David Chisnall

ESSENTIAL CODE AND COMMANDS

Objective-C 2.0

PHRASEBOOK

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

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

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

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

ISBN-13: 978-0-321-74362-6
ISBN-10: 0-321-74362-8

Text printed in the United States on recycled paper at RR Donnelly in Crawfordsville, Indiana.

First printing February 2011

Editor-in-Chief
Mark Taub

Acquisitions Editor
Mark Taber

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Bart Reed

Proofreader
Charlotte Kughen

Publishing Coordinator
Vanessa Evans

Cover Designer
Gary Adair

Compositor
Gloria Schurick

Table of Contents

This page intentionally left blank

About the Author

David Chisnall is a freelance writer and consultant. While studying for his PhD, he co-founded the Étoilé project, which aims to produce an open-source desktop environment on top of GNUstep, an open-source implementation of the OpenStep and Cocoa APIs. He is an active contributor to GNUstep and is the original author and maintainer of the GNUstep Objective-C 2 runtime library and the associated compiler support in the Clang compiler.

After completing his PhD, David hid in academia for a while, studying the history of programming languages. He finally escaped when he realized that there were places off campus with an equally good view of the sea and without the requirement to complete quite so much paperwork. He occasionally returns to collaborate on projects involving modeling the semantics of dynamic languages.

David has a great deal of familiarity with Objective-C, having worked both on projects using the language and on implementing the language itself. He has also worked on implementing other languages, including dialects of Smalltalk and JavaScript, on top of an Objective-C runtime, allowing mixing code between all of these languages without bridging.

When not writing or programming, David enjoys dancing Argentine Tango and Cuban Salsa, playing badminton and ultimate frisbee, and cooking.

Acknowledgments

When writing a book about Objective-C, the first person I should thank is Nicolas Roard. I got my first Mac at around the same time I started my PhD and planned to use it to write Java code, not wanting to learn a proprietary language. When I started my PhD, I found myself working with Nicolas, who was an active GNUstep contributor. He convinced me that Objective-C and Cocoa were not just for Macs and that they were both worth learning. He was completely right: Objective-C is a wonderfully elegant language, and the accompanying frameworks make development incredibly easy.

The next person to thank is Fred Kiefer. Fred is the maintainer of the GNUstep implementation of the AppKit framework. He did an incredibly thorough (read: pedantic) technical review of this book, finding several places where things were not explained as well as they could have been. If you enjoy reading this book, then Fred deserves a lot of the credit.

Finally, I need to thank everyone else who was involved in bringing this book from my text editor to your hands, especially Mark Taber who originally proposed the idea to me.

We Want to Hear from You

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book——as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail:	mark.taber@pearson.com
Mail:	Mark Taber
	Associate Publisher
	Addison Wesley Publishing
	800 East 96th Street
	Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/aw for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

Blaise Pascal once wrote, "I didn't have time to write a short letter, so I wrote a long one instead." This phrasebook, at under 350 (small) pages, is the shortest book I've written, and trying to fit everything that I wanted to say into a volume this short was a challenge.

When Mark Taber originally suggested that I write an Objective-C Phrasebook, I was not sure what it would look like. A phrasebook for a natural language is a list of short idioms that can be used by people who find themselves in need of a quick sentence or two. A phrasebook for a programming language should fulfil a similar rôle.

This book is not a language reference. Apple provides a competent reference for the Objective-C language on the `http://developer.apple.com` site. This is not a detailed tutorial; unlike my other Objective-C book, *Cocoa Programming Developer's Handbook*, you won't find complete programs as code examples. Instead, you'll find very short examples of Objective-C idioms, which hopefully you can employ in a wide range of places.

One of the most frustrating things in life is finding that code examples in a book don't actually work. There are two sorts of code listings in this book. Code on a white background is intended to illustrate a simple point. This code may depend on some implied context and should not be taken as working, usable examples.

The majority of the code you will find in this book is on a gray background. At the bottom of each of these examples, you will find the name of the file that the listing was taken from. You can download these from the book's page on InformIT's website: `http://www.informit.com/title/0321743628`

A Note About Typesetting

This book was written in Vim, using semantic markup. From here, three different versions are generated. Two are created using pdflatex. If you are reading either the printed or PDF version, then you can see one of these. The only difference between the two is that the print version contains crop marks to allow the printer to trim the pages.

The third version is XHTML, intended for the ePub edition. This is created using the EtoileText framework, which first parses the LaTeX-style markup to a tree structure, then performs some transformations for handling cross-references and indexing, and finally generates XHTML. The code for doing this is all written in Objective-C.

If you have access to both, you may notice that the code listings look slightly nicer in the ePub edition. This is because EtoileText uses SourceCodeKit, another Étoilé framework, for syntax highlighting. This uses part of Clang, a modern Objective-C compiler, to mark up the code listings. This means that ranges of the code are annotated with exactly the same semantic types that the compiler sees. For example, it can distinguish between a function call and a macro instantiation.

You can find all of the code for doing this in the Étoilé subversion repository:

5

Numbers

One of the big differences between Objective-C and Smalltalk is that Objective-C inherits the full range of primitive (non-object) C types. These are, in ascending order of size, **`char`**, **`short`**, **`int`**, **`long`** and **`long long`** integers, with both **`signed`** and **`unsigned`** variants, as well as two floating-point types: **`float`** and **`double`**.

These all behave exactly as they do in C, complete with type promotion rules. You'll also find that Objective-C compilers support a **`long double`** type, which is architecture-dependent.

Note that this is very similar to Java, where you have a small selection of non-object types, but with some very important differences. In Java, the *intrinsic types* are defined to be a fixed size. In C, they are defined to have a minimum precision. For example, the specification says that an **`int`** has "the natural size suggested by the architecture of the execution environment," whereas in Java it is explicitly defined as a "32-bit signed two's complement integer."

As well as the primitive types, C supports defining new names for the existing types via the **typedef** keyword. The most common reason for this is that the specification does not require a particular size for any of the standard types, merely that each must be at least as big as the previous one. In particular, there are platforms currently deployed where **int** is 16, 32, and 64 bits, so you can't rely on any specific size for these.

OS X supports ILP32 and LP64 modes. This shorthand is used to describe which of the C types have which sizes. ILP32 means that **int**s, **long**s, and pointers are 32 bits. LP64 means that **long**s and pointers are 64-bit quantities, and that, implicitly, other values are smaller. Microsoft Windows, in contrast, is an LLP64 platform on 64-bit architectures; both **int** and **long** remain 32 bits and only pointers and **long long**s are 64 bits. This causes a problem if you assumed that you could safely cast a pointer to **long**—something that works on almost every platform in the world, including Win32, but does not work on Win64.

The problem of casting a pointer to an integer is a serious one. The **long long** type is at least 64 bits, so on any current platform it is guaranteed to be big enough to store any pointer, but on any 32- or 16-bit platform it can be much too big. C99 introduced the `intptr_t` typedef, which is exactly the size of a pointer. Apple introduced an equivalent: `NSInteger`. This is used throughout the Cocoa frameworks and is always the same size as a pointer. There is also an unsigned version, `NSUInteger`.

In GUI code, you will often come across `CGFloat` or `NSFloat`. These are equivalent to each other. Both are the size of a pointer, making them **`float`**s on 32-bit platforms and **`double`**s on 64-bit ones.

Storing Numbers in Collections

```
NSMutableArray *array = [NSMutableArray array];
[array addObject: [NSNumber numberWithInt: 12]];
```

From: numberInArray.m

All of the standard Objective-C collection classes let you store objects, but often you want to store primitive types in them as well. The solution to this is *boxing*—wrapping a primitive type up in an object.

The `NSValue` class hierarchy is used for this. `NSValue` is a class designed to wrap a single primitive value. This class is quite generic, and is an example of a *class cluster*. When you create an instance of an `NSValue`, you will get back some subclass, specialized for storing different kinds of data. If you store a pointer in an `NSValue`, you don't want the instance to take up as much space as one containing an `NSRect`—a C structure containing four `NSFloat`s.

One concrete subclass of `NSValue` is particularly important: `NSNumber`. This class is intended to wrap single numerical values and can be initialized from any of the C standard integer types.

The designated constructor for both of these classes is `+valueWithBytes:objCType`. The first argument is a pointer to some value and the second is the *Objective-C type encoding* of the type. Type encodings are strings representing a particular type. They are used a lot for introspection in Objective-C; you can find out the types of any method or instance variable in a class as a type encoding string and then parse this to get the relevant compile-time types.

You can get the type encoding of any type with the **`@encode()`** directive. This is analogous to **`sizeof()`** in C, but instead of returning the size as an integer it returns the type encoding as a C string. One very convenient trick when working with type encodings is to use the `typeof()` GCC extension. This returns the type of an expression. You can combine it with **`@encode()`**, like this:

```
NSValue *value =
  [NSValue valueWithBytes: &aPrimitive
       objCType: @encode(typeof(aPrimitive))];
```

This snippet will return an `NSValue` wrapping `aPrimitive`, and will work regardless of the type of the primitive. You could wrap this in a macro, but be careful not to pass it an expression with side effects if you do.

Note that you have to pass a pointer to the primitive value. This method will use the type encoding to find out how big the primitive type is and will then copy it.

More often, you will use one of the other constructors. For example, if you want to create an `NSNumber` instance from an integer, you would do so like this:

```
NSNumber *twelve = [NSNumber numberWithInt:
    12];
```

The resulting object can then be added to a collection. Unlike `NSValue`, `NSNumber` instances are ordered, so you can sort collections containing `NSNumber` instances.

```
NSArray *a = [NSArray arrayWithObjects:
  [NSNumber numberWithUnsignedLongLong:
      ULLONG_MAX],
  [NSNumber numberWithInt: -2],
  [NSNumber numberWithFloat: 300.057],
  [NSNumber numberWithInt: 1],
  [NSNumber numberWithDouble: 200.0123],
  [NSNumber numberWithLongLong: LLONG_MIN],
  nil];
NSArray *sorted =
  [a sortedArrayUsingSelector: @selector(compare
      :)];
NSLog(@"%@", sorted);
```

From: numberArray.m

The numberArray.m example stores a group of `NSNumber` instances in an array and then sorts them using the `-compare:` selector. As you can see from the output, the ordering is enforced irrespective of how the number was created.

```
2010-03-15 14:50:48.166 a.out[51465:903] (
    "-9223372036854775808",
    "-2",
    1,
    "200.0123",
    "300.057",
    18446744073709551615
)
```

Output from: numberArray.m

Performing Decimal Arithmetic

```
NSDecimalNumber *one =
  [NSDecimalNumber one];
NSDecimalNumber *fortyTwo =
  [NSDecimalNumber decimalNumberWithString: @"42"];
NSDecimalNumber *sum =
  [one decimalNumberByAdding: fortyTwo];
NSDecimal accumulator = [sum decimalValue];
NSDecimal temp = [fortyTwo decimalValue];
NSDecimalMultiply(&accumulator, &accumulator, &temp, NSRoundPlain);
temp = [one decimalValue];
NSDecimalAdd(&accumulator, &accumulator, &temp, NSRoundPlain);
NSDecimalNumber *result =
  [NSDecimalNumber decimalNumberWithDecimal: accumulator];
```

From: decimal.m

C gives you two options for working with numbers: integers and floating-point values. Floating-point values are made of two components: a mantissa and an exponent. Their value is two to the power of the exponent, multiplied by the mantissa.

The problem with floating-point values is that they are binary. This means that their precision is defined in terms of binary digits, which is not always what you want. For a financial application, for example, you may need to store amounts to exactly four decimal places. This is not possible with floating-point values; a value such as 0.1 cannot be represented by any finite binary floating-point, just as 0.1 in base three (one third) cannot be represented by any finite decimal sequence.

A binary number is the sum of a set of powers of two, just as a decimal number is a sum of powers of ten. With fractional values, the digits after the radix point indicate halves, quarters, eighths, and so on. If you try to create a value of 0.1 by adding powers of two, you never succeed, although you get progressively closer. Exactly the same thing happens when you try to create a third by adding powers of ten (a three tenths, plus three hundredths, plus three thousands, and so on).

One solution is to use fixed-point arithmetic. Rather than storing dollars, you might store hundredths of a cent. You must then remember to normalize your values, and you are limited by the range of an integer type. Objective-C provides another option: decimal floating-point types.

The `NSDecimal` type is a C structure that represents a decimal value. Somewhat strangely, there is no C API for creating these. You must create an `NSDecimalNumber` instance and then send it a `-decimalValue` message.

You then have two choices for arithmetic. `NSDecimalNumber` instances are immutable. You can create new ones as a result of arithmetic—for example, by sending a `decimalNumberByAdding:` message to one.

Alternatively, you can use the C API, which modifies the value of the structure directly.

If you are just performing one arithmetic operation and then storing the result in an object, the first option is simpler. If you are doing a number of steps then it is faster to use the C APIs. Because these modify the structure, they do not require you to create a new object for each intermediate step.

Note: The C1X specification includes decimal number types, and some compilers support these as an extension. The `NSDecimal` type is not compatible with these. On most platforms this is not important. If you are targeting something like IBM's POWER6, which has hardware for decimal arithmetic, then it is better to use the decimal types directly.

Neither of these is especially fast. The decimal number is represented as an array of digits, and these are operated on in pairs, after the two numbers have been normalized. You can expect to get similar performance to a software floating-point implementation—possibly slightly worse as `NSDecimal` is not widely used and therefore has not been the focus of much optimization effort.

`NSDecimalNumber` is a subclass of `NSNumber`, so all of the ways of converting `NSNumber`s to strings that we'll look at in the next section work as expected. You can also convert them to

C primitive types using the standard methods for accessing these on number objects, but these methods may truncate or approximate the decimal value.

Converting Between Strings and Numbers

```
int answer = [@"42" intValue];
NSString *answerString =
  [NSString stringWithFormat: @"%d", answer];
NSNumber *boxedAnswer =
  [NSNumber numberWithInt: answer];
NSCAssert([answerString isEqualToString:
    [boxedAnswer stringValue]],
    @"Both strings should be the same");
```

From: strtonum.m

There are several ways of converting between a number and a string. A lot of objects that represent simple data have methods like `-intValue`, for returning an integer representation of the receiver.

`NSString` has several methods in this family. If you have a string that contains a numerical value, you can send it a `-doubleValue`, `-floatValue`, `-intValue`, or `-longLongValue` message to convert it to any of these types. In 64-bit safe versions of Foundation, you can also send it an `-integerValue` message. This will return an `NSInteger`.

There are a few ways of going in the

opposite direction, getting a string from an integer. We look at one in Chapter 6: The `+stringWithFormat:` method on `NSString` lets you construct a string from any primitive C types, just as you would construct a C string with `sprintf()`.

If you already have a number in an `NSNumber` instance, there are two ways of getting a string, one of which is a wrapper around the other. The `-descriptionWithLocale:` method returns a string generated by formatting the number according to the specified locale.

In fact, this doesn't do the translation itself. It sends an `-initWithFormat:locale:` message to a new `NSString`. The format string depends on the type of the number: for example, a double will be converted using the `@"%0.16g"` format string. This uses up to 16 significant figures and an exponent if required.

The decimal separator depends on the locale. If you send an `NSNumber` a `-stringValue` message, this is the equivalent to sending a `-descriptionWithLocale:` message with `nil` as the argument. This uses the *canonical locale*, which means without any localization, so the result will be the same on any platform.

Reading Numbers from Strings

```
NSScanner *parser =
  [NSScanner scannerWithString: @"1 plus 2"];

int operands[2];
NSString *operation;

[parser setCharactersToBeSkipped:
  [NSCharacterSet whitespaceCharacterSet]];

[parser scanInt: operands];
[parser scanCharactersFromSet:
  [NSCharacterSet letterCharacterSet]
          intoString: &operation];
[parser scanInt: operands+1];
```

From: scanner.m

Two of the first things any C programmer learns to use are the `printf()` and `scanf()` functions. These are very, very similar—one is almost an inverse of the other—and they let you construct formatted strings and parse data from them. We've already seen that `NSString` has a rough analogue of `sprintf()`, so you can construct strings from format strings and variables, but what is the Objective-C equivalent of `sscanf()`? How, given a string, do we parse values from it?

The answer lies in the `NSScanner` class. This class is a very powerful tokenizer class. You create an instance of `NSScanner` attached to a string and then scan values from it, one at a time.

The messages you send to a scanner all have the same form. They take a pointer to a variable and return a **BOOL**, indicating whether they succeeded. The scanner stores the current scanning index in the string, and only increments it on a successful scan, so you can try parsing the next characters in different ways. You can also implement read-ahead and backtracking quite easily with `NSScanner`. If you send it a `-scanLocation` message, it returns the current index in the string. You can then try scanning a few things, get to an error, and backtrack by sending it a `-setScanLocation:` message, resetting the old index.

One of the most powerful methods in `NSScanner` is `-scanCharactersFromSet:intoString:`. This reads a string from the current scanning point until it encounters a character not present in the specified set. As we will see in Chapter 6, you can construct `NSCharacterSet` instances with any arbitrary set of characters, or you can use one of the standard ones.

The example at the start of this section reads a number, then a word, then another number from a string. The number is read using the built in `-scanInt:` method, but the word is a bit more complex. It uses an `NSCharacterSet`, in this case the set of all letters.

This isn't the only `NSCharacterSet` used in this example. This scanner is also configured to skip whitespace. The `setCharactersToBeSkipped:` message sent to the scanner tells it to ignore any characters in the set passed as the argument. Passing the whitespace character set tells it to skip any whitespace that occurs between calls. If there are characters in this set at the position where the scanner starts reading when you send it a scan message, it will skip past them. It will not skip these characters while parsing a token, so putting "1 2" in the string would be read as two separate numbers, not as 12.

Index

A

B

C

D

E

F

G

I

K

L

M

O