



Stephen G. Kochan

Updated
for Xcode 4.5
and iOS 6

Programming in Objective-C

Fifth Edition

Developer's Library



FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Programming in Objective-C

Fifth Edition

Developer's Library

ESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALS

Developer's Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.

All books in the *Developer's Library* are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.

Key titles include some of the best, most widely acclaimed books within their topic areas:

PHP & MySQL Web Development

Luke Welling & Laura Thomson

ISBN 978-0-672-32916-6

Python Essential Reference

David Beazley

ISBN-13: 978-0-672-32978-4

MySQL

Paul DuBois

ISBN-13: 978-0-672-32938-8

PostgreSQL

Korry Douglas

ISBN-13: 978-0-672-32756-8

Linux Kernel Development

Robert Love

ISBN-13: 978-0-672-32946-3

C++ Primer Plus

Stephen Prata

ISBN-13: 978-0321-77640-2

Developer's Library books are available in print and in electronic formats at most retail and online bookstores, as well as by subscription from Safari Books Online at **safari.informit.com**

**Developer's
Library**

informit.com/devlibrary

Programming in Objective-C

Fifth Edition

Stephen G. Kochan

◆ Addison-Wesley

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

Programming in Objective-C, Fifth Edition

Copyright © 2013 by 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: 978-0-321-88728-3

ISBN-10: 0-321-88728-X

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

Printed in the United States of America

First Printing: December 2012

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

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

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Acquisitions Editor
Mark Taber

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Copy Editor
Keith Cline

Indexer
Erika Millen

Proofreader
Dan Knott

Technical Editor
Michael Trent

Publishing
Coordinator
Vanessa Evans

Designer
Gary Adair

Cover Designer
Chuti Prasertsith

Compositor
Tricia Bronkella



To Roy and Ve, two people whom I dearly miss.

To Ken Brown, "It's just a jump to the left."



Contents at a Glance

1 Introduction 1

I: The Objective-C Language

2 Programming in Objective-C 7

3 Classes, Objects, and Methods 27

4 Data Types and Expressions 51

5 Program Looping 71

6 Making Decisions 93

7 More on Classes 127

8 Inheritance 153

9 Polymorphism, Dynamic Typing, and Dynamic Binding 179

10 More on Variables and Data Types 197

11 Categories and Protocols 223

12 The Preprocessor 237

13 Underlying C Language Features 251

II: The Foundation Framework

14 Introduction to the Foundation Framework 307

15 Numbers, Strings, and Collections 311

16 Working with Files 377

17 Memory Management and Automatic Reference Counting 407

18 Copying Objects 419

19 Archiving 431

III: Cocoa, Cocoa Touch, and the iOS SDK

20 Introduction to Cocoa and Cocoa Touch 449

21 Writing iOS Applications 453

Appendixes

- A Glossary 487
- B Address Book Example Source Code 495
- Index 501

Table of Contents

1 Introduction 1

What You Will Learn from This Book 2

How This Book Is Organized 3

Support 5

Acknowledgments 5

Preface to the Fifth Edition 6

I: The Objective-C Language

2 Programming in Objective-C 7

Compiling and Running Programs 7

Using Xcode 8

Using Terminal 16

Explanation of Your First Program 18

Displaying the Values of Variables 22

Summary 25

Exercises 25

3 Classes, Objects, and Methods 27

What Is an Object, Anyway? 27

Instances and Methods 28

An Objective-C Class for Working with Fractions 30

The `@interface` Section 33

Choosing Names 34

Class and Instance Methods 35

The `@implementation` Section 37

The `program` Section 39

Accessing Instance Variables and Data Encapsulation 45

Summary 49

Exercises 49

4 Data Types and Expressions 51

Data Types and Constants 51

Type `int` 51

Type `float` 52

Type <code>char</code>	52
Qualifiers: <code>long</code> , <code>long long</code> , <code>short</code> , <code>unsigned</code> , and <code>signed</code>	53
Type <code>id</code>	54
Arithmetic Expressions	55
Operator Precedence	55
Integer Arithmetic and the Unary Minus Operator	58
The Modulus Operator	60
Integer and Floating-Point Conversions	61
The Type Cast Operator	63
Assignment Operators	64
A Calculator Class	65
Exercises	67
5 Program Looping	71
The <code>for</code> Statement	72
Keyboard Input	79
Nested <code>for</code> Loops	81
<code>for</code> Loop Variants	83
The <code>while</code> Statement	84
The <code>do</code> Statement	89
The <code>break</code> Statement	91
The <code>continue</code> Statement	91
Summary	91
Exercises	92
6 Making Decisions	93
The <code>if</code> Statement	93
The <code>if-else</code> Construct	98
Compound Relational Tests	101
Nested <code>if</code> Statements	104
The <code>else if</code> Construct	105
The <code>switch</code> Statement	115
Boolean Variables	118
The Conditional Operator	123
Exercises	125

7 More on Classes 127

- Separate Interface and Implementation Files 127
- Synthesized Accessor Methods 133
- Accessing Properties Using the Dot Operator 135
- Multiple Arguments to Methods 137
 - Methods without Argument Names 139
 - Operations on Fractions 139
- Local Variables 143
 - Method Arguments 144
 - The `static` Keyword 144
- The `self` Keyword 148
- Allocating and Returning Objects from Methods 149
 - Extending Class Definitions and the Interface File 151
- Exercises 151

8 Inheritance 153

- It All Begins at the Root 153
 - Finding the Right Method 157
- Extension through Inheritance: Adding New Methods 158
 - A Point Class and Object Allocation 162
 - The `@class` Directive 163
 - Classes Owning Their Objects 167
- Overriding Methods 171
 - Which Method Is Selected? 173
- Abstract Classes 176
- Exercises 176

9 Polymorphism, Dynamic Typing, and Dynamic Binding 179

- Polymorphism: Same Name, Different Class 179
- Dynamic Binding and the `id` Type 182
- Compile Time Versus Runtime Checking 184
- The `id` Data Type and Static Typing 185
 - Argument and Return Types with Dynamic Typing 186
- Asking Questions about Classes 187
- Exception Handling Using `@try` 192
- Exercises 195

10	More on Variables and Data Types	197
	Initializing Objects	197
	Scope Revisited	200
	Directives for Controlling Instance Variable Scope	200
	More on Properties, Synthesized Accessors, and Instance Variables	202
	Global Variables	203
	Static Variables	205
	Enumerated Data Types	207
	The typedef Statement	211
	Data Type Conversions	212
	Conversion Rules	212
	Bit Operators	214
	The Bitwise AND Operator	215
	The Bitwise Inclusive-OR Operator	216
	The Bitwise Exclusive-OR Operator	217
	The Ones Complement Operator	217
	The Left-Shift Operator	219
	The Right-Shift Operator	219
	Exercises	220
11	Categories and Protocols	223
	Categories	223
	Class Extensions	228
	Some Notes about Categories	229
	Protocols and Delegation	230
	Delegation	233
	Informal Protocols	233
	Composite Objects	234
	Exercises	235
12	The Preprocessor	237
	The #define Statement	237
	More Advanced Types of Definitions	239
	The #import Statement	244

- Conditional Compilation 245
 - The `#ifdef`, `#endif`, `#else`, and `#ifndef` Statements 245
 - The `#if` and `#elif` Preprocessor Statements 247
 - The `#undef` Statement 248
- Exercises 249

13 Underlying C Language Features 251

- Arrays 252
 - Initializing Array Elements 254
 - Character Arrays 255
 - Multidimensional Arrays 256
- Functions 258
 - Arguments and Local Variables 259
 - Returning Function Results 261
 - Functions, Methods, and Arrays 265
- Blocks 266
- Structures 270
 - Initializing Structures 273
 - Structures within Structures 274
 - Additional Details about Structures 276
 - Don't Forget about Object-Oriented Programming! 277
- Pointers 277
 - Pointers and Structures 281
 - Pointers, Methods, and Functions 283
 - Pointers and Arrays 284
 - Operations on Pointers 294
 - Pointers and Memory Addresses 296
- They're Not Objects! 297
- Miscellaneous Language Features 297
 - Compound Literals 297
 - The `goto` Statement 298
 - The Null Statement 298
 - The Comma Operator 299
 - The `sizeof` Operator 299
 - Command-Line Arguments 300

How Things Work	302
Fact 1: Instance Variables Are Stored in Structures	303
Fact 2: An Object Variable Is Really a Pointer	303
Fact 3: Methods Are Functions, and Message Expressions Are Function Calls	304
Fact 4: The <code>id</code> Type Is a Generic Pointer Type	304
Exercises	304

II: The Foundation Framework

14 Introduction to the Foundation Framework	307
Foundation Documentation	307
15 Numbers, Strings, and Collections	311
Number Objects	311
String Objects	317
More on the <code>NSLog</code> Function	317
The <code>description</code> Method	318
Mutable Versus Immutable Objects	319
Mutable Strings	326
Array Objects	333
Making an Address Book	338
Sorting Arrays	355
Dictionary Objects	361
Enumerating a Dictionary	364
Set Objects	367
<code>NSIndexSet</code>	371
Exercises	373
16 Working with Files	377
Managing Files and Directories: <code>NSFileManager</code>	378
Working with the <code>NSData</code> Class	383
Working with Directories	384
Enumerating the Contents of a Directory	387
Working with Paths: <code>NSPathUtilities.h</code>	389
Common Methods for Working with Paths	392
Copying Files and Using the <code>NSProcessInfo</code> Class	394

Basic File Operations: <code>NSFileHandle</code>	398
The <code>NSURL</code> Class	403
The <code>NSBundle</code> Class	404
Exercises	405
17 Memory Management and Automatic Reference Counting	407
Automatic Garbage Collection	409
Manual Reference Counting	409
Object References and the Autorelease Pool	410
The Event Loop and Memory Allocation	412
Summary of Manual Memory Management Rules	414
Automatic Reference Counting	415
Strong Variables	415
Weak Variables	416
@autoreleasepool Blocks	417
Method Names and Non-ARC Compiled Code	418
18 Copying Objects	419
The <code>copy</code> and <code>mutableCopy</code> Methods	419
Shallow Versus Deep Copying	422
Implementing the <code><NSCopying></code> Protocol	424
Copying Objects in Setter and Getter Methods	427
Exercises	429
19 Archiving	431
Archiving with XML Property Lists	431
Archiving with <code>NSKeyedArchiver</code>	434
Writing Encoding and Decoding Methods	435
Using <code>NSData</code> to Create Custom Archives	442
Using the Archiver to Copy Objects	446
Exercises	447

III: Cocoa, Cocoa Touch, and the iOS SDK**20 Introduction to Cocoa and Cocoa Touch 449**

Framework Layers 449

Cocoa Touch 450

21 Writing iOS Applications 453

The iOS SDK 453

Your First iPhone Application 453

Creating a New iPhone Application Project 456

Entering Your Code 460

Designing the Interface 462

An iPhone Fraction Calculator 469

Starting the New Fraction_Calculator Project 471

Defining the View Controller 471

 The `Fraction` Class 477

A Calculator Class That Deals with Fractions 480

Designing the User Interface 482

Summary 482

Exercises 484

Appendixes**A Glossary 487****B Address Book Example Source Code 495**

Index 501

About the Author

Stephen Kochan is the author and coauthor of several bestselling titles on the C language, including *Programming in C* (Sams, 2004), *Programming in ANSI C* (Sams, 1994), and *Topics in C Programming* (Wiley, 1991), and several UNIX titles, including *Exploring the Unix System* (Sams, 1992) and *Unix Shell Programming* (Sams, 2003). He has been programming on Macintosh computers since the introduction of the first Mac in 1984, and he wrote *Programming C for the Mac* as part of the Apple Press Library. In 2003, Kochan wrote *Programming in Objective-C* (Sams, 2003), and followed that with another Mac-related title, *Beginning AppleScript* (Wiley, 2004).

About the Technical Reviewers

Michael Trent has been programming in Objective-C since 1997—and programming Macs since well before that. He is a regular contributor to Steven Frank’s www.cocoadev.com website, a technical reviewer for numerous books and magazine articles, and an occasional dabbler in Mac OS X open-source projects. Currently, he is using Objective-C and Apple Computer’s Cocoa frameworks to build professional video applications for Mac OS X. Michael holds a Bachelor of Science degree in computer science and a Bachelor of Arts degree in music from Beloit College of Beloit, Wisconsin. He lives in Santa Clara, California, with his lovely wife, Angela.

Wendy Mui is a programmer and software development manager in the San Francisco Bay Area. After learning Objective-C from the second edition of Steve Kochan’s book, she landed a job at Bump Technologies, where she put her programming skills to good use working on the client app and the API/SDK for Bump’s third-party developers. Prior to her iOS experience, Wendy spent her formative years at Sun and various other tech companies in Silicon Valley and San Francisco. She got hooked on programming while earning a Bachelor of Arts degree in mathematics from the University of California Berkeley.

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 directly to let us 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 we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we 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.

Email: feedback@developers-library.info

Mail: Reader Feedback
Addison-Wesley Developer's Library
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

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

This page intentionally left blank

Classes, Objects, and Methods

In this chapter, you learn about some key concepts in object-oriented programming and start working with classes in Objective-C. You need to learn a little bit of terminology, but we keep it fairly informal. We also cover only some of the basic terms here because you can easily get overwhelmed. Refer to Appendix A, “Glossary,” at the end of this book for more precise definitions of these terms.

What Is an Object, Anyway?

An object is a thing. Think about object-oriented programming as a thing and something you want to do to that thing. This is in contrast to a programming language such as C, known as a procedural programming language. In C, you typically think about what you want to do first and then you worry about the objects, almost the opposite of object orientation.

Consider an example from everyday life. Let’s assume that you own a car, which is obviously an object, and one that you own. You don’t have just any car; you have a particular car that was manufactured in a factory, maybe in Detroit, maybe in Japan, or maybe someplace else. Your car has a vehicle identification number (VIN) that uniquely identifies that car here in the United States.

In object-oriented parlance, your particular car is an *instance* of a car. Continuing with the terminology, `car` is the name of the *class* from which this instance was created. So each time a new car is manufactured, a new instance from the class of cars is created, and each instance of the car is referred to as an *object*.

Your car might be silver, have a black interior, be a convertible or hardtop, and so on. In addition, you perform certain actions with your car. For example, you drive your car, fill it with gas, (hopefully) wash it, take it in for service, and so on. Table 3.1 depicts this.

Table 3.1 Actions on Objects

Object	What You Do with It
Your car	Drive it
	Fill it with gas
	Wash it
	Service it

The actions listed in Table 3.1 can be done with your car, and they can be done with other cars as well. For example, your sister drives her car, washes it, fills it with gas, and so on.

Instances and Methods

A unique occurrence of a class is an *instance*, and the actions that are performed on the instance are called *methods*. In some cases, a method can be applied to an instance of the class or to the class itself. For example, washing your car applies to an instance. (In fact, all the methods listed in Table 3.1 can be considered instance methods.) Finding out how many types of cars a manufacturer makes would apply to the class, so it would be a class method.

Suppose you have two cars that came off the assembly line and are seemingly identical: They both have the same interior, same paint color, and so on. They might start out the same, but as each car is used by its respective owner, its unique characteristics or *properties* change. For example, one car might end up with a scratch on it, and the other might have more miles on it. Each instance or object contains not only information about its initial characteristics acquired from the factory but also its current characteristics. Those characteristics can change dynamically. As you drive your car, the gas tank becomes depleted, the car gets dirtier, and the tires get a little more worn.

Applying a method to an object can affect the *state* of that object. If your method is to “fill up my car with gas,” after that method is performed, your car’s gas tank will be full. The method then will have affected the state of the car’s gas tank.

The key concepts here are that objects are unique representations from a class, and each object contains some information (data) that is typically private to that object. The methods provide the means of accessing and changing that data.

The Objective-C programming language has the following particular syntax for applying methods to classes and instances:

```
[ ClassOrInstance method ] ;
```

In this syntax, a left bracket is followed by the name of a class or instance of that class, which is followed by one or more spaces, which is followed by the method you want to perform. Finally, it is closed off with a right bracket and a terminating semicolon. When you ask a class

or an instance to perform some action, you say that you are sending it a *message*; the recipient of that message is called the *receiver*. So another way to look at the general format described previously is as follows:

```
[ receiver message ];
```

Let's go back to the previous list and write everything in this new syntax. Before you do that, though, you need to get your new car. Go to the factory for that, like so:

```
yourCar = [Car new];    get a new car
```

You send a new message to the `Car` class (the receiver of the message) asking it to give you a new car. The resulting object (which represents your unique car) is then stored in the variable `yourCar`. From now on, `yourCar` can be used to refer to your instance of the car, which you got from the factory.

Because you went to the factory to get the car, the method `new` is called a *factory* or *class* method. The rest of the actions on your new car will be instance methods because they apply to your car. Here are some sample message expressions you might write for your car:

```
[yourCar prep];        get it ready for first-time use
[yourCar drive];       drive your car
[yourCar wash];        wash your car
[yourCar getGas];      put gas in your car if you need it
[yourCar service];     service your car

[yourCar topDown];     if it's a convertible
[yourCar topUp];

currentMileage = [yourCar odometer];
```

This last example shows an instance method that returns information—presumably, the current mileage, as indicated on the odometer. Here, we store that information inside a variable in our program called `currentMileage`.

Here's an example of where a method takes an *argument* that specifies a particular value that may differ from one method call to the next:

```
[yourCar setSpeed: 55]; set the speed to 55 mph
```

Your sister, Sue, can use the same methods for her own instance of a car:

```
[suesCar drive];
[suesCar wash];
[suesCar getGas];
```

Applying the same methods to different objects is one of the key concepts of object-oriented programming, and you'll learn more about it later.

You probably won't need to work with cars in your programs. Your objects will likely be computer-oriented things, such as windows, rectangles, pieces of text, or maybe even a calculator or a playlist of songs. And just like the methods used for your cars, your methods might look similar, as in the following:

<code>[myWindow erase];</code>	Clear the window
<code>theArea = [myRect area];</code>	Calculate the area of the rectangle
<code>[userText spellCheck];</code>	Spell-check some text
<code>[deskCalculator clearEntry];</code>	Clear the last entry
<code>[favoritePlaylist showSongs];</code>	Show the songs in a playlist of favorites
<code>[phoneNumber dial];</code>	Dial a phone number
<code>[myTable reloadData];</code>	Show the updated table's data
<code>n = [aTouch tapCount];</code>	Store the number of times the display was tapped

An Objective-C Class for Working with Fractions

Now it's time to define an actual class in Objective-C and learn how to work with instances of the class.

Once again, you'll learn procedure first. As a result, the actual program examples might not seem very practical. We get into more practical stuff later.

Suppose that you need to write a program to work with fractions. Maybe you need to deal with adding, subtracting, multiplying, and so on. If you didn't know about classes, you might start with a simple program that looked like this.

Program 3.1

```
// Simple program to work with fractions

#import <Foundation/Foundation.h>

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int numerator = 1;
        int denominator = 3;
        NSLog(@"The fraction is %i/%i", numerator, denominator);
    }
    return 0;
}
```

Program 3.1 Output

The fraction is 1/3

In Program 3.1, the fraction is represented in terms of its numerator and denominator. After the `@autoreleasepool` directive, the two lines in `main` both declare the variables `numerator` and `denominator` as integers and assign them initial values of 1 and 3, respectively. This is equivalent to the following lines:

```
int numerator, denominator;

numerator = 1;
denominator = 3;
```

We represented the fraction $1/3$ by storing 1 in the variable `numerator` and 3 in the variable `denominator`. If you needed to store a lot of fractions in your program, this could be cumbersome. Each time you wanted to refer to the fraction, you'd have to refer to the corresponding numerator and denominator. And performing operations on these fractions would be just as awkward.

It would be better if you could define a fraction as a single entity and collectively refer to its numerator and denominator with a single name, such as `myFraction`. You can do that in Objective-C, and it starts by defining a new class.

Program 3.2 duplicates the functionality of Program 3.1 using a new class called `Fraction`. Here, then, is the program, followed by a detailed explanation of how it works.

Program 3.2

```
// Program to work with fractions - class version

#import <Foundation/Foundation.h>

//---- @interface section ----

@interface Fraction: NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end

//---- @implementation section ----

@implementation Fraction
{
    int numerator;
```



```

    int denominator;
}
-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end

//---- program section ----

int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *myFraction;

        // Create an instance of a Fraction

        myFraction = [Fraction alloc];
        myFraction = [myFraction init];

        // Set fraction to 1/3

        [myFraction setNumerator: 1];
        [myFraction setDenominator: 3];

        // Display the fraction using the print method

        NSLog(@"The value of myFraction is:");
        [myFraction print];
    }
    return 0;
}

```

Program 3.2 Output

```
The value of myFraction is:
1/3
```

As you can see from the comments in Program 3.2, the program is logically divided into three sections:

- @interface section
- @implementation section
- program section

The @interface section describes the class and its methods, and the @implementation section describes the data (the *instance variables* that objects from the class will store) and contains the actual code that implements the methods declared in the interface section. The program section contains the program code to carry out the intended purpose of the program.

Note

You can also declare the instance variables for a class in the interface section. The ability to do it in the implementation section was added as of Xcode 4.2 and is considered a better way to define a class. You learn more about why in a later chapter.

Each of these sections is a part of every Objective-C program, even though you might not need to write each section yourself. As you'll see, each section is typically put in its own file. For now, however, we keep it all together in a single file.

The @interface Section

When you define a new class, you have to tell the Objective-C compiler where the class came from. That is, you have to name its *parent* class. Next, you need to define the type of operations, or *methods*, that can be used when working with objects from this class. And, as you learn in a later chapter, you also list items known as *properties* in this special section of the program called the @interface section. The general format of this section looks like this:

```
@interface NewClassName: ParentClassName
    propertyAndMethodDeclarations;
@end
```

By convention, class names begin with an uppercase letter, even though it's not required. This enables someone reading your program to distinguish class names from other types of variables by simply looking at the first character of the name. Let's take a short diversion to talk a little about forming names in Objective-C.

Choosing Names

In Chapter 2, “Programming in Objective-C,” you used several variables to store integer (`int`) values. For example, you used the variable `sum` in Program 2.4 to store the result of the addition of the two integers 50 and 25.

The Objective-C language allows you to store data types other than just integers in variables as well, as long as the proper declaration for the variable is made before it is used in the program. Variables can be used to store floating-point numbers, characters, and even objects (or, more precisely, references to objects).

The rules for forming names are quite simple: They must begin with a letter or underscore (`_`), and they can be followed by any combination of letters (uppercase or lowercase), underscores, or the digits 0 through 9. The following is a list of valid names:

- `sum`
- `pieceFlag`
- `i`
- `myLocation`
- `numberOfMoves`
- `sysFlag`
- `ChessBoard`

However, the following names are not valid for the stated reasons:

- `sum$value` `$`—Is not a valid character.
- `piece flag`—Embedded spaces are not permitted.
- `3Spencer`—Names cannot start with a number.
- `int`—This is a reserved word.

`int` cannot be used as a variable name because its use has a special meaning to the Objective-C compiler. This use is known as a *reserved name* or *reserved word*. In general, any name that has special significance to the Objective-C compiler cannot be used as a variable name.

Always remember that uppercase and lowercase letters are distinct in Objective-C. Therefore, the variable names `sum`, `Sum`, and `SUM` each refer to a different variable. As noted, when naming a class, start it with a capital letter. Instance variables, objects, and method names, however, typically begin with lowercase letters. To aid readability, capital letters are used inside names to indicate the start of a new word, as in the following examples:

- `AddressBook`—This could be a class name.
- `currentEntry`—This could be an object.

- `addNewEntry`—This could be a method name.

When deciding on a name, keep one recommendation in mind: Don't be lazy. Pick names that reflect the intended use of the variable or object. The reasons are obvious. Just as with the comment statement, meaningful names can dramatically increase the readability of a program and will pay off in the debug and documentation phases. In fact, the documentation task will probably be much easier because the program will be more self-explanatory.

Here, again, is the @interface section from Program 3.2:

```
//---- @interface section ----

@interface Fraction: NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end
```

The name of the new class is `Fraction`, and its parent class is `NSObject`. (We talk in greater detail about parent classes in Chapter 8, "Inheritance.") The `NSObject` class is defined in the file `NSObject.h`, which is automatically included in your program whenever you import `Foundation.h`.

Class and Instance Methods

You have to define methods to work with your `Fractions`. You need to be able to set the value of a fraction to a particular value. Because you won't have direct access to the internal representation of a fraction (in other words, direct access to its instance variables), you must write methods to set the numerator and denominator. You'll also write a method called `print` that will display the value of a fraction. Here's what the declaration for the `print` method looks like in the interface file:

```
-(void) print;
```

The leading minus sign (-) tells the Objective-C compiler that the method is an instance method. The only other option is a plus sign (+), which indicates a class method. A class method is one that performs some operation on the class itself, such as creating a new instance of the class.

An instance method performs some operation on a particular instance of a class, such as setting its value, retrieving its value, displaying its value, and so on. Referring to the car example, after you have manufactured the car, you might need to fill it with gas. The operation of filling it with gas is performed on a particular car, so it is analogous to an instance method.

Return Values

When you declare a new method, you have to tell the Objective-C compiler whether the method returns a value and, if it does, what type of value it returns. You do this by enclosing the return type in parentheses after the leading minus or plus sign. So this declaration specifies that the instance method called `currentAge` returns an integer value:

```
-(int) currentAge;
```

Similarly, this line declares a method that returns a double precision value. (You learn more about this data type in Chapter 4, “Data Types and Expressions.”)

```
-(double) retrieveDoubleValue;
```

A value is returned from a method using the Objective-C `return` statement, similar to the way in which we returned a value from `main` in previous program examples.

If the method returns no value, you indicate that using the type `void`, as in the following:

```
-(void) print;
```

This declares an instance method called `print` that returns no value. In such a case, you do not need to execute a `return` statement at the end of your method. Alternatively, you can execute a `return` without any specified value, as in the following:

```
return;
```

Method Arguments

Two other methods are declared in the `@interface` section from Program 3.2:

```
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
```

These are both instance methods that return no value. Each method takes an integer argument, which is indicated by the `(int)` in front of the argument name. In the case of `setNumerator`, the name of the argument is `n`. This name is arbitrary and is the name the method uses to refer to the argument. Therefore, the declaration of `setNumerator` specifies that one integer argument, called `n`, will be passed to the method and that no value will be returned. This is similar for `setDenominator`, except that the name of its argument is `d`.

Notice the syntax of the declaration for these methods. Each method name ends with a colon, which tells the Objective-C compiler that the method expects to see an argument. Next, the type of the argument is specified, enclosed in a set of parentheses, in much the same way the return type is specified for the method itself. Finally, the symbolic name to be used to identify that argument in the method is specified. The entire declaration is terminated with a semicolon. Figure 3.1 depicts this syntax.

The members declared in this section are known as the *instance variables*. Each time you create a new object, a new and unique set of instance variables also is created. Therefore, if you have two `Fractions`, one called `fracA` and another called `fracB`, each will have its own set of instance variables—that is, `fracA` and `fracB` each will have its own separate numerator and denominator. The Objective-C system automatically keeps track of this for you, which is one of the nicer things about working with objects.

The *methodDefinitions* part of the `@implementation` section contains the code for each method specified in the `@interface` section. Similar to the `@interface` section, each method's definition starts by identifying the type of method (class or instance), its return type, and its arguments and their types. However, instead of the line ending with a semicolon, the code for the method follows, enclosed inside a set of curly braces. It's noted here that you can have the compiler automatically generate methods for you by using a special `@synthesize` directive. Chapter 7 covers this in detail.

Consider the `@implementation` section from Program 3.2:

```
//---- @implementation section ----
@implementation Fraction
{
    int numerator;
    int denominator;
}

-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end
```

The `print` method uses `NSLog` to display the values of the instance variables `numerator` and `denominator`. But to which `numerator` and `denominator` does this method refer? It refers to the instance variables contained in the object that is the receiver of the message. That's an important concept, and we return to it shortly.

The `setNumerator:` method stores the integer argument you called `n` in the instance variable `numerator`. Similarly, `setDenominator:` stores the value of its argument `d` in the instance variable `denominator`.

The program Section

The `program` section contains the code to solve your particular problem, which can be spread out across many files, if necessary. Somewhere you must have a routine called `main`, as previously noted. That's where your program always begins execution. Here's the program section from Program 3.2:

```
//---- program section ----

int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *myFraction;

        // Create an instance of a Fraction and initialize it

        myFraction = [Fraction alloc];
        myFraction = [myFraction init];

        // Set fraction to 1/3

        [myFraction setNumerator: 1];
        [myFraction setDenominator: 3];

        // Display the fraction using the print method

        NSLog(@"The value of myFraction is:");
        [myFraction print];
    }

    return 0;
}
```

Inside `main`, you define a variable called `myFraction` with the following line:

```
Fraction *myFraction;
```

This line says that `myFraction` is an object of type `Fraction`; that is, `myFraction` is used to store values from your new `Fraction` class. The asterisk that precedes the variable name is described in more detail later.

Now that you have an object to store a `Fraction`, you need to create one, just as you ask the factory to build you a new car. This is done with the following line:

```
myFraction = [Fraction alloc];
```

`alloc` is short for *allocate*. You want to allocate memory storage space for a new fraction. This expression sends a message to your newly created `Fraction` class:

```
[Fraction alloc]
```

You are asking the `Fraction` class to apply the `alloc` method, but you never defined an `alloc` method, so where did it come from? The method was inherited from a parent class. Chapter 8 deals with this topic in detail.

When you send the `alloc` message to a class, you get back a new instance of that class. In Program 3.2, the returned value is stored inside your variable `myFraction`. The `alloc` method is guaranteed to zero out all of an object's instance variables. However, that doesn't mean that the object has been properly initialized for use. You need to initialize an object after you allocate it.

This is done with the next statement in Program 3.2, which reads as follows:

```
myFraction = [myFraction init];
```

Again, you are using a method here that you didn't write yourself. The `init` method initializes the instance of a class. Note that you are sending the `init` message to `myFraction`. That is, you want to initialize a specific `Fraction` object here, so you don't send it to the class; you send it to an instance of the class. Make sure that you understand this point before continuing.

The `init` method also returns a value—namely, the initialized object. You store the return value in your `Fraction` variable `myFraction`.

The two-line sequence of allocating a new instance of class and then initializing it is done so often in Objective-C that the two messages are typically combined, as follows:

```
myFraction = [[Fraction alloc] init];
```

This inner message expression is evaluated first:

```
[Fraction alloc]
```

As you know, the result of this message expression is the actual `Fraction` that is allocated. Instead of storing the result of the allocation in a variable, as you did before, you directly apply the `init` method to it. So, again, first you allocate a new `Fraction` and then you initialize it. The result of the initialization is then assigned to the `myFraction` variable.

As a final shorthand technique, the allocation and initialization is often incorporated directly into the declaration line, as in the following:

```
Fraction *myFraction = [[Fraction alloc] init];
```

Returning to Program 3.2, you are now ready to set the value of your fraction. These program lines do just that:

```
// Set fraction to 1/3

[myFraction setNumerator: 1];
[myFraction setDenominator: 3];
```

The first message statement sends the `setNumerator:` message to `myFraction`. The argument that is supplied is the value 1. Control is then sent to the `setNumerator:` method you defined for your `Fraction` class. The Objective-C system knows that it is the method from this class to use because it knows that `myFraction` is an object from the `Fraction` class.

Inside the `setNumerator:` method, the passed value of 1 is stored inside the variable `n`. The single program line in that method stores that value in the instance variable `numerator`. So, you have effectively set the numerator of `myFraction` to 1.

The message that invokes the `setDenominator:` method on `myFraction` follows next. The argument of 3 is assigned to the variable `d` inside the `setDenominator:` method. This value is then stored inside the `denominator` instance variable, thus completing the assignment of the value 1/3 to `myFraction`. Now you're ready to display the value of your fraction, which you do with the following lines of code from Program 3.2:

```
// Display the fraction using the print method

NSLog(@"The value of myFraction is:");
[myFraction print];
```

The `NSLog` call simply displays the following text:

```
The value of myFraction is:
```

The following message expression invokes the `print` method:

```
[myFraction print];
```

Inside the `print` method, the values of the instance variables `numerator` and `denominator` are displayed, separated by a slash character.

Note

In the past, iOS programmers were responsible for telling the system when they were done using an object that they allocated by sending the object a `release` message. That was done in accordance with a memory management system known as *manual reference counting*. As of Xcode 4.2, programmers no longer have to worry about this and can rely on the system to take care of releasing memory as necessary. This is done through a mechanism known as *Automatic Reference Counting*, or ARC for short. ARC is enabled by default when you compile new applications using Xcode 4.2 or later.

It seems as if you had to write a lot more code to duplicate in Program 3.2 what you did in Program 3.1. That’s true for this simple example here; however, the ultimate goal in working with objects is to make your programs easier to write, maintain, and extend. You’ll realize that later.

Let’s go back for a second to the declaration of `myFraction`

```
Fraction *myFraction;
```

and the subsequent setting of its values.

The asterisk (*) in front of `myFraction` in its declaration says that `myFraction` is actually a reference (or *pointer*) to a `Fraction` object. The variable `myFraction` doesn’t actually store the fraction’s data (that is, its numerator and denominator values). Instead, it stores a reference—which is actually a memory address—indicating where the object’s data is located in memory. When you first declare `myFraction` as shown, its value is undefined as it has not been set to any value and does not have a default value. We can conceptualize `myFraction` as a box that holds a value. Initially the box contains some undefined value, as it hasn’t been assigned any value, as shown in Figure 3.2.

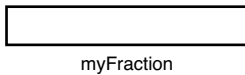


Figure 3.2 Declaring `Fraction *myFraction;`

When you allocate a new object (using `alloc`, for example) enough space is reserved in memory to store the object’s data, which includes space for its instance variables, plus a little more. The location of where that data is stored (the reference to the data) is returned by the `alloc` routine, and assigned to the variable `myFraction`. This all takes place when this statement is executed in Program 3.2:

```
myFraction = [Fraction alloc];
```

The allocation of the object and the storage of the reference to that object in `myFraction` is depicted in Figure 3.3.

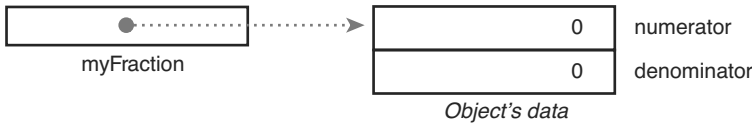


Figure 3.3 Relationship between `myFraction` and its data

Note

More data is stored with the object than just that indicated, but you don't need to worry about that here. You'll note that the instance variables are shown as being set to 0. That's currently being handled by the `alloc` method. However, the object still has not been properly initialized. You still need to use the `init` method on the newly allocated object.

Notice the directed line in Figure 3.3. This indicates the connection that has been made between the variable `myFraction` and the allocated object. (The value stored inside `myFraction` is actually a memory address. It's at that memory address that the object's data is stored.)

Subsequently in Program 3.2, the fraction's numerator and denominator are set. Figure 3.4 depicts the fully initialized `Fraction` object with its numerator set to 1 and its denominator set to 3.

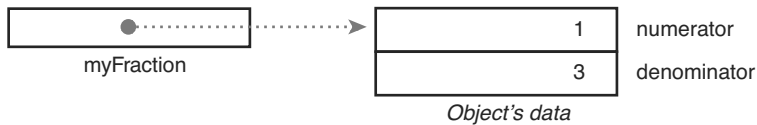


Figure 3.4 Setting the fraction's numerator and denominator

The next example shows how you can work with more than one fraction in your program. In Program 3.3, you set one fraction to $2/3$, set another to $3/7$, and display them both.

Program 3.3

```
// Program to work with fractions - cont'd

#import <Foundation/Foundation.h>

//---- @interface section ----

@interface Fraction: NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end

//---- @implementation section ----

@implementation Fraction
{
    int numerator;
```

```

    int denominator;
}

-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end

//---- program section ----

int main (int argc, char * argv[])
{
    @autoreleasepool {

        Fraction *frac1 = [[Fraction alloc] init];
        Fraction *frac2 = [[Fraction alloc] init];

        // Set 1st fraction to 2/3

        [frac1 setNumerator: 2];
        [frac1 setDenominator: 3];

        // Set 2nd fraction to 3/7

        [frac2 setNumerator: 3];
        [frac2 setDenominator: 7];

        // Display the fractions

        NSLog(@"First fraction is:");

        [frac1 print];

        NSLog(@"Second fraction is:");
        [frac2 print];
    }
}

```

```

    }
    return 0;
}

```

Program 3.3 Output

```

First fraction is:
2/3
Second fraction is:
3/7

```

The `@interface` and `@implementation` sections remain unchanged from Program 3.2. The program creates two `Fraction` objects, called `frac1` and `frac2`, and then assigns the value $2/3$ to the first fraction and $3/7$ to the second. Realize that when the `setNumerator:` method is applied to `frac1` to set its numerator to 2, the instance variable `frac1` gets its instance variable `numerator` set to 2. Also, when `frac2` uses the same method to set its numerator to 3, its distinct instance variable `numerator` is set to the value 3. Each time you create a new object, it gets its own distinct set of instance variables. Figure 3.5 depicts this.

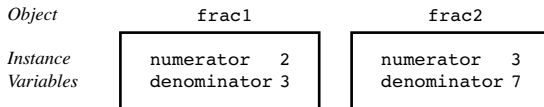


Figure 3.5 Unique instance variables

Based on which object is getting sent the message, the correct instance variables are referenced. Therefore, here `frac1`'s `numerator` is referenced whenever `setNumerator:` uses the name `numerator` inside the method:

```
[frac1 setNumerator: 2];
```

That's because `frac1` is the receiver of the message.

Accessing Instance Variables and Data Encapsulation

You've seen how the methods that deal with fractions can access the two instance variables `numerator` and `denominator` directly by name. In fact, an instance method can always directly access its instance variables. A class method can't, however, because it's dealing only with the class itself, not with any instances of the class. (Think about that for a second.) But what if you wanted to access your instance variables from someplace else (for example, from inside your `main` routine)? You can't do that directly because they are hidden. The fact that they are hidden from you is a key concept called *data encapsulation*. It enables someone writing class definitions to extend and modify the class definitions, without worrying about whether programmers (that

is, users of the class) are tinkering with the internal details of the class. Data encapsulation provides a nice layer of insulation between the programmer and the class developer.

You can access your instance variables in a clean way by writing special methods to set and retrieve their values. We wrote `setNumerator:` and `setDenominator:` methods to set the values of the two instance variables in our `Fraction` class. To retrieve the values of those instance variables, you need to write two new methods. For example, create two new methods called, appropriately enough, `numerator` and `denominator` to access the corresponding instance variables of the `Fraction` that is the receiver of the message. The result is the corresponding integer value, which you return. Here are the declarations for your two new methods:

```
-(int) numerator;
-(int) denominator;
```

And here are the definitions:

```
-(int) numerator
{
    return numerator;
}

-(int) denominator
{
    return denominator;
}
```

Note that the names of the methods and the instance variables they access are the same. There's no problem doing this (although it might seem a little odd at first); in fact, it is common practice. Program 3.4 tests your two new methods.

Program 3.4

```
// Program to access instance variables - cont'd

#import <Foundation/Foundation.h>

//---- @interface section ----

@interface Fraction: NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;

@end
```

```

//---- @implementation section ----

@implementation Fraction
{
    int numerator;
    int denominator;
}

-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

-(int) numerator
{
    return numerator;
}

-(int) denominator
{
    return denominator;
}

@end

//---- program section ----

int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *myFraction = [[Fraction alloc] init];

        // Set fraction to 1/3

        [myFraction setNumerator: 1];
        [myFraction setDenominator: 3];
    }
}

```



```

    // Display the fraction using our two new methods

    NSLog(@"The value of myFraction is: %i/%i",
          [myFraction numerator], [myFraction denominator]);
}

return 0;
}

```

Program 3.4 Output

```
The value of myFraction is 1/3
```

This `NSLog` statement displays the results of sending two messages to `myFraction`: the first to retrieve the value of its `numerator`, and the second the value of its `denominator`:

```

NSLog(@"The value of myFraction is: %i/%i",
      [myFraction numerator], [myFraction denominator]);

```

So, in the first message call, the `numerator` message is sent to the `Fraction` object `myFraction`. In that method, the code returns the value of the `numerator` instance variable for that fraction. Remember, the context of a method while it is executing is the object that is the receiver of the message. So, when the `numerator` method accesses and returns the value of the `numerator` instance variable, it's `myFraction`'s `numerator` that is accessed and returned. That returned integer value is then passed along to `NSLog` to be displayed.

For the second message call, the `denominator` method is called to access and return the value of `myFraction`'s `denominator`, which is then passed to `NSLog` to be displayed.

Incidentally, methods that set the values of instance variables are often collectively referred to as *setters*, and methods used to retrieve the values of instance variables are called *getters*. For the `Fraction` class, `setNumerator:` and `setDenominator:` are the setters, and `numerator` and `denominator` are the getters. Collectively, setters and getters are also referred to as *accessor* methods.

Make sure that you understand the difference between setters and the getters. The setters don't return a value, because their purpose is to take an argument and to set the corresponding instance variable to the value of that argument. No value needs to be returned in that case. That's the purpose of a setter: to set the value of an instance variable, so setters typically do not return values. In contrast, the purpose of the getter is to "get" the value of an instance variable stored in an object and to send it back to the program. To do that, the getter must return the value of the instance variable using the `return` statement.

Again, the idea that you can't directly set or get the value of an instance variable from outside of the methods written for the class, but instead have to write setter and getter methods to do so is the principle of data encapsulation. So, you have to use methods to access this data that is normally hidden to the "outside world." This provides a centralized path to the instance

variables and prevents some other code from indirectly changing these values, which would make your programs harder to follow, debug, and modify.

We should also point out that a method called `new` combines the actions of an `alloc` and `init`. So, this line could be used to allocate and initialize a new `Fraction`:

```
Fraction *myFraction = [Fraction new];
```

It's generally better to use the two-step allocation and initialization approach so that you conceptually understand that two distinct events are occurring: You're first creating a new object and then you're initializing it.

Summary

Now you know how to define your own class, create objects or instances of that class, and send messages to those objects. We return to the `Fraction` class in later chapters. You'll learn how to pass multiple arguments to your methods, how to divide your class definitions into separate files, and also how to use key concepts such as inheritance and dynamic binding. However, now it's time to learn more about data types and writing expressions in Objective-C. First, try the exercises that follow to test your understanding of the important points covered in this chapter.

Exercises

1. Which of the following are invalid names? Why?

<code>Int</code>	<code>playNextSong</code>	<code>6_05</code>
<code>_calloc</code>	<code>Xx</code>	<code>alphaBetaRoutine</code>
<code>clearScreen</code>	<code>_1312</code>	<code>z</code>
<code>ReInitialize</code>	<code>_</code>	<code>A\$</code>

2. Based on the example of the car in this chapter, think of an object you use every day. Identify a class for that object and write five actions you do with that object.
3. Given the list in exercise 2, use the following syntax to rewrite your list in this format:


```
[instance method];
```
4. Imagine that you own a boat and a motorcycle in addition to a car. List the actions you perform with each of these. Do you have any overlap between these actions?

5. Based on question 4, imagine that you have a class called `Vehicle` and an object called `myVehicle` that could be either `Car`, `Motorcycle`, or `Boat`. Suppose that you write the following:

```
[myVehicle prep];  
[myVehicle getGas];  
[myVehicle service];
```

Do you see any advantages of being able to apply an action to an object that could be from one of several classes?

6. In a procedural language such as C, you think about actions and then write code to perform the action on various objects. Referring to the car example, you might write a procedure in C to wash a vehicle and then inside that procedure write code to handle washing a car, washing a boat, washing a motorcycle, and so on. If you took that approach and then wanted to add a new vehicle type (see the previous exercise), do you see advantages or disadvantages to using this procedural approach over an object-oriented approach?
7. Define a class called `XYPoint` that will hold a Cartesian coordinate (x, y) , where x and y are integers. Define methods to individually set the x and y coordinates of a point and retrieve their values. Write an Objective-C program to implement your new class and test it.

Index

Symbols

+ (addition) operator, 54-58
& (address) operator, 278
+= (assignment) operator, 64
= (assignment) operator, 64-65, 74
-= (assignment) operator, 64
* (asterisk), 42
@ (at symbol), 20, 317
& (bitwise AND) operator, 215-216
| (bitwise OR) operator, 216-217
^ (bitwise XOR) operator, 217
^ (caret), 267
: (colon), 123
, (comma) operator, 299
/* */ comment syntax, 19
// comment syntax, 19
{ } (curly braces), 20
- (decrement) operator, 78, 291-294
/ (division) operator, 54-58
\$ (dollar sign), 16
. (dot) operator, 135-136
" (double quotes), 132
== (equal to) operator, 74
> (greater than) operator, 74
>= (greater than or equal to) operator, 74
++ (increment) operator, 78, 291-294
* (indirection) operator, 278
<< (left-shift) operator, 219
< (less than) operator, 74

- <= (less than or equal to) operator, 74
- && (logical AND) operator, 101
- ! (logical negation) operator, 121
- || (logical OR) operator, 101
- (minus sign), 35
- % (modulus) operator, 60-61
- * (multiplication) operator, 54-58
- != (not equal to) operator, 74
- ~ (ones complement) operator, 217-219
- # (pound sign), 237
- ? (question mark), 123
- >> (right-shift) operator, 219-220
- ;(semicolon), 84
- (subtraction) operator, 54
- ~ (tilde), 378
- (unary minus) operator, 58-60
- _ (underscore), 34, 202

A

- absolute value, calculating, 94
- abstract classes, 176, 487
- accessing
 - instance variables, 45-49
 - properties with dot operator, 135-136
- accessor methods
 - definition of, 487
 - explained, 48-49
 - synthesized accessors, 133-135, 202-203, 493
- add: method, 139-143, 149-151, 411
- addition (+) operator, 54-58
- addObject: method, 358, 370
- address (&) operator, 278
- address book program, 2
 - AddressBook class
 - custom archives, 442-445
 - defining, 344-347

- encoding/decoding methods, 438-441
- fast enumeration, 347-349
- @implementation section, 345-346
- @implentation section, 497-500
- @interface section, 345, 496
- lookup: method, 349-351
- removeCard: method, 351-355
- sortedArrayUsingComparator: method, 357
- sortUsingComparator: method, 357-358
- sortUsingSelector: method, 355-358
- AddressCard class
 - defining, 338-341
 - @implementation section, 339-342
 - @implentation section, 496-497
 - @interface section, 338-339, 495
 - synthesized methods, 341-344
- entries
 - looking up, 349-351
 - removing, 351-355
 - sorting, 355-358
- fast enumeration, 347-349
- overview, 338
- source code, 495-500

- AddressBook class**
 - custom archives, 442-445
 - defining, 344-347
 - encoding/decoding methods, 438-441
 - fast enumeration, 347-349
 - @implementation section, 345-346, 497-500
 - @interface section, 345, 496
 - lookup: method, 349-351
 - removeCard: method, 351-355

sortedArrayUsingComparator: method, 357

sortUsingComparator: method, 357-358

sortUsingSelector: method, 355-358

AddressCard class

defining, 338-341

@implementation section, 339-340, 342, 496-497

@interface section, 338-339, 495

synthesized methods, 341-344

addresses

memory addresses, 296-297

URL addresses, reading files from, 403-404

algorithms, greatest common divisor (gcd), 86-87

allKeys method, 365

alloc method, 40

allocation

instances, 40

memory, 135-137

objects, 149-151, 162-163

allocF method, 205-207

allocWithZone: method, 425

alternative names, assigning to data types, 211-212

AND operators

& (bitwise AND), 215-216

&& (logical AND), 101

anyObject method, 370

AppDelegate class, 460

appending files, 402-403

appendString: method, 333

AppKit, 307, 487

application bundles, 404-405

Application Kit, 307, 487

Application Services layer, 450

application templates, 457

ARC (Automatic Reference Counting), 41

@autoreleasepool blocks, 417-418

definition of, 488

explained, 415

with non-ARC compiled code, 418

strong variables, 415-416

weak variables, 416-417

archiveRootObject: method, 434

archiving

copying objects with, 446-447

definition of, 431, 487

encoding/decoding methods, 435-442

with NSData, 442-445

with NSKeyedArchiver, 434-435

with XML property lists, 431-433

arguments

argument types, 263-265

command-line arguments, 300-302

function arguments, 259-261

method arguments

declaring, 36-37

local variables, 144

methods without argument names, 139

multiple arguments, 137-143

arguments method, 396

arithmetic operators

binary arithmetic operators, 54-58

integer and floating-point conversions, 61-63

integer arithmetic, 58-60

modulus (%) operator, 60-61

precedence, 54-58

type cast operator, 63-64

unary minus (-) operator, 58-60

array method, 358

arrays

- array objects
 - address book example. *See* address book program
 - defining, 331-337
 - NSNumber class, 360-361
- character arrays, 255-256
- declaring, 252-254
- definition of, 487
- initializing, 254-255
- limitations, 297
- multidimensional arrays, 256-258
- NSArray class, 311
- passing to methods/functions, 265-266
- pointers to, 284-294
 - increment and decrement operators, 291-294
 - pointers to character strings, 289-291
 - valuesPtr example, 284-288
- arrayWithCapacity: method, 358
- arrayWithContentsOfFile: method, 407, 433
- arrayWithObjects: method, 334, 358
- assignment operators, 64-65, 74
- asterisk (*), 42, 54-58
- at symbol (@), 20, 317
- AT&T Bell Laboratories, 1
- attributesOfItemAtPath: method, 378
- automatic garbage collection, 409
- automatic local variables, 261
- Automatic Reference Counting (ARC). *See* ARC (Automatic Reference Counting)
- automatic variables, 488
- autorelease message, 410
- autorelease pool, 20, 410-412, 488
- @autoreleasepool, 20, 410, 417-418
- availableData method, 398

B

- backslash (\), 22
- base 8 (octal) notation, 54
- base 16 (hexadecimal) notation, 54
- binary arithmetic operators, 54-58
- binding, dynamic, 182-184, 489
- bit operators
 - binary, decimal, and hexadecimal equivalents, 214
 - bitwise AND (&), 215-216
 - bitwise OR (|), 216-217
 - bitwise XOR (^), 217
 - left-shift (<<) operator, 219
 - ones complement (~) operator, 217-219
 - right-shift (>>) operator, 219-220
 - table of, 214
- bitfield, 488
- bitwise AND (&) operator, 215-216
- bitwise OR (|) operator, 216-217
- bitwise XOR (^) operator, 217
- blocks. *See also* statements
 - @autoreleasepool blocks, 417-418
 - definition of, 488, 492
 - explained, 266-270
- BOOL data type, 122-123
- Boolean variables, 118-123
- braces ({}), 20
- break statement, 91
- buffers, reading files to/from, 383-384
- bundles (application), 404-405
- buttons, adding, 466-468

C

- C programming language, 1
- calculate: method, 144

- calculateTriangularNumber method, 259-261**
- calculator. See fraction calculator**
- Calculator class, 65-67, 480**
 - @implementation section, 481-482
 - @interface section, 481
- capitalizedString method, 332**
- caret (^), 217, 267**
- case sensitivity, 19, 34**
- caseInsensitiveCompare: method, 322, 332**
- @catch blocks, 192-194**
- categories**
 - best practices, 229
 - class extensions, 228-229
 - defining, 223-228
 - definition of, 488
 - explained, 223-232
 - MathOps, 223-228
- CGPoint data type, 274**
- CGRect data type, 274**
- CGSize data type, 274**
- changeCurrentDirectoryPath: method, 385**
- char characters, 317**
- char data type, 52-53**
- character arrays, 255-256**
- character string objects. See string objects**
- characterAtIndex: method, 332**
- child classes, 153-155**
- clang compiler, 17-18**
- @class directive, 163-167**
- class extensions, 228-229**
- class methods, 29, 35, 488**
- class objects. See objects**
- classes**
 - abstract classes, 176, 487
 - adding to projects, 127-130
- AddressBook**
 - custom archives, 442-445
 - defining, 344-347
 - encoding/decoding methods, 438-441
 - fast enumeration, 347-349
 - @implementation section, 345-346, 497-500
 - @interface section, 345, 496
 - lookup: method, 349-351
 - removeCard: method, 351-355
 - sortedArrayUsingComparator: method, 357
 - sortUsingComparator: method, 357-358
 - sortUsingSelector: method, 355-358
- AddressCard**
 - defining, 338-341
 - @implementation section, 339-340, 342, 496-497
 - @interface section, 338-339, 495
 - synthesized methods, 341-344
- AppDelegate, 460**
- Calculator, 65-67, 480-482**
 - @implementation section, 481-482
 - @interface section, 481
- categories**
 - best practices, 229
 - class extensions, 228-229
 - defining, 223-228
 - definition of, 488
 - explained, 223-232
- child classes, 153-155**
- class extensions, 228-229**
- Complex, 179-182**
- composite classes, 488**

- concrete subclasses, 488
- defining
 - Fraction example, 30-33
 - @implementation section, 37, 127-133
 - @interface section, 33-37, 127-133
 - program section, 39-45
- definition of, 488
- dynamic binding, 182-184
- extending through inheritance
 - @class directive, 163-167
 - classes owning their objects, 167-171
 - explained, 158-162
 - object allocation, 162-163
- Fraction, 30-33, 477-480
 - add: method, 139-143, 149-151, 411
 - adding to projects, 127-130
 - allocF method, 205-207
 - convertToNum method, 95-98
 - count method, 205-207
 - data encapsulation, 45-49
 - @implementation section, 37, 131-132, 141-142, 146-147, 478-480
 - initWith:over: method, 197-200
 - instance variables, accessing, 45-49
 - @interface section, 33-37, 130-131, 141, 146, 477
 - program section, 39-45
 - setTo:over: method, 137-139
- inheritance, 153-157, 490
- instances
 - allocation, 40
 - definition of, 490
 - explained, 28-30
 - initialization, 40
- local variables
 - explained, 143-144
 - method arguments, 144
 - static variables, 144-148
- methods. *See also* specific methods
 - accessor methods, 48-49, 133-135
 - arguments, 36-37, 137-143, 144
 - class methods versus instance methods, 29, 35
 - declaring, 35-37
 - explained, 28-30
 - methods without argument names, 139
 - overriding, 171-175
 - return values, 36
 - self keyword, 148-149
 - syntax, 28-29
- MusicCollection, 374-375
- naming conventions, 34-35
- NSArray, 311
 - archiving, 431-433
 - defining, 331-337
 - methods, 358
- NSBundle, 404-405
- NSCountedSet, 370
- NSData, 383-384, 431-433, 442-445
- NSDate, 431-433
- NSDictionary
 - archiving, 431-433
 - defining, 361-363
 - enumerating, 364-365
 - methods, 365
- NSFileHandle, 377, 398-403
- NSFileManager, 377
 - directory enumeration, 387-389
 - directory management, 384-387
 - file management, 378-383

- NSIndexSet, 371-372
- NSKeyedArchiver, 434-435
- NSMutableArray
 - defining, 331-337
 - methods, 358
- NSMutableDictionary
 - defining, 361-363
 - enumerating, 364-365
 - methods, 365
- NSMutableString, 326-330, 333-331
- NSNumber, 311-317, 431-433
- NSProcessInfo, 394-398
- NSSet, 367-370
- NSString, 317
 - archiving, 431-433
 - description method, 318-319
 - mutable versus immutable objects, 319-326
 - NSLog function, 317-318
- NSURL, 403-404
- NSNumber, 360-361
- objects
 - allocation, 149-151
 - returning from methods, 149-151
- parent classes, 153-155, 491
- Playlist, 374-375
- polymorphism, 179-182, 491
- properties, accessing with dot operator, 135-136
- Rectangle, 158-171
- returning information about, 187-192
- root classes, 153
- Song, 374-375
- Square, 160-162, 234-235
- subclasses, 492
- superclasses, 492
- ViewController
 - first iPhone application, 460-462
 - fraction calculator, 471-477
 - CGPoint, 162-165
- classroomM.com/objective-c, 5**
- clickDigit: method, 476, 482**
- closeFile method, 398**
- clusters, 488**
- Cocoa, 449**
 - definition of, 307, 488
 - development of, 1
 - framework layers, 449-450
- Cocoa Touch, 307, 450-451, 488**
- collections**
 - definition of, 488
 - set, 492
- colon (:), 123**
- comma (,) operator, 299**
- Command Line Tools, 16**
- command-line arguments, 300-302**
- comments, 19-20**
- compare: method, 315, 322, 332**
- comparing string objects, 322**
- compilation, 7-8**
 - conditional compilation, 245-248
 - with Terminal, 16-18
 - with Xcode, 8-15
- compile time, 184-185, 488**
- compilers**
 - gcc, 490
 - LLVM Clang Objective-C compiler, 17-18
- Complex class, 179-182**
- composite classes, 488**
- composite objects, 234-235**

compound literals, 297-298
compound relational tests, 101-104
concrete subclasses, 488
conditional compilation, 245-248
conditional operator, 123-124
conforming, 489
conformsToProtocol: method, 232
constant character strings, 489
constants
 defined names, 237-244
 definition of, 51
 PI, 238-239
 TWO_PI, 239-241
containIndex: method, 372
containsObject: method, 358, 369, 370
contentsAtPath: method, 378, 384
contentsEqualAtPath: method, 378
contentsOfDirectoryAtPath: method, 377, 387-389
continue statement, 91
conversions (data types)
 conversion rules, 212-214
 integer and floating-point conversions, 61-63
convertToNum method, 95-98
copy method, 419-421
copying, 419
 files
 with NSFileHandle class, 399-402
 with NSProcessInfo class, 394-398
 objects
 with archiver, 446-447
 copy method, 419-421
 deep copying, 422-424, 446-447
 mutableCopy method, 419-421
 <NSCopying> protocol, 424-426

 in setter/getter methods, 427-429
 shallow copying, 422-424
copyItemAtPath: method, 378, 385
copyString function, 293-294
copyWithZone: method, 425-428
Core Data, 307
Core Services layer, 449
count method, 205-207, 358, 365, 372
countForObject: methods, 370
Cox, Brad, 1
createDirectoryAtPath: method, 385
createFileAtPath: method, 378, 384
curly braces ({}), 20
currentDirectoryPath method, 385
custom archives, 442-445

D

data encapsulation, 45-49, 489
data method, 443
data types
 argument types, 263-265
 assigning alternative names to, 211-212
 BOOL, 122-123
 CGPoint, 274
 CGRect, 274
 CGSize, 274
 char, 52-53
 conversions
 conversion rules, 212-214
 integer and floating-point conversions, 61-63
 determining size of, 299-300
 dynamic typing
 argument and return types, 186-187
 definition of, 489

- explained, 182-184
 - methods for working with, 187-189
 - enumerated data types, 207-211
 - explained, 51
 - float, 52
 - id, 54, 304
 - definition of, 490
 - dynamic typing and binding and, 182-183, 186-187
 - static typing and, 185-186
 - int, 20, 51-52. *See also* integers
 - integer and floating-point conversions, 61-63
 - pointers to, 277-281
 - qualifiers, 53-51
 - return types, 263-265
 - static typing, 185-186, 492
 - table of, 55
- dataWithContentsOfURL: method, 404**
- date structure**
- defining, 270-273
 - initialization, 273-274
- debugging**
- gdb tool, 490
 - Xcode projects, 14-15
- decision-making constructs, 93. *See also* loops**
- Boolean variables, 118-123
 - conditional operator, 123-124
 - if statement
 - compound relational tests, 101-104
 - else if construct, 105-115
 - explained, 93-98
 - if-else construct, 98-101
 - nested if statements, 104-105
 - switch statement, 115-118
- declaring. *See also* defining**
- argument types, 263-265
 - arrays, 252-254
 - methods, 35
 - arguments, 36-37
 - return values, 36
 - return types, 263-265
 - strong variables, 415-416
 - weak variables, 416-417
- decodeIntForKey: method, 442**
- decodeObject: method, 436**
- decoding methods, writing, 435-442**
- decrement (-) operator, 78, 291-294**
- deep copying, 422-424, 446-447**
- #define statement, 237-244**
- defined names, 237-244**
- defining. *See also* declaring**
- array objects, 331-337
 - categories, 223-228
 - class extensions, 228-229
 - classes
 - AddressBook class, 344-347
 - AddressCard class, 338-341
 - Fraction class, 30-33
 - @implementation section, 37, 127-133
 - @interface section, 33-37, 127-133
 - program section, 39-45
 - pointers
 - to data types, 277-281
 - to structures, 281-283
 - protocols, 230-233
 - string objects, 317-318
 - structures, 270-273, 274-276
- delegation**
- definition of, 489
 - protocols, 233

deleteCharactersInRange: method, 329, 333

deleting files, 379

denominator method, 46-48

description method, 318-319

designated initializers, 489

development of Objective-C, 1-2

dictionary objects

creating, 361-363

enumerating, 364-365

NSDictionary methods, 365

NSMutableDictionary methods, 365

dictionaryWithCapacity: method, 365

dictionaryWithContentsOfFile: method, 433

dictionaryWithContentsOfURL: method, 404

dictionaryWithObjectsAndKeys: method, 364-365

digits of numbers, reversing, 89-90

directives

@autoreleasepool, 20, 410

@catch, 192-194

@class, 163-167

definition of, 489

@finally, 194

@optional, 231

@package, 201

@private, 201

@property, 133

@protected, 201

@protocol, 232

@public, 201

@selector, 188-189

@synthesize, 134, 202

@throw, 194

@try, 192-194

directories. See also files

common iOS directories, 393

enumerating, 387-389

managing with NSFileManager class, 384-387

dispatch tables, creating, 296

displaying variable values, 22-25

distributed objects, 489

division (/) operator, 54-58

do statement, 89-90

documentation for Foundation framework, 307-310

Documents directory, 393

dollar sign (\$), 16

dot (.) operator, 135-136

double quotes ("), 132

doubleValue method, 332

downloading

iOS SDK (software development kit), 453

Xcode, 8

Drawing protocol, 231-233

dynamic binding, 182-184, 489

dynamic typing

argument and return types, 186-187

definition of, 489

explained, 182-184

methods for working with, 187-189

E

#elif statement, 245-247

else if construct, 105-115

#else statement, 245-247

Empty Application template, 457

encapsulation, 45-49, 489

encodeIntForKey: method, 442

encodeWithCoder: method, 436-442

encoding methods, writing, 435-442

#endif statement, 245-247

enum keyword, 207

enumerated data types, 207-211

enumerateObjectsUsingBlock: method, 358

enumeration

- of dictionaries, 364-365
- of directories, 387-389
- fast enumeration, 347-349

enumeratorAtPath: method, 385-389

environment method, 396

equal to (==) operator, 74

event loop and memory allocation, 135-137

exception handling, 192-194

exchange function, 284

extending classes through inheritance

- @class directive, 163-167
- classes owning their objects, 167-171
- explained, 158-162
- object allocation, 162-163

Extensible Markup Language (XML). See XML (Extensible Markup Language)

extensions (class), 228-229

extern variables. See global variables

F

factory methods. See class methods

factory objects. See objects

fast enumeration, 347-349

Fibonacci numbers, generating, 253-254

fileExistsAtPath: method, 378, 385

fileHandleForReadingAtPath: method, 398

fileHandleForUpdatingAtPath: method, 398

fileHandleForWritingAtPath: method, 398

filename extensions, 12

files

- appending, 402-403
- application bundles, 404-405
- basic file operations with NSFileHandle class, 377, 398-403

- copying
 - with NSFileHandle class, 399-402
 - with NSProcessInfo class, 394-398
- deleting, 379
- directories
 - common iOS directories, 393
 - enumerating, 387-389
 - managing with NSFileManager class, 384-387
- filename extensions, 12
- header files, 490
- main.m, 13
- managing with NSFileManager class, 377-383
- moving, 382
- paths
 - basic path operations, 389-392
 - path utility functions, 393
 - path utility methods, 392-394
- reading to/from buffer, 383-384
- removing, 382
- system files, 20
- Web files, reading with NSURL class, 403-404
- xib files, 462

@finally directive, 194

finishEncoding message, 444

first iPhone application

- AppDelegate class, 460
- application templates, 457
- interface design, 462-469
 - button, 466-468
 - label, 464-465
- overview, 453-469
- project, creating, 456-458
- ViewController class, 460-462

firstIndex method, 372

float data type, 52, 61-63

floatValue method, 332

fnPtr pointer, 363-365

for statement

execution order, 75

explained, 72-79

infinite loops, 84

keyboard input, 79-83

nested loops, 81-83

syntax, 73-75

variants, 83-84

formal protocols, 489

forums, classroomM.com/objective-c, 5

forwarding, 489

forwardInvocation: method, 189

Foundation framework

address book program. *See* address book program

archiving

copying objects with, 446-447

definition of, 431

encoding/decoding methods, 435-442

with NSData, 442-445

with NSKeyedArchiver, 434-435

with XML property lists, 431-433

array objects

address book example. *See* address book program

defining, 331-337

classes

abstract classes, 176

NSArray, 311, 331-337, 358

NSBundle, 404-405

NSCountedSet, 370

NSData, 383-384, 442-445

NSFileHandle, 377, 398-403

NSFileManager, 377-387

NSIndexSet, 371-372

NSKeyedArchiver, 434-435

NSMutableArray, 331-337, 358

NSMutableSet, 367-370

NSMutableString, 326-330, 333-331

NSNumber, 311-317

NSProcessInfo, 394-398

NSSet, 367-370

NSString, 317-331

NSURL, 403-404

NSValue, 360-361

Cocoa, 449-450

Cocoa Touch, 450-451

copying objects

copy method, 419-421

deep copying, 422-424

mutableCopy method, 419-421

<NSCopying> protocol, 424-426

in setter/getter methods, 427-429

shallow copying, 422-424

definition of, 489

dictionary objects

creating, 361-363

enumerating, 364-365

NSDictionary methods, 365

NSMutableDictionary methods, 365

directories

enumerating, 387-389

managing with NSFileManager class, 384-387

documentation, 307-310

exercises, 373-375

explained, 307

- file paths
 - basic path operations, 389-392
 - path utility functions, 393
 - path utility methods, 392-394
- files, 377-378
 - appending, 402-403
 - application bundles, 404-405
 - basic file operations with
 - NSFileHandle class, 398-403
 - copying with NSFileHandle class, 399-402
 - copying with NSProcessInfo class, 394-398
 - deleting, 379
 - managing with NSFileManager class, 378-383
 - moving, 382
 - removing, 382
 - Web files, reading with NSURL class, 403-404
- memory management
 - ARC (Automatic Reference Counting), 415-418
 - autorelease pool, 20
 - explained, 407-408
 - garbage collection, 409, 490
 - manual reference counting, 409-415
- number objects, 311-317
- set objects
 - NSCountedSet class, 370
 - NSIndexSet, 371-372
 - NSMutableSet, 367-370
 - NSSet, 367-370
- string objects
 - comparing, 322
 - defining, 317-318
 - description method, 318-319
 - explained, 317
 - immutable strings, 319-326
 - joining, 321
 - mutable strings, 326-330
 - NSLog function, 317-318
 - NSString methods, 331-332
 - substrings, 323-326
 - testing equality of, 322
- fraction calculator**
 - Calculator class, 480-482
 - @implementation section, 481-482
 - @interface section, 481
 - creating project, 471
 - Fraction class, 477-480
 - @implementation section, 478-480
 - @interface section, 477
 - overview, 469-470
 - summary, 482-484
 - user interface design, 482
 - ViewController class, 471-477
 - @implementation section, 473-476
 - @interface section, 472
- Fraction class, 30-33, 477-480**
 - add: method, 139-143, 149-151, 411
 - adding to projects, 127-130
 - allocF method, 205-207
 - convertToNum method, 95-98
 - count method, 205-207
 - data encapsulation, 45-49
 - @implementation section, 37, 131-132, 138, 146-147, 478-480
 - initWith:over: method, 197-200
 - instance variables, accessing, 45-49
 - @interface section, 33-37, 130-131, 141, 146, 477
 - program section, 39-45
 - setTo:over: method, 137-139

Fraction.h interface file, 130-131

Fraction.m implementation file, 131-132

FractionTest project

Fraction.h interface file, 130-131

Fraction.m implementation file, 131-132

main.m, 127-128

output, 133

framework layers, 449-450

frameworks, 489. See also Foundation framework

Free Software Foundation (FSF), 1

FSF (Free Software Foundation), 1

functions. See also methods

arguments, 259-261

pointers, 283-284

copyString, 293-294

definition of, 489

exchange, 284

explained, 258-259

gcd, 261-263

local variables, 259-261

minimum, 265-266

NSFullUserName, 393

NSHomeDirectory, 392-393

NSHomeDirectoryForUser, 393

NSLog, 317-318

NSSearchPathForDirectoriesInDomains,
393

NSTemporaryDirectory, 391-393

NSUserName, 393

passing arrays to, 265-266

pointers to, 295-296

qsort, 296

return values, 261-265

static functions, 492

G

garbage collection, 409, 490

gcc, 490

**gcd (greatest common divisor), calculating,
86-87, 261-263**

gcd function, 261-263

gdb, 490

getters

copying objects in, 427-429

definition of, 490

explained, 48-49

synthesizing, 133-135, 202-203

global variables

definition of, 490

scope, 203-205

globallyUniqueString method, 396

glossary, 487-493

GNU General Public License, 1

GNUStep, 1

goto statement, 298

greater than (>) operator, 74

greater than or equal to (>=) operator, 74

**greatest common divisor (gcd), calculating,
86-87, 261-263**

H

handling exceptions, 192-194

hasPrefix: methods, 332

hasSuffix: method, 332

header files, 490

help

classroomM.com/objective-c, 5

Foundation framework documentation,
307-310

Mac OS X reference library, 310

Quick Help panel, 309-310

hexadecimal (base 16) notation, 54
 history of Objective-C, 1-2
 hostName method, 396
 hyphen (-), 35

I

id data type, 54, 304
 definition of, 490
 dynamic typing and binding and, 182-183, 186-187
 static typing and, 185-186

#if statement, 245-247

if statement
 compound relational tests, 101-104
 else if construct, 105-115
 explained, 93-98
 if-else construct, 98-101
 nested if statements, 104-105

#ifdef statement, 245-247

if-else construct, 98-101

#ifndef statement, 245-247

immutable objects
 definition of, 490
 immutable strings, 319-326

@implementation section, 37
 AddressBook class, 345-346, 497-500
 AddressCard class, 339-342, 496-497
 Calculator class, 481-482
 Complex class, 180
 definition of, 490
 Fraction class, 127-133, 131-132, 138, 141-142, 146-147, 478-480
 ViewController class, 473-476

#import statement, 244-245

increment (++) operator, 78, 291-294

indexesOfObjectsPassingTest: method, 372

indexesPassingTest: method, 372

indexLessThanIndex: method, 372

indexOfObject: method, 358

indexOfObjectPassingTest: method, 358, 371

indirection (*) operator, 278

infinite loops, 84

informal protocols, 233-234, 490

inheritance
 definition of, 490
 explained, 153-158
 extending classes with, 158-171

init method, 40, 197
 overriding, 198

initialization
 arrays, 254-255
 designated initializers, 489
 instances, 40
 objects, 197-200
 structures, 273-274

initWithCapacity: method, 333, 358, 365, 370

initWithCoder: method, 436-442

initWithContentsOfFile: method, 332

initWithContentsOfURL: method, 332

initWithName: method, 346

initWithObjects: method, 370

initWithObjectsAndKeys: method, 365

initWith:over: method, 197-200

initWithString: method, 332

insertObject:, 358

insertString: method, 333

insertString:atIndex: method, 329

installation, Xcode Command Line Tools, 16

instance methods, 29, 35, 490

instance variables, 38

- accessing, 45-49
- definition of, 490
- scope, 200-203
- storing in structures, 303

instances

- allocation, 40
- definition of, 490
- explained, 28-30
- extending classes with
 - @class directive, 163-167
 - classes owning their objects, 167-171
 - explained, 158-162
 - object allocation, 162-163
- initialization, 40

instancesRespondToSelector: method, 187

int data type, 20, 51-52. See also integers

integers

- arithmetic, 58-60
- calculating absolute value of, 94
- conversions, 61-63
- int data type, 20, 51-52
- NSInteger, 313

integerValue method, 332

Interface Builder, 490

interface design (first iPhone application), 462-469

- button, 466-468
- label, 464-465

@interface section, 33-37

- AddressBook class, 345, 496
- AddressCard class, 338-339, 495
- Calculator class, 481
- class names, 34-35
- definition of, 490

Fraction class, 127-133, 141, 146, 477

method declarations, 35

arguments, 36-37

class methods versus instance methods, 35

return values, 36

ViewController class, 472

internationalization. See localization

intersect: method, 369

intersectSet: method, 370

intersectsSet: methods, 370

intNumber method, 313

intValue method, 332

iOS applications

- application templates, 457
- first iPhone application, 453-469
 - AppDelegate class, 460
 - interface design, 462-469
 - overview, 453-456
 - project, creating, 456-458
 - ViewController class, 460-462
- fraction calculator
 - Calculator class, 480-482
 - creating project, 471
 - Fraction class, 477-480
 - overview, 469-470
 - summary, 482-484
 - user interface design, 482
 - ViewController class, 471-477
- iOS SDK, 453

iOS SDK (software development kit), 2, 453

iPhone applications. See iOS applications

IS_LOWER_CASE macro, 243

isa variable, 490

isEqual: method, 352-353

isEqualToNumber: method, 315

isEqualToSet: method, 370
isEqualToString: method, 322, 332
isKindOfClass: method, 187
isMemberOfClass: method, 187
isReadableFileAtPath: method, 378
isSubclassOfClass: method, 187
isSubsetOfSet: method, 370
isWritableFileAtPath: method, 378

J-K

joining character strings, 321
keyed archives, 434-435
keyEnumerator method, 365
keysSortedByValueUsingSelector: method, 365
keywords

- enum, 207
- main, 20
- self, 148-149
- static, 144-148
- __strong, 416
- super, 492
- __weak, 417

L

labels, adding, 464-465
lastIndex method, 372
lastObject method, 358
lastPathComponent method, 391-392
layers (framework), 449-450
leap years, determining, 102-103
left-shift (<<) operator, 219
length method, 332
less than (<) operator, 74
less than or equal to (<=) operator, 74

Library/Caches directory, 393
Library/Preferences directory, 393
linking, 490
LinuxSTEP, 1
literals, compound, 297-298
LLVM Clang Objective-C compiler, 17-18
local variables

- definition of, 491
- explained, 143-144
- function arguments, 259-261
- method arguments, 144
- static variables, 144-148

localization, 491
logical AND (&&) operator, 101
logical negation (!) operator, 121
logical OR (||) operator, 101
long qualifier, 53-54
looking up address book entries, 349-351
lookup: method, 349-351, 371-372
loops

- break statement, 91
- continue statement, 91
- do statement, 89-90
- explained, 71-72
- for statement
 - execution order, 75
 - explained, 72-84
 - infinite loops, 84
 - keyboard input, 79-83
 - nested loops, 81-83
 - syntax, 73-75
 - variants, 83-84
- while statement, 84-89

lowercaseString method, 332

M

M_PI, 239**Mac OS X reference library, 310****macros, 242-244**

IS_LOWER_CASE, 243

MakeFract, 243

MAX, 243

SQUARE, 242-243

TO_UPPER, 244

main keyword, 20**mainBundle method, 405****main.m, 13****MakeFract macro, 243****makeObjectsPerform Selector: method, 358****manual memory management rules,
414-415****manual reference counting**

autorelease pool, 410-412

event loop and memory allocation,
135-137

explained, 409-410

manual memory management rules,
414-415**Master-Detail application template, 457****MathOps category, defining, 223-228****MAX macro, 243****member: method, 370****memberDeclarations (@implementation section), 37****memory addresses, pointers to, 296-297****memory management**

ARC (Automatic Reference Counting)

@autoreleasepool blocks, 417-418

explained, 415

with non-ARC compiled code, 418

strong variables, 415-416

weak variables, 416-417

autorelease pool, 20

explained, 407-408

garbage collection, 409, 490

manual reference counting

autorelease pool, 410-412

event loop and memory allocation,
135-137

explained, 409-410

manual memory management rules,
414-415**messages**

autorelease, 410

definition of, 491

finishEncoding, 444

message expression, 491

release, 409

retain, 409

methodDefinitions (@implementation section), 38**methods. See also functions**

accessor methods

definition of, 487

explained, 48-49

synthesized accessors, 133-135,
202-203, 493

add:, 139-143, 149-151, 411

adding to classes

@class directive, 163-167

classes owning their objects,
167-171

explained, 158-162

object allocation, 162-163

addObject:, 358, 370

allKeys, 365

alloc, 40

- allocF, 205-207
- allocWithZone:, 425
- anyObject, 370
- appendString:, 333
- archiveRootObject:, 434
- arguments, 396
 - local variables, 144
 - methods without argument names, 139
 - multiple arguments, 137-143
 - pointers, 283-284
- array, 358
- arrayWithCapacity:, 358
- arrayWithContentsOfFile:, 407, 433
- arrayWithObjects:, 334, 358
- attributesOfItemAtPath:, 378
- availableData, 398
- calculate:, 144
- calculateTriangularNumber, 259-261
- capitalizedString, 332
- caseInsensitiveCompare:, 322, 332
- changeCurrentDirectoryPath:, 385
- characterAtIndex:, 332
- class methods versus instance methods, 29, 35, 488-490
- clickDigit:, 476, 482
- closeFile, 398
- compare:, 315, 322, 332
- conformsToProtocol:, 232
- containIndex:, 372
- containsObject:, 358, 369-370
- contentsAtPath:, 378, 384
- contentsEqualAtPath:, 378
- contentsOfDirectoryAtPath:, 377, 387-389
- convertToNum, 95-98
- copy, 419-421
- copyItemAtPath:, 378, 385
- copyWithZone:, 425-428
- count, 205-207, 358, 365, 372
- countForObject:, 370
- createDirectoryAtPath:, 385
- createFileAtPath:, 378, 384
- currentDirectoryPath, 385
- data, 443
- dataWithContentsOfURL:, 404
- declaring, 35
 - arguments, 36-37
 - return values, 36
- decodeIntForKey:, 442
- decodeObject:, 436
- definition of, 491
- deleteCharactersInRange:, 329, 333
- description, 318-319
- dictionaryWithCapacity:, 365
- dictionaryWithContentsOfFile:, 433
- dictionaryWithContentsOfURL:, 404
- dictionaryWithObjectsAndKeys:, 364-365
- doubleValue, 332
- encodeIntForKey:, 442
- encodeWithCoder:, 436-442
- encoding/decoding methods, 435-442
- enumerateObjectsUsingBlock:, 358
- enumeratorAtPath:, 385-389
- environment, 396
- explained, 28-30, 304
- fileExistsAtPath:, 378, 385
- fileHandleForReadingAtPath:, 398
- fileHandleForUpdatingAtPath:, 398
- fileHandleForWritingAtPath:, 398
- firstIndex, 372
- floatValue, 332
- forwardInvocation:, 189

- getters
 - copying objects in, 427-429
 - definition of, 490
 - explained, 48-49
 - synthesizing, 133-135, 202-203
- globallyUniqueString, 396
- hasPrefix:, 332
- hasSuffix:, 332
- hostName, 396
- indexesOfObjectsPassingTest:, 372
- indexesPassingTest:, 372
- indexLessThanIndex:, 372
- indexOfObject:, 358
- indexOfObjectPassingTest:, 358, 371
- indexSet
- init, 40, 197
 - overriding, 198
- initWithCapacity:, 333, 358, 365, 370
- initWithCoder:, 436-442
- initWithContentsOfFile:, 332
- initWithContentsOfURL:, 332
- initWithName:, 346
- initWithObjects:, 370
- initWithObjectsAndKeys:, 365
- initWith:over:, 197-200
- initWithString:, 332
- insertObject:, 358
- insertString:, 333
- insertString:atIndex:, 329
- instancesRespondToSelector:, 187
- integerValue, 332
- intersect:, 369
- intersectSet:, 370
- intersectsSet:, 370
- intValue, 313
- intValue, 332
- isEqual:, 352-353
- isEqualToNumber:, 315
- isEqualToSet:, 370
- isEqualToString:, 322, 332
- isKindOfClass:, 187
- isMemberOfClass:, 187
- isReadableFileAtPath:, 378
- isSubclassOfClass:, 187
- isSubsetOfSet:, 370
- isWritableFileAtPath:, 378
- keyEnumerator, 365
- keysSortedByValueUsingSelector:, 365
- lastIndex, 372
- lastObject, 358
- lastPathComponent, 391
- length, 332
- lookup:, 349-351, 371-372
- lowercaseString, 332
- mainBundle, 405
- makeObjectsPerform Selector:, 358
- member:, 370
- minusSet:, 370
- moveItemAtPath:, 378, 385
- mutableCopy, 419-421
- mutableCopyWithZone:, 425
- new, 49
- numberWithInt:, 315
- numberWithInteger:, 315
- objectAtIndex:, 334, 358
- objectEnumerator, 365, 370
- objectForKey:, 363-365
- offsetInFile, 398
- operatingSystem, 396
- operatingSystemName, 396
- operatingSystemVersionString, 396
- overriding, 171-175
- passing arrays to, 265-266
- pathComponents, 392

pathExtension, 391-392
 pathsForResourcesOfType:, 405
 pathWithComponents:, 392
 performSelector:, 187-189
 print, 369
 processDigit:, 476
 processIdentifier, 396
 processInfo, 396
 processName, 396
 rangeOfString:, 325, 329
 readDataToEndOfFile, 398
 reduce, 143-144
 removeAllObjects, 365, 370
 removeObjectAtPath:, 378, 385
 removeObject:, 358, 370
 removeObjectAtIndex:, 358
 removeObjectForKey:, 365
 replaceCharactersInRange:, 333
 replaceObject:, 424
 replaceObjectAtIndex:, 358
 replaceOccurrencesOfString:withString:
 options:range:, 330, 333
 respondsToSelector:, 187, 189
 returning objects from, 149-151
 seekToEndOfFile, 398
 seekToFileOffset:, 398
 self keyword, 148-149
 set:, 139
 setAttributesOfItemAtPath:, 378
 setDenominator:, 39-41
 setEmail:, 340
 setName:, 340
 setName:andEmail:, 343
 setNumerator:, 39-41
 setNumerator:andDenominator:
 method, 137
 setObject:, 365
 setProcessName:, 396
 setString:, 330, 333
 setters
 copying objects in, 427-429
 definition of, 492
 explained, 48-49
 synthesizing, 133-135, 202-203
 setTo:over:, 137-139
 setWithCapacity:, 370
 setWithObjects:, 369-370
 sortedArrayUsing Selector:, 358
 sortedArrayUsingComparator:, 357-358
 sortUsingComparator:, 357-358
 sortUsingSelector:, 355-358
 string, 332
 stringByAppendingPathComponent:
 391-392
 stringByAppendingPathExtension:, 392
 stringByAppendingString:, 321
 stringByDeletingLastPathComponent,
 392
 stringByDeletingPathExtension, 392
 stringByExpandingTildeInPath, 392
 stringByResolvingSymlinksInPath, 392
 stringByStandardizingPath, 392
 stringWithCapacity:, 333
 stringWithContentsOfFile:, 332, 433
 stringWithContentsOfURL:, 332
 stringWithFormat:, 319, 332
 stringWithString:, 329, 332, 424
 substringFromIndex:, 325, 332
 substringToIndex:, 325, 332
 substringWithRange:, 325, 332
 syntax, 28-29
 truncateFileAtOffset:, 398

- unarchiveObjectWithFile:, 435
- union:, 369
- unionSet:, 370
- uppercaseString, 332
- URLWithString:, 403
- UTF8String, 332
- writeData:, 398
- writeToFile:, 358
- writeToFile:atomically:, 431-432
- minimum function, 265-266**
- minus sign (-), 35, 54, 58-60**
- minusSet: method, 370**
- modulus (%) operator, 60-61**
- moveItemAtPath: method, 378, 385**
- moving files, 382**
- multidimensional arrays, 256-258**
- multiple arguments to methods, 137-143**
- multiplication (*) operator, 54-58**
- MusicCollection class, 374-375**
- mutable objects**
 - definition of, 491
 - NSMutableArray class
 - defining, 331-337
 - methods, 358
 - NSMutableDictionary class
 - defining, 361-363
 - enumerating, 364-365
 - methods, 365
 - NSMutableSet class, 367-370
 - NSMutableString class, 326-330, 333-331
- mutableCopy method, 419-421**
- mutableCopyWithZone: method, 425**
- myFraction variable, 39**

N

- \n (newline character), 22**
- names**
 - assigning to data types, 211-212
 - class names, 34-35
 - defined names, 237-244
- native applications, 2**
- nested for loops, 81-83**
- nested if statements, 104-105**
- new method, 49**
- newline character, 22**
- NeXT Software, 1**
- NEXTSTEP, 1**
- nib files, 462**
- nil objects, 491**
- not equal to (!=) operator, 74**
- notification, 491**
- NSArray class, 311**
 - archiving, 431-433
 - defining, 331-337
 - methods, 358
- NSBundle class, 404-405**
- NSCopying protocol, 230-231**
- <NSCopying> protocol, 424-426**
- NSCountedSet class, 370**
- NSData class, 383-384, 431-433, 442-445**
- NSDate class, archiving, 431-433**
- NSDictionary class**
 - archiving, 431-433
 - defining, 361-363
 - enumerating, 364-365
 - methods, 365
- NSFileHandle class, 377, 398-403**

NSFileManager class, 377
 directory enumeration, 387-389
 directory management, 384-387
 management, 378-383

NSFullUserName function, 393

NSHomeDirectory function, 392-393

NSHomeDirectoryForUser function, 393

NSIndexSet class, 371-372

NSInteger, 313

NSKeyedArchiver class, 434-435

NSLog routine, 317-318
 displaying text with, 21-22
 displaying variable values with, 22-25

NSMutableArray class
 defining, 331-337
 methods, 358

NSMutableDictionary class
 defining, 361-363
 enumerating, 364-365
 methods, 365

NSMutableSet class, 367-370

NSMutableString class, 326-330, 333-331

NSNumber class, 311-317, 431-433

NSObject, 491

NSPathUtilities.h, 389-392

NSProcessInfo class, 394-398

NSSearchPathForDirectoriesInDomains function, 393

NSSet class, 367-370

NSString class
 archiving, 431-433
 description method, 318-319
 explained, 317
 mutable versus immutable objects, 319-326
 NSLog function, 317-318

NSTemporaryDirectory function, 391-393

NSURL class, 403-404

NSUserName function, 393

NSNumber class, 360-361

null character, 491

null pointers, 491

null statement, 298-299

numbers
 determining whether even or odd, 93-98
 Fibonacci numbers, generating, 253-254

integers
 arithmetic, 58-60
 calculating absolute value of, 94
 conversions, 61-63
 int data type, 20, 51-52
 integer and floating-point conversions, 61-63

number objects, 311-317
 prime numbers, generating, 119-123
 reversing digits of, 89-90
 triangular numbers, generating, 259-261

numberWithInt: method, 315

numberWithInteger: method, 315

numerator method, 46-48, 71-82

O

object variables, 303

objectAtIndex: method, 334, 358

objectEnumerator method, 365, 370

objectForKey: method, 363-365

object-oriented programming, 491

objects
 allocation, 149-151, 162-163
 archiving
 copying objects with, 446-447
 definition of, 431, 487

- encoding/decoding methods, 435-442
 - with NSData, 442-445
 - with NSKeyedArchiver, 434-435
 - with XML property lists, 431-433
- array objects
 - address book example. *See* address book program
 - defining, 331-337
- class objects, 488
- composite objects, 234-235
- copying
 - with archiver, 446-447
 - copy method, 419-421
 - deep copying, 422-424, 446-447
 - mutableCopy method, 419-421
 - <NSCopying> protocol, 424-426
 - in setter/getter methods, 427-429
 - shallow copying, 422-424
- definition of, 488, 491
- dictionary objects
 - creating, 361-363
 - enumerating, 364-365
 - NSDictionary methods, 365
 - NSMutableDictionary methods, 365
- distributed objects, 489
- explained, 27-28
- immutable objects
 - definition of, 490
 - immutable strings, 319-326
- initialization, 197-200
- mutable objects, 326-330, 491
- nil objects, 491
- NSObject, 491
- number objects, 311-317
- returning from methods, 149-151
- root objects, 492
- set objects
 - NSCountedSet class, 370
 - NSIndexSet, 371-372
 - NSMutableSet, 367-370
 - NSSet, 367-370
- string objects
 - comparing, 322
 - defining, 317-318
 - description method, 318-319
 - explained, 317
 - immutable strings, 319-326
 - joining, 321
 - mutable strings, 326-330
 - NSLog function, 317-318
 - NSMutableString methods, 333-331
 - NSString methods, 332-331
 - substrings, 323-326
 - testing equality of, 322
- octal (base 8) notation, 54**
- offsetInFile method, 398**
- ones complement (~) operator, 217-219**
- OOP (object-oriented programming), 491**
- OpenGL Game application template, 457**
- OPENSTEP, 1**
- operatingSystem method, 396**
- operatingSystemName method, 396**
- operatingSystemVersionString method, 396**
- operators**
 - address (&), 278
 - arithmetic operators
 - binary arithmetic operators, 54-58
 - integer and floating-point conversions, 61-63
 - modulus (%) operator, 60-61
 - type cast operator, 63-64
 - unary minus (-) operator, 58-60
 - assignment operators, 64-65, 74

bit operators

binary, decimal, and hexadecimal equivalents, 214

bitwise AND (&), 215-216

bitwise OR (|), 216-217

bitwise XOR (^), 217

left-shift (<<) operator, 219

ones complement (~) operator, 217-219

right-shift (>>) operator, 219-220

table of, 214

comma (,), 299

conditional operator, 123-124

decrement (--), 78, 291-294

dot (.), 135-136

increment (++), 78, 291-294

indirection (*), 278

logical AND (&&), 101

logical negation (!), 121

logical OR (||), 101

relational operators, 74-75

sizeof, 299-300

@optional directive, 231

OR operator (|), 216-217

OS X, 1

overriding methods, 171-175, 198

P

@package directive, 201

Page-Based Application template, 457

parent classes, 153-155, 491

pathComponents method, 392

pathExtension method, 391-392

paths

basic path operations, 389-392

path utility functions, 393

path utility methods, 392-394

pathsForResourceOfType: method, 405

pathWithComponents: method, 392

performSelector: method, 187-189

PI constant, 238-239

Playlist class, 374-375

plists. See property lists

plus sign (+), 54-58

pointers

to arrays, 284-294

increment and decrement operators, 291-294

pointers to character strings, 289-291

valuesPtr example, 284-288

to character strings, 289-291

to data types, 277-281

definition of, 491

to functions, 295-296

and memory addresses, 296-297

object variables as, 303

operations, 294-295

passing to methods/functions, 283-284

to structures, 281-283

polymorphism, 179-182, 491

pound sign (#), 237

precedence

arithmetic operators, 54-58

relational operators, 74

preprocessor

conditional compilation, 245-248

definition of, 491

explained, 237

statements

#define, 237-244

#elif, 245-247

#else, 245-247

#endif, 245-247

- #if, 245-247
- #ifdef, 245-247
- #ifndef, 245-247
- #import, 244-245
- #undef, 245-247
- prime numbers, generating, 119-123**
- print method, 38, 41, 369**
- @private directive, 201**
- procedural programming languages, 491**
- processDigit: method, 476**
- processIdentifier method, 396**
- processInfo method, 396**
- processName method, 396**
- "Programming is fun!" sample program**
 - code listings, 7, 18-22
 - compiling and running, 7-8
 - with Terminal, 16-18
 - with Xcode, 8-15
 - explained, 18-22
- programs, compiling and running, 7-8. See also iOS applications**
 - with Terminal, 16-18
 - with Xcode, 8-15
- projects (Xcode). See also iOS applications**
 - adding classes to, 127-130
 - application templates, 457
 - creating, 15
 - debugging, 14-15
 - filename extensions, 12
 - first iPhone application
 - AppDelegate class, 460
 - creating project, 456-458
 - interface design, 462-469
 - overview, 453-456
 - ViewController class, 460-462
 - fraction calculator
 - Calculator class, 480-482
 - creating project, 471
 - Fraction class, 477-480
 - overview, 469-470
 - summary, 482-484
 - user interface design, 482
 - ViewController class, 471-477
 - FractionTest
 - Fraction.h interface file, 130-131
 - Fraction.m implementation file, 131-132
 - main.m, 127-128
 - output, 133
 - main.m, 13
 - project window, 10-11
 - running, 14
 - starting, 8-11
- properties**
 - accessing with dot operator, 135-136
 - property declarations, 492
 - property lists. *See* property lists
- property declarations, 492**
- @property directive, 133**
- property lists**
 - archiving with, 431-433
 - definition of, 492
- @protected directive, 201**
- @protocol directive, 232**
- protocols**
 - defining, 230-233
 - definition of, 492
 - delegation, 233
 - explained, 230
 - formal protocols, 489
 - informal protocols, 233-234, 490
 - NSCopying, 230-231
 - <NSCopying> protocol, 424-426
- @public directive, 201**

Q

qsort function, 296

qualifiers, 51-53

- long, 53-54
- short, 54
- unsigned, 54

question mark (?), 123

Quick Help pane, 309-310

R

rangeOfString: method, 329

readDataToEndOfFile method, 398

reading files to buffer, 383-384

receivers, 492

Rectangle class, 158-171

reduce method, 143-144

reference counting

- ARC (Automatic Reference Counting)
 - @autoreleasepool blocks, 417-418
 - explained, 415
 - with non-ARC compiled code, 418
 - strong variables, 415-416
 - weak variables, 416-417
- manual reference counting
 - autorelease pool, 410-412
 - event loop and memory allocation, 135-137
 - explained, 409-410
 - manual memory management rules, 414-415

relational operators, 74-75

release message, 409

removeAllObjects method, 365, 370

removeCard: method, 351-355

removeItemAtPath: method, 378, 385

removeObject: method, 358, 370

removeObjectAtIndex: method, 358

removeObjectForKey: method, 365

removing

- address book entries, 351-355
- files from directories, 382

replaceCharactersInRange: method, 333

replaceObject: method, 424

replaceObjectAtIndex: method, 358

replaceOccurrencesOfString:withString:options:range: method, 330, 333

reserved words. See keywords; statements

respondToSelector: method, 187-189

retain count, 492. See also reference counting

retain message, 409

return types, declaring, 263-265

return values

- function return values, 261-265
- method return values, 36

returning objects from methods, 149-151

reversing digits of numbers, 89-90

right-shift (>>) operator, 219-220

Ritchie, Dennis, 1

root classes, 153

root objects, 492

routines

- NSLog
 - displaying text with, 21-22
 - displaying variable values with, 22-25
- scanf, 79-83

running programs, 7-8

- with Terminal, 16-18
- with Xcode, 8-15

runtime, 184-185, 492

S

scanf routine, 79-83**scope**

- global variables, 203-205
- instance variables, 200-203
- static variables, 205-207

SDK (software development kit). See software development kit (SDK)

seekToEndOfFile method, 398

seekToFileOffset: method, 398

@selector directive, 188-189

selectors, 492

self keyword, 148-149

self variable, 492

semicolon (;), 84

set collection, 492

set:: method, 139

set objects

- NSCountedSet class, 370
- NSIndexSet, 371-372
- NSMutableSet, 367-370
- NSSet, 367-370

setAttributesOfItemAtPath: method, 378

setDenominator: method, 39-41

setEmail: method, 340

setName: method, 340

setName:andEmail:, 343

setNumerator: method, 39-41

setNumerator:andDenominator: method, 137

setObject: method, 365

setProcessName: method, 396

setString: method, 330, 333

setters

- copying objects in, 427-429
- definition of, 492

explained, 48-49

synthesizing, 133-135, 202-203

setTo:over: method, 137-139

setWithCapacity: method, 370

setWithObjects: method, 369-370

shallow copying, 422-424

short qualifier, 54

sign function, implementing, 106-107

Single View Application template, 457

size of data types, determining, 299-300

sizeof operator, 299-300

slash (/), 54-58

software development kit (SDK), 2, 453

Song class, 374-375

sortedArrayUsing Selector: method, 358

sortedArrayUsingComparator: method, 357-358

sorting address book entries, 355-358

sortUsingComparator: method, 357-358

sortUsingSelector: method, 355-358

Square class, 160-162, 234-235

SQUARE macro, 242-243

starting Xcode projects, 8-11

statement blocks. *See* blocks

statements

- break, 91
- continue, 91
- definition of, 492
- do, 89-90
- for
 - execution order, 75
 - explained, 72-79
 - infinite loops, 84
 - keyboard input, 79-83
 - nested loops, 81-83

- syntax, 73-75
 - variants, 83-84
- goto, 298
- if
 - compound relational tests, 101-104
 - else if construct, 105-115
 - explained, 93-98
 - if-else construct, 98-101
 - nested if statements, 104-105
- null, 298-299
- preprocessor statements
 - #define, 237-244
 - #elif, 245-247
 - #else, 245-247
 - #endif, 245-247
 - #if, 245-247
 - #ifdef, 245-247
 - #ifndef, 245-247
 - #import, 244-245
 - #undef, 245-247
- switch, 115-118
- typedef, 211-212, 274
- while, 84-89
- static analyzer (Xcode), 15
- static functions, 492
- static keyword, 144-148
- static local variables, 261
- static typing, 185-186, 492
- static variables, 144-148
 - definition of, 492
 - scope, 205-207
- string method, 332
- string objects
 - character strings, 488
 - comparing, 322
 - constant character strings, 489
 - defining, 317-318
 - definition of, 488
 - description method, 318-319
 - explained, 317
 - immutable strings, 319-326
 - joining, 321
 - limitations, 297
 - mutable strings, 326-330
 - NSLog function, 317-318
 - NSMutableString methods, 333-331
 - NSString methods, 332-331
 - pointers to, 289-291
 - substrings, 323-326
 - testing equality of, 322
- stringByAppendingPathComponent: method, 391-392**
- stringByAppendingPathExtension: method, 392**
- stringByAppendingString: method, 321**
- stringByDeletingLastPathComponent method, 392**
- stringByDeletingPathExtension method, 392**
- stringByExpandingTildeInPath method, 392**
- stringByResolvingSymlinksInPath method, 392**
- stringByStandardizingPath method, 392**
- stringWithCapacity: method, 333**
- stringWithContentsOfFile: method, 332, 433**
- stringWithContentsOfURL: method, 332**
- stringWithFormat: method, 319, 332**
- stringWithString: method, 329, 332, 424**
- __strong keyword, 416**
- strong variables, 415-416**
- structures
 - date
 - defining, 270-273
 - initialization, 273-274

- defining, 270-276
- definition of, 492
- initialization, 273-274
- instance variables stored in, 303
- limitations, 297
- pointers to, 281-283
- structures within structures, 274-276
- subclasses, 153-155**
 - concrete subclasses, 488
 - definition of, 492
- substringFromIndex: method, 325, 332**
- substrings, 323-326**
- substringToIndex: method, 325, 332**
- substringWithRange: method, 325, 332**
- subtraction (-) operator, 54**
- super keyword, 492**
- superclasses, 153-155, 492**
- support**
 - classroomM.com/objective-c, 5
 - Foundation framework documentation, 307-310
 - Mac OS X reference library, 310
 - Quick Help panel, 309-310
- switch statement, 115-118**
- @synthesize directive, 134, 202**
- synthesized accessors, 133-135, 202-203, 341-344, 493**
- system files, 20**

T

- Tabbed Application template, 457**
- tables, dispatch tables, 296**
- templates, application templates, 457**
- Terminal, compiling programs with, 16-18**
- text, displaying with NSLog routine, 21-22**

- @throw directive, 194**
- tilde (~), 217-219, 378**
- tmp directory, 393**
- TO_UPPER macro, 244**
- triangular numbers**
 - calculating, 71-82
 - generating, 259-261
- triangularNumber program, 71-72**
- truncateFileAtOffset: method, 398**
- @try blocks, 192-194**
- TWO_PI constant, 239-241**
- two-dimensional arrays, 256-258**
- type cast operator, 63-64**
- typedef statement, 211-212, 274**
- types. See data types**

U

- UIKit, 493**
- unarchiveObjectWithFile: method, 435**
- unary minus (-) operator, 58-60**
- #undef statement, 245-247**
- underscore (_), 34, 202**
- unichar characters, 317**
- Unicode characters, 493**
- union: method, 369**
- unions, 493**
- unionSet: method, 370**
- unsigned qualifier, 54**
- uppercaseString method, 332**
- URL addresses, reading files from, 403-404**
- URLWithString: method, 403**
- UTF8String method, 332**
- Utility Application template, 457**

V

values

- displaying, 22-25
- return values
 - function return values, 261-265
 - method return values, 36

valuesPtr pointer, 284-288**variables**

- automatic variables, 488
- Boolean variables, 118-123
- global variables
 - definition of, 490
 - scope, 203-205
- instance variables, 38
 - accessing, 45-49
 - definition of, 490
 - scope, 200-203
 - storing in structures, 303
- isa, 490
- local variables
 - definition of, 491
 - explained, 143-144
 - in functions, 259-261
 - method arguments, 144
 - static variables, 144-148
- myFraction, 39
- object variables, 303
- scope
 - global variables, 203-205
 - instance variables, 200-203
 - static variables, 205-207
- self, 492
- static variables
 - definition of, 492
 - scope, 205-207
- strong variables, 415-416

- values, displaying, 22-25
- weak variables, 416-417

ViewController class

- first iPhone application, 460-462
- fraction calculator, 471-477
 - @implementation section, 473-476
 - @interface section, 472

W

__weak keyword, 417**weak variables, 416-417****Web files, reading with NSURL class, 403-404****Web-based applications, 2****while statement, 84-89****writeData: method, 398****writeToFile: method, 358****writeToFile:atomically: method, 431-432****writing files from buffer, 383-384**

X-Y-Z

Xcode, 8-15

- Command Line Tools, 16
- definition of, 493
- downloading, 8
- projects
 - adding classes to, 127-130
 - creating, 15
 - debugging, 14-15
 - filename extensions, 12
 - FractionTest, 127-133
 - main.m, 13
 - project window, 10-11
 - running, 14
 - starting, 8-11
- static analyzer, 15

532 xib files

xib files, 462

XML (Extensible Markup Language)

definition of, 493

XML property lists, archiving with,
431-433

XYPPoint class, 162-165

zones, 493