



Stephen G. Kochan

Third Edition

Programming in Objective-C 2.0

A complete introduction to the Objective-C
language for Mac OS X and iPhone development

Developer's Library



Programming in Objective-C

Third 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

MySQL

Paul DuBois

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

Linux Kernel Development

Robert Love

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

Python Essential Reference

David Beazley

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

PostgreSQL

Korry Douglas

ISBN-13: 978-0-672-32756-2

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

Third 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, Third Edition

Copyright © 2011 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: 978-0-321-71139-7

ISBN-10: 0-321-71139-4

Library of Congress Cataloging-in-Publication Data:

Kochan, Stephen G.

Programming in objective-C / Stephen G. Kochan. – 3rd ed.

p. cm.

Includes index.

ISBN 978-0-321-71139-7 (pbk.)

1. Objective-C (Computer program language) 2. Object-oriented programming (Computer science) 3. Macintosh (Computer)-Programming.

I. Title.

QA76.64.K655 2011

005.1'17-dc23

2011015714

Printed in the United States of America

First Printing, June 2011

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

Development Editor

Michael Thurston

Managing Editor

Sandra Schroeder

Project Editor

Mandie Frank

Indexer

Larry Sweazy

Proofreader

Kathy Ruiz

Technical Editor

Michael Trent

Publishing Coordinator

Vanessa Evans

Designer

Gary Adair

Compositor

Mark Shirar



To Roy and Ve, two people whom I dearly miss



Contents at a Glance

1 Introduction **1**

Part 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 **69**

6 Making Decisions **91**

7 More on Classes **123**

8 Inheritance **149**

9 Polymorphism, Dynamic Typing,
and Dynamic Binding **179**

10 More on Variables and Data
Types **197**

11 Categories and Protocols **219**

12 The Preprocessor **233**

13 Underlying C Language
Features **247**

Part II The Foundation Framework

14 Introduction to the Foundation
Framework **305**

15 Numbers, Strings, and
Collections **309**

16 Working with Files **369**

17 Memory Management **397**

18 Copying Objects **417**

19 Archiving **429**

Part III Cocoa, Cocoa Touch, and the iOS SDK

20 Introduction to Cocoa and
Cocoa Touch **445**

21 Writing iOS Applications **449**

Appendices

A Glossary **481**

B Address Book Source Code **489**

Index **495**

Table of Contents

1 Introduction 1

What You Will Learn from This Book	2
How This Book Is Organized	3
Support	5
Acknowledgments	5

Part I The Objective-C 2.0 Language

2 Programming in Objective-C 7

Compiling and Running Programs	7
Using Xcode	8
Using Terminal	15
Explanation of Your First Program	17
Displaying the Values of Variables	21
Summary	23

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	32
Choosing Names	33
Instance Variables	35
Class and Instance Methods	35
The @implementation Section	37
The program Section	38
Accessing Instance Variables and Data	
Encapsulation	45
Summary	48

4 Data Types and Expressions 51

Data Types and Constants	51
Type int	51
Type float	52
Type char	52

Qualifiers: <code>long</code> , <code>long long</code> , <code>long 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	62
Assignment Operators	63
A Calculator Class	64

5 Program Looping 69

The <code>for</code> Statement	70
Keyboard Input	76
Nested <code>for</code> Loops	78
<code>for</code> Loop Variants	80
The <code>while</code> Statement	81
The <code>do</code> Statement	85
The <code>break</code> Statement	87
The <code>continue</code> Statement	87
Summary	88

6 Making Decisions 91

The <code>if</code> Statement	91
The <code>if-else</code> Construct	95
Compound Relational Tests	98
Nested <code>if</code> Statements	101
The <code>else if</code> Construct	102
The <code>switch</code> Statement	111
Boolean Variables	114
The Conditional Operator	118

7 More on Classes 123

Separate Interface and Implementation Files	123
Synthesized Accessor Methods	128
Accessing Properties Using the Dot Operator	129

Multiple Arguments to Methods	130
Methods Without Argument Names	132
Operations on Fractions	133
Local Variables	135
Method Arguments	136
The <code>static</code> Keyword	137
The <code>self</code> Keyword	140
Allocating and Returning Objects from Methods	141
Extending Class Definitions and the Interface File	146

8 Inheritance 149

It All Begins at the Root	149
Finding the Right Method	153
Extension Through Inheritance: Adding New Methods	154
A Point Class and Memory Allocation	157
The <code>@class</code> Directive	159
Classes Owning Their Objects	163
Overriding Methods	167
Which Method Is Selected?	169
Overriding the <code>dealloc</code> Method and the Keyword <code>super</code>	171
Extension Through Inheritance: Adding New Instance Variables	173
Abstract Classes	175

9 Polymorphism, Dynamic Typing, and Dynamic Binding 179

Polymorphism: Same Name, Different Class	179
Dynamic Binding and the <code>id</code> Type	182
Compile Time Versus Runtime Checking	184
The <code>id</code> Data Type and Static Typing	185
Argument and Return Types with Dynamic Typing	186
Asking Questions About Classes	187
Exception Handling Using <code>@try</code>	191

10 More on Variables and Data Types 197

Initializing Objects	197
Scope Revisited	200
Directives for Controlling Instance Variable Scope	200
External Variables	201
Static Variables	203
Enumerated Data Types	205
The <code>typedef</code> Statement	208
Data Type Conversions	209
Conversion Rules	210
Bit Operators	211
The Bitwise AND Operator	213
The Bitwise Inclusive-OR Operator	214
The Bitwise Exclusive-OR Operator	214
The Ones Complement Operator	215
The Left Shift Operator	216
The Right Shift Operator	217

11 Categories and Protocols 219

Categories	219
Some Notes About Categories	224
Protocols and Delegation	225
Delegation	228
Informal Protocols	228
Composite Objects	229

12 The Preprocessor 233

The <code>#define</code> Statement	233
More Advanced Types of Definitions	235
The <code>#import</code> Statement	240
Conditional Compilation	241
The <code>#ifdef</code> , <code>#endif</code> , <code>#else</code> , and <code>#ifndef</code> Statements	241
The <code>#if</code> and <code>#elif</code> Preprocessor Statements	243
The <code>#undef</code> Statement	244

13 Underlying C Language Features 247

Arrays 248	
Initializing Array Elements 250	
Character Arrays 251	
Multidimensional Arrays 252	
Functions 254	
Arguments and Local Variables 255	
Returning Function Results 257	
Functions, Methods, and Arrays 260	
Blocks 261	
Structures 265	
Initializing Structures 268	
Structures Within Structures 269	
Additional Details About Structures 271	
Don't Forget About Object-Oriented Programming! 273	
Pointers 274	
Pointers and Structures 277	
Pointers, Methods, and Functions 279	
Pointers and Arrays 280	
Operations on Pointers 290	
Pointers and Memory Addresses 291	
Unions 292	
They're Not Objects! 295	
Miscellaneous Language Features 295	
Compound Literals 295	
The <code>goto</code> Statement 296	
The <code>null</code> Statement 296	
The Comma Operator 297	
The <code>sizeof</code> Operator 297	
Command-Line Arguments 298	
How Things Work 300	
Fact #1: Instance Variables are Stored in Structures 300	
Fact #2: An Object Variable is Really a Pointer 301	

Fact #3: Methods are Functions, and Message Expressions are Function Calls	301
Fact #4: The <code>id</code> Type is a Generic Pointer Type	302

Part II The Foundation Framework

14 Introduction to the Foundation Framework 305

Foundation Documentation 305

15 Numbers, Strings, and Collections 309

Number Objects 309

 A Quick Look at the Autorelease Pool 311

String Objects 314

 More on the `NSLog` Function 314

 The `description` Method 315

 Mutable Versus Immutable Objects 316

 Mutable Strings 322

 Where Are All Those Objects Going? 326

Array Objects 328

 Making an Address Book 332

 Sorting Arrays 350

Dictionary Objects 356

 Enumerating a Dictionary 357

Set Objects 360

`NSIndexSet` 363

16 Working with Files 369

Managing Files and Directories:

`NSFileManager` 370

 Working with the `NSData` Class 374

 Working with Directories 376

 Enumerating the Contents of a Directory 378

Working with Paths: `NSPathUtilities.h` 380

 Common Methods for Working with Paths 383

 Copying Files and Using the `NSProcessInfo` Class 385

Basic File Operations: `NSFileHandle` 389

The `NSURL` Class 393

The `NSBundle` Class 394

17 Memory Management 397

- The Autorelease Pool 397
- Reference Counting 398
 - Reference Counting and Strings 401
 - Instance Variables 403
- An Autorelease Example 409
- Summary of Memory-Management Rules 410
- More on the Event Loop and Memory Allocation 411
- Finding Memory Leaks 413
- Garbage Collection 413

18 Copying Objects 417

- The `copy` and `mutableCopy` Methods 418
- Shallow Versus Deep Copying 420
- Implementing the `<NSCopying>` Protocol 422
- Copying Objects in Setter and Getter Methods 425

19 Archiving 429

- Archiving with XML Property Lists 429
- Archiving with `NSKeyedArchiver` 431
- Writing Encoding and Decoding Methods 433
- Using `NSData` to Create Custom Archives 440
- Using the Archiver to Copy Objects 443

Part III Cocoa, Cocoa Touch, and the iOS SDK**20 Introduction to Cocoa and Cocoa Touch 445**

- Framework Layers 445
- Cocoa Touch 446

21 Writing iOS Applications 449

- The iOS SDK 449
- Your First iPhone Application 449
 - Creating a New iPhone Application Project 452
 - Entering Your Code 455
 - Designing the Interface 458

An iPhone Fraction Calculator	464
Starting the New <code>Fraction_Calculator</code>	
Project	465
Defining the View Controller	468
The <code>Fraction</code> Class	472
A <code>Calculator</code> Class That Deals with Fractions	475
Designing the UI	477
Summary	478

Appendices

A Glossary 481

B Address Book Source Code 489

Index 495

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 Reviewer

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 B.A. in Mathematics from University of California Berkeley. When not working, Wendy is pursuing her 4th Dan Tae Kwon Do black belt.

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 Web site, 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.

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 Mark 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.

E-mail: 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

3

Classes, Objects, and Methods

In this chapter, you'll learn about some key concepts in object-oriented programming and start working with classes in Objective-C. You'll 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. Additionally, 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:

[myWindow erase];	Clear the window
theArea = [myRect area];	Calculate the area of the rectangle
[userText spellCheck];	Spell-check some text
[deskCalculator clearEntry];	Clear the last entry
[favoritePlaylist showSongs];	Show the songs in a playlist of favorites
[phoneNumber dial];	Dial a phone number
[myTable reloadData];	Show the updated table's data
n = [aTouch tapCount];	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 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[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int numerator = 1;
    int denominator = 3;
    NSLog (@"The fraction is %i/%i", numerator, denominator);

    [pool drain];
    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 autorelease pool is created, 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
{
    int numerator;
    int denominator;
}

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

@end

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

@implementation Fraction
-(void) print
{
    NSLog(@"%@", numerator, denominator);
}

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

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

@end

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

```

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    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];
    [myFraction release];

    [pool drain];
    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, its data components, and its methods, whereas the @implementation section contains the actual code that implements these methods. Finally, the program section contains the program code to carry out the intended purpose of the program.

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 do a few things. First, you have to tell the Objective-C compiler where the class came from. That is, you have to name its *parent* class. Second, you have to specify what type of data is to be stored in the objects of this

class. That is, you have to describe the data that members of the class will contain. These members are called the *instance variables*. Finally, you need to define the type of operations, or *methods*, that can be used when working with objects from this class. This is all done in a special section of the program called the `@interface` section. The general format of this section looks like this:

```
@interface NewClassName: ParentClassName
{
    memberDeclarations;
}

methodDeclarations;
@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 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 (upper- or lowercase), underscores, or the digits 0–9. The following is a list of valid names:

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

On the other hand, 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 can't 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 upper- 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, on the other hand, 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.
- current_entry—Some programmers use underscores as word separators.
- 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
{
    int numerator;
    int denominator;
}

-(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`.

Instance Variables

The *memberDeclarations* section specifies what types of data are stored in a `Fraction`, along with the names of those data types. As you can see, this section is enclosed inside its own set of curly braces. For your `Fraction` class, these declarations say that a `Fraction` object has two integer members, called `numerator` and `denominator`:

```
int numerator;
int denominator;
```

The members declared in this section are known as the instance variables. As you'll see, 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.

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'll 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.

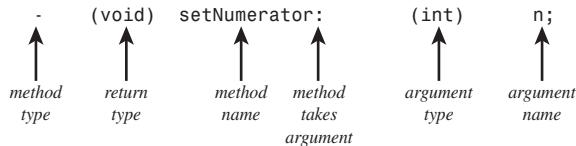


Figure 3.1 Declaring a method

When a method takes an argument, you also append a colon to the method name when referring to the method. Therefore, `setNumerator:` and `setDenominator:` is the correct way to identify these two methods, each of which takes a single argument. Also, identifying the `print` method without a trailing colon indicates that this method does not take any arguments. In Chapter 7, “More on Classes,” you’ll see how methods that take more than one argument are identified.

The @implementation Section

As noted, the `@implementation` section contains the actual code for the methods you declared in the `@interface` section. Just as a point of terminology, you say that you declare the methods in the `@interface` section and that you *define* them (that is, give the actual code) in the `@implementation` section.

The general format for the `@implementation` section is as follows:

```

@implementation NewClassName
  methodDefinitions;
@end
  
```

`NewClassName` is the same name that was used for the class in the `@interface` section. You can use the trailing colon followed by the parent class name, as we did in the `@interface` section:

```
@implementation Fraction: NSObject
```

However, this is optional and typically not done.

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.

Consider the `@implementation` section from Program 3.2:

```

//---- @implementation section ----
@implementation Fraction
-(void) print
{
  NSLog(@"%@", 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 we've 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[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    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];

[myFraction release];
[pool drain];

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 below.

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, “Inheritance” 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 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];
```

We use this coding style often throughout the remainder of this book, so it's important that you understand it. You've seen in every program up to this point with the allocation of the autorelease pool:

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

Here an `alloc` message is sent to the `NSAutoreleasePool` class requesting that a new instance be created. The `init` message then is sent to the newly created object to get it initialized.

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.

The message in the program releases or frees the memory that was used for the `Fraction` object:

```
[myFraction release];
```

This is a critical part of good programming style. Whenever you create a new object, you are asking for memory to be allocated for that object. Also, when you're done with the object, you are responsible for releasing the memory it uses. Although it's true that the memory will be released when your program terminates anyway, after you start developing more sophisticated applications, you can end up working with hundreds (or thousands) of objects that consume a lot of memory. Waiting for the program to terminate for the memory to be released is wasteful of memory, can slow your program's execution, and is not good programming style. So get into the habit of releasing memory when you can right now.

The Apple runtime system provides a mechanism known as *garbage collection* that facilitates automatic cleanup of memory. However, it's best to learn how to manage your memory usage yourself instead of relying on this automated mechanism. In fact, you can't rely on garbage collection when programming for certain platforms on which garbage collection is not supported, such as the iPhone or iPad. For that reason, we don't talk about garbage collection until much later in this book.

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.

Note

Note that Xcode inserts a space after the * when it generates the first line in `main` that creates an `NSAutoreleasepool` object. It's not needed, so we won't do that for our objects, as most programmers don't either.

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. This is depicted in Figure 3.2.

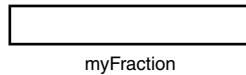


Figure 3.2 Declaring
`Fraction *myFraction;`

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:

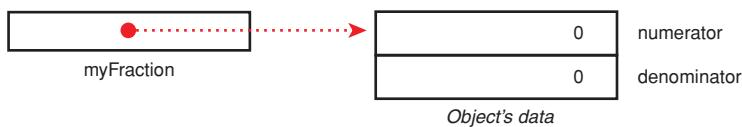


Figure 3.3 Relationship between `myFraction`
and its data

Note

There's some more data stored with the object than 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.

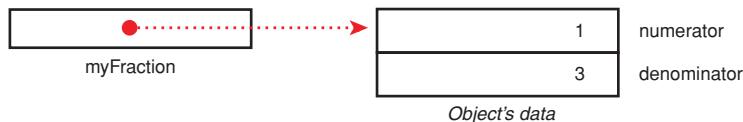


Figure 3.4 Setting the fraction's numerator and denominator

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
{
    int numerator;
    int denominator;
}

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

@end

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

@implementation Fraction
-(void) print
{
    NSLog(@"%@", numerator, denominator);
}

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

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

```

@end

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

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    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];

    [frac1 release];
    [frac2 release];

    [pool drain];
    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.

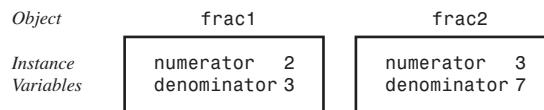


Figure 3.5 Unique instance variables

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'll need to write two new methods. For example, you'll 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
{
    int numerator;
    int denominator;
}

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

@end

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

@implementation Fraction
-(void) print
{
    NSLog(@"%@", 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[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    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]);
    [myFraction release];
    [pool drain];

    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 will be sent to the `Fraction` object `myFraction`. In that method, the code will return 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 will be accessed and returned. That returned integer value is then passed along to `NSLog` to be displayed.

For the second message call, the `denominator` method will be 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.

Note

Soon you'll learn a convenient feature of Objective-C 2.0 that allows for the automatic creation of getter and setter methods.

Make sure 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. On the other hand, 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. In order 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 there's also a method called `new` that 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 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?

Int	playNextSong	6_05
_calloc	Xx	alphaBetaRoutine
clearScreen	_1312	z
ReInitialize	_	A\$

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 owned a boat and a motorcycle in addition to a car. List the actions you would perform with each of these. Do you have any overlap between these actions?
5. Based on question 4, imagine that you had a class called `Vehicle` and an object called `myVehicle` that could be either `Car`, `Motorcycle`, or `Boat`. Imagine that you wrote 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.

This page intentionally left blank

Index

SYMBOLS

== (double equals sign), 98
; (semicolon), 19
_ (underscore), 33
\'0\' (null characters), 251
**** (backslash), 20
!= (not equal to) operator, 72
#define statement, 233-239
#elif statements, 243-244
#else statements, 241-243
#endif statements, 241-243
#ifdef statements, 241-243
#ifndef statements, 241-243
#if statements, 243-244
#import statements, 234, 240-241
#include statements, 234
#undef statements, 244
% (modulus operator), 60-61
& operator, 239
& (address) operator, 274
***** (asterisk), 42, 55,
++ (increment operators), 282, 287
+ (plus sign), 55
+ (unary plus operator), 59
+= (plus equals) operator, 63
++ (increment) operators, 287
, (commas), 268
— (decrement operators), 282
- (minus sign), 35, 55
/ (slash), 18, 55
: (colons), 118, 188
; (semicolons), 236
< (less than) operator, 72
<= (less than or equal to) operator, 72
= (equal sign), 262
== (equal to) operator, 72
> (greater than) operator, 72
>= (greater than or equal to) operator, 72
? (question marks), 118
@catch block, 193
@class directive, 159-163
@finally block, 193
@implementation section, 37-38
@interface directive, 423
@interface section, 32-37
@optional directive, 226
@package directive, 200
@private directive, 200
@property directive, 128
@protected directive, 200
@protocol directive, 225
@public directive, 200
@sign, 19
@throw directive, 193
@try, handling exceptions, 191-193
\ (curly braces), 256
~(tilde), 370, 383

A

- absoluteValue function, 259**
- abstract classes, 175**
- accessing**
 - documentation, 306
 - files, 369
 - instance variables, 45–48
 - local variables, 137
 - MAC OS X reference library, 308
 - properties, 130
- accessor methods. *See also* getter methods; setter methods**
 - event loops, 412
 - synthesized, 128–130
- actions, 449**
 - adding, 462
 - in objects, 28
- addCard: method, 340**
- adding**
 - actions, 462
 - categories, 219
 - classes, 124
 - description methods, 315
 - else clauses, 101
 - extensions, 220
 - fractions, 30
 - instance variables, 173–175
 - labels, 460
 - methods, 154–167, 219
- add: method, 133, 140, 141, 182, 220**
 - testing, 142
- AddressBook class, 226, 340-342**
 - archiving, 433
- AddressCard class, 332-337**
 - synthesized methods, 337–339
- addresscard.h interface file, 434**
- addresses**
 - books, formatting, 332–337
 - memory pointers, 291–292
- addSubview: method, 401**
- advanced type definitions, 235**
- algorithms**
 - common divisor, 257
 - while statements, 82
- allKeys method, 358**
- allocating**
 - memory, 19, 39, 423
 - event loops, 411–413
 - inheritance, 157–158
 - objects, 141–146, 413, 424
- allocF class, 203**
- alloc message, 39**
- alloc method, 153, 204**
- alternative names, assigning data types, 208**
- analyzing**
 - characters, 104
 - memory leaks, 413
- AND operators, 213, 239**
 - compound relational tests, 98
- AppKit, 446**
- Apple Computer, 1**
- applications**
 - Fraction_Calculator project, 465–467
 - hierarchies, 445
- iOS**
 - SDK (Software Development Kit), 449
 - writing, 449
- iPhone**
 - fraction calculators, 464–477
 - templates, 452
 - writing, 449–464

Terminal, 15-17
terminating, 411
Xcode, 8. *See also* Xcode

Applications folder, 15

Applications Services layer, 446

applying
directories, 376
fractions, 30-32
NSData class, 374
paths, 383
Xcode, 8-15

archiving, 429
decoding methods, 433-440
encoding methods, 433-440
NSData, 440-443
NSKeyedArchiver, 431-433
objects, copying, 443-444
XML propertylists, 429-431

arguments
command-line, 298
with dynamic typing, 186-187
functions, 255
instance variables, referring to, 134
methods, 36-37, 136-137
multiple, 130-135
names, methods without, 133
zone, 423

arithmetic
expressions, 55-63
integers, 58-59
operators, combining assignment operators, 63

arrays, 248-254, 309
characters, 251
copying, 421
elements, initializing, 250
indexes, 330
methods, 412
multidimensional, 252-254

objects, 328-356
address books, 332-337
references, 419
saving, 431
sorting, 350-355
passing, 260-261
pointers, 280-290
sorting, 350-355
two-dimensional, 252

arraySum function, 251

arrayWithObjects: method, 330

assigning
alternative names to data types, 208
blocks, 262
constants, 235
integer values to enumerated data types, 208

assignment operators, 63-64

assignment statements, 417

asterisk (*), 42, 55,

atomically parameter, 430

AT&T Bell Laboratories, 1

attributes
labels, modifying, 461
nonatomic, 337

attributesOfItemAtPath: method, 371

automatic local variables, 256

autorelease pools, 22, 30, 142, 411, 425
examples of, 409-410
memory, 397-398
NSAutoreleasePool, 306
number objects, 311-314

availableData method, 389

B

backslash (), 20

backup files, 430

basic data types, 54-55

binary notation conversions, 212

binding
 dynamic, 54, 179
 id type, 182–184

bit fields, 271–273

bit operators, 211–218

blocks,
 printFoo, 264
 Using, sorting, 352–355

BOOL data types, 117

Boolean variables, 114–118

brace pairs, 253

break statements, 87, 111, 113

buffers, configuring, 374

buttons, adding actions, 462

bytes, files, 392. See also files

C

Caches directory, 385

calculateTriangularNumber function, 256

calculating
 arraySum function, 283
 fractions, iPhone applications, 464–477
 triangle numbers, 77

Calculator class, 64–66, 106, 475–477

Calculator.h interface file, 475

Calculator.m implementation file, 240

calls, functions, 301

capitalization, 235

Cartesian coordinate systems, 157

categories, 219–224
 MathOps, 220–221

centralizing definitions, 241

CGSizeMake function, 270

changeCurrentDirectoryPath: method, 438

characters
 @ sign, 19
 analyzing, 104

arrays, 251
 pointers, 276
 slash (/), 18
 strings
 immutable, 316
 pointers to, 284–286
 tildes (~), 370, 383
 underscore (_), 33
 unichar, 314

char data types, 52–54

checking
 compile time, 184–185
 runtime, 184–185

child classes, 147. See also classes

Circle method, 234

C language,

classes, 27, 187–191
 @interface section, 35–37
 abstract, 175
 adding, 124
 AddressBook, 226, 340–342, 433
 AddressCard, 332–337–339
 allocF, 203
 Calculator, 64–66, 106, 475–477
 Complex, 179, 183
 copying, 422
 declaring, 124
 defining, 33, 229
 extending, 146, 154
 Foundation framework, 309
 Fraction, 31, 66, 128, 183,
 422, 472–475
 GraphicObject, 226
 initializing, 197–199
 instances, 28. *See also* instances
 local variables, 135–140
 members, releasing, 171

methods
multiple arguments to, 130–135
syntax for applying to, 28
without argument names, 133

NSArray, 198, 329, 354
NSBundle, 370, 394–395
NSCountedSet, 362
NSData, applying, 374–375
NSDictionary methods, 359
NSIndexSet, 363–366
NSMutableArray, 329, 355
NSMutableDictionary methods, 359
NSMutableSet methods, 363
NSMutableString, 326, 328
NSObject, 147
NSProcessInfo, copying files, 385
NSSet methods, 363
NSString methods, 326–328
NSURL, 370, 374–394
NSNumber, 355–356
objects, owning, 163–167
operations on fractions, 133–135
overview of, 123
polymorphism. *See* polymorphism
Printer, 200
properties, accessing, 130
Rectangle, 155, 185, 189
root, 147–154
self keyword, 140–141
separate files, 123–128
Square, 155, 189
defining, 229
testing, 157
synthesized accessor methods, 128–130
UIKit, 229
CGPoint, 159, 190
implementation files, 160
interface files, 160
memory, 164

class method, 191

clauses, else, 110
adding, 101
if statements, 103

cleanup, memory, 41

clickDigit: method, 472

clickEquals method, 472

closeFile method, 390

Cocoa, 411
framework layers, 445–446
overview of, 445

Cocoa Touch, 445–447

code, 7. *See also* programming
interfaces, viewing, 462
iOS, entering, 455–458
looping, 69–70
program section, 38–45

collections, 309. *See also* sets

colons (:), 118, 188

colors
windows, modifying, 459

Xcode, 13

command-line arguments, 298–300

commands
gcc, 16
Terminal, 15

comma operators, 297

commas (,), 268

comments, 12
reducing need for, 237
statements, 18

common divisors
algorithms, 257
finding, 84

compare: method, 313

compareNames: method, 351

comparing pointer variables, 282–290

- compilers**
 - enumeration identifiers, 206
 - errors, 166
 - static typing, 185
- compile time, checking, 184-185**
- compiling**
 - conditional compilation, 241-244
 - preprocessors, 233. *See also* preprocessors
 - programs, 7-17
- Complex class, 179, 183**
- composite objects, 229-230**
- compound literals, 295**
- compound relational tests, 98-101**
- conditional compilation, 241-244**
- conditional operators, 118-119**
- configuring**
 - buffers, 374
 - subclasses, 175
- conformsToProtocol: method, 227**
- connecting outlet variables, 462**
- Console Interfaces, 332**
- constants**
 - assigning, 235
 - characters, 52
 - data types and, 51-55
 - floating-point, 52
 - hexadecimal floating, 55
 - metric conversions, 240
- constructs**
 - else if, 102-111
 - if-else, 95-98
- contentsAtPath: method, 371, 375**
- contentsEqualAtPath: method, 371**
- contentsOfDirectoryAtPath: method, 376**
- continue statements, 87-88**
- controllers**
 - defining, 468-472
 - views, 465
- conversions**
 - binary notation, 212
 - data types, 209
 - floating-point, 61-62
 - hexadecimal notation, 212
 - integers, 61-62
 - metric, 240
 - objects, 355
 - rules, 210-211
 - type cast operators, 62-63
- convertToString method, 472**
- coordinates, 158**
- copying**
 - deep, 420-422, 443
 - files, 385-389
 - getter/setter methods, 425-427
 - objects, 417, 443-444
 - shallow, 420-422
- copyItemAtPath: method, 371**
- copy method, 418-424**
- copyString function, 289**
- copyWithZone: method, 423**
- Core Data frameworks, 445**
- counting references, 398-409**
 - instance variables, 403-409
 - strings, 401-403
- Cox, Brad J., 1**
- C programming language**
 - arrays, 248-254
 - characters, 251-252
 - initializing elements, 250-251
 - multidimensional, 252-254
 - passing, 260-261
 - blocks, 261

command-line arguments, 298
comma operators, 297
compound literals, 295
features, 247, 295
functions, 254
 arguments/local variables, 255
 declaring return/argument types, 258–260
 returning results, 257
goto statements, 296
null statements, 296
Objective-C connections (how things work), 300–302
objects, 295
pointers, 274
 arrays, 280
 functions/methods, 279
 memory addresses, 291
 operations on, 290
 structures, 277
sizeof operator, 297
structures, 265
 defining, 271
 initializing, 268
 OOP (object-oriented programming), 273
 within structures, 269
unions, 292

createDirectoryAtPath: method, 376

createFileAtPath: method, 371, 375

curly braces (), 256

current directories, 378. *See also* **directories**

currentDirectory method, 382

currentDirectoryPath method, 376

customizing

- archiving, 440–443
- preprocessors, 233. *See also* **preprocessors**

D

dataArray, 418, 443

data encapsulation, 45–48

data types, 51, 197

- alternative names, assigning, 208
- basic, 54–55
- BOOL, 117
- char, 52–54
- constants, 51–55
- conversions, 209
- enumeration, 205–208
- float, 52
- id, 54–55, 182–187. *See also* **pointers**
- int, 22, 51–52
- storage, 33

date structures, 266

dealloc method, 336, 408

- overriding, 171–172

debugging

- Cocoa, 445
- conditional compilation, 243
- description method, 316
- Terminal, 17
- Xcode, 13

decimal notation conversions, 212

decisions, making, 91

declaring

- classes, 124
- functions, 260
- immutable string objects, 318
- methods, 36
- unions, 292

decoding methods, 433–440

decrement operators (—), 282

deep copying, 420–422, 443

defaultManager message, 371

defining

- advanced types, 235
- arrays. *See arrays*
- classes, 33, 146, 229
- enumerated data types, 206
- external variables, 201
- fractions, 31
- global variables, 202
- methods, 35
- multiple arguments, 131
- new type names, 209
- pointers, 274
- preprocessor identifiers, 242
- protocols, 225, 227
- structures, 266
- unions, 293
- view controllers, 468–472

delegates, 449

delegation, protocols and, 225–229

deleteCharactersInRange: method, 325

denominator instance variable, 128

description method, 315–316, 362

design

- iPhone interfaces, 458–464
- UIs (user interfaces), 477

Developer folder, Xcode, 8

dictionaries, 309

- copying, 422
- enumeration, 357–359
- objects, 356–359, 431

dictionaryWithObjectsAndKeys: method, 357

digits, int data types, 51

directives

- `@ class`, 159–163
- `@interface`, 423
- `@optional`, 226

@package, 200

@private, 200

@property, 128

@protected, 200

@protocol, 225

@public, 200

for controlling instances, 200–201

directories

- applying, 376
- Caches, 385
- enumeration, 378
- files, 369. *See also files*
- iOS, 385
- managing, 370
- methods, 376

dividing

- fractions, 30
- integers, 268

division, if-else constructs, 95–98

division-by-zero problem, 110

documentation

- Foundation framework, 305–308
- Xcode, 306

do statements, 85–87

dot (.) operator, 266

- properties, accessing, 130

double equals sign (==), 98

double quotes, 127, 240, 314

drain messages, 398

Drawing3D protocol, 227

Drawing protocol, 226

draw method, 184

duplication of functionality, 446

dynamic binding, 54, 179

- id data type, 182–184

dynamic typing, 179

arguments, 186–187
methods, 187
with return types, 186–187

E

editing files, 12

Xcode, 306

edit windows, 10**elements**

arrays, 250. *See also* arrays
copying, 421
initializing, 250
references, 284
structures, 265. *See also* structures

else clauses, 110

adding, 101
if statements, adding, 103

else if constructs, 102-111**enabling garbage collection, 414****encapsulation, data, 45-48****encodeObject: method, 435****encodeWithCoder: method, 433****encoding**

methods, 433–440
objects, 441

entering code, iOS, 455-458**enumeration**

data types, 205–208
dictionaries, 357–359
directories, 378
fast, 330, 342–344

enumeratorAtPath: method, 376**enum flags, 205****environment method, 386****equality tests, 236****equal sign (=), 262****equal to (==) operator, 72****errors**

compilers, 166

Xcode, 13

Euclid, 82**evaluating**

expressions, 210
multiple operations, 57

even numbers, if-else constructs, 95-98**event loops, allocating memory, 411-413****exceptions, handling, 191-193****exclusive OR operators, 214****executing**

break statements, 87
continue statements, 88
do statements, 85
loops, Boolean variables, 115
methods, 182
programs, terminating, 192
for statements, 73

expressions, 51

arithmetic, 55–63
compound relational, 99
conditional operators, 118
evaluating, 210
messages, 40,

extending class definitions, 146**extending classes, 154**

instance variables, adding, 173–175

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

adding, 220
filenames, 12

external variables, 201-202

F**FALSE**, 233**fast enumeration**, 330, 342-344**features, C programming language**, 247, 295**Fibonacci numbers**, 249**fields, bit**, 271-273**fileExistsAtPath: method**, 371**fileHandleForReadingAtPath: method**, 371**fileHandleForUpdatingAtPath: method**, 389**fileHandleForWritingAtPath: method**, 389**filename extensions**, 12**files**, 369

addresscard.h interface, 434

backup, 430

copying, 385-389

editing, 12

formatting, 124

Foundation.h, 18, 309

glossary, 430

implementation, 123, 180

AddressBook.m, 340

AddressCard.m, 333, 337

Calculator.m, 240

Fraction_CalculatorApp

Delegate.m, 467

Fraction_CalculatorView

Controller.m, 469

Fraction.m, 126, 131, 138, 473

iPhone_1AppDelegate.m, 457

Rectangle.m, 154, 161

Square.me, 156

XYPoint class, 160

interfaces, 123, 179

AddressBook.h, 340, 347

AddressCard.h, 333

Calculator.h, 475

extending, 146

Fraction_CalculatorApp

Delegate.h, 467

Fraction_CalculatorView

Controller.h, 468

Fraction.h, 124, 131, 138, 473**Rectangle.h**, 161**Square.h**, 156**XYPoint class**, 160

managing, 370

NSFileHandle, 389

NSPathUtilities.h, 380

objects, saving, 431

opening, 369

positioning, 392

separate, classes, 123-128

temporary storage areas, 374

troubleshooting, 373

Xcode, 306

XML, 430

flags, 115

enum, 205

tests, 116

float data types, 52**floating-point**

conversions, 61-62

numbers, 175

folders

Applications, 15

Developer, Xcode, 8

naming, 8

for loops, 73, 145

arrays, 330

autorelease pools, 398

list method, 342

nested, 78-80

variants, 80-81

for statements, 70-81
 executing, 73
 keyboard input, 76-78
 nested for loops, 78-80
 programs, 73

formatting
 @implementation section, 37
 address books, 332-337
 archives, customizing, 440-443
 comments, 18
 files, 124
 iPhone interfaces, 458-464
 mutable copies of dataArray, 419
 pathnames, 382
 for statements, 71
 subclasses, 175
 UIs (user interfaces), 477

forwardInvocation: method, 189

Foundation framework, 305
 abstract classes, 175
 autorelease pools, 397-398
 copy method, 418-420
 documentation, 305-308
 files, accessing, 18, 369
 memory, 397. *See also* memory
 mutablecopy method, 418-420
 NSCountedSet class, 362
 objects, 309
 pointers, 291
 protocols, 225
 Terminal window, 16

Foundation.h file, 18, 309

Fraction_CalculatorAppDelegate.h interface file, 467

Fraction_CalculatorAppDelegate.m implementation file, 467

Fraction_Calculator project, 465-467

Fraction_CalculatorViewController.h interface file, 468

Fraction_CalculatorViewController.m implementation file, 469

Fraction class, 31, 66, 128, 183, 422, 472-475

Fraction.h interface file, 124, 131, 138, 473

Fraction.m implementation file, 126, 131, 138, 473

fractions, 30-32
 calculators, 464-477
 multiplying, 465
 operations on, 133-135
 reduce method, 135

FractionTest.m file, 123, 127, 132, 134, 139, 144

frameworks
 Cocoa, layers, 445-446
 Cocoa Touch, 447
 Core Data, 445
 Foundation, 224-225

FSF (Free Software Foundation), 1

functionality, duplication of, 446

functions
 absoluteValue, 259
 arguments, 255
 arrays, passing, 260-261
 arraySum, 251
 calculateTriangularNumber, 256
 calls, 301
 CGSizeMake, 270
 copyString, 289
 declaring, 260
 Foundation framework, 309
 gcd, 302
 invoking, 261
 methods, 301

multidimensional arrays, 252
NSFullUserName, 384
NSHomeDirectory, 384
NSHomeDirectoryForUser, 384
NSLog, 314-315
NSSearchPathForDirectoriesInDomains, 384
NSTemporaryDirectory, 382
NSUserName, 384
numberOfDays, 395
paths, 384
results, returning, 257
returning, 258
sign, implementing, 103
squareRoot, 260

G

garbage collection, 41, 413-414
gcc command, 16
gcd function, 302
generating
 objects, 188
 prime numbers, 114
 selectors, 188
getter methods, 48, 203
 objects, copying, 425-427
globallyUniqueString method, 386
global variables, defining, 202
glossary files, 430
gMoveNumber, 201
GNU General Public License, 1
goto statements, 296
graphical user interfaces. See **GUIs**
GraphicObject class, 226
greater than (>) operator, 72
greater than or equal to (>=) operator, 72

grouping elements, structures, 265.
See also structures
GUIs (graphical user interfaces), 3, 445

H

handling exceptions, 191-193
hard-coding pathnames, 370
headers
 Foundation framework, 309
 NSObjCRuntime.h, 244
hexadecimal floating constants, 55
hexadecimal notation conversions, 212
hierarchies
 applications, 445
 inheritance, 191
home directories, 370. *See also* **directories**
hostName method, 386

I-J

id data types, 54-55, 182-187.
See also **pointers**
identifiers
 enumeration, 206
 preprocessors, defining, 242
if-else constructs, 95-98
if statements, 91-111
 compound relational tests, 98-101
 else if constructs, 102-111
 if-else constructs, 95-98
 nested, 101-102
immutable character strings, 316
immutable objects, 316-322
immutable storage areas, 374
implementation
 categories, 223
 files, 123, 180
 AddressBook.m, 340
 AddressCard.m, 333, 337

Calculator.m, 240
Fraction_CalculatorApp
 Delegate.m, 467
Fraction_CalculatorView
 Controller.m, 469
Fraction.m, 126, 131, 138, 473
iPhone_1AppDelegate.m, 457
Rectangle.m, 154, 161
Square.me, 156
XYPoint class, 160
NSCopying protocol, 422
protocols, 225
sign functions, 103
importing Foundation header files, 309
inclusive OR operators, 214
increment (++) operators, 282, 287
indexes,
 arrays, 330
 strings, 325
indexesOfObhectsPassingTest: method, 365
indirection (*) operators, 274. *See also pointers*
inDirectory: parameter, 395
informal protocols, 228-229
inheritance, 149, 417
 abstract classes, 175
 hierarchies, 191
 instance variables, adding, 173-175
 memory, allocating, 157-158
 methods
 adding, 154-167
 overriding, 167-172
 overview of, 149-154
initialization
 array elements, 250
 classes, 197-199
 comments, 18
 immutable string objects, 318
 methods, 198
 structures, 268
 two-dimensional arrays, 252
 two-step allocation approach, 48
initialize method, 205
init method, 39, 153
initVar method, 150
initWithCoder: method, 433
initWith:: method, 198
input, for statements, 76-78
inserting. *See also adding*
 comment statements, 18
 loops, 78
instances
 @interface section, 35-37
 directives for controlling, 200-201
 methods and, 28-30
 objects, 197
 variables
 @interface section, 35
 accessing, 45-48
 adding, 173-175
 counting references, 403-409
 inheritance, 151
 storage, 300
 unique, 44
int data types, 22, 51-52
integers, 51. *See also int data types*
 abstract classes, 175
 arithmetic, 58-59
 conversions, 61-62
 dividing, 268
 Euclid, 82
 NSUInteger, 365
 pointers, 278. *See also pointers*
 variables, 22

- Interface Builder, 449**
- MainWindow.xib, 458
- interfaces**
- addresscard.h interface files, 434
 - categories, 223
 - Console Interfaces, 332
 - design, 477
 - files, 123, 179
 - AddressBook.h, 340, 347
 - AddressCard.h, 333
 - Calculator.h, 475
 - extending, 146
 - Fraction_CalculatorApp Delegate.h, 467
 - Fraction_Calculator ViewController.h, 468
 - Fraction.h, 124, 131, 138, 473
 - Rectangle.h, 161
 - Square.h, 156
 - XYPoint class, 160
 - GUIs (graphical user interfaces), 3, 445
 - iPhones, designing, 458–464
 - source code, viewing, 462
- intersect: method, 362**
- INTOPBJ macro, 362**
- intWithName: method, 341**
- invoking functions,**
- iOS, 411**
- applications, writing, 449
 - code, entering, 455–458
 - directories, 385
 - Fraction_Calculator project, 465–467
 - interfaces, formatting, 458–464
 - iPhone applications
 - fraction calculators, 464–477
 - writing, 449–464
 - SDK (Software Development Kit), 449
- iPads, Cocoa Touch, 446**
- iPhone_1AppDelegate.m implementation file, 457**
- iPhones, 2**
- applications
 - templates, 452
 - writing, 449–464
 - Cocoa Touch, 446
 - fraction calculators, 464–477
 - interfaces, designing, 458–464
- iPods, 2**
- Cocoa Touch, 446
- isEqualToString: method, 313**
- islower routine, 106**
- isPrime variable, 115**
- isReadableFileAtPath: method, 371**
- isupper routine, 106**
- isWritableFileAtPath: method, 371**
-
- K**
-
- keyboard input, for statements, 76–78**
- keyed archives, 431. *See also* archiving**
- keying operations, 465**
- key-object pairs, 356**
- keys, dictionaries, 357. *See also* dictionaries**
- keywords**
- overriding, 171–172
 - self, 140–141
 - static, 137–140
 - struct, 293
-
- L**
-
- labels**
- adding, 460
 - modifying, 461

lastPathComponent method, 282

layers, Cocoa, 445-446

Leaks, memory, 413

leap years, 237

left shift operators, 216-217

length method, 318

less than (<) operator, 72

less than or equal to (≤) operator, 72

letters, lowercase, 239

libraries

- references, accessing MAC OS X, 308
- Standard Library, 291

lists

- method, 342
- protocols, 225. *See also* protocols
- XML propertylists, 429-431

literals, compound, 295

local static variables, 137

local variables, 135-140

- automatic, 256
- functions, 255

locations, folders, 8

logical negation operator, 117

long qualifiers, 53-54

lookupAll: method, 365

lookup functions, 291

lookup: method, 344-346

loops

- for, 73, 145
- arrays, 330
- autorelease pools, 398
- list method, 342
- nested, 78-80
- variants, 80-81

Boolean variables, 115

break statements, 87

continue statements, 87-88

dictionaries, enumerating, 358

do statements, 85-87

events, allocating memory, 411-413

inserting, 78

programs, 69-70

while, 326,

while statements, 81-85

lowercase letters, 17, 239

M

MAC OS X

- Cocoa Touch, 446-447
- reference library, accessing, 308
- support, 445

macros, 238

- INTOPBJ, 362
- MAX, 239
- SQUARE, 238

mainBundle method, 396

main function, 254

main.m program, 10, 180

main routines, 19

Main Test Program, FractionTest.m, 127, 132-134, 139

MainWindow.xib, 458

making decisions, 91

management

- files, 370-380
- memory, 397. *See also* memory
- NSFileManager, 369-380
- rules for memory, 410-411

MathOps category, 220

- programs, 221

matrices, 252

MAX macro, 239

members

- releasing, 171
- unions, 294. *See also* unions

memberships, 191

memory

- add: method, 142
- addresses, pointers, 291
- allocating, 19, 39, 423
 - event loops, 411–413
 - inheritance, 157–158
 - autorelease pools, 311, 397–398.
 - See also* autorelease pools
 - buffers, reading files, 369
 - cleanup, 41
 - dataArray, 418
 - garbage collection, 413–414.
 - See also* garbage collection
 - Leaks, 413
 - management, 397
 - references, counting, 398–409
 - rules, 410–411
 - XYPoint class, 164

menus, New File (Xcode), 124

messages

- alloc, 39
- compiler errors, 166
- defaultManager, 371
- drain, 398
- expressions, 40
- release, 398, 410
- retainCount, 399
- skipDescendants, 378
- uppercaseString, sending, 319

methods, 27

- @implementation section, 37–38
- @interface section, 35–37
- accessor
 - event loops, 412
 - synthesized, 128–130
- add:, 133, 140–142, 182, 220
- addCard:, 340
- adding, 219
- AddressCard class, synthesized, 337–339
- addSubview:, 401
- allKeys, 358
- alloc, 153, 204
- arguments, 36–37, 136–137
- arrays, 412
- arrayWithObjects:, 330
- attributesOfItemAtPath:, 371
- autorelease pools, 311
- availableData, 389
- changeCurrentDirectoryPath:, 438
- Circle, 234
- classes, 191
 - multiple arguments to, 130–135
 - without argument names, 133
- clickDigit:, 472
- clickEquals, 472
- closeFile, 390
- compare:, 313
- compareNames:, 351
- conformsToProtocol:, 227
- contentsAtPath:, 371
- contentsEqualAtPath:, 371
- contentsOfDirectoryAtPath:, 376
- convertToString, 472
- copy, 418–424
- copyItemAtPath:, 371
- copyWithZone:, 423
- createDirectoryAtPath:, 376
- createFileAtPath:, 371
- currentDirectory, 382
- currentDirectoryPath, 376
- dealloc, 171–172, 336, 408
- declaring, 36

decoding, 433–440
defining, 35
deleteCharactersInRange:, 325
description, 315–316, 362
dictionaryWithObjectsAndKeys:, 357
directories, 376
draw, 184
dynamic typing, 187
encodeObject:, 435
encodeWithCoder:, 433
encoding, 433–440
enumeratorAtPath:, 376
environment, 386
executing, 182
fileExistsAtPath:, 371
fileHandleForReadingAtPath:, 371
fileHandleForUpdatingAtPath:, 389
fileHandleForWritingAtPath:, 389
forwardInvocation:, 189
Foundation framework, 309
functions, 301
getter, 48, 203
globallyUniqueString, 386
hostName, 386
indexesOfObhectsPassingTest:, 365
inheritance, 151, 154–167
init, 39, 153
initialize, 205
iVar, 150
initWith::, 198
initWithCoder:, 433
instances and, 28–30
intersect:, 362
intWithName:, 341
isEqualToString:, 313
isEqualToString:, 318
isReadableAtPath:, 371
isWritableAtPath:, 371
lastPathComponent, 382
length, 318
list, 342
lookup:, 344–346
lookupAll:, 365
mainBundle, 386
moveItemAtPath:, 371
multiply, 66
mutablecopy, 418–420
name, 182
names, 182
newObject, 54
NSArray class, 354
NSDictionary class, 359
NSFileHandle, 389
NSFileManager, 370
NSIndexSet class, 366
NSMutableArray class, 355
NSMutableDictionary class, 359
NSMutableSet class, 363
NSMutableString class, 328
NSProcessInfo class, 386
NSSet class, 363
NSString class, 326–328
numberWithInt:, 313
numberWithInteger:, 313
objectAtIndex:, 330
objects
 allocating, 141–146, 413
 returning, 141–146
offsetInFile, 389
operatingSystem, 386
operatingSystemName, 386
operatingSystemVersionString, 386

outline, tests, 227
overriding, 167–172, 224
pathComponents, 382–383
pathExtension, 382–383
pathForResource:, 395
paths, 384–385
pathWithComponents, 383
performSelector:, 188
pointers, 279–280
print, 35, 182, 335
printVar, 170
processDigit:, 472
processIdentifier, 386
processInfo, 386
processName, 386
rangeOfString:, 325
readDataOfLength:, 389–391
readDataToEndOfFile, 389–391
reduce, 135, 140
release, 171
removeCard:, 346–349
removeItemAtPath:, 374–376
removeObjectAtIndex:, 401
respondsToSelector:, 189
retain, 408
return values, 36, 311
saveFilePath, 384
seekToEndOfFile, 389
seekToFileOffset:, 389
selecting, 153–154, 169–171
set::, 133
setAttributesOfItemsAtPath:, 371
setDenominator:, 41
setEmail:, 334
setName:, 334, 425
setNumerator:, 40
setOrigin:, 164, 171
setReal:andImaginary:, 184
setSide:, 156
setStr:, 405, 408
setter, 48, 203
setTo:over:, 188
setWidth:, 156
setWithObjects:, 362
sortUsingFunction:, 350
sortUsingSelector:, 350
stringByDeletingLastPathComponent, 383
stringByDeletingPathExtension, 383
stringByExpandingTildeInPath, 383
stringByResolvingSymlinksInPath, 383
stringByStandardizingPath, 383
stringsByAppendingPathComponent:, 383
stringsByAppendingPathExtension:, 383
stringWithString:, 324
substringFromIndex:, 321
substringWithRange:, 321
syntax, applying to classes, 28
tests, 204, 423
truncateFileAtOffset:, 390
union:, 362
unwrapper, 355
wrapper, 355
writeData:, 388
writeToFile:, 430
metric conversions, 240
minus sign (-), 35, 55
modifying
 labels, 461
 order of evaluation, 57
 window colors, 459

modularization, 219
modulus operator (%), 60-61
moveItemAtPath: method, 371
multidimensional arrays, 252-254
multiple arguments, 130-135
multiple operations, evaluating, 57
multiplying fractions, 30, 465
multiply method, 66
mutablecopy method, 418-420
mutable objects, 316-322
mutable strings, 322-326

N

name method, 387
names
 alternative, assigning to data types, 208
 arguments, methods without, 133
 folders, 8
 methods, 182
 pathnames to files, 370
 selecting, 33-35
nested for loops, 78-80
nested if statements, 101-102
New File menu, Xcode, 124
newObject method, 54
New Project window, 466
nil objects, 336, 356
nonatomic attributes, 337
nonzero, 116
notations
 multidimensional arrays, 252
 sigma, 144
not equal to (!=) operator, 72
NSArray class, 198, 329
 methods, 354
NSAutoreleasePool class, 306
NSBundle class, 370, 394-395
NSCopying protocol, 225
 implementation, 422-423
NSCountedSet class, 362
NSData class
 applying, 374
 archiving, 440-443
NSDictionary class methods, 359
NSException object, 193
NSFileHandle class, 369, 389-391
NSFileManager class, 369-380
NSFullUserName function, 384
NSHomeDirectoryForUser function, 382-384
NSHomeDirectory function, 382-384
NSIndexSet class, 363-366
NSKeyedArchiver, 431-433
NSLog function, 314-315
NSLog statements, 19, 22, 47, 59, 267
 for statements, 74
 while statements, 84
NSMutableArray class, 329
 methods, 355
NSMutableCopying protocol, 418
NSMutableDictionary class methods, 359
NSMutableSet class methods, 363
NSMutableString class, 326
 methods, 328
NSNumber objects, 311
 creation/retrieval methods, 312
NSObjectRuntime.h header, 244
NSObject class, 149
NSPathUtilities.h file, 380, 383
NSProcessInfo class, copying files, 385, 388
NSRange structure, 322
NSSearchPathForDirectoriesInDomains
 function, 384
NSSet class methods, 363
NSString object, 314

NSString class methods, 326-328

NSTemporaryDirectory function, 382-384

NSURL class, 370, 399

NSUserName function, 384

NSNumber class, 355-356

null characters ('0'), 248

null statements, 296

numberOfDays function, 269

numbers, 309

 autorelease pools, 311-314

 Fibonacci, 249

 floating-point, 175

 if-else constructs, 95-98

 indexes, 330

 objects, 309-314

 prime, 114

 triangle arrangements, 255.

See also triangle arrangements

numberWithInteger: method, 313

numberWithInt: method, 313

numerator instance variable, 128

O

objectAtIndex: method, 330

**object-oriented programming. See OOP
(object-oriented programming)**

objects, 27

 actions in, 28

 allocating, 413, 424

 arrays, 328-356

 address books, 332-337

 references, 419

 sorting, 350-355

 autorelease pools, 311-314, 398.

See also autorelease pools

 composite, 229-230

 copying, 417, 443-444

C programming language, 295

dictionaries, 356-359

encoding, 441

Foundation framework, 309

generating, 188

getter/setter methods, 425-427

instances, 197

methods

 allocating, 141-146

 returning, 141-146

nil, 336, 356

NSEException, 193

NSNumber, 311-312

NSString, 314

numbers, 309-314

overview of, 27-28

releasing, 410

Rounded Rect Button, 461

saving, 431

sets, 360-366

storage, id data types, 54-55

strings, 314-328

 description method, 315-316

 mutable, 322-326

 mutable/imutable objects, 316-322

 NSLog function, 314-315

 returning, 326-328

structure conversions, 355

variables, 301-302

viewing, 439

odd numbers, if-else constructs, 95-98

offsetInFile method, 389

old-style property lists, 429

ones complement operators, 215-216

OOP (object-oriented programming), 273

OpenGL, 446

opening files, 369

operands, 118

operatingSystem method, 386

operatingSystemName method, 386

operatingSystemVersionString method, 386

operations

- on fractions, 133–135
- keying, 465
- on pointers, 690–691

operators

- AND, 98, 213, 239
- ##-240
- &, 239
- address (&), 274
- assignment, 63–64
- bit, 211–218
- comma, 297
- conditional, 118–119
- decrement (—), 282
- dot, 130, 266
- equal to (==), 72
- greater than (>), 72
- greater than or equal to (>=), 72
- increment (++), 290
- indirection (*), 276
- left shift, 216–217
- less than (<), 72
- less than or equal to (<=), 72
- logical negation, 117
- modulus (%), 60–61
- not equal to (!=), 72
- ones complement, 215–216

OR

- compound relational tests, 98
- exclusive, 214
- inclusive, 214
- plus equals (+=), 63
- precedence, 55–58

relational, 72

right shift, 217–218

sizeof, 297–298

ternary, 118

type cast, 62–63, 208, 210

unary plus (+), 59

options, selecting projects, 453

order

- of conversions, 198
- of evaluation, modifying, 57

OR operators

- compound relational tests, 98
- exclusive, 214
- inclusive, 214

outlets, 449

- variables, connecting, 462

outline method, tests, 227

overriding

- dealloc methods, 171–172
- methods, 167–172, 224

owning objects, classes, 163–167

P

pageCount variables, 138

pairs, key-object, 356

parameters

- atomically, 430
- inDirectory:, 380

passing

- arrays, 261, 284
- blocks, 262
- pointers, 291

rectangle origins to methods, 164

pathComponents method, 382–383

pathExtension method, 382–383

pathForResource: method, 395

pathnames
 files, 370
 formatting, 382

paths
 applying, 385
 functions, 385
 methods, 384–385
`NSPathUtilities.h`, 380

PATH variables, 17

pathWithComponents method, 383

performSelector: method, 188

PI, 235

plus equals (+=) operator, 63

plus sign (+), 55

pointers, 273
 arrays, 281, 289
 characters, 280
 constant character strings, 287
 functions, 289–290
 memory addresses, 291–292
 methods, 279–280
 operations on, 290–291
 passing, 290
 structures, 279

polymorphism, 54, 179–182

pools, autorelease, 22, 30, 142, 411
 examples of, 409–410
 memory, 397–398
`NSAutoreleasePool`, 306

positioning
`#define` statements, 234
 files, 392

precedence, operators, 55–58

preprocessors, 233
`#define` statement, 233–239
`#elif` statements, 243–244

`#if` statements, 243–244
`#import` statements, 240–241
`#undef` statements, 244
 conditional compilation, 241–244
 identifiers, defining, 242

prime numbers, 114

Printer class, 200

printFoo block, 264

print method, 35, 182, 335

printVar method, 170

processDigit: method, 472

processIdentifier method, 386

processInfo method, 386

processName method, 386

programming, 7
 pointers, characters, 276
 Xcode, applying, 8–15

programs. See also programming
`#import` statements, 240
`@implementation` section, 37
`add:` method, 142
 address books, 437
`addresscard.h` interface file, 434
 address (&) operators, 277
 archiving, customizing, 430
 arguments, 255
 array objects, 329
 bit operators, 215
 blocks
 accessing variables, 264
 defining, 263
 executing, 262
 modifying values, 264
 Boolean variables, 114–117
 Calculator class, 64–66
 character arrays, 251

char data types, 52-53
classes, 189
defining, 404
owning objects, 163
common divisor algorithms, 258
compiling, 7-17
compound relational tests, 99-100
conversions, 61
copying, 420
copyString function, 289
dataArray, 418
date structures, 268
dictionaries, 356-357
directories, 376, 378
do statements, 86-87
dummy classes, defining, 409
dynamic binding, 182
else if constructs, 103-108
encoding/decoding, 436
enumerated data types, 207
exception handling, 191-192
fast enumeration, 330
Fibonacci numbers, 249
files, 372, 375
for statements, 70, 73
Fraction class, 31
fractions, 30
FractionTest.m, 123, 144
if-else constructs, 96-98
if statements, 92-94
immutable character strings, 316
implementation files, 126, 131, 138, 180
initialization methods, 199
instance variables, 46-47
adding, 173
releasing values, 407
integer arithmetic, 58
interface files, Fraction.h, 124, 131, 138
iPhone_1AppDelegate.m
implementation file, 457
iPhone applications, 456
looping, 69-70
main function, 255
main.m, 10, 180
Main Test Program, FractionTest.m,
127, 132, 134, 139
MathOps category, 221
methods
adding, 149
overriding, 167
tests, 423
modulus operator (%), 60
mutable strings, 322
nested for loops, 78
NSFileHandle, 391-392
NSKeyedArchiver, 432
NSLog statements, 22
NSProcessInfo class, 395
NSSstring object, 314
NSURL class, 392-394
number objects, 309
objects
copying, 443-444
viewing, 439
operator precedence, 56
overview of, 17-21
paths, 280
pointers
arraySum function, 283
indexes, 284
returning, 280
structures, 279
polymorphism, 179

program section, 43–44
 readability, 34
`Rectangle.h` interface files, 161
`Rectangle.m` added methods, 160
`Rectangle.m` implementation files, 161
 references, 399–401
 restoring data, 442
 root classes, 149
 sections, 38–45
 sets, 360
`setStr:` method, 405
 statements, 22
 static variables, 204
 strings, 320
 structures, 287, 279
 switch statements, 112–113
 Terminal, 15–17
 termination, 20, 192
 testing, 109
 triangle arrangements, 69, 77
 values of variables, displaying, 21
 while statements, 81–84
 XML propertylists, 429–431
`XYPoint.h` interface files, 160
`XYPoint.m` implementation files, 160

projects
 classes, adding, 124
`Fraction_Calculator`, 465–467
 options, selecting, 453
 starting, 8

properties
 accessing, 130
`UILabel`, 455

protocols, 219
 defining, 227
 and delegation, 225–229

Drawing, 226
`Drawing3D`, 227
 informal, 228–229
`NSCopying`, 225, 422–423

Q

qualifiers, 53–54
Quartz, 446
question marks (?), 118
Quick Help pane, 307
QuickTime, 446

R

rangeOfString: method, 325
readability, 34
readDataOfLength: method, 389–391
readDataToEndOfFile: method, 389–391
reading XML propertylists, 431
Rectangle class, 155, 189
Rectangle.h interface file, 161
Rectangle.m implementation file, 161
rectangles in windows, 157
Rectangle variable, 185
recursion, directories, 378
reduce method, 135, 140
references
 array objects, 419
 counting, 398–409
 elements, 284
 instance variables, 403–409
 libraries, accessing MAC OS X, 308
 strings, counting, 401–403
 tracking, 399
relational operators, 72
release messages, 398, 410

release method, 171
releasing
 members, 171
 objects, 398, 410. *See also* autorelease pools
removeCard: method, 346-349
removeItemAtPath: method, 374-376
removeObjectAtIndex: method, 401
repetition, 70
reserved words, Xcode, 13
respondsToSelector: method, 189
results, returning functions, 257-258
retainCount message, 399
retain method, 408
returning
 objects from methods, 141-146
 results from functions, 257-258
return statements, 48
return types with dynamic typing, 186-187
return values, 36-37
right shift operators, 217-218
root classes, 147-154
Rounded Rect Button object, 461
routines
 islower, 106
 isupper, 106
 main, 19
 NSLog statements, 23
 program section, 38
rules
 conversions, 210-211
 external variables, 201
 memory management, 410-411
 multiple operations, evaluating, 57
 names, creating, 33
running programs, 7-17

runtime
 checking, 184-185
 dynamic typing, 187
 garbage collection, 413
 systems, 41

S

saveFilePath method, 384
saving
 folders, 8
 objects, 431
scanf calls, 108
scope
 blocks, 261
 enumerated identifiers, 208
 functions, 260
 variables, 200-205
 directives for controlling instances, 200-201
 external variables, 201-202
 static variables, 203-205
SDK (Software Development Kit), 2
 iOS, 449
sections
 @implementation, 37-38
 @interface, 32-37
 interfaces, 224
 program, 38-45
seekToEndOfFile method, 389
seekToFileOffset: method, 389
selecting
 application types, 8
 methods, 153-154, 169-171
 names, 33-35
 project options, 453
selectors, generating, 188

self keyword, [140-141](#)

semicolons (`;`), [19](#), [236](#)

sending

- release messages, [410](#)
- uppercaseString messages, [319](#)

separate files, classes, [123-128](#)

sequences

- int data types, [51-50](#)
- unions, [294](#). *See also* unions

setAttributesOfItemsAtPath: method, [371](#)

setDenominator: method, [41](#)

setEmail: method, [334](#)

set:: method, [133](#)

setName: method, [334](#), [425](#)

setNumerator: method, [40](#)

setOrigin: method, [164](#), [171](#)

setReal:andImaginary: method, [184](#)

sets, [309](#)

- copying, [422](#)
- objects, [360-366](#)

setSide: method, [156](#)

setStr: method, [405](#), [408](#)

setter methods, [48](#), [203](#)

- objects, copying, [425-427](#)
- structures, [266](#)

setTo:over: method, [188](#)

setWidth: method, [156](#)

setWithObjects: method, [362](#)

shallow copying, [420-422](#)

shells, UNIX, [17](#)

short qualifiers, [53-54](#)

sigma notations, [144](#)

signed qualifiers, [53-54](#)

sign extensions, [211](#)

sign functions, implementing, [103](#)

simulators, iPhones, [450](#), [455](#)

single characters, char data types, [52-53](#)

sizeof operator, [297-298](#)

sizing, modifying labels, [461](#)

skipDescendants message, [379](#)

slash (/), [18](#), [55](#)

Software Development Kit. *See* **SDK**

sorting

- arrays, [350-355](#)
- Using blocks, [352-355](#)

sortUsingFunction: method, [350](#)

sortUsingSelector: method, [350](#)

source code, viewing interfaces, [462](#)

specifiers, classes, [205](#)

Square class, [155](#), [189](#)

- defining, [229](#)
- testing, [157](#)

Square.h interface file, [156](#)

SQUARE macro, [238](#)

squareRoot function, [261](#)

Standard Library, [291](#)

starting

- projects, [8](#)
- Terminal, [15](#)

statements. *See also* **constructs**

- for, [70-81](#)
 - executing, [73](#)
 - keyboard input, [76-78](#)
 - nested for loops, [78-80](#)
 - programs, [73](#)
- #define, [233-239](#)
- #elif, [243-244](#)
- #else, [241-243](#)
- #endif, [241-243](#)
- #if, [243-244](#)
- #ifdef, [241-243](#)
- #ifndef, [241-243](#)
- #import, [234](#), [240-241](#)

#include, 234
#undef, 244
assignment, 417
break, 87, 111-113
comments, 18
continue, 87-88
debugging, 243
do, 85-87
goto, 296
if, 91-111
 compound relational tests, 98-101
 else if constructs, 102-111
 if-else constructs, 95-98
 nested, 101-102
NSLog, 19, 22, 47, 59
 for statements, 74
 while statements, 84
null, 296
programs, 22
return, 48
switch, 111-114
termination, 19
typedef, 208-209
while, 81-85

static functions, 260. *See also* functions

static keyword, 137-140

static typing, id data types, 185-187

static variables, 203-205

storage, 203-205

- characters, 293
- data types, 33
- instance variables, 300-301
- objects, id data types, 54-55
- projects, writing iPhone applications, 453
- temporary areas, files, 374
- XML propertylists, 429

stringByDeletingLastPathComponent
 method, 383

stringByDeletingPathExtension method, 383

stringByExpandingTildeInPath method, 383

stringByResolvingSymlinksInPath method, 383

stringByStandardizingPath method, 383

strings, 309

- characters
 - immutable, 316
 - pointers to, 316
- indexes, 325
- objects, 314-328
 - description method, 315-316
 - immutable, 316-322
 - mutable, 322-326
 - NSLog function, 314-315
 - returning, 326-328
 - saving, 431
- references, counting, 401-403

stringsByAppendingPathComponent:
 method, 383

stringsByAppendingPathExtension:
 method, 383

stringWithString: method, 324

struct keyword, 293

structures, 265

- arrays of, 268
- dates, 268
- defining, 268-270, 273
- initializing, 286-287
- NSRange, 322
- object conversions, 355
- OOP (object-oriented programming), 273
- pointers, 277-279. *See also* pointers
- within structures, 269-270

styles, comments, 18

subclasses, 147, 229. *See also* classes
 abstract classes, 175
 creating, 175

subdirectories, 380. *See also* directories

subfolders, 8. *See also* folders

subscribing to documents (Xcode), 308

substitutions, text, 236

substringFromIndex: method, 321

substrings, 320. *See also* strings

substringWithRange: method, 321

subtracting fractions, 30

superclasses, 150. *See also* classes

support, 5
 Mac OS X, 445

switch statements, 111-114

syntax
 do statements, 85
 methods, applying to classes, 28

synthesized accessor methods, 128-130

synthesized AddressCard class methods, 337-339

systems, runtime, 41

T

templates, iPhone applications, 452

temporary storage areas, files, 374

Terminal
 debugging, 17
 programs, running, 15-17
 windows, 3

termination
 applications, 411
 continue statements, 87-88
 programs, 20, 192
 statements, 19

ternary operators, 118

tests
 add: method, 142
 AddressBook class, 342
 AddressCard class, 335
 compound relational, 98-101
 equality, 236
 flags, 116
 FractionTest.m, 144.
See also FractionTest.m
 lookup: method, 344
 lowercase letters, 239
 main.m program, 180
 MathOps category, 221
 methods, 204, 423
 outline method, 227
 Rectangle class, 162
 removeCard: method, 348
 sort method, 351
 Square class, 157
 synthesized setter methods, 339

text, substitutions, 236

throwing exceptions, 193

tilde (~), 370, 382

tools
 NSPathUtilities.h, 380, 389
 Terminal, 15-17
 Xcode, 8. *See also* Xcode

tracking references, 399

triangle arrangements, 69, 77, 255

troubleshooting
 files, 374
 programs, 109
 Xcode, 13

TRUE, 233

truncateFileAtOffset: method, 390

two-dimensional arrays, 252

two-step allocation approach, 48

typedef statements, 208-209, 269

types

- advanced, definitions, 235-240
- applications, selecting, 8
- cast operators, 62-63, 208, 210
- of comments, 18
- data, 51. *See also* data types
- encoding/decoding, 434
- of exceptions, 193
- return, with dynamic typing, 186-187

typing

- dynamic, 179
- arguments, 186-187
- methods, 187
- with return types, 186-187
- static id data types, 185-187

U

UIKit classes, 229

UILabel property, 455

UIs (user interfaces), designing, 477

unarchiving, 443. *See also* archiving

unary plus operator (+), 59

underscore (_), 33

unichar characters, 314

union: method, 362

unions, 292-294

unique instance variables, 44

UNIX shells, 17

unknown operators, keying, 109

unsigned qualifiers, 53-54

unwrapper methods, 355

uppercase letters, 17

uppercaseString messages, sending, 319

User Interface design pane, 458

user interfaces, 477

Using blocks, sorting, 352-355

utilities, NSPathUtilities.h, 254, 389

V

valid names, 33. *See also* names

values

- arrays, initializing elements, 250
- floating-point, 62
- nil objects, 356. *See also* nil objects
- return, 36-37
- switch statements, 113
- variables, viewing, 21-23
- Xcode, 13

variables, 197

- blocks, assigning, 262
- Boolean, 114-118
- characters, sign extensions, 211
- char data types, 52-54
- defining, 205
- float data types, 52
- global, defining, 202
- instances
 - @interface section, 35
 - accessing, 45-48
 - adding, 173-175
 - counting references, 403-409
 - inheritance, 151
 - storage, 300-301
 - unique, 44
- int data types, 52
- integers, 22
- isPrime, 115
- local, 135-140
 - automatic, 256
 - functions, 255, 257
- local static, 137

objects, 300–301. *See also* pointers
 outlet, connecting, 462
 pageCount, 138
 PATH, 17
 pointers, 274
 qualifiers, 53–54
 Rectangle, 185
 scope, 200–205
 directives for controlling instances, 200–201
 external variables, 201–202
 static variables, 203–205
 values, viewing, 21–23

variants for loops, 80–81

viewing
 code, interfaces, 462
 objects, 439
 values, variables, 21–23

View pane, 307

views
 controllers, 465
 defining, 468–472

W

while loops, 326

while statements, 81–85

windows
 colors, modifying, 459
 edit, 10
 New Project, 466
 rectangles in, 157
 Terminal, 3, 15
 Xcode prog1 project, 10

wrapper methods, 355

writeData: method, 388

writeToFile: method, 430

writing applications

iOS, 449. *See also* iOS
 iPhone, 449–464

X

Xcode, 6

applying, 8–15
 debugging, 13
 documentation, 306
 FractionTest.m, 123
 iOS SDKs (Software Development Kits), 449
 New File menu, 124
 projects, starting, 8

XML (Extensible Markup Language)

files, 430
 propertylists, 429–431

XYPoint class, 159, 190

implementation files, 160
 interface files, 160
 memory, 164

Y

years, leap, 237

Z

zeros

char data types, 52
 conditional operators, 118
 division-by-zero problem, 110
 nonzero, 116
 sets, 362

zone arguments, 423