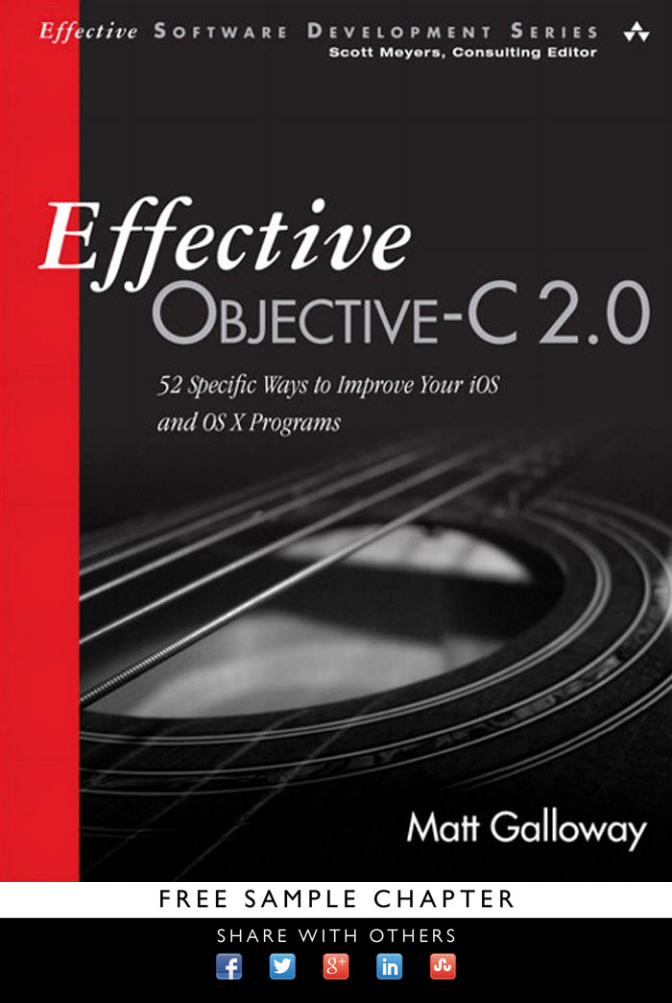
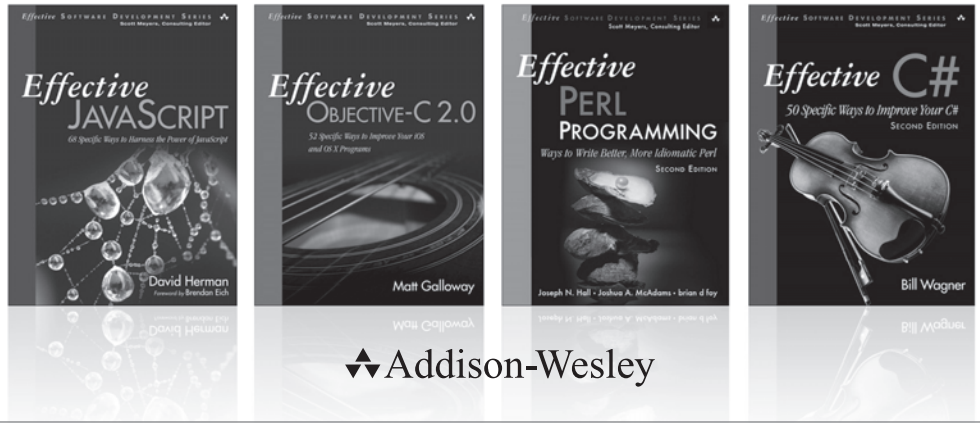

Effective Software Development Series
Scott Meyers, Consulting Editor
Effective
Objective-C 2.0
52 Specific Ways to Improve Your iOS and OS X Programs
Matt Galloway
FREE SAMPLE CHAPTER
SHARE WITH OTHERS

Effective Objective-C 2.0

The Effective Software Development Series

Scott Meyers, Consulting Editor



Visit **informit.com/esds** for a complete list of available publications.

The Effective Software Development Series provides expert advice on all aspects of modern software development. Titles in the series are well written, technically sound, and of lasting value. Each describes the critical things experts always do — or always avoid — to produce outstanding software.

Scott Meyers, author of the best-selling books *Effective C++* (now in its third edition), *More Effective C++*, and *Effective STL* (all available in both print and electronic versions), conceived of the series and acts as its consulting editor. Authors in the series work with Meyers to create essential reading in a format that is familiar and accessible for software developers of every stripe.

Effective Objective-C 2.0

52 SPECIFIC WAYS TO IMPROVE YOUR iOS AND OS X PROGRAMS

Matt Galloway

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The Cataloging-in-Publication Data is on file with the Library of Congress.

ISBN-13: 978-0-321-91701-0
ISBN-10: 0-321-91701-4
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing, May 2013

Editor-in-Chief
Mark Taub

Acquisitions Editor
Trina MacDonald

Development Editor
Chris Zahn

Managing Editor
John Fuller

Project Editor
Elizabeth Ryan

Packager
Vicki Rowland

Copy Editor
Evelyn W. Pyle

Indexer
Sharon Hilgenberg

Proofreader
Archie Brodsky

Technical Reviewers
Anthony Herron
Cesare Rocchi
Chris Wagner

Editorial Assistant
Olivia Bassegio

Cover Designer
Chuti Prasertsith

Compositor
Vicki Rowland

To Rosie

This page intentionally left blank

Contents

This page intentionally left blank

Preface

Objective-C is verbose. Objective-C is clunky. Objective-C is ugly. I have heard all these things said about Objective-C. On the contrary, I find it elegant, flexible, and beautiful. However, to get it to be these things, you must understand not only the fundamentals but also the quirks, pitfalls, and intricacies: the topic of this book.

About This Book

This book doesn't teach you the syntax of Objective-C. It is assumed that you know this already. Instead, this book teaches you how to use the language to its full potential to write good code. Objective-C is extremely dynamic, thanks to having its roots in Smalltalk. Much of the work that's usually done by a compiler in other languages ends up being done by the runtime in Objective-C. This leads to a potential for code to function fine during testing but to break in strange ways later down the line in production, perhaps when processing invalid data. Avoiding these problems by writing good code in the first place is, of course, the best solution.

Many of the topics are not, strictly speaking, related to core Objective-C. Reference is made to items found in system libraries, such as Grand Central Dispatch, which is part of libdispatch. Similarly, many classes from the Foundation framework are referred to, not least the root class, NSObject, because developing with modern Objective-C means developing for Mac OS X or iOS. When developing for either, you will undoubtedly be using the system frameworks, collectively known as Cocoa and Cocoa Touch, respectively.

Since the rise of iOS, developers have been flocking to join the ranks of Objective-C development. Some of these developers are new to programming, some have come from Java or C++ backgrounds, and some have come from web-development backgrounds. In any case, all developers should take the time to learn how to use a language effectively.

Doing so will yield code that is more efficient, easier to maintain, and less likely to contain bugs.

Even though I have been writing this book for only around six months, it has been years in the making. I bought an iPod Touch on a whim; then, when the first SDK for it was released, I decided to have a play with development. That led to me build my first "app," which I released as Subnet Calc, which immediately got many more downloads than I could have imagined. I became certain that my future was in this beautiful language I had come to know. Since then, I have been researching Objective-C, regularly blogging about it on my web site, www.galloway.me.uk/. I am most interested in the inner workings, such as the guts of blocks and how ARC works. When I got the opportunity to write a book about this language, I jumped at the chance.

In order to get the full potential from this book, I encourage you to jump around it, hunting for the topics that are of most interest or relevant to what you're working on right now. Each item can be read individually, and you can use the cross-references to go to related topics. Each chapter collates items that are related, so you can use the chapter headings to quickly find items relevant to a certain language feature.

Audience for This Book

This book is aimed at developers who wish to further their knowledge of Objective-C and learn to write code that will be maintainable, efficient, and less likely to contain bugs. Even if you are not already an Objective-C developer but come from another object-oriented language, such as Java or C++, you should still be able to learn. In this case, reading about the syntax of Objective-C first would be prudent.

What This Book Covers

It is not the aim of this book to teach the basics of Objective-C, which you can learn from many other books and resources. Instead, this book teaches how to use the language effectively. The book comprises Items, each of which is a bite-sized chunk of information. These Items are logically grouped into topic areas, arranged as follows:

- **Chapter 1: Accustoming Yourself to Objective-C**
  - Core concepts relating to the language in general are featured here.
- **Chapter 2: Objects, Messaging, and the Runtime**
  - Important features of any object-oriented language are how objects relate to one another and how they interact. This chapter deals with these features and delves into parts of the runtime.

- **Chapter 3: Interface and API Design**
  - Code is rarely written once and never reused. Even if it is not released to the wider community, you will likely use your code in more than one project. This chapter explains how to write classes that feel right at home in Objective-C.
- **Chapter 4: Protocols and Categories**
  - Protocols and categories are both important language features to master. Effective use of them can make your code much easier to read, more maintainable, and less prone to bugs. This chapter helps you achieve mastery.
- **Chapter 5: Memory Management**
  - Objective-C's memory-management model uses reference counting, which has long been a sticky point for beginners, especially if they have come from a background of a language that uses a garbage collector. The introduction of Automatic Reference Counting (ARC) has made life easier, but you need to be aware of a lot of important things to ensure that you have a correct object model that doesn't suffer from leaks. This chapter fosters awareness of common memory-management pitfalls.
- **Chapter 6: Blocks and Grand Central Dispatch**
  - Blocks are lexical closures for C, introduced by Apple. Blocks are commonly used in Objective-C to achieve what used to involve much boilerplate code and introduced code separation. Grand Central Dispatch (GCD) provides a simple interface to threading. Blocks are seen as GCD tasks that can be executed, perhaps in parallel, depending on system resources. This chapter enables you to make the most from these two core technologies.
- **Chapter 7: The System Frameworks**
  - You will usually be writing Objective-C code for Mac OS X or iOS. In those cases, you will have the full system frameworks stack at your disposal: Cocoa and Cocoa Touch, respectively. This chapter gives a brief overview of the frameworks and delves into some of their classes.

If you have any questions, comments, or remarks about this book, I encourage you to contact me. You can find my full contact details on the web site for this book at www.effectiveobjectivec.com.

This page intentionally left blank

Acknowledgments

When asked whether I would like to write a book about Objective-C, I instantly became excited. I had already read other books in this series and knew that the task of creating one for Objective-C would be a challenge. But with the help of many people, this book became a reality.

Much inspiration for this book has come from the many excellent blogs that are dedicated to Objective-C. Mike Ash, Matt Gallagher, and "bbum" are a few of the individuals whose blogs I read. These blogs have helped me over the years to gain a deeper understanding of the language. *NSHipster* by Mattt Thompson has also provided excellent articles that gave me food for thought while compiling this book. Finally, the excellent documentation provided by Apple has also been extremely useful.

I would not have been in a position to write this book had it not been for the excellent mentoring and knowledge transfer that happened while I was working at MX Telecom. Matthew Hodgson in particular gave me the opportunity to develop the company's first iOS application, building on top of a mature C++ code base. The knowledge I picked up from this project formed the basis of much of my subsequent work.

Over the years, I have had many excellent colleagues with whom I have always stayed in touch either for academic reasons or purely just being there for a beer and a chat. All have helped me while writing this book.

I've had a fantastic experience with the team from Pearson. Trina MacDonald, Olivia Basegio, Scott Meyers, and Chris Zahn have all provided help and encouragement when required. They have provided the tools for me to get the book written without distraction and answered my queries when necessary.

The technical editors I have had the pleasure of working with have been incredibly helpful. Their eagle eyes have pushed the content of the book to be the very best. They should all be proud of the level of detail they used when analyzing the manuscript.

Finally, I could not have written this book without the understanding and support from Helen. Our first child was born the day I was supposed to start writing, so I naturally postponed for a short time. Both Helen and Rosie have been fantastic at keeping me going throughout.

About the Author

Matt Galloway is an iOS developer from London, UK. He graduated from the University of Cambridge, Pembroke College, in 2007, having completed an M.Eng. degree, specializing in electrical and information sciences. Since then, he has been programming, mostly in Objective-C. He has been developing for iOS ever since the first SDK was released. You'll find him on Twitter as @mattjgalloway, and he is a regular contributor to *Stack Overflow* (http://stackoverflow.com).

This page intentionally left blank

1 Accustoming Yourself to Objective-C

Objective-C brings object-oriented features to C through an entirely new syntax. Often described as verbose, Objective-C syntax makes use of a lot of square brackets and isn't shy about using extremely long method names. The resulting source code is very readable but is often difficult for C++ or Java developers to master.

Writing Objective-C can be learned quickly but has many intricacies to be aware of and features that are often overlooked. Similarly, some features are abused or not fully understood, yielding code that is difficult to maintain or to debug. This chapter covers fundamental topics; subsequent chapters cover specific areas of the language and associated frameworks.

Item 1: Familiarize Yourself with Objective-C's Roots

Objective-C is similar to other object-oriented languages, such as C++ and Java, but also differs in many ways. If you have experience in another object-oriented language, you'll understand many of the paradigms and patterns used. However, the syntax may appear alien because it uses a messaging structure rather than function calling. Objective-C evolved from Smalltalk, the origin of messaging. The difference between messaging and function calling looks like this:

```
// Messaging (Objective-C)
Object *obj = [Object new];
[obj performWith:parameter1 and:parameter2];

// Function calling (C++)
Object *obj = new Object;
obj->perform(parameter1, parameter2);
```

The key difference is that in the messaging structure, the runtime decides which code gets executed. With function calling, the compiler

decides which code will be executed. When polymorphism is introduced to the function-calling example, a form of runtime lookup is involved through what is known as a virtual table. But with messaging, the lookup is always at runtime. In fact, the compiler doesn't even care about the type of the object being messaged. That is looked up at runtime as well, through a process known as dynamic binding, covered in more detail in Item 11.

The Objective-C runtime component, rather than the compiler, does most of the heavy lifting. The runtime contains all the data structures and functions that are required for the object-oriented features of Objective-C to work. For example, the runtime includes all the memory-management methods. Essentially, the runtime is the set of code that glues together all your code and comes in the form of a dynamic library to which your code is linked. Thus, whenever the runtime is updated, your application benefits from the performance improvements. A language that does more work at compile time needs to be recompiled to benefit from such performance improvements.

Objective-C is a superset of C, so all the features in the C language are available when writing Objective-C. Therefore, to write effective Objective-C, you need to understand the core concepts of both C and Objective-C. In particular, understanding the memory model of C will help you to understand the memory model of Objective-C and why reference counting works the way it does. This involves understanding that a pointer is used to denote an object in Objective-C. When you declare a variable that is to hold a reference to an object, the syntax looks like this:

```
NSString *someString = @"The string";
```

This syntax, mostly lifted straight from C, declares a variable called `someString` whose type is `NSString*`. This means that it is a pointer to an `NSString`. All Objective-C objects must be declared in this way because the memory for objects is always allocated in heap space and never on the stack. It is illegal to declare a stack-allocated Objective-C object:

```
NSString stackString;
// error: interface type cannot be statically allocated
```

The `someString` variable points to some memory, allocated in the heap, containing an `NSString` object. This means that creating another variable pointing to the same location does not create a copy but rather yields two variables pointing to the same object:

```
NSString *someString = @"The string";
NSString *anotherString = someString;
```

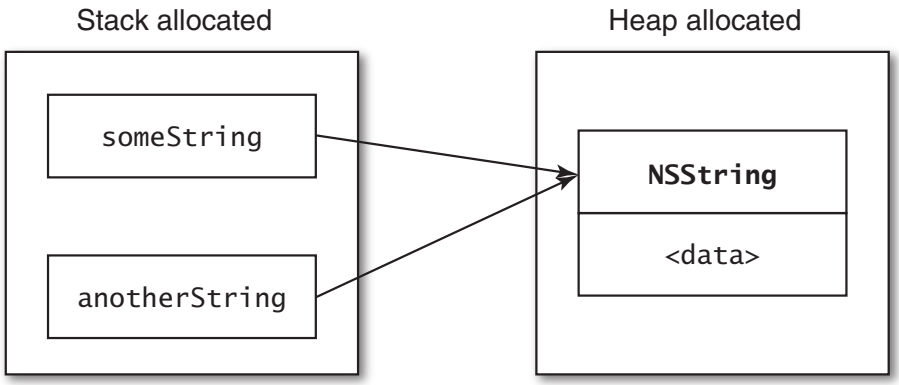


Figure 1.1 Memory layout showing a heap-allocated `NSString` instance and two stack-allocated pointers to it

There is only one `NSString` instance here, but two variables are pointing to the same instance. These two variables are of type `NSString*`, meaning that the current stack frame has allocated 2 bits of memory the size of a pointer (4 bytes for a 32-bit architecture, 8 bytes for a 64-bit architecture). These bits of memory will contain the same value: the memory address of the `NSString` instance.

Figure 1.1 illustrates this layout. The data stored for the `NSString` instance includes the bytes needed to represent the actual string.

The memory allocated in the heap has to be managed directly, whereas the stack-allocated memory to hold the variables is automatically cleaned up when the stack frame on which they are allocated is popped.

Memory management of the heap memory is abstracted away by Objective-C. You do not need to use `malloc` and `free` to allocate and deallocate the memory for objects. The Objective-C runtime abstracts this out of the way through a memory-management architecture known as reference counting (see Item 29).

Sometimes in Objective-C, you will encounter variables that don't have a * in the definition and might use stack space. These variables are not holding Objective-C objects. An example is `CGRect`, from the CoreGraphics framework:

```
CGRect frame;
frame.origin.x = 0.0f;
frame.origin.y = 10.0f;
frame.size.width = 100.0f;
```

```
frame.size.height = 150.0f;
```

A CGRect is a C structure, defined like so:

```
struct CGRect {
  CGPoint origin;
  CGSize size;
};
typedef struct CGRect CGRect;
```

These types of structures are used throughout the system frameworks, where the overhead of using Objective-C objects could affect performance. Creating objects incurs overhead that using structures does not, such as allocating and deallocating heap memory. When nonobject types (int, float, double, char, etc.) are the only data to be held, a structure, such as CGRect, is usually used.

Before embarking on writing anything in Objective-C, I encourage you to read texts about the C language and become familiar with the syntax. If you dive straight into Objective-C, you may find certain parts of the syntax confusing.

Things to Remember

- Objective-C is a superset of C, adding object-oriented features. Objective-C uses a messaging structure with dynamic binding, meaning that the type of an object is discovered at runtime. The runtime, rather than the compiler, works out what code to run for a given message.
- Understanding the core concepts of C will help you write effective Objective-C. In particular, you need to understand the memory model and pointers.

Item 2: Minimize Importing Headers in Headers

Objective-C, just like C and C++, makes use of header files and implementation files. When a class is written in Objective-C, the standard approach is to create one of each of these files named after the class, suffixed with .h for the header file and .m for the implementation file. When you create a class, it might end up looking like this:

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

@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
```

```
@end

// EOCPerson.m
#import "EOCPerson.h"

@implementation EOCPerson
// Implementation of methods
@end
```

The importing of `Foundation.h` is required pretty much for all classes you will ever make in Objective-C. Either that, or you will import the base header file for the framework in which the class's superclass lives. For example, if you were creating an iOS application, you would subclass `UIViewController` often. These classes' header files will import `UIKit.h`.

As it stands, this class is fine. It imports the entirety of Foundation, but that doesn't matter. Given that this class inherits from a class that's part of Foundation, it's likely that a large proportion of it will be used by consumers of `EOCPerson`. The same goes for a class that inherits from `UIViewController`. Its consumers will make use of most of `UIKit`.

As time goes on, you may create a new class called `EOCEmployer`. Then you decide that an `EOCPerson` instance should have one of those. So you go ahead and add a property for it:

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

@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, strong) EOCEmployer *employer;
@end
```

A problem with this, though, is that the `EOCEmployer` class is not visible when compiling anything that imports `EOCPerson.h`. It would be wrong to mandate that anyone importing `EOCPerson.h` must also import `EOCEmployer.h`. So the common thing to do is to add the following at the top of `EOCPerson.h`:

```
#import "EOCEmployer.h"
```

This would work, but it's bad practice. To compile anything that uses `EOCPerson`, you don't need to know the full details about what an `EOCEmployer` is. All you need to know is that a class called `EOCEmployer` exists. Fortunately, there is a way to tell the compiler this:

```
@class EOCEmployer;
```

This is called forward declaring the class. The resulting header file for EOCPerson would look like this:

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

@class EOCEmployer;

@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, strong) EOCEmployer *employer;
@end
```

The implementation file for EOCPerson would then need to import the header file of EOCEmployer, as it would need to know the full interface details of the class in order to use it. So the implementation file would end up looking like this:

```
// EOCPerson.m
#import "EOCPerson.h"
#import "EOCEmployer.h"

@implementation EOCPerson
// Implementation of methods
@end
```

Deferring the import to where it is required enables you to limit the scope of what a consumer of your class needs to import. In the example, if EOCEmployer.h were imported in EOCPerson.h, anything importing EOCPerson.h would also import all of EOCEmployer.h. If the chain of importing continues, you could end up importing a lot more than you bargained for, which will certainly increase compile time.

Using forward declaration also alleviates the problem of both classes referring to each other. Consider what would happen if EOCEmployer had methods to add and remove employees, defined like this in its header file:

```
- (void)addEmployee:(EOCPerson*)person;
- (void)removeEmployee:(EOCPerson*)person;
```

This time, the EOCPerson class needs to be visible to the compiler, for the same reasons as in the opposite case. However, achieving this by importing the other header in each header would create a chicken-and-egg situation. When one header is parsed, it imports the other,

which imports the first. The use of `#import` rather than `#include` doesn't end in an infinite loop but does mean that one of the classes won't compile correctly. Try it for yourself if you don't believe me!

Sometimes, though, you need to import a header in a header. You must import the header that defines the superclass from which you are inheriting. Similarly, if you declare any protocols that your class conforms to, they have to be fully defined and not forward declared. The compiler needs to be able to see the methods the protocol defines rather than simply that a protocol does exist from a forward declaration.

For example, suppose that a rectangle class inherits from a shape class and conforms to a protocol allowing it to be drawn:

```
// EOCRectangle.h
#import "EOCShape.h"
#import "EOCDrawable.h"

@interface EOCRectangle : EOCShape <EOCDrawable>
@property (nonatomic, assign) float width;
@property (nonatomic, assign) float height;
@end
```

The extra import is unavoidable. For such protocols, it is prudent to put them in their own header file for this reason. If the `EOCDrawable` protocol were part of a larger header file, you'd have to import all of that, thereby creating the same dependency and extra compilation-time problems as described before.

That said, not all protocols, such as delegate protocols (see Item 23), need to go in their own files. In such cases, the protocol makes sense only when defined alongside the class for which it is a delegate. In these cases, it is often best to declare that your class implements the delegate in the class-continuation category (see Item 27). This means that the import of the header containing the delegate protocol can go in the implementation file rather than in the public header file.

When writing an import into a header file, always ask yourself whether it's really necessary. If the import can be forward declared, prefer that. If the import is for something used in a property, instance variable, or protocol conformance and can be moved to the class-continuation category (see Item 27), prefer that. Doing so will keep compile time as low as possible and reduce interdependency, which can cause problems with maintenance or with exposing only parts of your code in a public API should ever you want to do that.

Things to Remember

- Always import headers at the very deepest point possible. This usually means forward declaring classes in a header and importing their corresponding headers in an implementation. Doing so avoids coupling classes together as much as possible.
- Sometimes, forward declaration is not possible, as when declaring protocol conformance. In such cases, consider moving the protocol-conformance declaration to the class-continuation category, if possible. Otherwise, import a header that defines only the protocol.

Item 3: Prefer Literal Syntax over the Equivalent Methods

While using Objective-C, you will come across a few classes all the time. They are all part of the Foundation framework. Although technically, you do not have to use Foundation to write Objective-C code, you usually do in practice. The classes are `NSString`, `NSNumber`, `NSArray`, and `NSDictionary`. The data structures that each represent are self-explanatory.

Objective-C is well known for having a verbose syntax. That's true. However, ever since Objective-C 1.0, there has been a very simple way to create an `NSString` object. It is known as a string literal and looks like this:

```
NSString *someString = @"Effective Objective-C 2.0";
```

Without this type of syntax, creating an `NSString` object would require allocating and initializing an `NSString` object in the usual `alloc` and then `init` method call. Fortunately, this syntax, known as literals, has been extended in recent versions of the compiler to cover `NSNumber`, `NSArray`, and `NSDictionary` instances as well. Using the literal syntax reduces source code size and makes it much easier to read.

Literal Numbers

Sometimes, you need to wrap an integer, floating-point, or Boolean value in an Objective-C object. You do so by using the `NSNumber` class, which can handle a range of number types. Without literals, you create an instance like this:

```
NSNumber *someNumber = [NSNumber numberWithInt:1];
```

This creates a number that is set to the integer 1. However, using literals makes this cleaner:

```
NSNumber *someNumber = @1;
```

As you can see, the literal syntax is much more concise. But there's more to it than that. The syntax also covers all the other types of data that NSNumber instances can represent. For example:

```
NSNumber *intNumber = @1;
NSNumber *floatNumber = @2.5f;
NSNumber *doubleNumber = @3.14159;
NSNumber *boolNumber = @YES;
NSNumber *charNumber = @'a';
```

The literal syntax also works for expressions:

```
int x = 5;
float y = 6.32f;
NSNumber *expressionNumber = @(x * y);
```

Making use of literals for numbers is extremely useful. Doing so makes using NSNumber objects much clearer, as the bulk of the declaration is the value rather than superfluous syntax.

Literal Arrays

Arrays are a commonly used data structure. Before literals, you would create an array as follows:

```
NSArray *animals =
    [NSArray arrayWithObjects:@"cat", @"dog",
                              @"mouse", @"badger", nil];
```

Using literals, however, requires only the following syntax:

```
NSArray *animals = @[@"cat", @"dog", @"mouse", @"badger"];
```

But even though this is a much simpler syntax, there's more to it than that with arrays. A common operation on an array is to get the object at a certain index. This also is made easier using literals. Usually, you would use the objectAtIndex: method:

```
NSString *dog = [animals objectAtIndex:1];
```

With literals, it's a matter of doing the following:

```
NSString *dog = animals[1];
```

This is known as subscripting, and just like the rest of the literal syntax, it is more concise and much easier to see what's being done. Moreover, it looks very similar to the way arrays are indexed in other languages.

However, you need to be aware of one thing when creating arrays using the literal syntax. If any of the objects is nil, an exception is

thrown, since literal syntax is really just syntactic sugar around creating an array and then adding all the objects within the square brackets. The exception you get looks like this:

```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '***
-[__NSPlaceholderArray initWithObjects:count:]: attempt to
insert nil object from objects[0]'
```

This brings to light a common problem when switching to using literals. The following code creates two arrays, one in each syntax:

```
id object1 = /* … */;
id object2 = /* … */;
id object3 = /* … */;

NSArray *arrayA = [NSArray arrayWithObjects:
                        object1, object2, object3, nil];
NSArray *arrayB = @[object1, object2, object3];
```

Now consider the scenario in which `object1` and `object3` point to valid Objective-C objects, but `object2` is nil. The literal array, `arrayB`, will cause the exception to be thrown. However, `arrayA` will still be created but will contain only `object1`. The reason is that the `arrayWithObjects:` method looks through the variadic arguments until it hits nil, which is sooner than expected.

This subtle difference means that literals are much safer. It's much better that an exception is thrown, causing a probable application crash, rather than creating an array having fewer than the expected number of objects in it. A programmer error most likely caused nil to be inserted into the array, and the exception means that the bug can be found more easily.

Literal Dictionaries

Dictionaries provide a map data structure in which you add key-value pairs. Like arrays, dictionaries are commonly used in Objective-C code. Creating one used to look like this:

```
NSDictionary *personData =
    [NSDictionary dictionaryWithObjectsAndKeys:
        @"Matt", @"firstName",
        @"Galloway", @"lastName",
        [NSNumber numberWithInt:28], @"age",
        nil];
```

This is rather confusing, because the order is <object>, <key>, <object>, <key>, and so on. However, you usually think about dictionaries the other way round, as in key to object. Therefore, it doesn't read particularly well. However, literals once again make the syntax much clearer:

```
NSDictionary *personData =
    @{@"firstName" : @"Matt",
      @"lastName" : @"Galloway",
      @"age" : @28};
```

This is much more concise, and the keys are before the objects, just as you'd expect. Also note that the literal number in the example shows where literal numbers are useful. The objects and keys have to all be Objective-C objects, so you couldn't store the integer 28; instead, it must be wrapped in an `NSNumber` instance. But the literal syntax means that it's simply one extra character.

Just like arrays, the literal syntax for dictionaries suffers from an exception being thrown if any values are nil. However, for the same reason, this is a good thing. It means that instead of creating a dictionary with missing values, owing to the `dictionaryWithObjectsAndKeys:` method stopping at the first nil, an exception is thrown.

Also similar to arrays, dictionaries can be accessed using literal syntax. The old way of accessing a value for a certain key is as follows:

```
NSString *lastName = [personData objectForKey:@"lastName"];
```

The equivalent literal syntax is:

```
NSString *lastName = personData[@"lastName"];
```

Once again, the amount of superfluous syntax is reduced, leaving an easy-to-read line of code.

Mutable Arrays and Dictionaries

In the same way that you can access indexes in an array or keys in a dictionary through subscripting, you can also set them if the object is mutable. Setting through the normal methods on mutable arrays and dictionaries looks like this:

```
[mutableArray replaceObjectAtIndex:1 withObject:@"dog"];
[mutableDictionary setObject:@"Galloway" forKey:@"lastName"];
```

Setting through subscripting looks like this:

```
mutableArray[1] = @"dog";
mutableDictionary[@"lastName"] = @"Galloway";
```

Limitations

A minor limitation with the literal syntax is that with the exception of strings, the class of the created object must be the one from the Foundation framework. There's no way to specify your own custom subclass that should be created instead. If you wanted to create an instance of your own custom subclass, you'd need to use the nonliteral syntax. However, since `NSArray`, `NSDictionary`, and `NSNumber` are class clusters (see Item 9), they are rarely subclassed, as it's nontrivial to do so. Also, the standard implementations are usually good enough. Strings can use a custom class, but it must be changed through a compiler option. Use of this option is discouraged because unless you know what you are doing, you will always want to use `NSString` anyway.

Also, in the case of strings, arrays, and dictionaries, only immutable variants can be created with the literal syntax. If a mutable variant is required, a mutable copy must be taken, like so:

```
NSMutableArray *mutable = [@[@1, @2, @3, @4, @5] mutableCopy];
```

This adds an extra method call, and an extra object is created, but the benefits of using the literal syntax outweigh these disadvantages.

Things to Remember

- Use the literal syntax to create strings, numbers, arrays, and dictionaries. It is clearer and more succinct than creating them using the normal object-creation methods.
- Access indexes of an array or keys in a dictionary through the subscripting methods.
- Attempting to insert nil into an array or dictionary with literal syntax will cause an exception to be thrown. Therefore, always ensure that such values cannot be nil.

Item 4: Prefer Typed Constants to Preprocessor `#define`

When writing code, you will often want to define a constant. For example, consider a UI view class that presents and dismisses itself using animations. A typical constant that you'd likely want to factor out is the animation duration. You've learned all about Objective-C and its C foundations, and so you take the approach of defining the constant like this:

```
#define ANIMATION_DURATION 0.3
```

This is a preprocessor directive; whenever the string ANIMATION_DURATION is found in your source code, it is replaced with 0.3. This might seem exactly what you want, but this definition has no type information. It is likely that something declared as a "duration" means that the value is related to time, but it's not made explicit. Also, the preprocessor will blindly replace all occurrences of ANIMATION_DURATION, so if that were declared in a header file, anything else that imported that header would see the replacement done.

To solve these problems, you should make use of the compiler. There is always a better way to define a constant than using a preprocessor define. For example, the following defines a constant of type NSTimeInterval:

```
static const NSTimeInterval kAnimationDuration = 0.3;
```

Note that with this style, there is type information, which is beneficial because it clearly defines what the constant is. The type is NSTimeInterval, and so it helps to document the use of that variable. If you have a lot of constants to define, this will certainly help you and other people who read the code later.

Also note how the constant is named. The usual convention for constants is to prefix with the letter k for constants that are local to a translation unit (implementation file). For constants that are exposed outside of a class, it is usual to prefix with the class name. Item 19 explains more about naming conventions.

It is important where you define your constants. Sometimes, it is tempting to declare preprocessor defines in header files, but that is extremely bad practice, especially if the defines are not named in such a way that they won't clash. For example, the ANIMATION_DURATION constant would be a bad name to appear in a header file. It would be present in all other files that imported the header. Even the static const as it stands should not appear in a header file. Since Objective-C has no namespaces, it would declare a global variable called kAnimationDuration. Its name should be prefixed with something that scopes it to the class it is to be used with, such as EOCViewClassAnimationDuration. Item 19 explains more about using a clear naming scheme.

A constant that does not need to be exposed to the outside world should be defined in the implementation file where it is used. For example, if the animation duration constant were used in a UIView subclass, for use in an iOS application that uses UIKit, it would look like this:

```
// EOCAnimatedView.h
#import <UIKit/UIKit.h>

@interface EOCAnimatedView : UIView
- (void)animate;
@end

// EOCAnimatedView.m
#import "EOCAnimatedView.h"

static const NSTimeInterval kAnimationDuration = 0.3;

@implementation EOCAnimatedView
- (void)animate {
    [UIView animateWithDuration:kAnimationDuration
                     animations:^(){
                         // Perform animations
                     }];
}
@end
```

It is important that the variable is declared as both `static` and `const`. The `const` qualifier means that the compiler will throw an error if you try to alter the value. In this scenario, that's exactly what is required. The value shouldn't be allowed to change. The `static` qualifier means that the variable is local to the translation unit in which it is defined. A translation unit is the input the compiler receives to generate one object file. In the case of Objective-C, this usually means that there is one translation unit per class: every implementation (`.m`) file. So in the preceding example, `kAnimationDuration` will be declared locally to the object file generated from `EOCAnimatedView.m`. If the variable were not declared `static`, the compiler would create an external symbol for it. If another translation unit also declared a variable with the same name, the linker would throw an error with a message similar to this:

```
duplicate symbol _kAnimationDuration in:
    EOCAnimatedView.o
    EOCOtherView.o
```

In fact, when declaring the variable as both `static` and `const`, the compiler doesn't end up creating a symbol at all but instead replaces occurrences just like a preprocessor define does. Remember, however, the benefit is that the type information is present.

Sometimes, you will want to expose a constant externally. For example, you might want to do this if your class will notify others using

NSNotificationCenter. This works by one object posting notifications and others registering to receive them. Notifications have a string name, and this is what you might want to declare as an externally visible constant variable. Doing so means that anyone wanting to register to receive such notifications does not need to know the actual string name but can simply use the constant variable.

Such constants need to appear in the global symbol table to be used from outside the translation unit in which they are defined. Therefore, these constants need to be declared in a different way from the static const example. These constants should be defined like so:

```
// In the header file
extern NSString *const EOCStringConstant;

// In the implementation file
NSString *const EOCStringConstant = @"VALUE";
```

The constant is "declared" in the header file and "defined" in the implementation file. In the constant's type, the placement of the const qualifier is important. These definitions are read backward, meaning that in this case, EOCStringConstant is a "constant pointer to an NSString." This is what we want; the constant should not be allowed to change to point to a different NSString object.

The extern keyword in the header tells the compiler what to do when it encounters the constant being used in a file that imports it. The keyword tells the compiler that there will be a symbol for EOCStringConstant in the global symbol table. This means that the constant can be used without the compiler's being able to see the definition for it. The compiler simply knows that the constant will exist when the binary is linked.

The constant has to be defined once and only once. It is usually defined in the implementation file that relates to the header file in which it is declared. The compiler will allocate storage for the string in the data section of the object file that is generated from this implementation file. When this object file is linked with other object files to produce the final binary, the linker will be able to resolve the global symbol for EOCStringConstant wherever else it has been used.

The fact that the symbol appears in the global symbol table means that you should name such constants carefully. For example, a class that handles login for an application may have a notification that is fired after login has finished. The notification may look like this:

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

```
extern NSString *const EOCLoginManagerDidLoginNotification;

@interface EOCLoginManager : NSObject
- (void)login;
@end

// EOCLoginManager.m
#import "EOCLoginManager.h"

NSString *const EOCLoginManagerDidLoginNotification =
    @"EOCLoginManagerDidLoginNotification";

@implementation EOCLoginManager

- (void)login {
    // Perform login asynchronously, then call 'p_didLogin'.
}

- (void)p_didLogin {
    [[NSNotificationCenter defaultCenter]
      postNotificationName:EOCLoginManagerDidLoginNotification
                    object:nil];
}

@end
```

Note the name given to the constant. Prefixing with the class name that the constant relates to is prudent and will help you avoid potential clashes. This is common throughout the system frameworks as well. UIKit, for example, declares notification names as global constants in the same way. The names include `UIApplicationDidEnterBackgroundNotification` and `UIApplicationWillEnterForegroundNotification`.

The same can be done with constants of other types. If the animation duration needed to be exposed outside of the `EOCAnimatedView` class in the preceding examples, you could declare it like so:

```
// EOCAnimatedView.h
extern const NSTimeInterval EOCAnimatedViewAnimationDuration;

// EOCAnimatedView.m
const NSTimeInterval EOCAnimatedViewAnimationDuration = 0.3;
```

Defining a constant in this way is much better than a preprocessor define because the compiler is used to ensure that the value cannot change. Once defined in `EOCAnimatedView.m`, that value is used

everywhere. A preprocessor define could be redefined by mistake, meaning that different parts of an application end up using different values.

In conclusion, avoid using preprocessor defines for constants. Instead, use constants that are seen by the compiler, such as `static const` globals declared in implementation files.

Things to Remember

- Avoid preprocessor defines. They don't contain any type information and are simply a find and replace executed before compilation. They could be redefined without warning, yielding inconsistent values throughout an application.
- Define translation-unit-specific constants within an implementation file as `static const`. These constants will not be exposed in the global symbol table, so their names do not need to be namespaced.
- Define global constants as external in a header file, and define them in the associated implementation file. These constants will appear in the global symbol table, so their names should be namespaced, usually by prefixing them with the class name to which they correspond.

Item 5: Use Enumerations for States, Options, and Status Codes

Since Objective-C is based on C, all the features of C are available. One of these is the enumeration type, `enum`. It is used extensively throughout the system frameworks but is often overlooked by developers. It is an extremely useful way to define named constants that can be used, for example, as error status codes and to define options that can be combined. Thanks to the additions of the C++11 standard, recent versions of the system frameworks include a way to strongly type such enumeration types. Yes, Objective-C has benefitted from the C++11 standard as well!

An enumeration is nothing more than a way of naming constant values. A simple enumeration set might be used to define the states through which an object goes. For example, a socket connection might use the following enumeration:

```
enum EOCConnectionState {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
```

Using an enumeration means that code is readable, since each state can be referred to by an easy-to-read value. The compiler gives a unique value to each member of the enumeration, starting at 0 and increasing by 1 for each member. The type that backs such an enumeration is compiler dependent but must have at least enough bits to represent the enumeration fully. In the case of the preceding enumeration, this would simply need to be a `char` (1 byte), since the maximum value is 2.

This style of defining an enumeration is not particularly useful, though, and requires the following syntax:

```
enum EOCConnectionState state = EOCConnectionStateDisconnected;
```

It would be much easier if you didn't have to type `enum` each time but rather use `EOCConnectionState` on its own. To do this, you add a `typedef` to the enumeration definition:

```
enum EOCConnectionState {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
typedef enum EOCConnectionState EOCConnectionState;
```

This means that `EOCConnectionState` can be used instead of the full `enum EOCConnectionState`:

```
EOCConnectionState state = EOCConnectionStateDisconnected;
```

The advent of the C++11 standard brought some changes to enumerations. One such change is the capability to dictate the underlying type used to store variables of the enumerated type. The benefit of doing this is that you can forward declare enumeration types. Without specifying the underlying type, an enumeration type cannot be forward declared, since the compiler cannot know what size the underlying type will end up being. Therefore, when the type is used, the compiler doesn't know how much space to allocate for the variable.

To specify the type, you use the following syntax:

```
enum EOCConnectionStateConnectionState : NSInteger { /* … */ };
```

This means that the value backing the enumeration will be guaranteed to be an `NSInteger`. If you so wished, the type could be forward declared like so:

```
enum EOCConnectionStateConnectionState : NSInteger;
```

It's also possible to define the value a certain enumeration member relates to rather than letting the compiler choose for you. The syntax looks like this:

```
enum EOCConnectionStateConnectionState {
    EOCConnectionStateDisconnected = 1,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
```

This means that `EOCConnectionStateDisconnected` will use the value 1 rather than 0. The other values follow, incrementing by 1 each time, just as before. Thus, `EOCConnectionStateConnected` will use the value 3, for example.

Another reason to use enumeration types is to define options, especially when the options can be combined. If the enumeration is defined correctly, the options can be combined using the bitwise OR operator. For example, consider the following enumeration type, found in the iOS UI framework, used to define which dimensions of a view can be resized:

```
enum UIViewAutoresizing {
    UIViewAutoresizingNone                 = 0,
    UIViewAutoresizingFlexibleLeftMargin   = 1 << 0,
    UIViewAutoresizingFlexibleWidth        = 1 << 1,
    UIViewAutoresizingFlexibleRightMargin  = 1 << 2,
    UIViewAutoresizingFlexibleTopMargin    = 1 << 3,
    UIViewAutoresizingFlexibleHeight       = 1 << 4,
    UIViewAutoresizingFlexibleBottomMargin = 1 << 5,
}
```

Each option can be either on or off, and using the preceding syntax enables this because each option has just a single bit set in the value represented by it. Multiple options can be bitwise OR'ed together: for example, `UIViewAutoResizingFlexibleWidth | UIViewAutoresizingFlexibleHeight`. Figure 1.2 shows the bit layout of each enumeration member and the combination of two of the members.

It's then possible to determine whether one of the options is set by using the bitwise AND operator:

```
enum UIVewAutoresizing resizing =
    UIViewAutoresizingFlexibleWidth |
    UIViewAutoresizingFlexibleHeight;
if (resizing & UIViewAutoresizingFlexibleWidth) {
    // UIViewAutoresizingFlexibleWidth is set
}
```

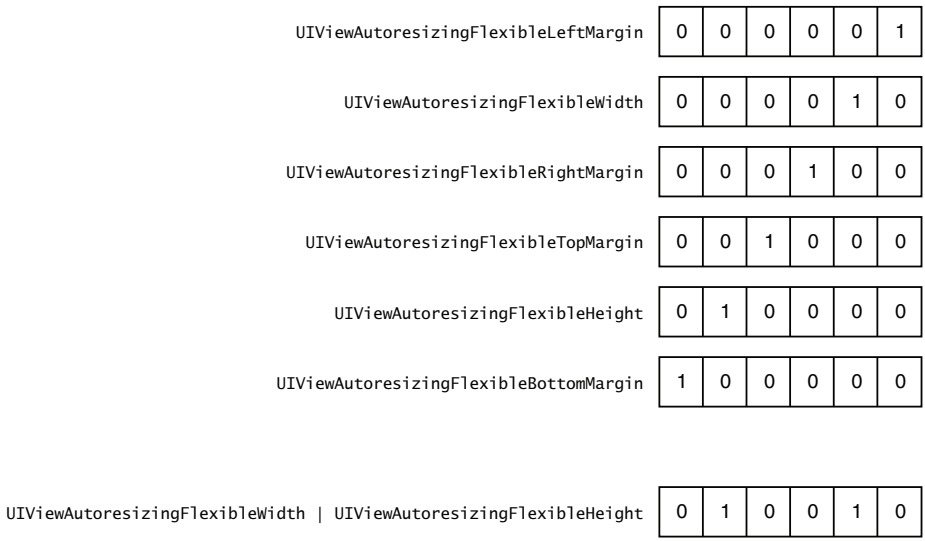


Figure 1.2 Binary representation of three options values and two of those values bitwise OR'ed together

This is used extensively throughout the system libraries. Another example from UIKit, the iOS UI framework, uses it as a way of telling the system what device orientations your view supports. It does this with an enumerated type called `UIInterfaceOrientationMask`, and you implement a method called `supportedInterfaceOrientations` to indicate the supported orientations:

```
- (NSUInteger)supportedInterfaceOrientations {
    return UIInterfaceOrientationMaskPortrait |
           UIInterfaceOrientationMaskLandscapeLeft;
}
```

A couple of helpers defined within the Foundation framework help define enumeration types that also allow you to specify the integral type that will be used to store values that use the enumeration type. These helpers provide backward compatibility such that if you're targeting a compiler that supports the new standard, that syntax is used, but it falls back to the old syntax if not. The helpers are provided in the form of preprocessor `#define` macros. One is provided for normal enumeration types, such as the `EOCConnectionState` example. The other is provided for the case in which the enumeration defines a list of options like the `UIViewAutoresizing` example. You use them as follows:

```
typedef NS_ENUM(NSUInteger, EOCConnectionState) {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
typedef NS_OPTIONS(NSUInteger, EOCPermittedDirection) {
    EOCPermittedDirectionUp    = 1 << 0,
    EOCPermittedDirectionDown  = 1 << 1,
    EOCPermittedDirectionLeft  = 1 << 2,
    EOCPermittedDirectionRight = 1 << 3,
};
```

This is what the macro definitions look like:

```
#if (__cplusplus && __cplusplus >= 201103L &&
        (__has_extension(cxx_strong_enums) ||
         __has_feature(objc_fixed_enum))
    ) ||
    (!__cplusplus && __has_feature(objc_fixed_enum))
    #define NS_ENUM(_type, _name)
            enum _name : _type _name; enum _name : _type
    #if (__cplusplus)
        #define NS_OPTIONS(_type, _name)
                _type _name; enum : _type
    #else
        #define NS_OPTIONS(_type, _name)
                enum _name : _type _name; enum _name : _type
    #endif
#else
    #define NS_ENUM(_type, _name) _type _name; enum
    #define NS_OPTIONS(_type, _name) _type _name; enum
#endif
```

The reason for the various ways of defining the macros is that there are different scenarios. The first case that is checked is whether the compiler supports the new style enumerations at all. This is checked with what looks like some rather complex Boolean logic, but all that it's checking is that the feature is there. If the feature is not there, it defines the enumeration by using the old style.

If the feature is available, the `NS_ENUM` type is defined such that it expands out like this:

```
typedef enum EOCConnectionState : NSUInteger EOCConnectionState;
enum EOCConnectionState : NSUInteger {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
```

```
    EOCConnectionStateConnected,
};
```

The NS_OPTIONS macro is defined in different ways if compiling as C++ or not. If it's not C++, it's expanded out the same as NS_ENUM. However, if it is C++, it's expanded out slightly differently. Why? The C++ compiler acts differently when two enumeration values are bitwise OR'ed together. This is something, as shown earlier, that is commonly done with the options type of enumeration. When two values are OR'ed together, C++ considers the resulting value to be of the type the enumeration represents: NSUInteger. It also doesn't allow the implicit cast to the enumeration type. To illustrate this, consider what would happen if the EOCPermittedDirection enumeration were expanded out as NS_ENUM:

```
typedef enum EOCPermittedDirection : int EOCPermittedDirection;
enum EOCPermittedDirection : int {
    EOCPermittedDirectionUp    = 1 << 0,
    EOCPermittedDirectionDown  = 1 << 1,
    EOCPermittedDirectionLeft  = 1 << 2,
    EOCPermittedDirectionRight = 1 << 3,
};
```

Then consider attempting the following:

```
EOCPermittedDirection permittedDirections =
    EOCPermittedDirectionLeft | EOCPermittedDirectionUp;
```

If the compiler were in C++ mode (or potentially Objective-C++), this would result in the following error:

```
error: cannot initialize a variable of type
'EOCPermittedDirection' with an rvalue of type 'int'
```

You would be required to put in an explicit cast to the result of the ORing, back to EOCPermittedDirection. So the NS_OPTIONS enumeration is defined differently for C++ such that this does not have to be done. For this reason, you should always use NS_OPTIONS if you are going to be ORing together the enumeration values. If not, you should use NS_ENUM.

An enumeration can be used in many scenarios. Options and states have been shown previously; however, many other scenarios exist. Status codes for errors are a good candidate as well. Instead of using preprocessor defines or constants, enumerations provide a means for grouping together logically similar status codes into one enumeration. Another good candidate is styles. For example, if you have a UI element

that can be created with different styles, an enumeration type is perfect for that situation.

One final extra point about enumerations has to do with using a `switch` statement. Sometimes, you will want to do the following:

```
typedef NS_ENUM(NSUInteger, EOCConnectionState) {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};

switch (_currentState) {
    EOCConnectionStateDisconnected:
        // Handle disconnected state
        break;
    EOCConnectionStateConnecting:
        // Handle connecting state
        break;
    EOCConnectionStateConnected:
        // Handle connected state
        break;
}
```

It is tempting to have a default entry in the `switch` statement. However, when used for switching on an enumeration that defines a state machine, it is best not to have a default entry. The reason is that if you add a state later on, the compiler will helpfully warn that the newly added state has not been cared for in the `switch` statement. A default block handles the new state, so the compiler won't warn. The same applies to any other type of enumeration defined using the `NS_ENUM` macro. For example, if used to define styles of a UI element, you would usually want to make sure that `switch` statements handled all styles.

Things to Remember

- Use enumerations to give readable names to values used for the states of a state machine, options passed to methods, or error status codes.
- If an enumeration type defines options to a method in which multiple options can be used at the same time, define its values as powers of 2 so that multiple values can be bitwise OR'ed together.

- Use the `NS_ENUM` and `NS_OPTIONS` macros to define enumeration types with an explicit type. Doing so means that the type is guaranteed to be the one chosen rather than a type chosen by the compiler.
- Do not implement a default case in `switch` statements that handle enumerated types. This helps if you add to the enumeration, because the compiler will warn that the switch does not handle all the values.

Index

B

C

D

M

N

O

S

T

U