case, the reference count for `str` is now 2. When you release `str`, `str2` will still be valid. When you are done with `str2`, you need to release it manually.

NOTE *As a general rule, if you own an object (using `alloc` or `retain`), you need to release it.*

release

When you are done with an object, you need to manually release it by using the `release` keyword:

```
NSString *str = [[NSString alloc] initWithString:@"Hello"];

//...do what you want with the object...

[str release];
```

When you use the `release` keyword on an object, it causes the reference count of that object to decrease by 1. When the reference count reaches 0, the memory used by the object is released.

One important aspect to keep in mind when using the `release` keyword is that you cannot release an object that is not owned by you. For example, consider the example used in the previous section:

```
NSString *str = [[NSString alloc] initWithString:@"Hello"];
NSString *str2 = str;
[str release];
[str2 release];  //---this is not OK as you do not own str2---
```

Attempting to release `str2` will result in a runtime error because you cannot release an object not owned by you. However, if you use the `retain` keyword to gain ownership of an object, you do need to use the `release` keyword:

```
NSString *str = [[NSString alloc] initWithString:@"Hello"];
NSString *str2 = str;
[str2 retain];
[str release];
[str2 release];  //---this is now OK as you now own str2---
```

Recall that earlier in the section on properties, you defined the `setName:` method, where you set the value of the `name` field:

```
-(void) setName:(NSString *) value {
    [value retain];
```

```
    [name release];
    name = value;
}
```

Notice that you first had to retain the `value` object, followed by releasing the `name` object and then finally assigning the `value` object to `name`. Why do you need to do that as opposed to the following?

```
-(void) setName:(NSString *) value {
    name = value;
}
```

Well, if you were using garbage collection, the preceding statement would be valid. However, because iPhone OS does not support garbage collection, the preceding statement will cause the original object referenced by the `name` object to be lost, thereby causing a memory leak. To prevent that leak, you first retain the `value` object to indicate that you wish to gain ownership of it; then you release the original object referenced by `name`. Finally, assign `value` to `name`:

```
[value retain];
[name release];
name = value;
```

Convenience Method and Autorelease

So far, you learned that all objects created using the `alloc` or `new` keywords are owned by you. Consider the following case:

```
NSString *str = [NSString stringWithFormat:@"%d", 4];
```

In this statement, do you own the `str` object? The answer is no, you don't. This is because the object is created using one of the *convenience methods* — static methods that are used for allocating and initializing objects directly. In the preceding case, you create an object but you do not own it. Because you do not own it, you cannot release it manually. In fact, objects created using this method are known as *autorelease* objects. All autorelease objects are temporary objects and are added to an *autorelease pool*. When the current method exits, all the objects contained within it are released. Autorelease objects are useful for cases in which you simply want to use some temporary variables and do not want to burden yourself with allocations and de-allocations.

The key difference between an object created using the `alloc` (or `new`) keyword and one created using a convenience method is that of ownership, as the following example shows:

```
NSString *str1 = [[NSString alloc] initWithFormat:@"%d", 4];
[str1 release]; //---this is ok because you own str1---

NSString *str2 = [NSString stringWithFormat:@"%d", 4];
[str2 release]; //---this is not ok because you don't own str2---
//---str2 will be removed automatically when the autorelease
// pool is activated---
```

UNDERSTANDING REFERENCE COUNTING USING AN ANALOGY

When you think of memory management using reference counting, it is always good to use a real-life analogy to put things into perspective.

Imagine a room in the library that you can reserve for studying purposes. Initially, the room is empty and hence the lights are off. When you reserve the room, the librarian increases a counter to indicate the number of persons using the room. This is similar to creating an object using the `alloc` keyword.

When you leave the room, the librarian decreases the counter, and if the counter is now 0, this means that the room is no longer being used and the lights can thus be switched off. This is similar to using the `release` keyword to release an object.

There may be times when you have booked the room and are the only one in the room (hence, the counter is 1) until a friend of yours comes along. He may simply come and visit you and therefore doesn't register with the librarian. Hence, the counter does not increase. Because he is just visiting you and hasn't booked the room, he has no rights to decide whether the lights should be switched off. This is similar to assigning an object to another variable without using the `alloc` keyword. In this case, if you leave the room (release), the lights will be switched off and your friend will have to leave.

Consider another situation in which you are using the room and another person also booked the room and shares it with you. In this case, the counter is now 2. If you leave the room, the counter goes down to 1, but the lights are still on because another person is in the room. This situation is similar when you create an object and assign it to another variable that uses the `retain` keyword. In such a situation, the object is released only when both objects release it.

If you want to take ownership of an object when using a convenience method, you can do so using the `retain` keyword:

```
NSString *str2 = [[NSString stringWithFormat:@"%d", 4] retain];
```

To release the object, you can use either the `autorelease` or `release` keyword. You learned earlier that the `release` keyword immediately decreases the reference count by 1 and that the object is immediately de-allocated from memory when the reference count reaches 0. In contrast, the `autorelease` keyword promises to decrease the reference count by 1, not *immediately*, but sometime later. It is like saying, "Well, I still need the object now, but later on I can let it go." The following code makes it clear:

```
NSString *str = [[NSString stringWithFormat:@"%d", 4] retain];
[str autorelease];  //you don't own it anymore; still available
NSlog(str);         //still accessible for now
```

NOTE *After you have autoreleased an object, do not release it anymore.*

Note that the statement

```
NSString *str2 = [NSString stringWithFormat:@"%d", 4];
```

has the same effect as

```
NSString *str2 = @"4";
```

Although autorelease objects seem to make your life simple by automatically releasing objects that are no longer needed, you have to be careful when using them. Consider the following example:

```
for (int i=0; i<=99999; i++){
    NSString *str = [NSString stringWithFormat:@"%d", i];
    //...
    //...
}
```

You are creating an `NSString` object for each iteration of the loop. Because the objects are not released until the function exits, you may well run out of memory before the autorelease pool (see next section) can kick in to release the objects.

One way to solve this dilemma is to use an autorelease pool, as discussed in the next section.

REFERENCE COUNTING: THE ANALOGY CONTINUES

Continuing with our analogy of the room in the library, imagine that you are about to sign out with the librarian when you realize that you have left your books in the room. You tell the librarian that you are done with the room and want to sign out now, but because you left your books in the room, you tell the librarian not to switch off the lights yet so that you can go back to get the books. At a later time, the librarian can switch off the lights at his or her own choosing. This is the behavior of autoreleased objects.

Autorelease Pools

All autorelease objects are temporary objects and are added to an *autorelease pool*. When the current method exits, all the objects contained within it are released. However, sometimes you want to control how the autorelease pool is emptied, rather than wait for it to be called by the OS. To do so, you can create an instance of the `NSAutoreleasePool` class, like this:

```
for (int i=0; i<=99999; i++){
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
```

```
        NSString *str1 = [NSString stringWithFormat:@"%d", i];
        NSString *str2 = [NSString stringWithFormat:@"%d", i];
        NSString *str3 = [NSString stringWithFormat:@"%d", i];
        //...
        //...
        [pool release];
    }
```

In this example, for each iteration of the loop, an `NSAutoreleasePool` object is created, and all the autorelease objects created within the loop — `str1`, `str2`, and `str3` — go into it. At the end of each iteration, the `NSAutoreleasePool` object is released so that all the objects contained within it are automatically released. This ensures that you have at most three autorelease objects in memory at any one time.

dealloc

You have learned that by using the `alloc` or the `new` keyword, you own the object that you have created. You have also seen how to release the objects you own using the `release` or `autorelease` keyword. So when is a good time for you to release them?

As a rule of thumb, you should release the objects as soon as you are done with them. So if you created an object in a method, you should release it before you exit the method. For properties, recall that you can use the `@property` compiler directive together with the `retain` keyword:

```
@property (retain, nonatomic) NSString *name;
```

Because the values of the property will be retained, it is important that you free it before you exit the application. A good place to do so is in the `dealloc` method of a class (such as a View Controller):

```
-(void) dealloc {
    [self.name release];    //---release the name property---
    [super dealloc];
}
```

The `dealloc` method of a class is fired whenever the reference count of its object reaches 0. Consider the following example:

```
SomeClass *sc1 = [[SomeClass alloc] initWithName:@"Wei-Meng Lee"
                                         andRate:35];
//...do something here…
[sc1 release];  //---reference count goes to 0; dealloc will be called---
```

The preceding example shows that when the reference count of `sc1` goes to 0 (when the `release` statement is called), the `dealloc` method defined within the class will be called. If you do not define this method in the class, its implementation in the base class will be called.

Memory Management Tips

Memory management is a tricky issue in iPhone programming. Although there are tools that you can use to test for memory leaks, this section presents some simple things you can do to detect memory problems that might affect your application.

First, ensure that you implement the `didReceiveMemoryWarning` method in your View Controller:

```
- (void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];
    //---insert code here to free unused objects---
    // Release any cached data, images, etc that aren't in use.
}
```

The `didReceiveMemoryWarning` method will be called whenever your iPhone runs out of memory. You should insert code in this method so that you can free resources/objects that you do not need.

In addition, you should also handle the `applicationDidReceiveMemoryWarning:` method in your application delegate:

```
- (void)applicationDidReceiveMemoryWarning:(UIApplication *)application {
    /*
     Free up as much memory as possible by purging cached
     data objects that can be recreated (or reloaded from
     disk) later.
     */
    //---insert code here to free unused objects---
}
```

In this method, you should stop all memory-intensive activities, such as audio and video playback. You should also remove all images cached in memory.

PROTOCOLS

In Objective-C, a *protocol* declares a programmatic interface that any class can choose to implement. A protocol declares a set of methods, and an adopting class may choose to implement one or more of its declared methods. The class that defines the protocol is expected to call the methods in the protocols that are implemented by the adopting class.

The easiest way to understand protocols is to examine the `UIAlertView` class. As you have experienced in the various chapters in this book, you can simply use the `UIAlertView` class by creating an instance of it and then calling its `show` method:

```
UIAlertView *alert = [[UIAlertView alloc]
                      initWithTitle:@"Hello"
                            message:@"This is an alert view"
                           delegate:self
                  cancelButtonTitle:@"OK"
                  otherButtonTitles:nil];
[alert show];
```

The preceding code displays an Alert view with one button — OK. Tapping the OK button automatically dismisses the Alert view. If you want to display additional buttons, you can set the `otherButtonTitles:` parameter like this:

```
UIAlertView *alert = [[UIAlertView alloc]
                      initWithTitle:@"Hello"
                            message:@"This is an alert view"
                           delegate:self
                  cancelButtonTitle:@"OK"
                  otherButtonTitles:@"Option 1", @"Option 2", nil];
[alert show];
```

The Alert view now displays three buttons — OK, Option 1, and Option 2. But how do you know which button was tapped by the user? You can determine this by handling the relevant method(s) that will be fired by the Alert view when the buttons are clicked. This set of methods is defined by the `UIAlertViewDelegate` protocol. This protocol defines the following methods:

- `alertView:clickedButtonAtIndex:`
- `willPresentAlertView:`
- `didPresentAlertView:`
- `alertView:willDismissWithButtonIndex:`
- `alertView:didDismissWithButtonIndex:`
- `alertViewCancel:`

If you want to implement any of the methods in the `UIAlertViewDelegate` protocol, you need to ensure that your class, in this case the View Controller, conforms to this protocol. A class conforms to a protocol using angle brackets (<>), like this:

```
@interface ObjCTestViewController : UIViewController
    <UIAlertViewDelegate> {  //---this class conforms to the
                             // UIAlertViewDelegate protocol---

}

@end
```

NOTE *To conform to more than one delegate, separate the protocols with commas, such as* `<UIAlertViewDelegate, UITableViewDataSource>`.

After the class conforms to a protocol, you can implement the method in your class:

```
- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {

    NSLog([NSString stringWithFormat:@"%d", buttonIndex]);

}
```

Delegate

In Objective-C, a delegate is just an object that has been assigned by another object as the object responsible for handling events. Consider the case of the `UIAlertView` example that you have seen previously:

```
UIAlertView *alert = [[UIAlertView alloc]
                      initWithTitle:@"Hello"
                            message:@"This is an alert view"
                           delegate:self
                  cancelButtonTitle:@"OK"
                  otherButtonTitles:@"Option 1", @"Option 2", nil];
[alert show];
```

The initializer of the `UIAlertView` class includes a parameter called the `delegate`. Setting this parameter to `self` means that the current object is responsible for handling all the events fired by this instance of the `UIAlertView` class. If you don't need to handle events fired by this instance, you can simply set it to `nil`:

```
UIAlertView *alert = [[UIAlertView alloc]
                      initWithTitle:@"Hello"
                            message:@"This is an alert view"
                           delegate:nil
                  cancelButtonTitle:@"OK"
                  otherButtonTitles:@"Option 1", @"Option 2", nil];
[alert show];
```

If you have multiple buttons on the Alert view and want to know which button was tapped, you need to handle the methods defined in the `UIAlertViewDelegate` protocol. You can either implement it in the same class in which the `UIAlertView` class was instantiated (as shown in the previous section), or create a new class to implement the method, like this:

```
//--SomeClass.m--
@implementation SomeClass

- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {

    NSLog([NSString stringWithFormat:@"%d", buttonIndex]);

}
@end
```

To ensure that the Alert view knows where to look for the method, create an instance of `SomeClass` and then set it as the delegate:

```
SomeClass *myDelegate = [[SomeClass alloc] init];
UIAlertView *alert = [[UIAlertView alloc]
                      initWithTitle:@"Hello"
                            message:@"This is an alert view"
                           delegate:myDelegate;
                  cancelButtonTitle:@"OK"
                  otherButtonTitles:@"Option 1", @"Option 2", nil];
[alert show];
```

SELECTORS

In Objective-C, a selector is the name used to select a method to execute for an object. It is used to identify a method. You have seen the use of a selector in some of the chapters in this book. Here is one of them:

```
//---create a Button view---
CGRect frame = CGRectMake(10, 50, 300, 50);
UIButton *button = [UIButton buttonWithType:UIButtonTypeRoundedRect];
button.frame = frame;
[button setTitle:@"Click Me, Please!"
        forState:UIControlStateNormal];
button.backgroundColor = [UIColor clearColor];
[button addTarget:self
           action:@selector(buttonClicked:)
 forControlEvents:UIControlEventTouchUpInside];
```

The preceding code shows that you are dynamically creating a `UIButton` object. In order to handle the event (for example, the `Touch Up Inside` event) raised by the button, you need to call the `addTarget:action:forControlEvents:` method of the `UIButton` class:

```
[button addTarget:self
           action:@selector(buttonClicked:)
 forControlEvents:UIControlEventTouchUpInside];
```

The `action:` parameter takes in an argument of type `SEL` (selector). In the preceding code, you pass in the name of the method that you have defined — `buttonClicked:` — which is defined within the class:

```
-(IBAction) buttonClicked: (id) sender {
    //...
}
```

Alternatively, you can create an object of type `SEL` and then instantiate it by using the `NSSelectorFromString` function (which takes in a string containing the method name):

```
NSString *nameOfMethod = @"buttonClicked:";
SEL methodName = NSSelectorFromString(nameOfMethod);
```

The call to the `addTarget:action:forControlEvents:` method now looks like this:

```
[button addTarget:self
           action:methodName
 forControlEvents:UIControlEventTouchUpInside];
```

NOTE *When naming a selector, be sure to specify the full name of the method. For example, if a method name has one or more parameters, you need to add a ":" in the sector, such as:*

```
NSString *nameOfMethod = @"someMethod:withPara1:andPara2:";
```

NOTE *Because Objective-C is an extension of C, it is common to see C functions interspersed throughout your Objective-C application. C functions use the parentheses () to pass in arguments for parameters.*

CATEGORIES

A category in Objective-C allows you to add methods to an existing class without the need to subclass it. You can also use a category to override the implementation of an existing class.

NOTE *In some languages (such as C#), a category is known as an extension method.*

As an example, imagine that you want to test whether a string contains a valid e-mail address. You can add an `isEmail` method to the `NSString` class so that you can call the `isEmail` method on any `NSString` instance, like this:

```
NSString *email = @"weimenglee@gmail.com";
if ([email isEmail]) {
    //...
}
```

To do so, you can simply create a new class file and code it as follows:

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

//---NSString is the class you are extending---
@interface NSString (stringUtils)

//---the method you are adding to the NSString class---
-(BOOL) isEmail;

@end
```

Basically, it looks the same as declaring a new class except that it does not inherit from any other class. The `stringUtils` is a name that identifies the category you are adding, and you can use any name you want.

Next, you need to implement the method(s) you are adding:

```
//--Utils.m--
#import "Utils.h"

@implementation NSString (Utilities)

- (BOOL) isEmail {
```

```
    NSString *emailRegEx =
    @"(?:[a-z0-9!#$%\\&'*+/=?\\^_`{|}~-]+(?:\\.[a-z0-9!#$%\\&'*+/=?\\^_`{|}"
    @"~-]+)*|\"(?:[\\x01-\\x08\\x0b\\x0c\\x0e-\\x1f\\x21\\x23-\\x5b\\x5d-\\"
    @"x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-\\x7f])*\")@(?:(?:[a-z0-9](?:[a-"
    @"z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\\[(?:(?:25[0-5"
    @"]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-"
    @"9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\\x01-\\x08\\x0b\\x0c\\x0e-\\x1f\\x21"
    @"-\\x5a\\x53-\\x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-\\x7f])+)\\])";

    NSPredicate *regExPredicate = [NSPredicate
                                   predicateWithFormat:@"SELF MATCHES %@",
                                   emailRegEx];

    return [regExPredicate evaluateWithObject:self];
}

@end
```

NOTE *The code for validating an e-mail address using regular expression is adapted from* `http://cocoawithlove.com/2009/06/verifying-that-string-is-email-address.html`.

You can then test for the validity of an e-mail address using the newly added method:

```
NSString *email = @"weimenglee@gmail.com";
if ([email isEmail])
    NSLog(@"Valid email");
else
    NSLog(@"Invalid email");
```