David Chisnall

Cocoa® Programming
Developer's Handbook

Developer's Library

Cocoa® Programming Developer's Handbook

This page intentionally left blank

Cocoa® Programming Developer's Handbook

David Chisnall

Addison-Wesley

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

Cocoa® Programming Developer's Handbook
Copyright © 2010 Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-321-63963-9
ISBN-10: 0-321-63963-4

Library of Congress Cataloging-in-Publication Data

Chisnall, David.
Cocoa programming developer's handbook / David Chisnall.
p. cm.
Includes index.
ISBN 978-0-321-63963-9 (pbk. : alk. paper) 1. Cocoa (Application development environment) 2. Object-oriented programming (Computer science) 3. Application program interfaces (Computer software) 4. Mac OS. I. Title.
QA76.64.C485 2010
005.26'8—dc22

2009042661

Printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First Printing December 2009

Trademarks
All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Pearson cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer
Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales
Pearson offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact us by phone or email:

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact the International Sales group:

International Sales
international@pearson.com

Editor-in-Chief
Mark L. Taub

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Technical Reviewer
Gregory Casamento

Book Designer
Gary Adair

Composition
David Chisnall

Contents

List of Figures

This page intentionally left blank

List of Tables

This page intentionally left blank

Preface

This book aims to serve as a guide to the Cocoa APIs found on Mac OS X. The core frameworks are described in detail, as are many of the other components used to build rich applications.

These APIs are huge. In most Cocoa programs, you include the Cocoa.h header, which imports the two core frameworks that make up Cocoa. This header, when preprocessed, including all of the headers that it references, is well over 100,000 lines long. If you printed the preprocessed header out, you would get something over twice as long as this book, and you would still only have the core APIs, and not any of the more advanced parts discussed in later parts of this book.

This book aims to provide a guided tour, indicating features of interest to help visitors find their way around this enormous family of APIs. As with many travel books, this aims to include the same 'must-see' destinations that everyone will visit as well as some of the more interesting but often-overlooked parts.

Deep familiarity with something like Cocoa only comes from years of practice using the classes that are included as part of the frameworks. This book provides an introduction, but you will only become an expert OS X developer if you take the information contained in these pages and apply it, developing your own applications.

Who Should Read This Book

This book is aimed at people wanting to learn how to develop applications using the rich Cocoa APIs on OS X. It is not aimed at people wanting to learn iPhone development. The iPhone SDK is designed to be easy to learn for seasoned Mac programmers, and shares a lot of core concepts and frameworks with the desktop APIs, but it is a separate system. Reading this book will make it easy for you to learn iPhone development later and care has been taken to point out places where the desktop and mobile APIs diverge; however, this book does not cover the iPhone APIs directly.

If you want to learn how to develop rich applications for Mac OS X then this book will help you. This includes coverage of the core APIs that have remained largely unchanged since the early 1990s on NeXT workstations up to the latest additions for integration with an internetworked environment and handling rich multimedia content.

This book assumes some general knowledge of programming. The first chapters include an introduction to the Objective-C, which should be sufficient for readers already familiar with languages like C or Java. This section is not intended as a general introduction to programming concepts.

Overview and Organization

This book is divided into seven parts. Each covers part of the Cocoa APIs.

Introducing Cocoa covers the background of Cocoa, how it fits into OS X, and where it came from. This part introduces and describes the Objective-C language and provides the reader with an overview of the tools used to create Cocoa applications.

In *The Cocoa Frameworks* you will be introduced to the Foundation and Application Kit frameworks that form the core of the Cocoa APIs. Foundation provides low-level, core functions, while the Application Kit is layered on top and provides the features needed to build rich applications. This part introduces both, giving an overview of how they fit together and how to begin creating applications using them. You will see the basic concepts that underlie the Cocoa application model, including how events are delivered and how the drawing model works. By the end of this part you will understand how to create simple applications using Cocoa.

Cocoa Documents covers developing document-driven applications with Cocoa. A document driven application is one that creates identical windows representing some persistent model, typically a file. Cocoa includes a lot of code to support this kind of application. You will also be introduced in this part to the Core Data framework, which handles automatic persistence for documents.

Part IV, *Complex User Interfaces* goes deeper into the Application Kit. You will learn about the more advanced view objects that interact with your program via a data source and will learn how to provide data dynamically to them. You will also see how to create new view objects.

The next part, *Advanced Graphics*, builds on top of this knowledge by exploring some of the more complex graphical capabilities of Cocoa. This includes the Core Animation framework, found on both desktop and iPhone OS X, which enables you to create intricate animated effects with only a small amount of code. This part will also take a small diversion from the visual into the audio world and discuss how to provide audible feedback to your user interface. This includes using the speech recognition and synthesis APIs on OS X. By the end of this part, you should be able to write complex multimedia Cocoa applications.

User Interface Integration focusses on the parts of OS X that make an application feel like a part of the environment, rather than an isolated program. This includes integration with the systemwide search facilities as well as the various shared data stores, such as the address book and calendar.

The final part, *System Programming*, covers the low-level features of Cocoa, including network programming and concurrency. This ranges from creating sockets to fetching data from a remote URL, and explores the distributed objects system in the Foundation framework.

This book is not intended as a replacement for Apple's excellent documentation. Every class in Cocoa has an accompanying reference available both online and in the XCode environment. Many also include guides covering how a small set of classes relate to each other. This comes to a total of several tens of thousands of pages of material.

You will not find detailed descriptions of every method in a class in this book. If you want to learn exactly what a class can do, look it up in the Apple documentation. Instead, you will find descriptions of the most important and commonly used features of classes and how they relate together. The Apple documentation, while thorough, can be overwhelming. Reading this book will help you find the subset that you need to solve a particular problem.

The example programs provided by Apple are similarly different to the ones provided by this book. Each of the examples included with this book is intended to demonstrate a single aspect of the Cocoa API. In contrast, the Apple examples tend to be complete applications demonstrating a complete API. The TextEdit application included with OS X is one such example. This is a full-featured rich text editor, and is several thousand lines of code. If you want to see a detailed example of how all of the parts of the Cocoa document support and text system fit together, it is an invaluable resource, but trying to understand the whole of the code can be very difficult if you are not already quite familiar with Cocoa.

Typographical Conventions

This book uses a number of different typefaces and other visual hints to describe different types of material.

Filenames, such as `/bin/sh`, are all shown in `this font`. This is also used for commands that you might type into a terminal.

Variable or function names, such as `example()`, used in text will be typeset `like this`. Objective-C message names will be prefixed with a plus sign if they are indented to be sent to classes or a minus if they are sent to instances, for example, `+alloc` and `-init`.

This book contains two kinds of code listing. Short listings appear like this:

```
eg = example_function(arg1);
```

This kind of listing is intended to highlight a simple point and may contain shorthand or depend on variables or functions that are not given. You should not expect to be able to copy these listings into a source file and compile them; they are intended to aid understanding.

Longer listings will have line numbers down the left, and a gray background, as shown in Listing 1. In all listings, bold is used to indicate keywords, and italicized text represents strings and comments.

Listing 1: An example listing [from: example/hello.c]

```
1 #include <stdio.h>
2 
3 int main(void)
4 {
5     /* Print hello world */
6     printf("Hello␣World!\n");
7     return 0;
8 }
```

Listings that are taken from external files will retain the line numbers of the original file, allowing the referenced section to be found easily by the reader. The captions contain the original source in square brackets. Those beginning with example/ are from the example sources that accompany this book. You should be able to compile and run any of these on a modern OS X system.

Output from command-line interaction is shown in the following way:

```
$ gcc hello.c
$ ./a.out
Hello World!
```

A $ prompt indicates commands that can be run as any user, while a # is used to indicate that root access is likely to be required. Most of the time, example programs are intended to be compiled using XCode. A few single-file examples are intended to be compiled from the terminal.

Chapter 4

Foundation: The Objective-C Standard Library

The “core” Objective-C language only defines two classes: Object and Protocol. It is rare to use Objective-C without an implementation of OpenStep Foundation, whether it’s GNUstep, Cocoa, libfoundation, or Cocotron. The Portable Object Compiler provides its own set of core objects, but it is not widely used.

The OpenStep Foundation is the closest thing that Objective-C has to a standard library, the equivalent of the C standard library or C++’s STL. Of course since Objective-C is a pure superset of C, the C standard library can also be used. The original idea was to do exactly this, and use Objective-C for building components from C software.

Foundation was only introduced with OpenStep to hide the differences between NeXTSTEP’s Mach-based operating system and Solaris, and to make it easier to write endian-independent code. Most of Foundation is endian-independent, which was a huge benefit when Apple moved from the big-endian PowerPC to the little-endian x86 architecture.

4.1 General Concepts

Although the Foundation framework is very large, it is quite easy to learn. A lot of the classes share common design principles. When you understand these shared concepts, you can learn how to use each of the individual classes quickly.

4.1.1 Mutability

Objective-C does not have a concept of constant objects. This is not quite true; the `const` keyword from C still exists, but it only applies to direct access to instance variables. Methods cannot be marked as mutators and so any messages sent to an object may modify it, irrespective of whether the object pointer is `const`-qualified.

In many cases, however, it is useful to have mutable and immutable versions of objects. This is often done in object-oriented systems by having mutable and immutable classes. Strings are a common example. If you create an Objective-C string literal `@"like␣this"` then you are creating a constant string. The compiler will put this string in the constants section of the binary—attempting to modify it will cause a segmentation fault. Having to create a new string and copy can make a program very slow, however. This is one of the reasons Java code has a reputation for being slow; Java's `String` class is immutable, and since it is declared `final` you can't use Cocoa's solution to the problem, a mutable subclass.

The `NSString` object is an immutable string. It has a subclass, `NSMutableString`. Because the mutable version is a subclass, it can be used anywhere that the immutable version can. It implements all of the same methods.

The distinction between mutable and immutable objects is most apparent in the implementation of the `-copy` method. When you send a `-copy` message to an immutable object, you often get the same object back (but with the retain count incremented). Because you cannot modify either "copy" they can never become different from each other.

This ability is one of the reasons why Objective-C programs are often faster than C++, in spite of microbenchmarks showing the opposite. In a C++ program, the equivalent with `std::string` objects would result in a real copy. A C++ string might be copied half a dozen times, whereas a Cocoa string will only have its reference count incremented and decremented.

4.1.2 Class Clusters

Although `NSString` is the class for immutable strings, your string literal will not really be an `NSString`. Instead, it will be an `NSConstantString` or similar. This class is a private subclass of `NSString`, used for a specific purpose.

This is very common in Cocoa. There might be half a dozen or so different implementations of common classes, such as `NSDictionary`, all optimized for different uses. When you initialize one, you will get back a specific subclass, rather than the abstract superclass.

There are two ways in which this can be done. The first is to return a different subclass from each constructor or initializer. The second is to use the same instance variable layout and use a trick known as *isa-swizzling*. The `isa` pointer, the pointer to the object's class, is just another instance variable. In keeping with the "no magic" philosophy of Objective-C, there is nothing special about it. You can assign a new value to it if you wish. As long as both the new and old classes have the same layout in memory, everything will keep working. (If they don't, you will get some difficult-to-debug memory corruption.)

Class clusters make subclassing slightly difficult. Typically, each of the hidden classes in a cluster implements only a small number of primitive methods. In `NSString` these are `-characterAtIndex:` and `-length`. All of the others are implemented in the superclass in terms of these. If you want to create a new `NSString` subclass, you must implement these methods yourself. It is common to do this by having a concrete instance as an instance variable and delegating to it, although you can implement the primitive methods yourself.

Of course, there is nothing stopping you from implementing more than just these two primitive methods. You may be able to implement more efficient versions of some of them.

More isa-swizzling

The `isa`-swizzling trick is useful in a lot of cases, not just class clusters. It can be used for debugging use-after-free memory problems, by having the `-dealloc` method simply change the class to one that throws an exception if it receives any messages. You can also use it to implement state machines, where each state is in a separate subclass of a common class. To enter a new state, simply change the `isa` pointer to that pointer's class.

You can implement class clusters of your own very easily. Typically, you will have a set of different initializers in the public class, and each of these will return an instance of a different subclass. To demonstrate this, we will define a simple class encapsulating a pair of values. Listing 4.1 shows this interface. Note that no instance variables are declared here.

In the implementation file, we define two concrete subclasses of the `Pair` class, one for storing integers and one for floating point values. These are shown in Listing 4.2. Neither of these defines any new methods. Since these interfaces are private, there would be no point in adding new methods since no one would know to call them. They do, however, define the structure. Class clusters implemented like this allow entirely different data layouts for different implementations.

The implementation of the public class, shown in Listing 4.3, is very simple. Most of the methods just return simple default values, since they should not be called. A more robust implementation might throw an exception.

The important thing to note is the `[self release]` line in both initializers.

Listing 4.1: The public interface to the pair class. [from: examples/ClassCluster/Pair.h]

```
@interface Pair : NSObject {}
- (Pair*) initWithFloat:(float)a float:(float)b;
- (Pair*) initWithInt:(int)a int:(int)b;
- (float) firstFloat;
- (float) secondFloat;
- (int) firstInt;
- (int) secondInt;
@end
```

Listing 4.2: The private interfaces to the concrete pair classes. [from: examples/Class-Cluster/Pair.m]

```
@interface IntPair : Pair {
    int first;
    int second;
}
@end
@interface FloatPair : Pair {
    float first;
    float second;
}
@end
```

Listing 4.3: The implementation of the public pair class. [from: examples/ClassCluster/-Pair.m]

```
@implementation Pair
- (Pair*) initWithFloat: (float)a float: (float)b
{
    [self release];
    return [[FloatPair alloc] initWithFloat: a float: b];
}
- (Pair*) initWithInt: (int)a int: (int)b
{
    [self release];
    return [[IntPair alloc] initWithInt: a int: b];
}
- (float) firstFloat { return 0; }
- (float) secondFloat { return 0; }
- (int) firstInt { return 0; }
- (int) secondInt { return 0; }
@end
```

Typically, an object will be created by first sending `+alloc` to the `Pair` class and then sending the result the initialization message. The object returned from `+alloc` is not required, and so is released here and a new object returned instead.

Listing 4.4 shows the implementations of the private pair classes. Each of these only implements a single constructor, the one relevant to its data type. The accessor methods then either return instance variables or casts of instance variables, allowing both kinds of pair to return **`int`**s or **`float`**s. One method from `NSObject` is implemented by both, `-description`, which provides a human-readable description of the object. Note that neither of these call the designated initializer in the superclass; this is quite bad style, but was done to simplify the example.

Listing 4.4: The implementation of the private pair classes. [from: examples/ClassCluster/Pair.m]

```
@implementation IntPair
- (Pair*) initWithInt: (int)a int: (int)b
{
    first = a;
    second = b;
    return self;
}
- (NSString*) description
{
    return [NSString stringWithFormat: @"(%d, %d)",
            first, second];
}
- (float) firstFloat { return (float)first; }
- (float) secondFloat { return (float)second; }
- (int) firstInt { return first; }
- (int) secondInt { return second; }
@end
@implementation FloatPair
- (Pair*) initWithFloat: (float)a float: (float)b
{
    first = a;
    second = b;
    return self;
}
- (NSString*) description
{
    return [NSString stringWithFormat: @"(%f, %f)",
            (double)first, (double)second];
}
- (float) firstFloat { return first; }
- (float) secondFloat { return second; }
- (int) firstInt { return (int)first; }
- (int) secondInt { return (int)second; }
@end
```

Users of the pair class now don't have to be aware of either of the private classes. A simple test program that creates one of each can demonstrate this. Listing 4.5 shows a short program that just creates two pair objects and logs them. The format string provided to `NSLog` will cause the `-description` method in each to be called.

Listing 4.5: Demonstrating the pair classes. [from: examples/ClassCluster/test.m]

```
#import "Pair.h"

int main(void)
{
    [NSAutoreleasePool new];
    Pair *floats = [[Pair alloc] initWithFloat:0.5 float:12.42];
    Pair *ints= [[Pair alloc] initWithInt:1984 int:2001];
    NSLog(@"Two␣floats:␣%@", floats);
    NSLog(@"Two␣ints:␣%@", ints);
    return 0;
}
```

Running this program gives the following output:

```
2009-01-14 14:27:55.091 a.out[80326:10b] Two floats: (0.500000, 12.420000)
2009-01-14 14:27:55.093 a.out[80326:10b] Two ints: (1984, 2001)
```

A more full implementation of this cluster would have named constructors, such as `+pairWithInt:int:`, which would avoid the need to allocate and then free an instance of the `Pair` object. The alternate way of avoiding this, as mentioned earlier, is to use `isa`-swizzling. The `Pair` class might have two instance variables that were unions of an `int` and a `float`. Implemented in this way, the initializers would look like this:

```
- (Pair*) initWithFloat: (float)a float: (float)b
{
    isa = [FloatPair class];
    return [self initWithFloat: a float: b];
}
```

This first line in this implementation sets the class pointer to the subclass, and the second calls the method again. Because the class pointer has changed, the second call will invoke the subclass implementation of this method. Each subclass would then refer to the correct field in the union.

4.2 Core Foundation Types

The *Core Foundation* (*CF*) library contains a set of C opaque types that have a similar interface to a number of Cocoa Foundation objects. This similarity is not accidental. The aim of Core Foundation was to produce a rich common set of fundamental types that both Cocoa and Carbon applications could use. This is no longer important, since Carbon did not make the 64-bit switch, but Core Foundation is still used in a lot of low-level parts of OS X, such as `Launchd`.

Although C does not have a notion of inheritance on types, Core Foundation types are built into a hierarchy. At the root is `CFType`, which implements basic memory management for CF types. Just as Cocoa objects are reference counted with `-retain` and `-release` messages, Core Foundation types are reference counted by calling the `CFRetain()` and `CFRelease()` functions with them as an argument.

Many of the Core Foundation types use the *toll-free bridging* mechanism to interoperate with their Cocoa equivalents. The first field in any CF structure is an `isa` pointer, just as with an Objective-C object. Unlike Cocoa objects, however, this value is always between 0 and 2^{16}, a region of memory where no Objective-C classes will be. When you send a message to a CF object, the message send function will use a special case for class pointers in this range.

Similarly, when you call a Core Foundation function with a Cocoa object, it will test that the `isa` pointer is greater than 0xFFFF and, if it is, then call the Objective-C runtime functions for method dispatch, bouncing the call back to Objective-C. This allows you to use the Core Foundation types and Cocoa objects interchangeably.

A lot of the Cocoa Foundation objects have Core Foundation analogues. The most common is probably `CFString`, the equivalent of Cocoa's `NSString`. In fact, both `NSString` and `NSMutableString` are class clusters on Cocoa, meaning that their instances may not really be versions of that class. Under the hood, all three types are implemented by the `NSCFString` type. This is true for a lot of class clusters in Cocoa.

4.3 Basic Data Types

Foundation provides a number of data types, some as primitive C types, and some as object types. Some of these represent some kind of structured data, such as a string or a date, while others are collections of arbitrary types.

Any nontrivial Cocoa program is likely to make heavy use of some of these. All of them provide a rich set of methods for manipulating them, and so you should take care to check the documentation carefully before implementing new features for them.

4.3.1 Non-Object Types

OpenStep was originally designed to work on very slow computers by today's standards. One of the big improvements in performance over Smalltalk came from the judicious use of non-object types. The most obvious of these are the various primitive integer and floating point types. There are also a small number of structures, such as `NSRange`, which are used throughout the Foundation frameworks.

There are several reasons why these are not objects. The first is their size. Most of these structures are pairs of values. A range is a start and a length, for example. Adding on four bytes for an `isa` pointer and four bytes for a reference count would double their size. By making them structures, they can be passed by value in registers, which makes calling methods (and functions) that use or return them faster. Finally, they are rarely aliased. When you set a range or a point or rectangle somewhere, you want to set a copy.

The most common structures used in Cocoa are

- `NSRange`, a pair of positive integers representing an offset and length in a sequence. These are most commonly used with `NSStrings` for defining substrings, but can be used with arrays and other similar data structures.
- `NSPoint`, which contains two floating-point values representing x and y coordinates.
- `NSSize`, which is structurally equivalent to `NSPoint`. The difference between `NSSize` and `NSPoint` is that the values for a size should never be negative. As a structure it is unable to enforce this constraint; however, assigning a negative value to either field may cause exceptions or subtle failures.
- `NSRect`, an aggregate of an `NSPoint` and an `NSSize` that allows a rectangle to be defined in 2D space.

Note that the last three of these are all most commonly used for drawing functions in AppKit, even though they are defined in Foundation.

CGFloat, NSUInteger, and Friends

Prior to 10.5, most of these structures used **`int`**, **`float`**, and similar types. With 10.5, Apple redefined a lot of types and functions to use the `CGFloat` and `NSUInteger` types. The new `NSUInteger` and `NSInteger` types are identical to C99's `uintptr_t` and `intptr_t` respectively. They are provided for compatibility with code using C89 or older dialects. `CGFloat` is defined as a **`float`** on 32-bit platforms and a **`double`** on 64-bit.

Foundation also includes a number of other primitive data types, including a large number of enumerated types. Common examples of these include `NSComparisonResult`, which defines `NSOrderedAscending`, `NSOrderedSame`, and `NSOrderedDescending`, and is used to define how two objects should be ordered. If you sort a collection of Cocoa objects, the order will be defined by calling a function or a method that returns one of these three values on pairs of objects in the collection.

4.3.2 Strings

One of the most commonly used classes in Foundation is `NSString`. Technically speaking, this means subclasses of `NSString`, since it is a *class cluster* and is never directly used.

Each concrete subclass of `NSString` must override at least two of the methods defined by this class: `-length` and `-characterAtIndex:`. The first of these returns the length of the string, and the second returns a unicode (32-bit) character at a specified index. Note that the internal format of the string may be in any format. The class cluster design allows 8-, 16-, and 32-bit strings to all be stored internally when a given string does not include any characters from outside the set that can be expressed with these. The programmer can be largely oblivious to this and use these strings interchangeably: The `NSString` subclass will transparently handle any conversion required.

Although these are the only methods that need to be overridden, most of the methods in `NSString` will call `getCharacters:range:`, which writes a substring into a buffer provided by the caller. Subclasses that implement this directly, rather than using the superclass implementation that repeatedly calls `-characterAtIndex:`, will be much faster.

Note that this method name begins with the `get` prefix. This is a common Cocoa idiom for methods that return a value into space provided by the caller. Contrast this with the `length` method, which does not have the `get` prefix, and just returns the length.

Although it is possible to create your own subclass of `NSString`, it is generally a better option to compose objects without subclassing. An example of this in the Foundation framework is `NSAttributedString`. This responds to `-stringValue` messages to return the string for which it stores attributes, but cannot be used directly in place of a string. We will look at this class in a lot more detail in Chapter 8.

`NSString` has one public subclass (which is also a class cluster), for representing strings that can be modified: `NSMutableString`. This adds methods for modifying characters. Only seven new methods are added by this class, with six being defined in terms of the one primitive method: `replaceCharactersInRange:withString:`.

The `NSString` class has a huge number of methods, and 10.5 added a lot more. A lot of these are to do with path handling. One of the problems that OS X developers encountered a lot in the early days was the fact that MacOS and OPENSTEP had different ways of representing paths. MacOS used a multi-routed file hierarchy, with one file for each disk, with path components separated by colons. OPENSTEP used a UNIX-style file hierarchy, with a single root and path components separated by slashes. Mac OS X applications often had to deal with both.

Fortunately, this was a problem that NeXT had already encountered. OpenStep applications were able to run on Solaris, OPENSTEP, and Windows. Windows file paths were similar in structure to classic MacOS paths. `NSString` has a set of methods for adding and deleting path components and splitting paths apart in a way that is independent of the underlying filesystem representation. It is good practice to use these, rather than manually constructing paths.

Recent versions of OS X have begun to move away from using file paths entirely, with a lot of methods now using *URLs* in the file:// namespace instead of file paths. There are fewer methods on `NSString` for dealing with these; however, the `NSURL` class provides a lot more.

4.3.3 Boxed Numbers and Values

The advantage of using primitive types is speed. The disadvantage is that they don't integrate well with collections that expect objects. There are three classes that are provided for working around these. Each of them boxes a specific kind of primitive value.

Boxing

Boxing is a term used to describe wrapping a primitive value in an object. High-level languages like Lisp and Smalltalk perform auto-boxing, and so you can interact with primitive values as if they were objects. Objective-C requires manual boxing.

The most general boxing class is `NSValue`, which can contain any primitive type. This is most commonly used for encapsulating the Foundation **`struct`** types, such as `NSRange` and storing them in collections. This class has a subclass (actually, a class cluster), `NSNumber`, which is used to store single numerical values. Any value from a **`char`** to a **`long long`** stored in one of these, and it will correctly cast the result if any of the `-somethingValue` family of methods is called. For example, you can create an `NSNumber` from a primitive **`unsigned int`** like this:

```
[NSNumber numberWithUnsignedInt: myInt];
```

It could then be stored in a collection, retrieved, passed to another method, and then turned into a 64-bit value like this:

```
[aNumber longLongValue];
```

Be careful when doing this, however. If you do the reverse operation—create an `NSNumber` with a 64-bit value and then retrieve a 32-bit or smaller value—then there will be silent truncation of the result.

Decimal Arithmetic

In addition to the standard binary types inherited from C, and their boxed equivalents, Foundation defines an `NSDecimal` structure and a `NSDecimalNumber` boxed equivalent. These can be used for performing decimal floating point arithmetic. Some decimal numbers, such as 0.1, cannot be represented as finite binary values. This is problematic for financial applications, where a fixed number of decimal digits of precision is required. The `NSDecimal` type can be used to accomplish this.

There is one remaining boxed value, which is often overlooked. `NSNull` is a singleton—only one instance of it ever exists—representing a boxed version of `NULL`.

The Many Types of Zero

In C, `NULL` is defined as `(void*)0`; a pointer value of zero. Because the `void*` type can be silently cast to any pointer, the `NULL` value can be used for any pointer. Objective-C adds two new types of zero; `nil` and `Nil`, meaning `(id)0` and `(Class)0` respectively. In addition, there is the boxed version, `NSNull` and zero values boxed in `NSValue` and `NSNumber` objects. This means that there are a lot of different ways of expressing zero in Cocoa, depending on the use.

Unlike many of the other classes in Foundation, there is no `NSMutableNumber` or `NSMutableDecimalNumber`. If you need to modify a boxed value, you need to first unbox it, then perform primitive operations on it, and then box it again. This makes sense, since operations on primitive values are typically a lot faster than message sends. In a language like Smalltalk or Lisp, the compiler would try to transparently turn the object into a primitive value and do this for you, but Objective-C compilers are not (yet) clever enough to do so.

4.3.4 Data

In C, arbitrary data is typically represented in the same way as strings; by `char*`s. In Cocoa, using string objects would not work, since they perform character set conversion. The `NSData` class exists to encapsulate raw data. You can think of it as a boxed version of `void*`, although it also stores a length, preventing pointer arithmetic bugs from overwriting random memory locations.

You can get a pointer to the object's data by sending it a `-bytes` message. It may seem that this will be more efficient; however, this is not always the case. In some cases, the underlying representation may be a set of non-contiguous memory regions, or data in a file that has not been read into memory. When you call `-bytes` the object is required to ensure that all of the data is in a contiguous memory region, which may be an expensive operation. Subsequent operation on the data will, in the absence of swapping, be very fast.

You can use `NSData` and its mutable subclass, `NSMutableData`, for doing simple file I/O operations. Data objects can be initialized using the contents of a file, either using file reading operations or using `mmap()`. Using a memory-mapped `NSData` object is often a very convenient way of doing random access on a file. On 32-bit platforms you can exhaust your address space fairly quickly doing this, but on 64-bit systems you have a lot of spare address space for memory mapped files.

One big advantage of accessing files in this way is that it is very VM-friendly. If you read the contents of a file into memory and then the system is low on RAM, then it has to write out your copy to the swap file, even if you haven't modified it. If you use a `NSData` object created with `dataWithContentsOfMappedFile:` or similar, then it will simply evict the pages from memory and read them back from the original file when needed.

Since `NSData` objects can be initialized from URLs, they provide a very simple means of accessing the system's URL loading services. OS X has code for loading data from a wide variety of URL types, including files, HTTP, and FTP.

4.3.5 Caches and Discardable Data

Memory conservation is an important problem for a lot of modern applications. In recent years, the price of memory has fallen considerably, and so it becomes increasingly tempting to use some of it to store results from calculations or data received over the network. This suddenly becomes a problem when you want to port your code to a device that has a small amount of memory, like the iPhone, or when everyone is doing it.

With OS X 10.6, Apple introduced the `NSDiscardableContent` protocol. This defines a transactional API for working with objects. Before you use an object that implements this protocol, you should send it a `-beginContentAccess` message.

If this returns `YES`, then you can use the object as you would and then send an `-endContentAccess` message when you are finished. Other code may send the object a `-discardContentIfPossible` message, and if this message is received outside of a transaction, then the receiver will discard its contents.

This is easiest to understand with a concrete implementation, such as that provided by a new subclass of `NSMutableData` called `NSPurgeableData`. This behaves in exactly the same way as `NSMutableData`, but also implements the `NSDiscardableContent` protocol. When it receives a `-discardContentIfPossible` message, it will free the data that it encapsulates unless it is currently being accessed.

You may want to combine objects that uses the `NSDiscardableContent` protocol with existing code. The `-autoContentAccessingProxy` method, declared on `NSObject`, lets you do this safely. This returns a proxy object that calls `-beginContentAccess` on the receiver when it is created, and `-endContentAccess` when it is destroyed, passing all other messages on to the original object. This prevents the contents of the object from being freed as long as the proxy exists.

This is useful for storing cached data, for example, images rendered from other data in the application, that can be regenerated if required. The object remains valid, but its contents do not. This means that you can use it as a form of zeroing weak reference in non-garbage-collected environments. It is more flexible than a weak reference, however, because it provides fine-grained control over when it can be freed.

Most commonly, you will use objects that implement this protocol in conjunction with `NSCache`. This class is conceptually similar to a dictionary but is designed for storing discardable content. When you add an object to a cache, you use the `-setObject:forKey:cost:` method. The third argument defines the cost of keeping this object in the cache. When the total cost exceeds the limit set by `-setTotalCostLimit:`, the cache will attempt to discard the contents of some objects (and, optionally, the objects themselves) to reduce the cost.

Most commonly the cost is memory. When using `NSPurgeableData` instances, you would use the size as the limit. You might also use caches to limit the number of objects holding some other scarce resource, such as file handles, or even some remote resources hosted on a server somewhere.

4.3.6 Dates and Time

Time on POSIX systems is stored in `time_t` values. In a traditional UNIX system, this was a 32-bit signed value counting seconds since the UNIX epoch (the start of 1970). This means that there will be a new version of the Y2K bug some time in 2038, when this value overflows. On OS X, the `time_t` is a `long`, meaning that it is 32 bit on 32-bit systems and 64 bit on 64-bit systems. If people are still using

OS X in three hundred trillion years, when this overflows, then they probably will have had enough time to port their software to some other system.

Since the implementation of `time_t` is implementation-dependent, it was not a good fit for Cocoa. On some platforms it is an integer, on others a floating point value. Cocoa defines a `NSTimeInterval` type, which is a `double`. As a floating point value, the accuracy of an `NSTimeInterval` depends on the size of the value. A `double` has a 53-bit mantissa and a 10-bit exponent. If the least significant bit of the mantissa is a millisecond, then the value can store 9×10^{12} seconds, or around 285,427 years. If you use a range of under around a hundred thousand years, it will store half milliseconds, and so on. For any value that could be stored in a 32-bit `time_t`, the value will be accurate to under a microsecond, which is usually more accurate than is needed. The time slicing quantum for most UNIX-like systems is around 10ms, meaning that you are very unlikely to get timer events more accurately than every few tens of milliseconds.

As with other primitive values, Foundation defines both the basic primitive type and a number of classes for interacting with them in a more friendly way. These gain a little more precision by using the start of 2001 (the year OS X was publicly released) as their reference date.

Date handling is much more complex than time handling. While an `NSTimeInterval` can represent a time four hundred years ago easily, getting the corresponding calendar date is much more complex. The Gregorian calendar was introduced in 1582, but Britain didn't switch over until 1752 and Russia didn't switch until 1918. The existence of leap years and leap seconds further complicates matters, meaning that a `NSTimeInterval` may represent different dates in different locales. And all of this is before you get into the matter of time zones.

The `NSDate` class is a fairly simple wrapper around a time interval from some reference date (2001 by default, although the UNIX epoch and the current timestamp are other options). The `NSCalendarDate` subclass provides a version in the Gregorian calendar, although its use is discouraged.

With 10.4, Apple introduced the `NSCalendar` class, which encapsulates a *calendar*. A calendar is a mechanism from mapping between time intervals and dates. Early calendars were simple means of mapping between fixed dates, such as the summer and winter solstices, and seasons. Modern calendars map between time intervals and more complex dates. Cocoa understands a number of different calendars, including the Gregorian, Buddhist, Chinese, and Islamic calendars.

If you create an `NSCalendar` with `+autoupdatingCurrentCalendar`, then the calendar will automatically update depending on the currently specified locale. This means you should avoid caching values returned from the calendar, since they may change at arbitrary points in the future.

A `NSCalendar` allows you to turn a `NSDate` into an `NSDateComponents` object. This object is roughly equivalent to the POSIX `struct tm`. It allows the year, month, day, day of the week, and so on to be extracted, based on the interpretation of an `NSDate` in a specified calendar.

In general, you should always store dates in `NSDate` objects and only convert them to a given calendar when you want to display them in the user interface. This is one of the reasons why using `NSCalendarDate` is discouraged—as an `NSDate` subclass it is very tempting to use it for long-term storage—the other being that it is limited to the Gregorian calendar, making it unsuitable for use in Japan, China, and much of the rest of the world outside the Americas and Europe.

4.4 Collections

A big part of any language's standard library is providing collections, and Foundation is no exception. It includes a small number of primitive collection types defined as opaque C types and then uses these to build more complex Objective-C types.

In contrast with the C++ standard template library, Cocoa collections are heterogeneous and can contain any kind of object. All objects are referenced by pointer, so the amount of space needed to store pointers to any two objects is always the same: one word.

4.4.1 Comparisons and Ordering

For ordered collections, objects implement their own comparison. While almost any object can be stored in an array, there are more strict requirements for those that are to be stored in a set (which doesn't allow duplicates) or used as keys in a dictionary. Objects that are stored in this way must implement two methods: `-hash` and `-isEqual:`. These have a complex relationship.

1. Any two objects that are equal must return **`YES`** to `isEqual:` when compared in either order.

2. Any two objects that are equal must return the same value in response to `-hash`.

3. The hash of any object must remain constant while it is stored in a collection.

The first of these is somewhat difficult to implement by itself. It means that the following must always be true:

```
[a isEqual: b] == [b isEqual: a]
```

If this is ever not true, then some very strange and unexpected behavior may occur. This may seem very easy to get right, but what happens when you compare an object to its subclass or to an object of a different class? Some classes may allow comparisons with other classes; for example, an object encapsulating a number may decide it is equal to another object if they both return the same result to `intValue`.

An example of when this can cause problems is in the use of objects as keys in dictionaries. When you set a value for a given key in a dictionary, the dictionary first checks if the key is already in the dictionary. If it is, then it replaces the value for that key. If not, then it inserts a new value.

If `[a isEqual: b]` returns **YES** but `[b isEqual: a]` returns **NO**, then you will get two different dictionaries depending on whether you set a value for the key `a` first and then the value for the key `b`. In general, therefore, it is good practice to only use one kind of object as keys in any given collection (most commonly `NSString`s).

Listing 4.6 gives a simple example of this. This defines three new classes. The first, `A`, is a simple superclass for both of the others, which returns a constant value for the hash. It implements `copyWithZone:` in a simple way. Since this object is immutable (it has no instance variables, therefore no state, therefore no mutable state), instead of copying we just return the original object with its reference count incremented. This is required since the dictionary will attempt to copy keys, to ensure that they are not modified outside the collection (more on this later).

Listing 4.6: An invalid implementation of `isEqual:`[from: examples/isEqualFailure/dict.m]

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

@interface A : NSObject {}
@end
@interface B : A {}
@end
@interface C : A {}
@end
@implementation A
- (id) copyWithZone: (NSZone*)aZone { return [self retain]; }
- (NSString*)description { return [self className]; }
- (NSUInteger)hash { return 0; }
@end
@implementation B
- (BOOL) isEqual: (id)other { return YES; }
@end
@implementation C
- (BOOL) isEqual: (id)other { return NO; }
@end

```

```
int main(void)
{
    id pool = [NSAutoreleasePool new];
    NSObject *anObject = [NSObject new];
    NSMutableDictionary *d1 = [NSMutableDictionary new];
    [d1 setObject: anObject forKey: [B new]];
    [d1 setObject: anObject forKey: [C new]];
    NSMutableDictionary *d2 = [NSMutableDictionary new];
    [d2 setObject: anObject forKey: [C new]];
    [d2 setObject: anObject forKey: [B new]];
    NSLog(@"d1: %@", d1);
    NSLog(@"d2: %@", d2);
    return 0;
}
```

The two subclasses, `B` and `C`, have similarly trivial implementations of the `-isEqual:` method. One always returns **`YES`**; the other returns **`NO`**. In the `main()` function, we create two mutable dictionaries and set two objects for them, one with an instance of `A` and one with an instance of `B` as keys.

When we run the program, we get the following result:

```
$ gcc -framework Foundation dict.m &&./a.out
2009-01-07 16:54:15.735 a.out[28893:10b] d1: {
    B = <NSObject: 0x1003270>;
}
2009-01-07 16:54:15.737 a.out[28893:10b] d2: {
    B = <NSObject: 0x1003270>;
    C = <NSObject: 0x1003270>;
}
```

The first dictionary only contains one object, the second one contains two. This is a problem. In a more complex program, the keys may come from some external source. You could spend a long time wondering why in some instances you got a duplicate key and in others you got different ones.

Equality on objects of different classes makes the hash value even more tricky, since both objects must have the same hash value if they are equal. This means that both classes must use the same hash function, and if one has some state not present in the other, then this cannot be used in calculating the hash. Alternatively, both can return the same, constant, value for all objects. This is simple, but if taken to its logical conclusion means all objects must return 0 for their hash, which is far from ideal.

The third requirement is the hardest of all to satisfy in theory, but the easiest in practice. An object has no way of knowing when it is in a collection. If you use an object as a key in a dictionary, or insert it into a set, then modify it, then its hash might change. If its hash doesn't change, then it might now be breaking the second condition.

In practice, you can avoid this by simply avoiding modifying objects while they are in collections.

4.4.2 Primitive Collections

As mentioned earlier, Foundation provides some primitive collections as C opaque types. As of 10.5, these gained an `isa` pointer and so can be used both via their C and Objective-C interfaces. The biggest advantage of this is that they can be stored in other collections without wrapping them in `NSValue` instances. Most of the time, if you use these, you will want to use them via their C interfaces. These are faster and provide access to more functionality. The object interfaces are largely to support collections containing weak references in a garbage-collected environment. If you are not using garbage collection, or wish to use the primitive collections to store other types of value, then the C interfaces are more useful.

The simplest type of collection defined in Foundation, beyond the primitive C types like arrays and structures, is `NSHashTable`. This is a simple hash table implementation. It stores a set of unique values identified by pointers. A hash table is created using a `NSHashTableCallBacks` structure, which defines five functions used for interacting with the objects in the collection:

- `hash` defines a function returning hash value for a given pointer.
- `isEqual` provides the comparison function, used for testing whether two pointers point to equal values.
- `retain` is called on every pointer as it is inserted into the hash table.
- `release` is the inverse operation, called on objects that are removed.
- `describe` returns an `NSString` describing the object, largely for debugging purposes.

All of these correspond to methods declared by `NSObject`, and you can store these in a hash table by using the predefined set of callbacks called `NSObjectHashCallBacks` or `NSNonRetainedObjectHashCallBacks`, depending on whether you want the hash table to retain the objects when they are inserted.

The hash table model is extended slightly by `NSMapTable`. An `NSMapTable` is effectively a hash table storing pairs and only using the first element for comparisons. These are defined by two sets of callbacks, one for the key and one for the value.

Unlike other Cocoa collections, both of these can be used to store non-object types, including integers that fit in a pointer, or pointers to C structures or arrays.

4.4.3 Arrays

Objective-C, as a pure superset of C, has access to standard C arrays, but since these are just pointers to a blob of memory they are not very friendly to use. OpenStep defined two kinds of arrays: mutable and immutable. The `NSArray` class implements the immutable kind and its subclass `NSMutableArray` implements the mutable version.

Unlike C arrays, these can only store Objective-C objects. If you need an array of other objects, you can either use a C array directly or create a new Objective-C class that contains an array of the required type.

`NSArray` is another example of a class cluster. The two primitive methods in this case are `-count` and `-objectAtIndex:`. These have almost identical behavior to their counterparts in `NSString`, although the latter returns objects instead of unicode characters.

As with strings, immutable arrays can be more efficient in terms of storage than their C counterparts. When you create an array from a range in another array, for example, you may get an object back that only stores a pointer to the original array and the range—a view on the original array—avoiding the need to copy large numbers of elements.

Since Cocoa arrays are objects, they can do a lot of things that plain data arrays in C can't. The best example of this is the `-makeObjectsPerformSelector:` method, which sends a selector to every single element in an array. You can use this to write some very concise code.

With 10.5, Apple added `NSPointerArray`. This can store arbitrary pointers (but not non-pointer types). Unlike `NSArray`, it can store `NULL` values and in the presence of garbage collection can be configured to use weak references. In this case, a `NULL` value will be used for any object that is destroyed while in the array.

Variadic Initializers

Most Cocoa collections have a variadic constructor and initializer. Examples of this include `+arrayWithObjects:` and `+dictionaryWithObjectsAndKeys:`. These take a variable number of arguments, terminated with `nil` and return a constant array or dictionary with the named elements. These can be very useful for quickly constructing collections where the number of elements is known at compile time.

The Cocoa arrays are very flexible. They can be used as both *stacks* and *queues* without modification since they allow insertion at both ends with a single method. Using an array as a stack is very efficient. A stack is defined by three operations: push, pop, and top. The first of these adds a new object to the top of the stack. `NSMutableArray`'s `-addObject:` method does this. The pop operation removes the last object to have been pushed onto the stack, which is exactly what `-removeLastObject` does. The remaining operation, top, gets the object currently on the top of the stack (at the end of the array) and is provided by `NSArray`'s `-lastObject` method.

Using an array as a queue is less efficient. A queue has objects inserted at one end and removed from the other. You can cheaply insert objects at the end of the array, but inserting them at the front is very expensive. Similarly, you can remove the object from the end of an array very efficiently, but removing the first one is more expensive. The `removeObjectAtIndex:` method may not actually move the objects in the array up one if you delete the first element, however. Since `NSMutableArray` is a class cluster, certain implementations may be more efficient for removing the first element, but there is no way to guarantee this.

4.4.4 Dictionaries

Dictionaries, sometimes called *associative arrays* are implemented by the `NSDictionary` class. A dictionary is a mapping from objects to other objects, a more friendly version of `NSMapTable` that only works for objects.

It is common to use strings as keys in dictionaries, since they meet all of the requirements for a key. In a lot of Cocoa, keys for use in dictionaries are defined as constant strings. Somewhere in a header file you will find something like:

```
extern NSString *kAKeyForSomeProperty;
```

Then in a private implementation file somewhere it will say

```
NSString *kAKeyForSomeProperty = @"kAKeyForSomeProperty";
```

This pattern is found all over Cocoa and in various third-party frameworks. Often you can just use the literal value, rather than the key, but this will use a bit more space in the binary and be slightly slower, so there isn't any advantage in doing so.

As you might expect, the mutable version of a dictionary is an `NSMutableDictionary`, which adds `-setObject:forKey:` and `-removeObjectForKey:` primitive methods, and a few convenience methods.

Dictionaries can often be used as a substitute for creating a new class. If all you need is something storing some structured data, and not any methods on this data, then dictionaries are quite cheap and are very quick to create. You can create a dictionary in a single call, like this:

```
[NSDictionary dictionaryWithObjectsAndKeys:
    image, @"image",
    string, @"caption", nil];
```

This is a variadic constructor that takes a `nil`-terminated list of objects as arguments and inserts each pair into the dictionary as object and key. You can then access these by sending a `-objectForKey:` message to the resulting dictionary.

Cocoa uses this in quite a few places. Notifications store a dictionary, with a specific set of keys defined for certain notification types. This makes it easy to add additional data in the future.

4.4.5 Sets

Just as `NSDictionary` is an object built on top of the primitive `NSMapTable`, `NSSet` is an object built on top of the primitive `NSHashTable`. As in mathematics, *sets* in Cocoa are unordered collections of unique objects. Unlike an array, an object can only be in a set once.

The rules for determining whether two objects are equal are very simple. Objects in a set are first split into buckets using their hash, or some bits of the their hash for small sets. When a new object is inserted, the set first finds the correct bucket for its hash. It then tests it with every object in that bucket using `-isEqual:`. If none of them match it, then the new object is inserted.

For a `NSSet`, this is only done when the set is initialized from an array or a list of objects as arguments. `NSMutableSet` allows objects to be added to an existing set and will perform this check every time. As you might imagine, this is very slow ($\mathcal{O}(n)$) if all of the objects have the same hash value.

In addition to basic sets, OpenStep provided `NSCountedSet`. This is a subclass of `NSMutableSet` and so is also mutable. Unlike normal sets, *counted sets* (also known as *bags*) allow objects to exist more than once in the collection. Like sets, they are unordered. Another way of thinking of them is unordered arrays, although an array allows distinct-but-equal objects to exist in the same collection, while a counted set just keeps a count of objects.

With 10.3, `NSIndexSet` was also added. This is a set of integers that can be used as indexes in an array or some other integer-indexed data structure. Internally, `NSIndexSet` stores a set of non-overlapping ranges, so if you are storing sets containing contiguous ranges, then it can be very efficient.

`NSIndexSet` is not very useful by itself. It is made useful by `NSArray` methods such as `-objectsAtIndexes:`, which returns an array containing just the specified elements. Since the indexes are all within a certain range, operations on an `NSArray` using an index set only require bounds checking once, rather than for every lookup, which can make things faster.

4.5 Enumeration

The traditional way of performing enumeration on Foundation collections is via the `NSEnumerator`. This is a very simple object that responds to a `-nextObject` message and returns either the next object, or `nil` if there is no next object. To enumerate a collection using an enumerator, you simply call a method like `-objectEnumerator` on the collection and then loop sending `-nextObject` to the returned enumerator until it returns `nil`.

With 10.5, Apple added a fast enumeration system. This uses a new **for** loop construct, part of Objective-C 2.0, which handles collections.

A lot of the time, however, you don't need to use enumeration directly at all. You can use something like `NSArray`'s `-makeObjectsPerformSelector:` method. Listing 4.7 shows an example of all three ways of sending a single message to all objects in an array.

Listing 4.7: The three ways of sending a message to an object in Cocoa.[from: examples/Enumeration/enum.m]

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

@interface NSString (printing)
- (void) print;
@end
@implementation NSString (printing)
- (void) print
{
    fprintf(stderr, "%s\n", [self UTF8String]);
}
@end

int main(void)
{
    [NSAutoreleasePool new];
    NSArray* a =
        [NSArray arrayWithObjects: @"this", @"is", @"an", @"array", nil];

    NSLog(@"The␣Objective-C␣1␣way:");
    NSEnumerator *e=[a objectEnumerator];
    for (id obj=[e nextObject]; nil!=obj ; obj=[e nextObject])
    {
        [obj print];
    }
    NSLog(@"The␣Leopard␣way:");
    for (id obj in a)
27     {
28         [obj print];
29     }
30     NSLog(@"The␣simplest␣way:");
31     [a makeObjectsPerformSelector: @selector(print)];
32     return 0;
33 }
```

Lines 20–24 show how to use an enumerator. This is quite complex and easy to make typos in, so in Étoilé we hide this pattern in a `FOREACH` macro (which also does some caching to speed things up slightly). A simpler version is shown in lines 26–29, using the fast enumeration pattern. This is both simpler code and faster, which is quite a rare achievement. The final version, on line 31, is even simpler. This is a single line. If you want to send more than one message, or messages with more than one argument, then this mechanism is unavailable.

Running this code, we get

```
$ gcc -std=c99 -framework Foundation enum.m && ./a.out
2009-01-07 18:06:41.014 a.out[30527:10b] The Objective-C 1 way:
this
is
an
array
2009-01-07 18:06:41.020 a.out[30527:10b] The Leopard way:
this
is
an
array
2009-01-07 18:06:41.021 a.out[30527:10b] The simplest way:
this
is
an
array
```

4.5.1 Enumerating with Higher-Order Messaging

An additional way of performing enumeration, among other things, was proposed by Marcel Weiher. The mechanism, called *higher-order messaging* (*HOM*) uses the proxy capabilities of Objective-C. It adds methods like `-map` to the collection classes. When these are called, they return a proxy object that bounces every message sent to them to every object in the array.

Listing 4.8 shows a `-map` method added as a category on `NSArray`. This is taken from the EtoileFoundation framework, with the Étoilé-specific macros removed.

This framework is available under a BSD license, and so you can use it in your own projects if you wish.

Listing 4.8: A example of a map method implemented using higher-order messaging. [from: examples/HOM/NSArray+map.m]

```
@interface NSArrayMapProxy : NSProxy {
    NSArray * array;
}
- (id) initWithArray:(NSArray*)anArray;
@end

@implementation NSArrayMapProxy
- (id) initWithArray:(NSArray*)anArray
{
    if (nil == (self = [self init])) { return nil; }
    array = [anArray retain];
    return self;
}
- (id) methodSignatureForSelector:(SEL)aSelector
{
    for (object in array)
    {
        if([object respondsToSelector:aSelector])
        {
            return [object methodSignatureForSelector:aSelector];
        }
    }
    return [super methodSignatureForSelector:aSelector];
}
- (void) forwardInvocation:(NSInvocation*)anInvocation
{
    SEL selector = [anInvocation selector];
    NSMutableArray * mappedArray =
        [NSMutableArray arrayWithCapacity:[array count]];
    for (object in array)
    {
        if([object respondsToSelector:selector])
        {
            [anInvocation invokeWithTarget:object];
            id mapped;
            [anInvocation getReturnValue:&mapped];
            [mappedArray addObject:mapped];
        }
    }
    [anInvocation setReturnValue:mappedArray];
}
- (void) dealloc
{
    [array release];
    [super dealloc];
}
@end

@implementation NSArray (AllElements)
- (id) map
{
    return [[[NSArrayMapProxy alloc] initWithArray:self] autorelease];
}
@end
```

The `-map` method itself is relatively simple; it just creates an instance of the proxy, associates it with the array, and returns it. You would use this category like this:

```
[[array map] stringValue];
```

This would return an array containing the result of sending `-stringValue` to every element in `array`. When you send the `-stringValue` message to the proxy, the runtime calls the `-methodSignatureForSelector:` method. This is used to find out the types of the method. This implementation simply calls the same method on every object in the array until it finds one which returns a value.

Next, the `-forwardInvocation:` method will be called. This has an encapsulated message as the argument. The body of this method sends this message to every object in the array and then adds the result to a new array.

Unlike the `-makeObjectsPerformSelector:`, messages sent to objects using higher-order messaging can have an arbitrary number of arguments. Exactly the same mechanism can be used to implement a variety of other high-level operations on collections, such as folding or selecting.

Although the use of the forwarding mechanism makes this relatively slow, compared with other enumeration mechanisms, the fact that it preserves high-level information in the source code can make it attractive. It results in less duplicated code and code that is easier to write. HOM is used a lot in modern Smalltalk implementations, although the initial implementation was in Objective-C.

Higher-order messaging is not limited to enumeration. It is also used for a wide number of other tasks, including sending messages between threads. We'll look more at how to use it for asynchronous messaging in Chapter 23.

4.5.2 Enumerating with Blocks

OS X 10.6 added blocks, which we looked at in the last chapter, to the C family of languages. Blocks by themselves are quite useful, but their real power comes from their integration with the rest of the Foundation framework. This integration comes from a number of new methods, such as this one on `NSArray`:

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

The argument is a block taking three arguments: an object, the index at which that object appears in the array, and a pointer to a boolean value to set if enumeration should stop. We could rewrite the same enumeration example that we used earlier with a block as:

```
[a enumerateObjectsUsingBlock:
        ^(id obj, NSUInteger idx, BOOL *stop) { [obj print]; } ];
```

The requirement to put the types of the block arguments inline makes this quite difficult to read, but you could split it up a bit by declaring the block separately and then calling it. In this example, the block doesn't refer to anything other than its arguments, so using a block is equivalent to using a function pointer, with the exception that a block can be declared inline.

The method shown above is a simplified version. The more complex variant includes an options parameter that is an `NSEnumerationOptions` value. This is an enumerated type that specifies whether the enumeration should proceed forward, in reverse, or in parallel. If you specify `NSEnumerationConcurrent`, then the array may spawn a new thread or use a thread from a pool to split the enumeration across multiple processors. This is usually only a good idea for large arrays or blocks that take a long time to execute.

Foundation defines two other kinds of blocks for use with collections: *test blocks* and *comparator blocks*. A test block returns a `BOOL`, while a comparator is defined by the `NSComparator` `typedef`:

```
typedef NSComparisonResult (^NSComparator)(id obj1, id obj2);
```

Comparator blocks are used everywhere that sorting might be performed. Both mutable and immutable arrays can be sorted with comparators but so can sets and dictionaries. This includes some quite complex methods, such as this one from `NSDictionary`:

```
- (NSArray*)keysSortedByValueUsingComparator: (NSComparator)cmptr;
```

The argument to this method is a comparator block that defines the ordering of two objects. This will be called with all of the values in the dictionary, and the method will return an array containing all of the keys in the order that their values are listed. This can then be used to visit the values in this order by sending `-valueForKey:` messages to the dictionary.

Test blocks are used for filtering. Unlike comparators, they do not have a specific type associated with them because each class defines the arguments that a test block takes. For example, `NSIndexSet` test blocks take an `NSUInteger` argument, while tests for `NSDictionary` take both the key and value as arguments.

Most collection classes, including those outside of Foundation, such as those managed by the *Core Data* framework support `NSPredicate` as a means of filtering. As of OS X 10.6, you can also create `NSPredicate` instances from test blocks. You can also create `NSSortDescriptor` instances, which are heavily used in conjunction with *Cocoa bindings* from comparator blocks and, using the `-comparator` method turn an `NSSortDescriptor` object into a comparator block.

4.5.3 Supporting Fast Enumeration

From time to time, you will want to implement your own collection classes, and want to use them with the new `for...in` loops. If your collection can create enumerators, then you can use the enumerator as the enumerator's support for fast enumeration, but this is slightly unwieldy and slow. Full support requires collections to conform to the `NSFastEnumeration` protocol and implement the following method:

```
- (NSUInteger)countByEnumeratingWithState: (NSFastEnumerationState*)state
                                  objects: (id*)stackbuf
                                    count: (NSUInteger)len;
```

Understanding this method requires understanding the `NSFastEnumerationState` structure. This is defined in the NSEnumerator.h Foundation header, as shown in Listing 4.9.

Listing 4.9: The fast enumeration state structure. [from: NSEnumerator.h]

```
typedef struct {
    unsigned long state;
    id *itemsPtr;
    unsigned long *mutationsPtr;
    unsigned long extra[5];
} NSFastEnumerationState;
```

The first time this method is called, it should initialize the `mutationsPtr` field of this structure. This must be a valid pointer to something at least as big as a `long`. The caller will automatically cache the pointed-to value when the method is first called and then compare this on every subsequent iteration through the loop. If it has changed, then an exception will be thrown. In contrast, an `NSEnumerator` has no way of knowing if the collection it is enumerating has changed.

The second argument is a pointer to a buffer allocated by the caller, and the third is the size of this buffer. If the collection stores objects internally in a C array, it can return a pointer to this directly by setting `state->itemsPtr` to the array and returning the number of elements. Otherwise, it copies up to `len` elements into `stackbuf` and returns the number it copies. The compiler currently sets `len` to 16, and so only a single message send is required for every 16 items enumerated. In contrast, at least 32 will be required when using an enumerator (one to the enumerator and one from the enumerator to the collection). It is easy to see why Apple calls this the 'fast enumeration' system.

To see how you can support fast enumeration in your own collections, we will create two new classes, as shown in Listing 4.10. These both conform to the `NSFastEnumeration` protocol. One is mutable and the other immutable. Supporting fast enumeration is done slightly differently for mutable and immutable objects.

Listing 4.10: Integer array interfaces. [from: examples/FastEnumeration/IntegerArray.h]

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

@interface IntegerArray : NSObject<NSFastEnumeration> {
    NSUInteger count;
    NSInteger *values;
}
- (id)initWithValues: (NSInteger*)array count: (NSUInteger)size;
- (NSInteger)integerAtIndex: (NSUInteger)index;
@end

@interface MutableIntegerArray : IntegerArray {
    unsigned long version;
}
- (void)setInteger: (NSInteger)newValue atIndex: (NSUInteger)index;
@end
```

The most noticeable difference in the interface is that the mutable version has a `version` instance variable. This is used to track whether the object has changed during enumeration.

The immutable version is shown in Listing 4.11. The first two methods are very simple; they just initialize the array and allow values to be accessed. The array is a simple C buffer, created with `malloc()`. The `-dealloc` method frees the buffer when the object is destroyed.

The fast enumeration implementation here returns a pointer to the instance variable, on line 28. This code is written to support partial enumeration, where the caller only requests some subset of the total collection. This is not currently supported by the compiler, but, because this is just an Objective-C method, you cannot guarantee that it will not be called directly by some code wanting just the

Listing 4.11: A simple immutable integer array supporting fast enumeration.
[from: examples/FastEnumeration/IntegerArray.m]

```
@implementation IntegerArray
- (id)initWithValues: (NSInteger*)array count: (NSUInteger)size
{
    if (nil == (self = [self init])) { return nil; }
    count = size;
    NSInteger arraySize = size * sizeof(NSInteger);
    values = malloc(arraySize);
    memcpy(values, array, arraySize);
    return self;
}
- (NSInteger)integerAtIndex: (NSUInteger)index
{
    if (index >= count)
    {
        [NSException raise: NSRangeException
                    format: @"Invalid␣index"];
    }
    return values[index];
}
- (NSUInteger)countByEnumeratingWithState: (NSFastEnumerationState*)state
                                  objects: (id*)stackbuf
                                    count: (NSUInteger)len
{
    NSUInteger n = count - state->state;
    state->mutationsPtr = (unsigned long *)self;
    state->itemsPtr = (id*)(values + state->state);
    state->state += n;
    return n;
}
- (void)dealloc
{
    free(values);
    [super dealloc];
}
@end
```

values after a certain element. The `state` field will be set to the first element that the caller wants. In normal use, this will be either 0 or `count`. The items pointer is set to the correct offset in the instance variable array using some simple pointer arithmetic.

The `state` field is updated to equal the index of the last value and the array is returned. Any `for...in` loop will call this method twice. After the first call the `state` field will have been set to `count`. In the second call, the value of `n` will be set to 0 and the loop will terminate.

Note that the mutations pointer is set to **`self`**. Dereferencing this will give the `isa` pointer. This class does not support modifying the values, but some other code may change the class of this object to a subclass that does. In this case, the mutation pointer will change. This is very unlikely; for most cases the **`self`** pointer is a convenient value because it is a pointer that is going to remain both valid and constant for the duration of the loop.

The mutable case is a bit more complicated. This is shown in Listing 4.12. This class adds a method, allowing values in the array to be set. Note that on line 42 the version is incremented. This is used to abort enumeration when the array is modified.

The enumeration method in this class sets the mutation pointer to the address of the `version` instance variable. The initial value of this is cached by the code generated from the loop construct, and every loop iteration will be compared against the current value to detect changes.

Listing 4.12: A simple mutable integer array supporting fast enumeration. [from: examples/FastEnumeration/IntegerArray.m]

```
39 @implementation MutableIntegerArray
40 - (void)setInteger: (NSInteger)newValue atIndex: (NSUInteger)index
41 {
42     version++;
43     if (index >= count)
44     {
45         values = realloc(values, (index+1) * sizeof(NSInteger));
46         count = index + 1;
47     }
48     values[index] = newValue;
49 }
50 - (NSUInteger)countByEnumeratingWithState: (NSFastEnumerationState*)state
51                                   objects: (id*)stackbuf
52                                     count: (NSUInteger)len
53 {
54     NSInteger n;
55     state->mutationsPtr = &version;
56     n = MIN(len, count - state->state);
57     if (n >= 0)
58     {
59         memcpy(stackbuf, values + state->state, n * sizeof(NSInteger));
60         state->state += n;
61     }
62     else
```

```
    {
        n = 0;
    }
    state->itemsPtr = stackbuf;
    return n;
}
@end
```

Because the mutable array's internal array can be reallocated and become invalid, we copy values out onto the stack buffer. This is not technically required; the collection is not thread-safe anyway, and so the array cannot be accessed in a way that would cause problems, but it's done here as an example of how to use the stack buffer.

The stack buffer has a fixed size. This is typically 16 entries. On line 56, we find which is smaller out of the number of slots in the stack buffer and the number of elements left to return. We then copy this many elements on line 59. The items pointer is then set to the stack buffer's address.

Using the stack buffer is entirely optional. Our immutable array didn't use it, while this one does. It is there simply as a convenient place to put elements if the class doesn't use an array internally.

To test these two classes, we use the simple program shown in Listing 4.13. This creates two integer arrays, one mutable and one immutable, and iterates over both of them using the fast enumeration `for...in` loop construct.

Listing 4.13: Testing the fast enumeration implementation. [from: examples/FastEnumeration/test.m]

```
#import "IntegerArray.h"

int main(void)
{
    [NSAutoreleasePool new];
    NSInteger cArray[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
        15, 16, 17, 18, 19, 20};
    IntegerArray *array = [[IntegerArray alloc] initWithValues: cArray
                                                          count: 20];
    NSInteger total = 0;
    for (id i in array)
    {
        total += (NSInteger)i;
    }
    printf("total:␣%d\n", (int)total);
    MutableIntegerArray *mutablearray =
        [[MutableIntegerArray alloc] initWithValues: cArray
17                                                   count: 20];
18      [mutablearray setInteger: 21 atIndex: 20];
19      for (id i in mutablearray)
20      {
21          total += (NSInteger)i;
22      }
23      printf("total:␣%d\n", (int)total);
24      for (id i in mutablearray)
25      {
26          total += (NSInteger)i;
27          printf("value:␣%d\n", (int)(NSInteger)i);
28          [mutablearray setInteger: 22 atIndex: 21];
29      }
30      return 0;
31  }
```

Note that the type of the element in these loops has to be an **id**. This is only a requirement of the type checker. The compiler does not insert any message sends to the returned objects, so as long as they are the same size as an **id** they can be returned.

On line 28, we modify the collection inside a loop. This is exactly the kind of thing that the fast enumeration structure's mutation pointer field is intended to detect. If we have written the implementation correctly, then an exception will be thrown. Running the program, we see that this does happen:

```
$ gcc -framework Foundation *.m && ./a.out
total: 210
total: 441
value: 1
2009-02-21 16:24:56.278 a.out[4506:10b] *** Terminating app
due to uncaught exception 'NSGenericException', reason:
'*** Collection <MutableIntegerArray: 0x1004bb0> was mutated
while being enumerated.'
```

We didn't have to do anything explicit here to raise the exception. Just modifying the `version` instance variable did it. Note that the exception was raised after the first loop iteration, even though the first 16 values were all returned at once. This is why the mutation pointer is a pointer and not just a value. If it had been a simple value that we had set to the value of the `version` ivar in each call, the loop would not have been able to detect the mutation until the next call. Because it is a pointer, it can be dereferenced and tested very cheaply at each loop iteration and so the mutation is caught the first time the caller tries to load a value from the array that has since changed.

Because we modified the version counter before making any changes to the array, we guaranteed that the mutation will always be caught. If you add any other mutation methods to the class, just remember to add `version++` at the start, and this will keep working.

4.6 Property Lists

Property lists (*plists*) are a feature of OPENSTEP and OS X that crop up in a lot of places. The simplicity of the form and its utility have caused a number of other systems to implement support for property lists.

The original NeXT property lists were a simple ASCII format with single characters representing the borders of different types of collection. This had a few limitations. It was difficult to extend, and relatively difficult to embed in other formats. To address these problems, OS X added an XML format for property lists. This can store a few more formats than the original format and, with proper namespacing, can be embedded in any XML format.

Unfortunately, XML is a very verbose format. Parsing it is relatively expensive and storing it requires a lot of space. To address this, Apple added a third format, which is a proprietary (and undocumented) binary encoding. This encoding is very fast to parse and very dense.

The NeXT format is largely deprecated on OS X, although a few command-line tools (such as the `defaults` utility) still use it since it is the most human-readable. The XML format is primarily used for interchange and the binary format is used for local storage. Table 4.1 shows a comparison of the XML and OpenStep property list formats. In particular, you will see how much more verbose arrays and dictionaries are in the new XML format.

It is worth noting that GNUstep has extended the OpenStep format to allow `NSValue` and `NSDate` objects to be stored, making GNUstep OpenStep-style property lists as expressive as XML ones, at the cost of interoperability.

The `plutil` tool can convert to the XML and binary formats. It can also convert from the original NeXT format, but not to it. This is due to the fact that this format is less expressive than the new versions—for example, it cannot store `NSDate` objects—and so the tool cannot guarantee that property lists can be safely converted. As well as converting, the `-lint` option to the tool causes it to check a `plist` file and report errors. This is very useful if you ever need to edit a property list by hand.

Type	Cocoa Class	NeXT	XML
String	`NSString`	`"a string"`	`<string>a string </string>`
Boolean	`NSNumber`	N/A	`<true />` or `<false />`
Integer	`NSNumber`	12	`<integer>12</integer>`
Floating Point	`NSNumber`	12.42	`<real>12.42</real>`
Date	`NSDate`	N/A	`<date>2009-01-07T13:39Z </date>`
Binary data	`NSData`	<666f6f>	`<data>fooZm9v</data>`
Arrays	`NSArray`	`( "a" )`	`<array> <string>a</string> </array>`
Dictionaries	`NSDictionary`	`{"key" = "value";}`	`<dict> <key> <string>key</string> </key> <value> <string>value</string> </value> </dict>`

Table 4.1: The data types that can be stored in OpenStep (NeXT) and XML (Apple) property lists.

4.6.1 Serialization

The main use for property lists is to store collections of data in a format that can be easily written out and read back in. Table 4.1 showed the Cocoa objects that correspond to various elements in a property list, but because only a relatively small set of general formats of data are supported, Cocoa is not the only system that can handle plists.

The Core Foundation library also supports all of the data types, and so does CFLite. This means that property lists can be used by simple C applications without requiring all of Cocoa or even Core Foundation. There are other libraries available for parsing property lists. NetBSD's proplib is a BSD-licensed C library for handling XML property lists without any Apple code.

This means that Cocoa can be used to create configuration or simple data files in a format that is easy to parse and usable by non-Cocoa apps using a variety of toolkits. A lot of core configuration information for OS X is stored in property lists and accessed long before any Cocoa applications start.

The Cocoa collections can be written to property list files directly and read from them with a single call. Listing 4.14 shows an example of this. The program creates a simple array, writes it to a file, and reads it back into a new array. Logging the two arrays shows that they are equivalent.

Listing 4.14: Storing an array in a property list. [from: examples/PropertyList/plist.m]

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

int main(void)
{
    [NSAutoreleasePool new];
    NSArray *a = [NSArray arrayWithObjects:@"this", @"is", @"an", @"array",
         nil];
    [a writeToFile:@"array.plist" atomically:NO];
    NSArray *b = [NSArray arrayWithContentsOfFile:@"array.plist"];
    NSLog(@"a:␣%@", a);
    NSLog(@"b:␣%@", b);
    return 0;
}
```

When we run this program, we can verify that it works correctly:

```
$ gcc -framework Foundation plist.m && ./a.out
2009-01-07 19:13:15.299 a.out[34155:10b] a: (
    this,
    is,
    an,
    array
)
2009-01-07 19:13:15.300 a.out[34155:10b] b: (
    this,
    is,
    an,
    array
)
```

Although this simple example just contains strings, the same code will work on an array containing any of the types that can be stored in a property list.

This basic functionality is enough for a lot of uses, but for more advanced cases the `NSPropertyListSerialization` class is helpful. This provides the ability to validate property lists, and to load and store them from `NSData` objects in memory, rather than from files. The plutil utility mentioned earlier is a very simple wrapper around this class.

To create a property list with NSPropertyListSerialization, you would use this method:

```
+ (NSData *)dataFromPropertyList: (id)plist
                          format: (NSPropertyListFormat)format
                errorDescription: (NSString**)errorString;
```

The plist object is an object that will be turned into property list form. The format can be either NSPropertyListXMLFormat_v1_0 for the XML format, or NSPropertyListBinaryFormat_v1_0 for binary property lists. There is also NSPropertyListOpenStepFormat defined by the NSPropertyListFormat enumeration, but this is only valid for reading OpenStep property lists—OS X no longer has the capability to write them. The final parameter is a pointer to a string that will be used to return an error message.

This method is quite unusual in taking a pointer to a string as a parameter for returning an error. This is due to its age. It was introduced with OS X 10.2. Prior to this, exceptions were always used for returning errors and methods that could soft-fail returned a BOOL. With 10.2.7 (or earlier versions with Safari installed), Apple introduced the NSError class. A pointer to a pointer to an instance of this class is often passed in as a final argument, and set to non-nil on return, but this method was written just a few months too early to take advantage of it.

For deserializing property lists, the converse method is

```
+ (id)propertyListFromData: (NSData*)data
          mutabilityOption: (NSPropertyListMutabilityOptions)opt
                    format: (NSPropertyListFormat*)format
          errorDescription: (NSString**)errorString
```

Most of the parameters are the same here. The format is now an output parameter, which is set to the format of the property list, which is autodetected. The new parameter, opt, defines whether deserialized objects should be mutable. Property lists do not store mutability options—an NSString and an NSMutableString will be stored in the same way—so you must specify this when deserializing. You can define whether everything (NSPropertyListMutableContainersAndLeaves), only containers (NSPropertyListMutableContainers), or nothing (NSPropertyListImmutable) should be mutable.

4.6.2 User Defaults

One of the problems on any system is how to store preferences for an application. Numerous solutions have been suggested for this problem, from individual configuration files to a centralized registry. OS X picks a path somewhere in the middle. Each application has a property list file containing a dictionary associated with it.

This is accessed via the `NSUserDefaults` class, which also handles notifying parts of an application of changes to individual keys.

This combines most of the benefits of both approaches. You can treat user defaults as a system-maintained database. The `defaults` command-line tool can browse and modify the defaults for any application. Since they are just property lists, you can also modify them outside of the defaults system and delete them for unwanted applications.

To get a user defaults object for your application, you do

```
[NSUserDefaults standardUserDefaults];
```

This returns a singleton object. You can call this as many times as you want from your application and still be using the same object. It maintains a copy of the defaults in memory and periodically synchronizes it with the copy stored on disk. Alternatively, you can explicitly synchronize it by sending a `-synchronize` message.

The shared defaults object is similar to an `NSMutableDictionary`. It has a `setObject:forKey:` and an `objectForKey:` method, which set and get objects, respectively. There are also some convenience methods, like `boolForKey:` that fetches the boxed value, unboxes it, and returns a `BOOL`.

Since user defaults supports *key-value coding* (*KVC*), you can also use the standard KVC methods to access defaults. In particular, this includes the `valueForKeyPath:` method. This is very useful when you have a set of attributes stored in a dictionary in defaults. You can get at a nested key with a single call:

```
[[NSUserDefaults standardUserDefaults] valueForKeyPath: @"dict.key"];
```

As long as the user defaults system contains a dictionary called "dict" that contains a key called "key," this will return the corresponding value. You can use this for even deeper-nested dictionaries using a longer key path. Unfortunately, using the corresponding `setValue:forKeyPath:` method will cause a run-time exception.

Defaults and the Command Line

The user defaults system searches for values in a set of *defaults domains*. One of the most often overlooked is `NSArgumentDomain`. This can be very useful as a way of getting values from the command line. If you start a Cocoa application from the command line, you can override user defaults settings by specifying them on the command line in the form `-default value`. You can also use this as a quick and easy way of defining command-line options for tools that use the Foundation framework. To do this, you just need to pass the name of the command-line option as the key when loading a value from defaults.

There are a few difficulties with mutable collections in defaults. This is not directly supported, and so you must do it by creating a new copy of the collection and inserting this back into defaults. This typically involves the following steps:

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
NSMutableDictionary *d = [[defaults dictionaryForKey: aKey] mutableCopy];
// Some operations on d
[defaults setObject: d forKey: aKey];
```

This is somewhat clumsy, and so it is common to wrap it up in a method. In particular, you should remember that categories allow you to add methods to `NSUserDefaults`. If your application stores a dictionary of sounds, then you might consider adding a `-setSound:forAction:` or similar method that sets an entry in the sounds dictionary in defaults.

User defaults only supports storing objects that can be saved in property lists. A notable exception to this is the `NSColor` object. Apple suggests adding a category on `NSUserDefaults` for storing these, using the `NSArchiver` mechanism.

`NSArchiver` allows objects that support creating a serialized form of objects that implement the `NSCoding` mechanism. If the object you want to store in defaults implements these, then `NSArchiver` can turn it into a `NSData` instance and `NSUnarchiver` can restore it. This provides a mechanism for storing the object in defaults, since the defaults system can handle `NSData` instances.

Often, a more lightweight approach is possible. Some objects, like `NSURL` can easily be stored as strings. Listing 4.15 shows a simple category for storing URLs as strings in defaults. If your object already implements `-stringValue` and `-initWithString:` methods, then you might find this mechanism simpler than implementing `NSCoding`.

Listing 4.15: A category for storing URLs in defaults. [from: examples/URLDefaults/urldefaults.m]

```
@implementation NSUserDefaults (NSURL)
- (void) setURL: (NSURL*)aURL forKey: (NSString*)aKey
{
    [self setObject: [aURL absoluteString] forKey: aKey];
}
- (NSURL*) URLForKey: (NSString*)aKey
{
    return [NSURL URLWithString: [self stringForKey: aKey]];
}
@end
```

Either mechanism can be used for your own objects, depending on their complexity.

4.7 Interacting with the Filesystem

Most nontrivial programs need to interact with the filesystem in some way. On most UNIX-like systems, including OS X, the filesystem is the only persistent storage facility provided. User defaults is just a high-level interface to a small part of the filesystem, providing access to specific files via a dictionary-like interface.

How you want to interact with the filesystem depends a lot on the task at hand. Cocoa provides a number of facilities exposing files as UNIX-style streams of bytes, or as structured data of some kind. Which you should use depends on your requirements.

4.7.1 Bundles

Bundles are a very important part of OS X. They were used on NeXT systems and have gradually replaced resource forks from earlier versions of Mac OS. The big advantage is not needing any special filesystem support.

Applications on OS X are bundles and can have other resources as well as the code. On NeXT systems, application bundles were used to store different versions of the executable for different platforms; you could have a single .app on an NFS share and run it on OPENSTEP, Solaris, or any other platform that it supported. This legacy is still found in OS X today. The binary is in the Contents/MacOS directory inside the bundle. In theory, you could add binaries for other platforms, although this is not currently supported by the Apple tools.

Prior to the release (and naming) of OS X, the in-development successor to Classic MacOS was called *Rhapsody*. Three “boxes” were announced by Apple. Two eventually became part of OS X. *Blue box* was the virtualized compatibility layer for MacOS that was called *Classic* on early versions of OS X and is not present on Intel Macs. The *yellow box* was the OpenStep environment that was later rebranded Cocoa. The final box, the *red box*, never made it to a shipping product and was a Windows environment for OS X similar to WINE. There was also a planned Windows version of the yellow box, based on the *OPENSTEP Enterprise* (*OSE*) product from NeXT, including Project Builder and Interface Builder and allowing Cocoa applications to be developed for Windows.

It seems likely that Apple still maintains descendants of the Windows version of the yellow box internally and uses them for porting applications like Safari to Windows, although Apple does not use the bundle architecture for these applications. Although the red box was not shipped, it was seen as a possible future product for long enough for OS X to retain the ability to run application bundles with executables in entirely different formats.

OS X, like OPENSTEP, uses the *Mach-O* binary format, which supports different format executables in the same binary files (sharing constants and data when the endian is the same). This is more efficient than having independent binaries for each version and allows Intel and PowerPC, 32-bit and 64-bit executables to be included in the same file. NeXT called these *fat binaries*, while Apple opted for the more politically correct *universal binaries*.

Because applications are bundles, every application has at least one bundle that it will want to load resources from. In a very simple application this happens automatically. The main nib for the application will be loaded when it starts and connected to the application delegate and any other objects.

Other resources can be loaded from the application bundle with the `NSBundle` class. In general, you will have one instance of this class for each bundle you want to interact with. You can get the application bundle with

```
[NSBundle mainBundle];
```

Be careful when doing this. At some point in the future you may decide that your class is very useful and that you want to reuse it. When you do this, you will move it and a framework—another kind of bundle containing a loadable library, headers, and resources—along with any resources it might want to load. When you get the main bundle from your class, you will get the application bundle for the application that linked against the framework, rather than the framework bundle. If you are getting a bundle to load resources that are included with the class then this is not what you want. Instead, you should use

```
[NSBundle bundleForClass: [self class]];
```

This is relatively slow, so it is best done in the `+initialize` method for the class and cached in a file-static variable, like this:

```
static NSBundle *frameworkBundle;
+ (void) initialize
{
   frameworkBundle = [[NSBundle bundleForClass: self] retain];
}
```

In real code, you would probably want to wrap this in a check to ensure that it was only being called on the correct class, as shown in Chapter 3. Because this is a class method, it only needs to pass `self`, rather than `[self class]` as the parameter. You can also use `+bundleWithIdentifier`, which is generally faster. This loads the bundle that has the identifier provided as the argument. The bundle identifier is set in the bundle's property list by the CFBundleIdentifier key.

Once you have a bundle, you can load resources from it. This is a two-step process. The first is to find the path of the resource, using a method like this:

```
- (NSString*)pathForResource: (NSString*)name
                      ofType: (NSString*)extension
                 inDirectory: (NSString*)subpath
             forLocalization: (NSString*)localizationName
```

There are two wrapper versions of this method where the last parameters are filled in with default values. The simplest form just has the first two and finds resources using the user's preferred localization in the top-level resource directory in the bundle.

If you want to load all of the resources of a specific type in a bundle, there is a form that returns an array instead of a string:

```
- (NSArray*)pathsForResourcesOfType: (NSString*)extension
                        inDirectory: (NSString*)subpath
```

This, and the version that specifies a localization, finds all of the resources of a specific type, for example, all of the png files in a theme directory in the Resources directory in the bundle.

In addition to resources, you can load code from bundles, too. Listing 4.16 shows a simple framework loader. Because frameworks are just another kind of bundle with a well-known layout, the standard bundle loading code can be used to load them.

This example is taken from Étoilé's LangaugeKit and is used to allow scripts loaded and compiled by a running program to specify frameworks that they depend upon, without requiring the program that loads them to link against every possible framework that a script might want.

This example shows a number of Cocoa features. The first is the file manager, which we will look at in the next section. This is used in line 24 to test whether the framework exists at a given path. If it does, then `NSBundle` is used on lines 27 and 28 to load the code in the framework.

Listing 4.16: A simple framework loader. [from: examples/Loader/simpleLoader.m]

```
 7 @implementation SimpleLoader
 8 + (BOOL) loadFramework: (NSString*)framework
 9 {
10     NSFileManager *fm = [NSFileManager defaultManager];
11     NSArray *dirs =
12         NSSearchPathForDirectoriesInDomains(
13             NSLibraryDirectory,
14             NSAllDomainsMask,
15             YES);
16     FOREACH(dirs, dir, NSString*)
17     {
18         NSString *f =
19             [[[dir stringByAppendingPathComponent: @"Frameworks"]
20                 stringByAppendingPathComponent: framework]
21                     stringByAppendingPathExtension: @"framework"];
22         // Check that the framework exists and is a directory.
23         BOOL isDir = NO;
24         if ([fm fileExistsAtPath: f isDirectory: &isDir]
25             && isDir)
26         {
27             NSBundle *bundle = [NSBundle bundleWithPath: f];
28             if ([bundle load])
29             {
30                 NSLog(@"Loaded␣bundle␣%@", f);
31                 return YES;
32             }
33         }
34     }
35     return NO;
36 }
37 @end
```

The function on line 11 is one of the most useful, and most overlooked, parts of Cocoa, since it allows you to avoid hard-coding paths in a lot of instances. Line 19 shows some of `NSString`'s path manipulation code. This is used to assemble the correct path by appending the Frameworks directory, then the framework name as path components, and then the .framework extension. This could be done with `-stringWithFormat:` for OS X, but doing it this way means that it will continue to work if you try to move your code to a different platform with different path formats.

4.7.2 Workspace and File Management

Cocoa provides two ways of interacting with the filesystem, `NSFileManager` and `NSWorkspace`. The latter is part of AppKit and provides a higher-level interface. The `NSWorkspace` class does file operations in the background and posts a notification when they are done, while `NSFileManager` works synchronously. Both classes are *singletons*; you will only ever have (at most) one instance for each in an application.

We saw an example of one of the things you can do with a file manager in Listing 4.16. This used the `-fileExistsAtPath:isDirectory:` method, to see if a file existed. The second argument to this is a pointer to a `BOOL`, which is set to `YES` if the file is found and is a directory.

Most other common file manipulation operations are supported by the file manager, such as copying, moving, and linking files and directories. It can also enumerate the contents of folders and compare files. Most of `NSFileManager`'s functionality is exposed by a single method in `NSWorkspace`:

```
- (BOOL)performFileOperation: (NSString*)operation
                      source: (NSString*)source
                 destination: (NSString*)destination
                       files: (NSArray*)files
                         tag: (NSInteger)tag
```

This takes a source and destination directory as arguments and an array of files. It performs move, copy, link, destroy, or recycle operations and sets the value of the integer pointed to by `tag` to indicate whether the operation succeeded.

Most of `NSWorkspace`'s functionality deals with higher-level operations on files. While `NSFileManager` is for dealing with files as UNIX-style streams of bytes, `NSWorkspace` is for dealing with files as a user-level abstraction, representing documents or applications. Methods like `openFile:` are examples of this. This method opens a specified file with the default application and is used to implement the command-line `open` tool.

The low-level file manager methods are very easy to use. Listing 4.17 shows a simple tool for copying a file. This uses the user defaults system to read command-line arguments and then uses the file manager to copy the specified file.

Note that this example code does not check whether the input file exists or that the output is a valid destination. The file manager will call a delegate method in case of an error, but we did not set a handler on line 12, and so this will not allow error checking either. Implementing the handler is not required, it simply allows you to track the progress of the operation and to decide whether to proceed in case of an error. The return value from this method is a boolean indicating whether the copy succeeded. You can run this simple tool like this:

```
$ gcc -framework Foundation FileCopy.m -o FileCopy
$ ./FileCopy -in FileCopy -out CopyOfFileCopy
$ ls
CopyOfFileCopy FileCopy       FileCopy.m
```

Note that the file manager automatically resolved relative paths. These are treated as being relative to whatever the file manager returns from `-currentDirectoryPath`. You can alter the working directory for a running program by sending the file manager a `-changeCurrentDirectoryPath:` message. The working directory is much more important for command-line tools than it is for

Listing 4.17: A simple tool for copying files. [from: examples/FileCopy/FileCopy.m]

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

int main(void)
{
    [NSAutoreleasePool new];
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSString *source = [defaults stringForKey: @"in"];
    NSString *destination  = [defaults stringForKey: @"out"];
    NSFileManager *fm = [NSFileManager defaultManager];
    [fm copyPath: source
          toPath: destination
         handler: nil];
    return 0;
}
```

graphical applications. A command-line tool inherits its current working directory from the shell. The concept of a current directory does not make sense for an application invoked via the Finder or from the Dock.

Starting with 10.5, Apple began using the *uniform type identifier* (*UTI*) system for identifying file types. A UTI is a hierarchical arrangement of types. `NSWorkspace` is used to map between file extensions and UTIs.

4.7.3 Working with Paths

When working with filesystem paths, there are a number of helpful methods provided by `NSString`. These allow you to decompose paths into components and create various individual components without worrying about the kind of path that is represented.

On UNIX platforms, a tilde (˜) is commonly used as a shorthand for the user's home directory. You can get this explicitly by calling `NSHomeDirectory()` but often users will enter strings containing this shorthand and expect it to work. If you select Go to Folder in the Finder's Go menu, then enter "˜/Documents" then it will open a new window showing the Documents folder in your home directory.

The `NSString` class provides a convenient method for strings that may contain a tilde. If you send the string a `-stringByExpandingTildeInPath` message, then you will get back a new string containing the absolute path, without a tilde. Although less useful, it is also possible to go the other way by sending a full path a `-stringByAbbreviatingWithTildeInPath` message. If the path points to something inside the user's home directory, then it will be collapsed to only use a tilde character for this part of the path.

When interacting with the filesystem, you very often need to decompose a path into three parts: the file name, the file extension, and the path of the directory containing the file. You can do all of these from `NSString`, like this:

```
NSString *fullPath = @"/tmp/folder/file.extension";
// ext = @"extension";
NSString *ext = [fullPath pathExtension];
// file = @"file";
NSString *file = [[fullPath lastPathComponent]
    stringByDeletingPathExtension];
// dir = @"/tmp/folder";
NSString *dir = [fullPath stringByDeletingLastPathComponent];
```

There are also methods for constructing a path from individual parts, including appending components and setting the extension. Before you start writing code for parsing path strings yourself, make sure that `NSString` doesn't already have methods for doing what you need.

4.7.4 File Access

While `NSFileManager` lets you interact with the filesystem, and `NSWorkspace` lets you open files in other applications, neither provides a means of accessing the contents of files.

There is nothing stopping you from using the C library and POSIX functions for doing this, but they are not very convenient. Cocoa includes a wrapper around them in the form of the `NSFileHandle` class. This wraps a file handle as would be returned by `open()`. Four singleton instances exist, representing the C standard input, output and error streams, and a placeholder that discards all data written to it. Additional file handles can be created for reading, writing, or updating, with named constructors.

You can use the `NSFileHandle` class anywhere you have a C file handle using this initializer:

```
- (id)initWithFileDescriptor: (int)fileDescriptor
              closeOnDealloc: (BOOL)flag
```

The file descriptor is any C file descriptor, for example, the kind returned by `open()` or `socket()`. Whether the resulting object supports reading or writing depends on the underlying file descriptor. If you set `flag` to **`YES`**, then you can use this as a simple way of tracking how many parts of your program are using the file and ensuring that it is closed at the correct point. As long as you then only use the file descriptor with the object, it will stay open as long as some parts of your program are using it and close itself when it is no longer required.

If all you want to do is read data from a file, `NSData` objects can be created from a file using this method:

```
+ (id)dataWithContentsOfFile: (NSString*)path
                     options: (NSUInteger)mask
                       error: (NSError**)errorPtr
```

This creates a new `NSData` object from the contents of the file pointed to by `path` and sets the `errorPtr` to an `NSError` instance if it fails. The `mask` parameter allows two options to be set: `NSMappedRead` and `NSUncachedRead`. The first uses `mmap()` instead of file reading operations. As discussed earlier, this is a good idea if you know that the file is not going to be modified, for example, for read-only resources in an application bundle. If the system is low on memory, it can very cheaply free mapped data and load it back from the original file, while data read in will have to be written out to the swap file, even if they have not been modified. The second option allows the data to bypass the operating system's cache. If you know that you are just going to read the data once and then discard it, then it can improve performance. Otherwise it will use less RAM, but require more disk accesses.

`NSData` can also be used to write data back to a file. For small file output, using an `NSMutableData` to construct the file in memory and then using the `writeToFile:atomically:` or `writeToURL:atomically:` methods to output it is very simple. The second parameter to each of these is a **`BOOL`**, which, if set to **`YES`**, will write the data first to a temporary file and then rename the temporary file, ensuring on-disk consistency.

4.8 Notifications

Notifications are another example of *loose coupling* in Cocoa. They provide a layer of indirection between events and event handlers. Rather than every object keeping a list of objects that need to receive events, they go via the *notification center*, embodied by the `NSNotificationCenter` class. Objects can request notifications with a specific name, from a specific object, or both. When the specified object posts the named notification, the observer will receive a message with an `NSNotification` object as an argument.

This mechanism makes it very easy to join parts of your application together without explicitly hard-coding the connections. It is also a good example of the layering of Foundation and AppKit. Foundation defines notifications, but AppKit uses them heavily. A lot of view objects will post notifications when some user interaction has occurred, as well as delivering the message to their delegate. This allows multiple objects to be notified when some part of the user interface changes, with very little code.

By default, notifications are delivered synchronously. An object sends the notification to the notification center, and the notification center passes it on to all observers. Sometimes this is not the desired behavior. For asynchronous delivery, the `NSNotificationQueue` class integrates with the run loop and allows delivery of notifications to be deferred until the run loop is idle, or until the next iteration of the run loop.

4.8.1 Requesting Notifications

There are, generally speaking, two things you can do with notifications: send them and receive them. Receiving a notification is a two-step process. First, you tell the notification center what you want to receive, and then you wait.

Listing 4.18 shows a simple class for receiving notifications. This implements a single method, `receiveNotification:`, which takes a notification as an object. This is a very simple method that logs the name of the notification and the value associated with a key in the *user info dictionary*. The user info dictionary is what makes notifications so flexible. Any notification sender can attach an arbitrary set of key-value pairs to any notification. Because it is a dictionary, a receiver doesn't even need to know the specific keys; it has the option of enumerating every single key and doing something with all of the items in the dictionary, or it can simply ignore any keys it doesn't know about.

When the object is created, the default notification center is told to deliver notifications with the name "ExampleNotification" to its `receiveNotification:` method, from any object. It is also possible to specify a specific object to receive notifications from. In this case, you may leave the `name:` argument as `nil` and get every notification sent by that object.

Listing 4.18: Notification receiver object. [from: examples/Noti↓cations/notify.m]

```objc
@interface Receiver : NSObject {}
- (void) receiveNotification: (NSNotification*)aNotification;
@end

@implementation Receiver
- (id) init
{
    if (nil == (self = [super init]))
    {
        return nil;
    }
    // register to receive notifications
    NSNotificationCenter *center =
        [NSNotificationCenter defaultCenter];
    [center addObserver: self
               selector: @selector(receiveNotification:)
                   name: @"ExampleNotification"
                 object: nil];
    return self;
}
- (void) receiveNotification: (NSNotification*)aNotification
{
    printf("Received␣notification:␣%s\n",
            [[aNotification name] UTF8String]);
    printf("Received␣notification:␣%s\n",
        [[[aNotification userInfo] objectForKey: @"message"] UTF8String]);
}
- (void) dealloc
{
    NSNotificationCenter *center =
        [NSNotificationCenter defaultCenter];
    [center removeObserver: self];
    [super dealloc];
}
@end
```

There is one remaining part of handling notifications that is very important. You must remember to send the notification center a `removeObserver:` message when your object is destroyed. For this reason, it is good practice to ensure that the object that is registering as an observer in notifications is always **self**. This makes it very easy to make sure you call `removeObserver:` at the right time; just put the call in the `-dealloc` method. In a garbage-collected environment, the notification center should keep weak references to observers, so they will be automatically removed when no longer valid.

In this simple example, notifications are identified by literal strings. It is more common when creating public notifications to use shared global objects, initialized in one file and declared **extern** in a header.

4.8.2 Sending Notifications

The other half of the equation, sending messages, is even simpler. Listing 4.19 shows a simple object that sends a notification in response to a `sendMessage:` message. The string argument is inserted into the user info dictionary and delivered via a notification.

This is the companion of the `Receiver` class from the Listing 4.18. These two

Listing 4.19: Sending a notification. [from: examples/Notiﬁcations/notify.m]

```
@interface Sender : NSObject {}
- (void) sendMessage: (NSString*)aMessage;
@end
@implementation Sender
- (void) sendMessage: (NSString*)aMessage
{
    NSNotificationCenter *center =
        [NSNotificationCenter defaultCenter];

    NSDictionary *message =
        [NSDictionary dictionaryWithObject: aMessage
                                    forKey: @"message"];
    [center postNotificationName: @"ExampleNotification"
                          object: self
                        userInfo: message];
}
@end

int main(void)
{
    [NSAutoreleasePool new];
    // Set up the receiver
    Receiver *receiver = [Receiver new];
    // Send the notification
    [[Sender new] sendMessage: @"A short message"];
    return 0;
}
```

two classes communicate without either having a direct reference to the other. The `main()` method creates an instance of each class and calls the `sendMessage:` method on the sender. This posts a notification that is received by the receiver:

```
$ gcc -framework Foundation notify.m && ./a.out
Received notification: ExampleNotification
Message is: A short message
```

4.8.3 Sending Asynchronous Notification

Normally, sending a notification is a synchronous operation. You send a `-postNotification:` message to the notification center, it iterates over all of the objects that have registered to receive that notification, sends the notification to them, and then returns.

Every thread has its own notification center, and it also has a notification queue, an instance of `NSNotificationQueue`. This functions in a similar way to the notification center, but defers delivery of notifications until a future run-loop iteration.

Notification queues are particularly useful if you are generating a lot of the same sort of notification in quick succession. Often, the observers do not need to run immediately. Consider something like the spell checker in a text box. This could send a notification every time the user types a character. The spell checker could register for this notification, receive it, see if the typed word is valid, and update the display. This has two drawbacks. First, a word will be checked several times as it is typed, which is not always needed. Second, the spell checking will interrupt the typing.

A better design would flag the text box as needing spell checking as soon as the user typed something, but defer the actual checking until later. The notification queue does two things that help here. First, it performs *notification coalescing*, turning multiple identical notifications into a single one. This means that you can send a notification for every key press, but the spell checker will only receive one. The other useful feature is deferred delivery. When you post a notification via a queue, you can decide whether it should be delivered now, soon, or when the thread has nothing else to do. A typical program spends most of its time doing nothing. The notification queue allows you to defer handling of notifications until one of these times, for example, only running the spell checker when the user stops typing. In practice, no user types fast enough to keep a modern CPU busy, but this system applies equally to other data sources, such as the disk or network, which can provide data fast enough to keep a processor busy for a while.

A notification queue is a front end to a notification center. You insert notifications into the queue and it then sends them to the notification center, which sends them to the observers. This combination means that objects listening for a notification do not have to know whether the sender is using a notification queue or sending notifications synchronously.

There are two ways of getting a pointer to a notification queue. The most common way is to send a `+defaultQueue` message to the class. This will return the notification queue connected to the thread's default notification center. Notifications posted to this queue will be delivered in the thread that sent them.

Alternatively, you can explicitly create a queue for a given center. You can have different queues, as shown in Figure 4.1. Each object can send notifications to one or more notification queues, or to the notification center directly. The notification queues will coalesce the notifications and then pass them on to the notification center, which then distributes them to all of the registered observers.

Having multiple notification queues allows you to control how notifications are coalesced. You may want to have all notifications from every instance of a particular class to be coalesced, but to be delivered separately from notifications of the same kind delivered from other objects, you can create a new notification queue for the class. You must attach the queue to a notification center when you initialize it, by sending it an `-initWithNotificationCenter:` message.

You send a notification through a queue by sending it either this message or one of the simpler forms that omits one or more argument:

```
- (void)enqueueNotification: (NSNotification*)notification
               postingStyle: (NSPostingStyle)postingStyle
               coalesceMask: (NSUInteger)coalesceMask
                   forModes: (NSArray*)modes;
```

The first argument is a notification. You have to create this yourself; there are convenience methods for constructing them as there are on the notification center. You will typically do this by sending a `+notificationWithName:object:userInfo:` message to the notification class.

The second argument defines when the notification should be delivered. You have three options here. If you specify `NSPostNow`, then the behavior is similar sending the message directly to the notification center. The notification will be posted synchronously, but it will first be coalesced. You can use this to flush a set of notifications. If you have a set of operations that may all trigger a particular kind of notification, then you can have them all send their notifications into a queue and then send a notification with the posting style set to `NSPostNow` to ensure that exactly one notification will be sent.

The other options defer posting of the notification until a future run-loop iteration. If you specify `NSPostASAP`, then the notification will be posted as soon as possible. This may not be at the start of the next run-loop iteration, because there may be other notifications with the same priority already waiting for delivery, but it will be soon. If the notification is not very important, then you can set the posting style to `NSPostWhenIdle`. This will cause it to be delivered only when there are no more pressing events waiting for the run loop's attention.

Coalescing works best the longer notifications are allowed to stay in the queue. If everything is posted with `NSPostNow`, then the queue never has a chance to coalesce them. If everything is posted with `NSPostWhenIdle`, then notifications may stay in the queue for a long time, and will have a lot more opportunities for being combined.

The coalescing behavior is configured with the third argument. This is a mask formed by combining flags specifying whether notifications should be coalesced if they are from the same sender or of the same type. Most often you will specify either both of these flags, or neither. Coalescing notifications from the same sender

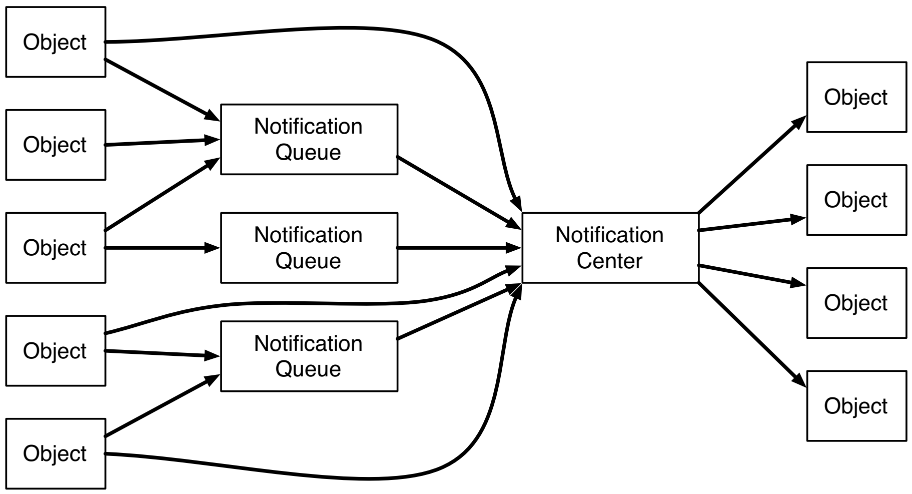

Figure 4.1: The flow of notifications in a Cocoa program.

but with different types may cause some very strange behavior. You can coalesce notifications of the same type from different senders, but this is generally not recommended either.

The final argument specifies the run-loop modes in which the notification will be delivered. You can use this to prevent notifications from being handled in certain modes, or to define new modes for handling certain kinds of notification and keep them queued until you explicitly tell the run loop to enter that mode. This provides a fairly simple way of deferring notification delivery until you explicitly decide you want to handle them.

Notification queues are very powerful, but rarely need to be used. They are most commonly treated as an optimization technique. If you profile your code and find that it is spending a lot of time handling duplicate notifications, then consider adding a notification queue.

4.8.4 Distributed Notifications

Notifications aren't solely subject to use in a single process. The `NSDistributedNotificationCenter` class is a subclass of `NSNotificationCenter` built using the *distributed objects* (*DO*) mechanism. This makes them the simplest form of *interprocess communication* (*IPC*) to use on OS X.

Although less flexible than using DO directly, distributed notifications provide

Notifications and Signals

If you come from a UNIX programming background, you may find notifications quite similar, conceptually, to UNIX signals. When combined with notification queues, they work in a similar way to signals sent using the POSIX realtime extensions to signal support. They are delivered asynchronously, can be enqueued, and carry something the size of a pointer as extra data with them.

a very simple way of communicating between different processes. In principle, distributed notifications could be integrated with *Bonjour* for sending notifications across the local network segment. The `-notificationCenterForType:` constructor hints at future possibilities for distributed notifications; however, it only supports `NSLocalNotificationCenterType` on OS X. The GNUstep implementation also supports `GSNetworkNotificationCenterType` for delivering notifications over the network using distributed objects, but there is currently no equivalent provided by Apple.

Registering to receive a distributed notification is almost the same as registering to receive a normal one. The main difference is the last parameter. Note the different method prototypes for registering observers in a notification center and a distributed notification center:

```
// NSNotificationCenter
- (void)addObserver: (id)notificationObserver
           selector: (SEL)notificationSelector
               name: (NSString*)notificationName
             object: (id)notificationSender;
// NSDistributedNotificationCenter
- (void)addObserver: (id)notificationObserver
           selector: (SEL)notificationSelector
               name: (NSString*)notificationName
             object: (NSString*)notificationSender;
```

The last argument is a pointer to the object from which you want to receive the notifications. In a distributed notification center, senders are identified by name, rather than pointer. The reason for this should be obvious: Distributed notifications can come from other processes, and pointers are only valid in the current process's address space.

Sending distributed notifications has the same change. The same method can be used, but the sender must be a string instead of an object. Typically, this is the name of the application sending the notification.

There are also some restrictions placed on the user info dictionary when sending distributed notifications. Because the notification is sent over DO, all of the objects in the dictionary must conform to the `NSCoding` protocol, allowing them to be serialized and deserialized in the remote process. Since the deserialization can potentially be done in a large number of listening processes, it is a good idea to keep notifications small.

4.9 Summary

In this chapter, we've gone through the most important aspects of the Foundation framework. This framework covers the core functionality of the Cocoa development environment and even provides a number of features that would typically be thought of as part of the language, such as reference counting and message forwarding.

We spent some time examining the core concepts of the Foundation library. In subsequent chapters, you will see examples of all of the things we've discussed here.

We looked at the collection classes provided by Foundation—sets, arrays, and dictionaries—and how enumeration of these types works. We saw the basic value types used to store non-object values in collections.

The most important aspects of the Foundation framework were covered in this chapter. This should not be taken as an exhaustive reference. The framework contains a large number of classes and functions. If you printed out just the class references from the Foundation framework, you would end up with something longer than this entire book, and many of the method descriptions would still be single-line comments.

In-depth understanding of all of the details of the Foundation library is almost impossible. The purpose of this chapter was to highlight the most important parts to look at. Familiarity with the classes discussed in this chapter goes a long way toward making a good Cocoa programmer.

Index