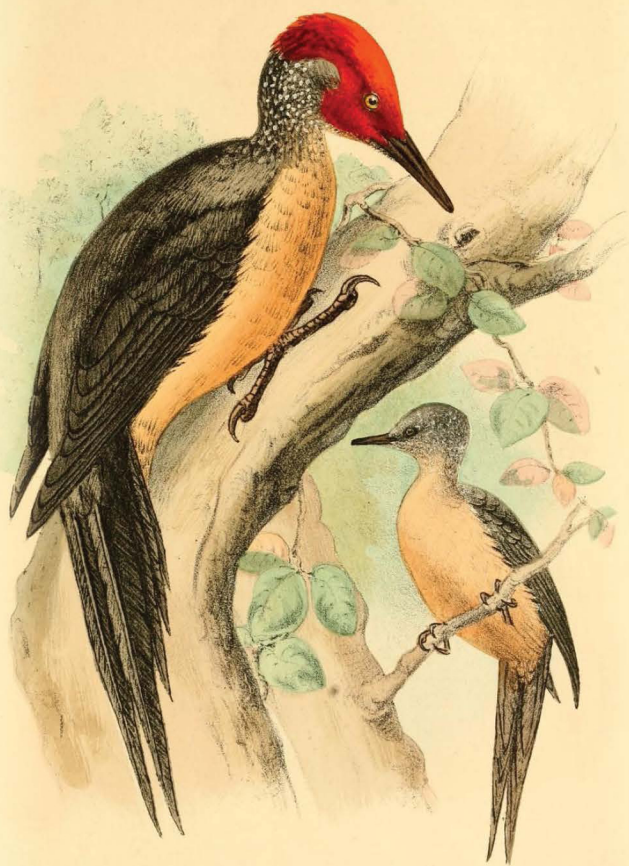


MASTERING COMPUTER SCIENCE

Mastering Objective-C

A Beginner's Guide



Adarász del. et lith.

W. Grund imp.

EDITED BY

Sufyan bin Uzayr



CRC Press
Taylor & Francis Group

Mastering Objective-C

Mastering Objective-C is a detailed guide for beginners that gives a modern programmer's viewpoint on Objective-C. It introduces the principles of Objective-C and current Apple programming skills and libraries allowing the reader to make the best use of the tools available.

Concise and easy to understand, this book covers many topics including an introduction to key concepts of language and primary toolkits, as well as the Foundation and AppKit frameworks to help quickly advance with the necessary information.

This book explains how to structure applications properly and organize code for optimum performance. The fundamental principles explained here are helpful to beginner and intermediate users interested in learning this highly technological and diverse language.

About the Series

The *Mastering Computer Science* covers a wide range of topics, spanning programming languages as well as modern-day technologies and frameworks. The series has a special focus on beginner-level content and is presented in an easy-to-understand manner, comprising:

- Crystal-clear text, spanning various topics sorted by relevance.
- Special focus on practical exercises, with numerous code samples and programs.
- A guided approach to programming, with step-by-step tutorials for the absolute beginners.
- Keen emphasis on real-world utility of skills, thereby cutting the redundant and seldom-used concepts and focusing instead of industry-prevalent coding paradigm.
- A wide range of references and resources, to help both beginner and intermediate-level developers gain the most out of the books.

Mastering Computer Science series of books start from the core concepts, and then quickly move on to industry-standard coding practices, to help learners gain efficient and crucial skills in as little time as possible. The books assume no prior knowledge of coding, so even the absolute newbie coders can benefit from this series.

Mastering Computer Science series is edited by Sufyan bin Uzayr, a writer and educator having over a decade of experience in the computing field.

For more information about this series, please visit: <https://www.routledge.com/Mastering-Computer-Science/book-series/MCS>

Mastering Objective-C

A Beginner's Guide

Edited by
Sufyan bin Uzayr



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

First Edition published 2024

by CRC Press

2385 NW Executive Center Drive, Suite 320, Boca Raton, FL 33431

and by CRC Press

2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2024 Sufyan bin Uzayr

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark Notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

ISBN: 9781032415307 (hbk)

ISBN: 9781032415284 (pbk)

ISBN: 9781003358534 (ebk)

DOI: [10.1201/9781003358534](https://doi.org/10.1201/9781003358534)

Typeset in Minion

by KnowledgeWorks Global Ltd.

For Mom



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Contents

About the Editor, xiii

Acknowledgments, xiv

Zeba Academy – Mastering Computer Science, xv

CHAPTER 1 ■ Objective-C Overview	1
WHY CHOOSE OBJECTIVE-C?	2
ADVANTAGES AND DISADVANTAGES	3
Advantages	3
Disadvantages	3
FACTORS TO CONSIDER	3
DIFFERENTIATING OBJECTIVE-C FROM C++	4
DIFFERENTIATING OBJECTIVE-C VERSUS SWIFT	4
FRAMEWORK FOR THE FOUNDATION	5
LEARNING OBJECTIVE-C	5
USING OBJECTIVE-C	5
OBJECT-ORIENTED PROGRAMMING	5
INSTANCE, OBJECT, AND CLASS	6
RETURN TO THE CUSTOMIZE TABLE CELL	6
MAKING THE RECIPE CLASS	8
Set up the Array of Recipe Objects	9
A Recipes Array Is Replacing TableData	12
WHAT DOES A COMPILER DO?	13
PREPROCESSING	14

CUSTOM MACROS	15
TOKENIZATION (LEXING)	17
PARSING	18
STATIC ANALYSIS	20
TYPE CHECKING	20
OTHER ANALYSES	21
CREATING CODE	21
OPTIMIZATIONS	23
OBJECTIVE-C DEVELOPERS' C# PRIMER	25
INTEROP OBJECTIVE-C	25
COMPARISON OF LANGUAGES	26
INTERFACES VS. PROTOCOLS	26
EXTENSION METHODS VS CATEGORIES	26
ASSEMBLIES VS. FRAMEWORKS	26
NAMED PARAMETERS VS. SELECTORS	27
NAMESPACES AND HEADERS	27
PROPERTIES	27
STATIC KEYWORD	27
LIST VS NSARRAY INITIALIZATION	28
LAMBDA EXPRESSIONS VS BLOCKS	28
NOTES	28
CHAPTER 2 ■ Environment Setup and Program Structure	29
CONFIGURING THE OBJECTIVE-C ENVIRONMENT	29
CONFIGURATION OF THE LOCAL ENVIRONMENT	29
EDITOR OF TEXT	30
GCC COMPILER	30
UNIX/LINUX INSTALLATION	30
Mac OS INSTALLATION	32
LAUNCHING Xcode	32
WINDOWS INSTALLATION	33
STRUCTURE OF THE OBJECTIVE-C PROGRAM	34
Objective-C Example of Hello Everyone	34

USING Xcode TO CREATE AN OBJECTIVE-C APPLICATION	35
USING THE COMMAND LINE TO COMPILE OBJECTIVE-C	36
FILENAME EXTENSION	37
USING THE TERMINAL	40
FIRST PROGRAM'S EXPLANATION	42
DISPLAYING THE VALUES OF VARIABLES	46
CLASSES, OBJECTS, AND METHODS	49
WHAT EXACTLY IS AN OBJECT?	49
INSTANCES AND METHODS	49
AN OBJECTIVE-C CLASS FOR THE WORKING WITH FRACTIONS	52
@interface SECTION	54
CLASS AND INSTANCE METHODS	57
RETURN VALUES	57
METHOD ARGUMENTS	58
THE @implementation SECTION	59
PROGRAM SECTION	61
ACCESSING INSTANCE VARIABLES AND DATA	
ENCAPSULATION	66
NOTE	70
CHAPTER 3 ■ Basic Syntax	71
OBJECTIVE-C TOKENS	71
SEMICOLONS;	72
COMMENTS	72
IDENTIFIERS	72
KEYWORDS	72
WHITESPACE IN OBJECTIVE-C	73
DATA TYPES	73
TYPES OF INTEGERS	74
TYPES OF FLOATING-POINT	75
VOID TYPE	75
VARIABLES IN OBJECTIVE-C	76

VARIABLE DEFINITION	76
VARIABLE DECLARATION	77
LVALUES AND RVALUES	79
CONSTANTS IN OBJECTIVE-C	79
INTEGER LITERALS	79
FLOATING-POINT LITERALS	80
CHARACTER CONSTANTS	80
STRING LITERALS	81
CONSTANT DEFINITION	81
#define Preprocessor	82
const Keyword	82
CHAPTER 4 ■ Data Types	83
DATA TYPES AND CONSTANTS	83
TYPE int	84
TYPE float	84
TYPE char	85
QUALIFIERS: long, long long, short, unsigned, and signed	86
TYPE id	86
THE void TYPE	87
CHAPTER 5 ■ Variables	88
OBJECTIVE-C VARIABLE DEFINITION	89
OBJECTIVE-C VARIABLE DECLARATION	90
OBJECTIVE-C LVALUES AND RVALUES	91
CHAPTER 6 ■ Constants	92
CONSTANTS IN OBJECTIVE-C	92
INTEGER LITERALS	93
FLOATING-POINT LITERALS	93
CHARACTER CONSTANTS	94
STRING LITERALS	94

CONSTANT DEFINITION	95
#define Preprocessor	95
const Keyword	95
CHAPTER 7 ■ Operators	97
ARITHMETIC OPERATORS IN OBJECTIVE-C	98
OPERATOR PRECEDENCE	98
UNARY MINUS OPERATOR AND INTEGER ARITHMETIC	100
THE MODULUS OPERATOR	102
INTEGER AND FLOATING-POINT CONVERSIONS	103
TYPE CAST OPERATOR	105
RELATIONAL OPERATORS	106
LOGICAL OPERATORS IN OBJECTIVE-C	106
BITWISE OPERATORS	107
ASSIGNMENT OPERATORS	108
MISC OPERATORS ⇨ SIZEOF & TERNARY	109
OPERATORS' PRECEDENCE IN THE OBJECTIVE-C	109
CALCULATOR CLASS	110
CHAPTER 8 ■ Loops and Control Statement	113
THE for Statement	115
RELATIONAL OPERATORS	116
KEYBOARD INPUT	121
Nested for Loops	122
for Loop Variants	123
THE while Statement	125
THE do Statement	129
do-while Loop	130
THE break Statement	131
continue Statement	131

CHAPTER 9 ■ Decision Making	132
THE if Statement	133
THE if-else CONSTRUCT	137
COMPOUND RELATIONAL TESTS	139
Nested if Statements	142
THE else if Construct	144
THE switch Statement	151
BOOLEAN VARIABLES	154
THE CONDITIONAL OPERATOR	159
CHAPTER 10 ■ Functions	161
FUNCTIONS IN OBJECTIVE-C	161
CREATING A METHOD	162
DECLARATIONS OF METHOD	163
CALLING A METHOD	163
FUNCTION ARGUMENTS	165
APPRAISAL, 166	
BIBLIOGRAPHY, 218	
INDEX, 223	

About the Editor

Sufyan bin Uzayr is a writer, coder, and entrepreneur having over a decade of experience in the industry. He has authored several books in the past, pertaining to a diverse range of topics, ranging from History to Computers/IT.

Sufyan is the Director of Parakozm, a multinational IT company specializing in EdTech solutions. He also runs Zeba Academy, an online learning and teaching vertical with a focus on STEM fields.

Sufyan specializes in a wide variety of technologies, such as JavaScript, Dart, WordPress, Drupal, Linux, and Python. He holds multiple degrees, including ones in Management, IT, Literature, and Political Science.

Sufyan is a digital nomad, dividing his time between four countries. He has lived and taught in universities and educational institutions around the globe. Sufyan takes a keen interest in technology, politics, literature, history, and sports, and in his spare time, he enjoys teaching coding and English to young students.

Learn more at sufyanism.com.

Acknowledgments

There are many people who deserve to be on this page, for this book would not have come into existence without their support. That said, some names deserve a special mention, and I am genuinely grateful to:

- My parents, for everything they have done for me.
- The Parakozm team, especially Divya Sachdeva, Jaskiran Kaur, and Simran Rao, for offering great amounts of help and assistance during the book-writing process.
- The CRC team, especially Sean Connelly and Danielle Zarfati, for ensuring that the book's content, layout, formatting, and everything else remain perfect throughout.
- Reviewers of this book, for going through the manuscript and providing their insight and feedback.
- Typesetters, cover designers, printers, and everyone else, for their part in the development of this book.
- All the folks associated with Zeba Academy, either directly or indirectly, for their help and support.
- The programming community, in general, and the web development community, in particular, for all their hard work and efforts.

Sufyan bin Uzayr

Zeba Academy – Mastering Computer Science

The “Mastering Computer Science” series of books are authored by the Zeba Academy team members, led by Sufyan bin Uzayr, consisting of:

- Divya Sachdeva
- Jaskiran Kaur
- Simran Rao
- Aruqqa Khateib
- Suleymen Fez
- Ibbi Yasmin
- Alexander Izbassar

Zeba Academy is an EdTech venture that develops courses and content for learners primarily in STEM fields and offers educational consulting and mentorship to learners and educators worldwide.

Additionally, Zeba Academy is actively engaged in running IT schools in the CIS countries and is currently working in partnership with numerous universities and institutions.

For more info, please visit <https://zeba.academy>



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Objective-C Overview

IN THIS CHAPTER

- Object-Oriented Programming
- Framework for Foundations
- Learning Objective-C
- Advantages and Use of Objective-C

A general-purpose, object-oriented computer language called Objective-C enhances the C language with Smalltalk-style messaging. Cocoa and Cocoa Touch are Apple's major programming languages for the OS X and iOS operating systems and their accompanying APIs. This book will guide you through a simple and practical approach to learning the Objective-C programming language.

Objective-C programming is a general-purpose programming language. Although it is not exclusive to any platform or system, it may be used to build various frameworks. Objective-C programming extends the language C with communication capabilities.

Objective-C is one of Apple's core programming languages for the iOS platform and is used to create mobile applications for this platform. It enables developers to be more detail-oriented and accept objects and other programming languages since it is a superset of the C language.

Objective-C is a good memory management language; there are compilers available that can convert Objective-C code into static code analysis,

which the language then utilizes to differentiate valuable information from “garbage.”

The most critical component of Objective-C to comprehend is its object-oriented nature. You can move graphs and alter files using this new language, but it is critical to grasp its limitations to appreciate its advantages.

Many aspects of Objective-C, a programming language developed in the 1980s, are still employed in iOS-specific mobile app development. While no breakthrough has been made that allows Objective-C to be used on all platforms, Objective-C is compatible with C and other languages for iOS applications.

Swift, a more contemporary iOS programming language, competes with Objective-C. There have been many debates on whether programming language produces better iOS mobile application development results.

WHY CHOOSE OBJECTIVE-C?

The Objective-C programming language was chosen for numerous reasons. First and foremost, it is an object-oriented language. Object-oriented methods are necessary to provide the capabilities found in the Cocoa frameworks. Second, since Objective-C is an extension of ANSI C, existing C programs are transformed to use the software frameworks without losing any of the work that went into their development. Since Objective-C contains C, we get all its benefits while interacting with it.

We may choose whether to employ object-oriented programming (OOP) methods (such as creating a new class) or procedural programming approaches (define a structure and some functions instead of a class).

In addition, Objective-C is a relatively simple programming language. It features a basic, unambiguous syntax that is easy to learn.

OOP presents a steep learning curve to new recruits with its self-conscious language and emphasis on abstract design. A well-structured language, such as Objective-C, may make it much simpler to become an adept object-oriented programmer.

When compared to other C-based object-oriented languages, Objective-C is very dynamic. The compiler maintains much information about the objects themselves for use at runtime. Decisions made at compile time are postponed until the program is run. Because of their dynamism, Objective-C programs offer a great deal of flexibility and power. It offers two significant benefits that other object-oriented languages do not.

Objective-C offers an open, dynamic binding method that allows for the creation of a rudimentary interactive user interface. Messages are not

necessarily restricted by the class or method name of the recipient. As a result, a software framework may allow users to make choices at runtime while giving developers creative freedom in their design. (Terms such as dynamic binding, message, class, and receiver will be described further down in this document.)

Dynamism enables the development of improved development tools. It is possible to build tools that monitor, intervene, and reveal the underlying structure and activities of Objective-C applications by using an interface to the runtime system, which provides access to running app information.

ADVANTAGES AND DISADVANTAGES

Advantages

- Compatibility with C++ as well as Objective-C++.
- Method swizzling is an effective characteristic.
- More dedicated support with Binary Frameworks coding.

Disadvantages

- Since Objective-C is based on C, namespacing is essential. Every class in an Objective-C application must be globally distinct. To minimize conflict, it is common to practice to prefix class names. We own the “NS” prefix for Foundation Framework classes and the “UI” prefix for UIKit classes.
- Specific pointers.
- The ability to send a message on a nil object without dropping and the absence of strict type make identifying and resolving problems more difficult.
- The language’s grammar is laborious and difficult.

FACTORS TO CONSIDER

Objective-C application development may be sped up and is an effective way to add object-based functionality to a program. This superset language has many essential considerations:

Upgrading Objective-C-built apps requires regular maintenance. Although the language is out of date, it is not obsolete. To stay updated, just little maintenance is required.

Many APIs still have much catching up to do for Swift-developed apps; thus little adaptation is required. This implies that Objective-C may be easier to implement in iOS mobile apps.

Object handling simplified Apple is obsessed with having an object network. These items can be easily moved using Objective-C.

Objective-C may make iOS mobile apps easier to develop for all Apple iOS devices, including smartphones and tablets.

DIFFERENTIATING OBJECTIVE-C FROM C++

According to Wilkerson, C++ was established in 1979 to integrate objects and instance methods with the original C programming language. The Objective-C programming language was created with the notion that OOP would be more productive and effective for large software projects; numerous senior experts believe that this led to C++'s widespread popularity in the years that followed. According to programming community specialists, most modern desktop apps are written in C++. Because of its popularity, various frameworks and libraries have been created to expand C++ for high-performance graphics, audio digital signal processing, and user interface design.

Both languages are developed from C, although they are very different. Objective-C depends primarily on its runtime library to manage inheritance and polymorphism, while C relies heavily on compile-time options. Nonetheless, with C++, the emphasis is often on compile-time conclusions. Middle-level computer language C++ is compatible with a number of cross-platform operating systems, including Windows, UNIX, and macOS. On the other hand, Apple uses Objective-C, a general-purpose, object-oriented programming language, among other things, in its Cocoa APIs and operating systems.

DIFFERENTIATING OBJECTIVE-C VERSUS SWIFT

Swift is a computer language launched by Apple in June 2014. Objective-C has all of the faults one would expect from a language based on C. Objective-C prefixes new keywords with @ to distinguish them from C types. Swift does not use C. As a result, it can aggregate all Objective-C types and object-related keywords and extract their many @ symbols.

Like other recent computer languages, Swift code closely resembles natural English. In contrast to the traumatic experience that Objective-C programmers had, this readability makes it simpler for existing JavaScript, Java, Python, C#, and C++ programmers to embrace Swift as part of their toolchain.

FRAMEWORK FOR THE FOUNDATION

The Foundation Framework has a plethora of functionality, which are described below:

- It includes several expanded datatypes such as NSArray, NSDictionary, NSSet, and others.
- It has an extensive collection of functions for handling files, strings, etc.
- It has URL processing functions and utilities like date formatting, data handling, error handling, etc.

LEARNING OBJECTIVE-C

When studying Objective-C, the essential thing to remember is to stay focused on principles rather than getting bogged down in language technicalities.

The goal of studying a programming language is to become a better programmer or become more successful at developing, implementing, and maintaining existing systems.

USING OBJECTIVE-C

As previously stated, Objective-C is utilized in iOS and macOS X. It has a sizable iOS user base and a rapidly growing macOS X user base. And because Apple prioritizes quality, this is fantastic news for individuals who have just begun learning Objective-C.

OBJECT-ORIENTED PROGRAMMING

Objective-C is characterized as an Object-Oriented Programming (OOP) language. OOP is a technique of designing software programs made of objects. In other terms, most of the code we've written in the program interacts with objects of some form. UIViewController, UIButton, UINavigationController, and UITableView are some items provided with the iOS SDK. Not only you've utilized the built-in objects in our iOS app, but we've also generated some objects of our own such as RecipeDetailViewController and SimpleTableViewCell.

So why OOP? One key reason is that we wish to break an extensive program into smaller components (or building blocks) which are simpler to create and maintain. Here, the smaller bits are the objects. One object has

its duty, and objects cooperate to make the program function. That is the fundamental tenet of OOP.

Consider the HelloWorld app as an illustration. The program is shown using the UIViewController object, which also serves as a placeholder for the HelloWorld button. The HelloWorld button's UIButton object is in charge of displaying a normal iOS button on the screen and monitoring touch events. The user is shown the alert message via the UIAlertView object, on the other hand. Most significantly, these items work together to make the HelloWorld program.

An object in OOP has two characteristics: properties and functionalities. Consider a real-world object – a car. A car has a unique color, model, top speed, manufacturer, etc. These are the characteristics of a car. A car's functionalities should include essential functions such as acceleration, braking, steering, etc.

Returning to the iOS world, consider the following properties and functionalities of the UIButton object in the HelloWorld app:

Background, size, color, and font are examples of UIButton properties.

When the button is pressed, it recognizes the event and displays an alert message on the screen. So the button's function is "showMessage."

INSTANCE, OBJECT, AND CLASS

In addition to method and object, you may encounter terms like instance and class in OOP. Let me explain in more detail.

A class is a model or blueprint for creating objects. A class is made up of properties and methods. Assume we want to create a Course class. A Course class has properties like name, course code, and the maximum number of students. This class represents a Course's blueprint. We can use it to create courses such as iOS Programming with course code IPC101, Cooking with course code CC101, and so on. In this case, the objects of the Course class are "iOS Programming course" and "Cooking course." They are also known as instances of the Course class. To keep things simple, consider instance to be another word for an object.

RETURN TO THE CUSTOMIZE TABLE CELL

So, why are we discussing OOP today? There is no better way to learn to program than by doing. Let's go over the "Customize Table Cell" option again.

In the Table Cell tutorial's viewDidLoad method, we created three arrays to store different types of data: the Recipe name, a thumbnail of the

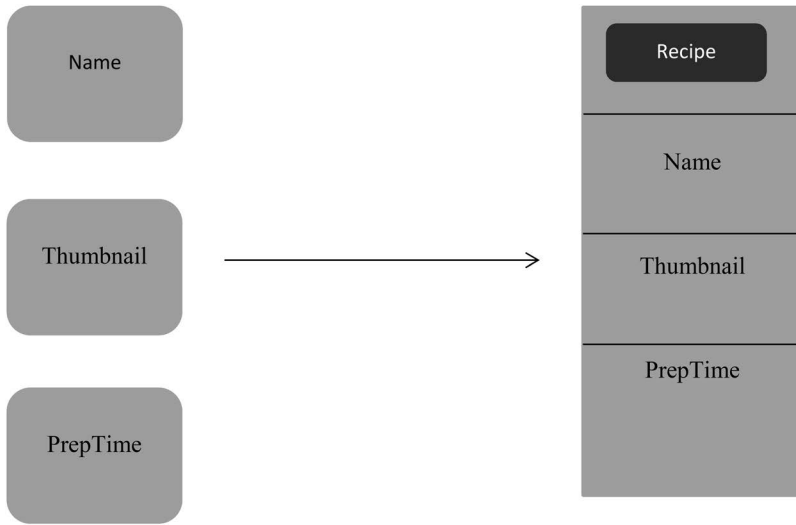
Recipe, and the preparation time. If we understand the OOP concept, we can think of these data as Recipe properties.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Initialize table data
    tableData = [NSArray arrayWithObjects:@"Egg
Benedict", @"Cheese Mushroom Risotto", @"Full
Breakfast", @"Hamburger", @"Ham and Eggwhite
Sandwich", @"Creme Brelee", @"White Dark Chocolate
Donut", @"Starbucks Coffee", @"Vegetable Curry",
@"Instant Noodle with Egg", @"Noodle with BBQ Pork",
@"Japanese Noodle with Pork", @"Green Tea", @"Thai
Shrimp Cake", @"Angry Birds Cake", @"Ham and Cheese
Panini", nil];

    // Initialize-thumbnails
    thumbnails = [NSArray arrayWithObjects:@"egg_
benedict.jpg", @"cheese_mushroom_risotto.jpg", @"full_
breakfast.jpg", @"hamburger.jpg",
@"ham_and_eggwhite_sandwich.jpg", @"creme_brelee.jpg",
@"white_dark_chocolate_donut.jpg", @"starbucks_coffee.
jpg", @"vegetable_curry.jpg", @"instant_noodle_with_
egg.jpg", @"noodle_with_bbq_pork.jpg", @"japanese_
noodle_with_pork.jpg", @"green_tea.jpg",
@"thai_shrimp_cake.jpg", @"angry_birds_cake.jpg",
@"ham_and_cheese_panini.jpg", nil];

    // Initialize the Preparation Time
    prepTime = [NSArray arrayWithObjects:@"35 min",
@"35 min", @"25 min", @"35 min", @"15 min", @"1 hour",
@"40 min", @"5 min", @"35 min", @"8 min", @"25 min",
@"25 min", @"5 min", @"1.5 hour", @"4 hours",
@"15 min", nil];
}
```

Given that the name (i.e., tableData), thumbnail (i.e., Recipe image), and preparation time (i.e., prepTime) are all related to a Recipe, it's better to create a Recipe class to model a Recipe rather than storing these data in separate arrays.



Class of Recipe.

MAKING THE RECIPE CLASS

First and foremost, we'll create the Recipe Class with the help of the following steps:

- Right-click the SimpleTable folder and choose “New File...”.
- Click “Next” after selecting the “Objective-C class” template (under Cocoa Touch).
- Make the class “Recipe” a subclass of “NSObject.”
- Save the file in our Xcode project folder by clicking “Next.”
- When finished, Xcode will generate the Recipe.h and Recipe.m files.

Add the following Recipe class attributes to the header file:

```
@interface Recipe : NSObject

@property (nonatomic, strong) NSString *name; // name
of the recipe
@property (nonatomic, strong) NSString *prep_Time;
// preparation-time
```

```
@property (nonatomic, strong) NSString *imageFile;
// image filename of the recipe

@end
```

The `@synthesis` directive is added to the implementation file (i.e., `Recipe.m`). The `@synthesize` directive instructs the compiler to create setters and getters for the properties defined in the header.

```
@implementation Recipe

@synthesize name;
@synthesize prep_Time;
@synthesize imageFile;

@end
```

We now have a `Recipe` class with three attributes. We'll need it later to instantiate other `Recipe` objects. So, how can we turn the arrays into a `Recipe` object array? We can use the new `Recipe` class to combine the arrays into a single array of `Recipe` objects, with each object storing the `Recipe` data.

Set up the Array of Recipe Objects

Let's get back to the code. Instead of defining three arrays (`tableData`, `thumbnails`, and `prepTime`), we'll use a single variable to represent the array of `Recipe` objects.

```
@implementation SimpleTableViewController
{
    NSArray *recipes;
}
```

We initialize the `Recipe` objects (a total of 16 `Recipe` objects) and place them in the “Recipes” array in the `viewDidLoad` function of `SimpleTableViewController.m`.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
```

10 ■ Mastering Objective-C

```
Recipe *recipe1 = [Recipe new];
recipe1.name = @"Egg Benedict";
recipe1.prep_Time = @"30 min";
recipe1.imageFile = @"egg_benedict1.jpg";

Recipe *recipe2 = [Recipe new];
recipe2.name = @"Mushroom Risotto";
recipe2.prep_Time = @"30 min";
recipe2.imageFile = @"mushroom_risotto1.jpg";

Recipe *recipe3 = [Recipe new];
recipe3.name = @"Full Breakfast";
recipe3.prep_Time = @"20 min";
recipe3.imageFile = @"full_breakfast1.jpg";

Recipe *recipe4 = [Recipe new];
recipe4.name = @"Hamburger";
recipe4.prep_Time = @"30 min";
recipe4.imageFile = @"hamburger1.jpg";

Recipe *recipe5 = [Recipe new];
recipe5.name = @"Ham and Egg Sandwich";
recipe5.prep_Time = @"10 min";
recipe5.imageFile = @"ham_and_egg_sandwich1.jpg";

Recipe *recipe6 = [Recipe new];
recipe6.name = @"Creme Brelee";
recipe6.prep_Time = @"1 hour";
recipe6.imageFile = @"creme_brelee1.jpg";

Recipe *recipe7 = [Recipe new];
recipe7.name = @"White Chocolate Donut";
recipe7.prep_Time = @"45 min";
recipe7.imageFile = @"white_chocolate_donut1.jpg";

Recipe *recipe8 = [Recipe new];
recipe8.name = @"Starbucks Coffee";
recipe8.prep_Time = @"5 min";
recipe8.imageFile = @"starbucks_coffee1.jpg";

Recipe *recipe9 = [Recipe new];
recipe9.name = @"Vegetable Curry";
recipe9.prep_Time = @"30 min";
recipe9.imageFile = @"vegetable_curry1.jpg";
```

```

Recipe *recipe10 = [Recipe new];
recipe10.name = @"Instant Noodle with Egg";
recipe10.prep_Time = @"8 min";
recipe10.imageFile = @"instant_noodle_with_egg1
.jpg";

Recipe *recipe11 = [Recipe new];
recipe11.name = @"Noodle with BBQ Pork";
recipe11.prep_Time = @"20 min";
recipe11.imageFile = @"noodle_with_bbq_pork1.jpg";

Recipe *recipe12 = [Recipe new];
recipe12.name = @"Japanese Noodle with Pork";
recipe12.prep_Time = @"20 min";
recipe12.imageFile = @"japanese_noodle_with_pork1
.jpg";

Recipe *recipe13 = [Recipe new];
recipe13.name = @"Green Tea";
recipe13.prep_Time = @"5 min";
recipe13.imageFile = @"green_tea1.jpg";

Recipe *recipe14 = [Recipe new];
recipe14.name = @"Thai Shrimp Cake";
recipe14.prep_Time = @"1.5 hours";
recipe14.imageFile = @"thai_shrimp_cake1.jpg";

Recipe *recipe15 = [Recipe new];
recipe15.name = @"Angry Birds Cake";
recipe15.prep_Time = @"4 hours";
recipe15.imageFile = @"angry_birds_cake1.jpg";

Recipe *recipe16 = [Recipe new];
recipe16.name = @"Ham and Cheese Panini";
recipe16.prep_Time = @"10 min";
recipe16.imageFile = @"ham_and_cheese_panini1
.jpg";

    recipes = [NSArray arrayWithObjects:recipe1,
recipe2, recipe3, recipe4, recipe5, recipe5, recipe6,
recipe7, recipe8, recipe9, recipe10, recipe11,
recipe12, recipe13, recipe14, recipe15, recipe16,
nil];

}

```

To create an object in Objective-C, we utilize the “new” method (a method supplied by NSObject). We have two options for determining the property’s worth. We utilize the dot syntax in the preceding code to assign the value. As an example,

```
recipe1.name = @"Egg Benedict";
```

We may also use square bracket notation to call the `set_Name` function (`[]`). The comparable code is as follows:

```
[recipe1 set_Name:@"Egg Benedict"];
```

Both syntaxes accomplish the same task. However, we’ll be using the dot syntax throughout our sessions.

A Recipes Array Is Replacing TableData

A few things must be changed in order to use the Recipes array. We replace “tableData” to “Recipes” for the `numberOfRowsInSection` method:

```
- (NSInteger)tableView:(UITableView *)tableView number
OfRowsInSection:(NSInteger) section
{
// DELETE LINE    return [tableData count];
    return [recipes count];
}
```

We replace “tableData,” “thumbnails,” and “prep_Time” with the Recipe array in the “`cellForRowAtIndexPath`” method:

```
- (UITableViewCell *)tableView:(UITableView *)
tableView cellForRowAtIndexPath:(NSIndexPath *)
indexPath
{
    static NSString *simpleTableIdentifier =
@"SimpleTableCell";

    SimpleTableCell *cell = (SimpleTableCell *)
[tableView dequeueReusableCellWithIdentifier:simpleTab
leIdentifier];
    if (cell == nil)
```

```

    {
        NSArray *nib = [[NSBundle mainBundle]
loadNibNamed:@"SimpleTableViewCell" owner:self
options:nil];
        cell = [nib objectAtIndex:0];
    }

    /* DELETE SECTION
    cell.nameLabel.text = [tableData
objectAtIndex:indexPath.row];
    cell.thumbnailImageView.image = [UIImage
imageName:[thumbnails objectAtIndex:indexPath.row]];
    cell.prep_TimeLabel.text = [prep_Time
objectAtIndex:indexPath.row];
    */

    // Locate current recipe object and assign label,
image & prep_Time
    Recipe *recipe = [recipes objectAtIndex:indexPath.
row];
    cell.nameLabel.text = recipe.name;
    cell.thumbnailImageView.image = [UIImage
imageName:recipe.imageFile];
    cell.prep_TimeLabel.text = recipe.prep_Time;

    return cell;
}

```

As we can see from the code, changing the three data arrays into the Recipes array improves readability and comprehension. Our app is now ready to use. The style and feel are identical to what we created in the “Custom Table Cell.” However, we beautify our code internally by building our Recipe object.

WHAT DOES A COMPILER DO?

In this part, we’ll look at what a compiler does and how we may utilize it to our advantage.

In general, the compiler has two tasks: translating our Objective-C code into low-level code and assessing our code to ensure there are no apparent errors.

Xcode now includes Clang as the compiler. We may read Clang anywhere we write a compiler.

Clang is a tool that analyzes Objective-C code and converts it to a lower-level representation that mimics assembly code: LLVM Intermediate Representation (LLVM IR).

LLVM IR is low-level and independent of operating systems. LLVM converts instructions into native bytecode for the target platform. This may be done either just in time or concurrently with compilation.

The advantage of having those LLVM instructions is that you can produce and execute them on any platform that LLVM supports. For example, suppose we develop an iOS app. In that case, it will automatically operate on two entirely different architectures (Intel and ARM), and LLVM will translate the IR code into native bytecode for both systems.

LLVM has a three-tier design, which means it supports a wide range of input languages (including C, Objective-C, and C++, as well as Haskell) in the first tier, a shared optimizer in the second tier (which optimizes the LLVM IR), and many targets in the third tier (e.g., Intel, ARM, and PowerPC). If we want to add a language, concentrate on the top layer; if we want to add another compilation target, don't worry too much about the input languages.

The compiler goes through various stages while compiling a source file.

To illustrate the various steps, we may ask Clang to build the file `helloo.m`:

```
% clang -ccc-print-phases helloo.m

0: input, "helloo.m", objective-c
1: preprocessor, {0}, objective-c-cpp-output
2: compiler, {1}, assembler
3: assembler, {2}, object
4: linker, {3}, image
5: bind-arch, "x86_64", {4}, image
```

PREPROCESSING

When compiling a source file, preprocessing is the first thing that happens. The preprocessor handles a macro processing language, which means it will substitute macro definitions in our text. For instance, consider the following:

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

The preprocessor will take that line with the contents of the file. Other macro definitions in that header file will replace as well.

This is why many advise us to keep our header files as clean of imports as possible, because every time we import something, the compiler needs to do more work. In our header file, for example, instead of writing:

```
#import "My_Class.h"
```

We might say:

```
@class My_Class;
```

And by doing so, we guarantee the compiler that a class, `MyClass`, will exist. The `My_Class.h` file may then be imported and used in the implementation file (the `.m` file).

Assume we have a basic pure C program called `helloo.c`:

```
#include <stdio.h>

int main() {
    printf("hello everyone\n");
    return 0;
}
```

We can try the preprocessor on this to see what happens:

```
clang -E helloo.c | less
```

Take a peek at that code now. If we add the following line at the beginning:

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

We can rerun the script and observe that our file has grown. While operating systems are created with fewer than a thousand lines of code.

Fortunately, the situation has just improved slightly. Modules a new feature, making this procedure a little more high level.

CUSTOM MACROS

Another case in point is when we develop or utilize custom macros, such as this:

```
#define MY_CONSTANT 4
```

Now, whenever we type `MY_CONSTANT` after this line, it will be substituted by 4 before the rest of the compilation begins. More interesting macros that take arguments can also be defined:

```
#define MY_MACRO(x) x
```

This chapter is too short to cover the whole spectrum of what the preprocessor can do, but it is a highly powerful tool. The preprocessor is frequently used to inline code. This is severely discouraged by us. Assume you have the following seemingly innocuous program:

```
#define MAX(x,y) x > y? x : y

int main() {
    printf("largest: %d\n", MAX(10,110));
    return 0;
}
```

This will work perfectly. But how about the following program:

```
#define MAX(x,y) x > y? x : y

int main() {
    int c = 210;
    printf("largest: %d\n", MAX(c++,110));
    printf("c: %d\n", c);
    return 0;
}
```

We get the following result when we build this using clang `max.c`.

This becomes clear when we run the preprocessor and expand all the macros with clang `-E max.c`:

```
int main() {
    int c = 210;
    printf("largest: %d\n", c++ > 110? c++ : 110);
    printf("c: %d\n", c);
    return 0;
}
```

This is an apparent illustration of what may go wrong with macros, but things can also go wrong in unanticipated and difficult-to-debug ways. We should use static inline functions instead of macros:

```
#include <stdio.h>

static const int MyConstant = 210;

static inline int max(int l, int r) {
    return l > r? l : r;
}

int main() {
    int c = MyConstant;
    printf("largest: %d\n", max(c++, 110));
    printf("c: %d\n", c);
    return 0;
}
```

This will provide the desired outcome (c: 201). Because the code is inlined, it will perform similarly to the macro variation but will be far less error-prone. We may also use breakpoints, type checking, and avoid unexpected behavior.

The only time macros make sense is for logging, when we may use `__FILE__`, `__LINE__`, and `assert` macros.

TOKENIZATION (LEXING)

After preprocessing, each source.m file now contains several definitions. This text is transformed from a string to a token stream. In the case of a basic Objective-C hello world application, for example:

```
int main() {
    NSLog(@"hello, %@", @"everyone");
    return 0;
}
```

By executing the command `clang -Xclang -dump-tokens`, we may request that Clang spill the tokens for this application. `helloo.m`:

```
int 'int' [StartOfLine] Loc=<helloo.m:4:1>
identifier 'main' [LeadingSpace]
Loc=<helloo.m:4:5>
```

```

l_paren '('                               Loc=<helloo.m:4:9>
r_paren ')'                               Loc=<helloo.m:4:10>
l_brace '{' [LeadingSpace] Loc=<helloo.m:4:12>
identifier 'NSLog' [StartOfLine] [LeadingSpace]
Loc=<helloo.m:5:3>
l_paren '('                               Loc=<helloo.m:5:8>
at '@' Loc=<helloo.m:5:9>
string_literal '"helloo, %@"'
Loc=<helloo.m:5:10>
comma ',' Loc=<helloo.m:5:21>
at '@' [LeadingSpace] Loc=<helloo.m:5:23>
string_literal '"Everyone"'
Loc=<helloo.m:5:24>
r_paren ')'                               Loc=<helloo.m:5:31>
semi ';' Loc=<helloo.m:5:32>
return 'return' [StartOfLine] [LeadingSpace]
Loc=<helloo.m:6:3>
numeric_constant '0' [LeadingSpace]
Loc=<helloo.m:6:10>
semi ';' Loc=<helloo.m:6:11>
r_brace '}' [StartOfLine] Loc=<helloo.m:7:1>
eof '' Loc=<helloo.m:7:2>

```

Each token, as we can see, consists of a bit of text and a source location. The source location is from before macro expansion, so Clang will direct us to the correct position if something goes wrong.

PARSING

The fun part comes: our token stream is processed into an abstract syntax tree. Because Objective-C is a complex language, parsing is not always simple. After parsing, a program is now available as an abstract syntax tree: a tree that reflects the original program. Assume we have the program `hello.m`:

```

#import <Foundation/Foundation.h>

@interface Everyone
- (void)helloo;
@end

@implementation Everyone
- (void)helloo {

```

```

    NSLog(@"helloo, everyone ");
}
@end

int main() {
    Everyone * everyone = [Everyone new];
    [everyone helloo];
}

```

When we run `clang -Xclang -ast-dump -fsyntax-only helloo.m`, the following message appears on the command line:

```

@interface Everyone - (void) helloo;
@end

@implementation Everyone
- (void) helloo (CompoundStmt 0x10372ded0
<helloo.m:8:15, line:10:1>
    (CallExpr 0x10372dea0 <line:9:3, col:24> 'void'
    (ImplicitCastExpr 0x10372de88 <col:3> 'void (*)
(NSString *, ...)' <FunctionToPointerDecay>
    (DeclRefExpr 0x10372ddd8 <col:3> 'void (NSString
*, ...)' Function 0x1023510d0 'NSLog' 'void (NSString
*, ...)''))
    (ObjCStringLiteral 0x10372de38 <col:9, col:10>
'NSString *'
    (StringLiteral 0x10372de00 <col:10> 'char [13]'
lvalue "hello, world"))))

@end

int main() (CompoundStmt 0x10372e118 <helloo.m:13:12,
line:16:1>
    (DeclStmt 0x10372e090 <line:14:4, col:30>
    0x10372dfe0 "Everyone *everyone =
    (ImplicitCastExpr 0x10372e078 <col:19, col:29>
'Everyone *' <BitCast>
    (ObjCMessageExpr 0x10372e048 <col:19, col:29>
'id':'id' selector=new class='Everyone'))")
    (ObjCMessageExpr 0x10372e0e8 <line:15:4, col:16>
'void' selector=hello
    (ImplicitCastExpr 0x10372e0d0 <col:5> 'Everyone *'
<LValueToRValue>
    (DeclRefExpr 0x10372e0a8 <col:5> 'Everyone *'
lvalue Var 0x10372dfe0 'everyone' 'Everyone *'))))

```

Every node in the abstract syntax tree is tagged with the source position so that if a problem arises later on, Clang may warn us and point us in the right direction.

STATIC ANALYSIS

Once the compiler has an abstract syntax tree, it may analyze it to find faults, such as type checking, which verifies whether our program types accurately.

When we send a message to an object, for example, it checks to see whether the object truly implements the message.

Clang also does additional in-depth assessments, such as walking through our program to ensure we're not doing anything strange.

TYPE CHECKING

When we're writing code, Clang is there to ensure we didn't make any errors.

One of the most obvious things to verify is whether our program delivers the appropriate messages to the appropriate objects and calls the appropriate functions on the data.¹

Since Clang would report an error, we can't simply send the welcome message to a normal `NSObject*`.

Additionally, if we construct a `Test` class that subclasses `NSObject`, such as this:

```
@interface Test : NSObject
@end
```

If we attempt to assign an object of a different type to that object, the compiler will assist us and tell us that we're probably doing something wrong.

Dynamic and static typing are the two forms of typing.

Static typing implies that types are verified at compile-time, while dynamic typing means that types are checked during runtime.

We could send any message to any object in the past, and it would be decided if the object responded at runtime.

It's termed dynamic typing when this is only tested at runtime.

This is tested at build time using static typing.

Because it has to know which objects it interacts with, the compiler examines a lot more types at build time when we use ARC.

For instance, we can no longer write the following code:

```
[myObject helloo]
```

OTHER ANALYSES

Clang does a variety of different analyses for you. Cloning the Clang repository and going to `lib/StaticAnalyzer/Checkers` will show us all the static checkers. `ObjCUnusedIVarsChecker.cpp`, for example, checks if ivars are unneeded. There's also `ObjCSelfInitChecker.cpp`, which checks if we called `[self initWith...]` or `[super init]` before using `self` within our initializer. Different tests are performed in other portions of the compiler. Line 2,534 of `lib/Sema/SemaExprObjC.cpp`, for example, has the following code:

```
Diag(SelLoc, diag::warn_arc_perform_selector_leaks);
```

This results in the dreaded “performSelector may create a leak since its selector is unknown” error message.

CREATING CODE

Clang can now create LLVM code for us once it has properly tokenized, parsed, and evaluated our code. To observe what occurs, we may revisit the program `hello.c`:

```
#include <stdio.h>

int main() {
    printf("hello everyone\n");
    return 0;
}
```

We can use the following command to compile this into LLVM IR:

```
clang -O3 -S -emit-llvm helloo.c -o helloo.ll
```

This produces a `helloo.ll` file with the following output:

```
; ModuleID = 'helloo.c'
target datalayout =
"e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.9.0"
```

22 ■ Mastering Objective-C

```
@str = private unnamed_addr constant [12 x i8] c"hello
everyone\00"
```

```
; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
    %puts = tail call i32 @puts(i8* getelementptr
inbounds ([12 x i8]* @str, i64 0, i64 0))
    ret i32 0
}
```

```
; Function Attrs: nounwind
declare i32 @puts(i8* nocapture readonly) #1
```

```
attributes #0 = { nounwind ssp uwtable "less-precise-
fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-
math"="false" "no-nans-fp-math"="false" "stack-
protector-buffer-size"="8" "unsafe-fp-math"="false"
"use-soft-float"="false" }
attributes #1 = { nounwind }
```

```
!llvm.ident = !{!0}
```

```
!0 = metadata !{metadata !"Apple LLVM version 6.0
(clang-600.0.41.2) (based on LLVM 3.5svn)"}
```

The main function is merely two lines long: one to display the string and one to return 0.

It's also fun to do the same thing with a very basic Objective-C program `five.m`, which we build and then inspect using `LLVM-dis <five.bc | less`:

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

int main() {
    NSLog(@"%@", [@"5 description"]);
    return 0;
}
```

There's a lot going on around here, but here's the main function:

```
define i32 @main() #0 {
    %1 = load %struct._class_t** @"_01L_OBJC_
CLASSLIST_REFERENCES_$_", align 8
```

```

    %2 = load i8** @"\01L_OBJC_SELECTOR_REFERENCES_",
align 8, !invariant.load !5
    %3 = bitcast %struct._class_t* %1 to i8*
    %4 = tail call %0* bitcast (i8* (i8*, i8*, ...) *
@objc_msgSend to %0* (i8*, i8*, i32)*) (i8* %3, i8* %2,
i32 5)
    %5 = load i8** @"\01L_OBJC_SELECTOR_REFERENCES_2",
align 8, !invariant.load !5
    %6 = bitcast %0* %4 to i8*
    %7 = tail call %1* bitcast (i8* (i8*, i8*, ...) *
@objc_msgSend to %1* (i8*, i8*)*) (i8* %6, i8* %5)
    tail call void (i8*, ...) * @NSLog(i8* bitcast
(%struct.NSConstantString* @_unnamed_cfstring_ to
i8*), %1* %7)
    ret i32 0
}

```

Line 4, which creates the NSNumber object, line 7, which sends the description message to the number object, and line 8, which records the string returned from the description message, are the essential lines.

OPTIMIZATIONS

It's worth looking at a somewhat more sophisticated C example, the recursively defined factorial function, to understand what type of optimizations LLVM and Clang can perform:

```

#include <stdio.h>

int factorial(int c) {
    if (c > 1) return c * factorial(c-1);
    else return 1;
}

int main() {
    printf("The factorial 10: %d\n", factorial(10));
}

```

Run the following command to compile it without optimizations:

```
clang -O0 -S -emit-llvm factorial.c -o factorial.ll
```

The code generated for the factorial function is very interesting:

```
define i32 @factorial(i32 %x) #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    store i32 %x, i32* %2, align 4
    %3 = load i32* %2, align 4
    %4 = icmp sgt i32 %3, 1
    br i1 %4, label %5, label %11

; <label>:5                                ; preds = %0
    %6 = load i32* %2, align 4
    %7 = load i32* %2, align 4
    %8 = sub nsw i32 %7, 1
    %9 = call i32 @factorial(i32 %8)
    %10 = mul nsw i32 %6, %9
    store i32 %10, i32* %1
    br label %12

; <label>:11                                ; preds = %0
    store i32 1, i32* %1
    br label %12

; <label>:12                                ; preds = %11, %5
    %13 = load i32* %1
    ret i32 %13
}
```

We can see that it calls itself recursively on the line highlighted with % 9. Because the stack grows with each recursive call, this is highly inefficient. We may enable optimizations by passing the parameter -O3 to Clang:

```
clang -O3 -S -emit-llvm factorial.c -o factorial.ll
```

The factorial function's code now looks like this:

```
define i32 @factorial(i32%x) #0 {
    %1 = icmp sgt i32%x, 1
    br i1%1, label %tailrecurse, label
%tailrecurse._crit_edge

tailrecurse:                                ; preds = %tailrecurse, %0
    %x.tr2 = phi i32 [ %2, %tailrecurse ], [ %x, %0 ]
```

```

    %accumulator.tr1 = phi i32 [ %3, %tailrecurse ],
[ 1, %0 ]
    %2 = add nsw i32%x.tr2, -1
    %3 = mul nsw i32%x.tr2, %accumulator.tr1
    %4 = icmp sgt i32%2, 1
    br i1%4, label %tailrecurse, label
%tailrecurse._crit_edge

tailrecurse._crit_edge:          ; preds =
%tailrecurse, %0
    %accumulator.tr.lcssa = phi i32 [ 1, %0 ],
[ %3, %tailrecurse ]
    ret i32%accumulator.tr.lcssa
}

```

OBJECTIVE-C DEVELOPERS' C# PRIMER

“Xamarin.iOS” enables platform-independent C# code to be shared between platforms.

Existing iOS apps, on the other hand, may desire to use Objective-C code that has already been written.

This chapter serves as a quick introduction to Xamarin and the C# language for Objective-C developers.

Apps written in Objective-C for iOS and macOS may benefit from Xamarin by using C# in areas where platform-specific code isn’t needed, enabling them to run on non-Apple devices.

Web services, JSON and XML parsing, and custom algorithms, for example, may all be utilized cross-platform.

To use Xamarin while keeping current Objective-C assets, the latter may be accessible to C# through bindings. This Xamarin technique exposes Objective-C code to the managed C# world.

Code may also be converted to C# line by line if required.

To properly use existing Objective-C code with “Xamarin.iOS,” regardless of the technique, whether binding or porting, some understanding of both Objective-C and C# is required.

INTEROP OBJECTIVE-C

There is currently no approved technique for creating a C# library that can be called from Objective-C using Xamarin. iOS.

The primary reason for this is that the Mono runtime is also needed in addition to the binding. Most of our logic, including user interfaces,

may still be written in Objective-C. Encapsulate the Objective-C code in a library and establish a binding to it.

“Xamarin.iOS”s is required to start the app (meaning it must create the Main entry point). Any other functionality may then be written in Objective-C and provided to C# through the binding (via P/Invoke). In this manner, we can build platform-agnostic sections in C# while keeping the platform-specific logic in Objective-C. This section compares and contrasts some significant similarities and differences between the two languages to serve as a primer for transitioning to C# using “Xamarin.iOS,” whether binding to existing Objective-C code or converting it to C#.

COMPARISON OF LANGUAGES

Both syntactically and in terms of runtime, Objective-C and C# are pretty distinct languages. C# is statically typed, while Objective-C is dynamic and employs a message-passing system.

Objective-C is similar to Smalltalk in terms of syntax; while C# borrows most of its basic grammar from Java, it has grown to incorporate many features beyond Java in recent years. However, numerous language features are functionally equivalent in both Objective-C and C#. Understanding these commonalities is essential when establishing a binding to Objective-C code from C# or when converting Objective-C to C#.

INTERFACES VS. PROTOCOLS

Single inheritance languages are both Objective-C and C#. On the other hand, both languages allow us to implement numerous interfaces in a single class. These logical interfaces are referred to as protocols in Objective-C and interfaces in C#. The critical difference between a C# interface and an Objective-C protocol in terms of implementation is that the latter might have optional methods.

EXTENSION METHODS VS CATEGORIES

Using Categories, Objective-C enables us to add methods to a class for which we may not have the implementation code.

Extension methods, analogous to extension methods in C#, provide a similar notion. Static methods in C# are equivalent to class methods in Objective-C, and extension methods allow us to add them to a class.

ASSEMBLIES VS. FRAMEWORKS

Frameworks are specific folders in which Objective-C organizes similar classes. Assemblies, on the other hand, are used in C# and .NET to offer

reusable chunks of precompiled code. Assembly files in systems other than iOS include intermediate language code (IL) compiled just in time (JIT) during runtime.

On the other hand, Apple does not enable JIT-compiled code to be executed in App Store-released iOS apps. As a result, Xamarin compiles C# code for iOS ahead of time (AOT), resulting in a single Unix executable and metadata files included in the application bundle.

NAMED PARAMETERS VS. SELECTORS

By their very nature, Objective-C methods incorporate parameter names in selectors. When used in code, a selection like `AddCrayon:WithColor:` makes it apparent what each argument implies.

NAMESPACES AND HEADERS

Objective-C, as a superset of C, employs headers for public declarations distinct from the implementation file. C# does not make use of header files. C# code, unlike Objective-C code, is enclosed in namespaces. If we wish to incorporate code from another namespace, add a `using` directive to the start of the implementation file, or qualify the type with the whole namespace.

PROPERTIES

The idea of properties is used in both Objective-C and C# to offer a high-level abstraction around accessor methods. The `@property` compiler directive is used in Objective-C to build accessor methods effectively. C#, on the other hand, comes with built-in support for properties.

STATIC KEYWORD

The static keyword in Objective-C and C# have pretty different meanings.

Static functions are used in Objective-C to restrict the scope of a function to the current file. The `public`, `private`, and `internal` keywords in C#, on the other hand, keep the scope under check. When we use the static keyword on a variable in Objective-C, it keeps its value across function calls. A static keyword is also available in C#.

When applied to a method, it performs the same function as the `+` modifier in Objective-C. It does this by creating a class method. When applied to other constructs like fields, properties, and events, it makes them part of the type they're defined in, rather than any instance of that type. We may also create a static class, which requires all of the class's methods to be static.²

LIST VS NSARRAY INITIALIZATION

The addition of literal syntax for usage with NSArray in Objective-C makes it easier to initialize. C#, on the other hand, C # has a richer type called a List that is generic, meaning that the type the list carries may be specified by the code that constructs the list (similar to templates in C++).

LAMBDA EXPRESSIONS VS BLOCKS

Objective-C employs blocks to generate closures, allowing us to create an inline function that may use the contained state. C# uses lambda expressions to implement a similar approach.

This chapter covered the Objective-C overview with the framework, advantages, disadvantages, compiler, and differentiation from other languages.

NOTES

1. What does a compiler do?
2. C# primer for Objective-C developers.

Environment Setup and Program Structure

IN THIS CHAPTER

- Local Environment Setup
- Text Editor
- The GCC Compiler
- Installation
- Hello World Example
- Compile and Execute Program

In the previous chapter, we covered Objective-C Overview, and in this chapter, we will cover the environment setup and structure of the program.

CONFIGURING THE OBJECTIVE-C ENVIRONMENT

CONFIGURATION OF THE LOCAL ENVIRONMENT

We must install Text Editor and The GCC Compiler on our computer if we want to develop our environment for the Objective-C programming language.

EDITOR OF TEXT

This is the location where we will type our program. Windows Notepad, the OS Edit command, Brief, Epsilon, EMACS, and vim or vi are among the editors.

The name and version of the text editor may change between operating systems. Notepad will be used on Windows, while vim or vi will be used on both Windows and Linux or UNIX.

We create source files with our editor that include program source code. The suffix “.m” is often used for Objective-C source files.

Before we begin programming, make sure we have a text editor and the expertise to create a computer program, save it in a file, compile it, and finally run it.

GCC COMPILER

The source code for our software is written in a source file and is human readable. Before our CPU can run the program as intended, it must be “compiled” into machine code.

Use the GCC compiler to turn our source code into an executable program. We’re going to assume you’re familiar with programming language compilers.

The GCC compiler is freely accessible on various platforms, and the procedure for installing it on those platforms is documented here.

UNIX/LINUX INSTALLATION

The first step is to install gcc and the gcc Objective-C package. This is accomplished by:

```
$ su -
$ yum install gcc
$ yum install gcc-objc
```

To specify package dependencies, use the following command:

```
$ yum install make libpng libpng-devel libtiff
libtiff-devel libobjc
libxml2 libxml2-devel libX11-devel libXt-devel
libjpeg libjpeg-devel
```

Download and install GNUStep to have access to all Objective-C features. We may get the package by going to <http://wwwmain.gnustep.org/resources/downloads.php>.

We must now browse to the downloaded location and unpack the file with:

```
$ tar xvfz gnustep-startup-.tar.gz
```

Now we must access the gnustep-startup folder, which was created by using:

```
$ cd gnustep-startup-<version>
```

After that, the construction technique must be configured.

```
$ ./configure
```

Then we may build by:

```
$ make
```

We will eventually have to design the environment.

```
$ . /usr/GNUstep/System/Library/Makefiles/GNUstep.sh
```

We have an Objective-C file called `heyworld.m` that looks like this:

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

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

    NSLog (@"hey world");
    [pool drain];
    return 0;
}
```

We can now create and run an Objective-C file, say `heyworld.m`, by using `cd` to get to the file's directory and then doing the following steps:

```
$ gcc 'gnustep-config --objc-flags'
-L/usr/GNUstep/Local/Library/Libraries
-lgnustep-base heyworld.m -o heyworld
$ ./heyworld
```

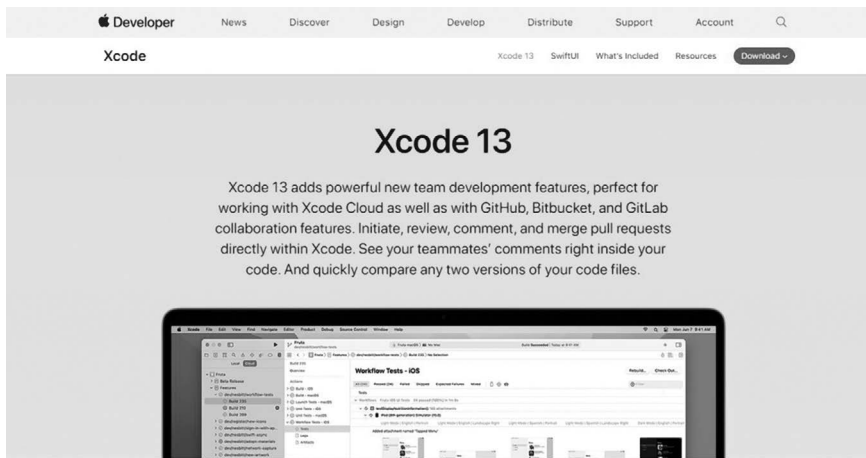
Mac OS INSTALLATION

Xcode may or may not pre-install on our Mac OS X machine.

To find out whether we already have it, use the Finder and search for it in the Developer subdirectory of the Applications folder.

If the Developer folder does not exist or does not include Xcode then you will need to install it. The simplest approach to acquiring Xcode is to download it from the Apple website.

The URL to download Xcode is <https://developer.apple.com/xcode/>.



Xcode on Mac installation.

We will either need to be a member of the iOS Developer Community or purchase a copy from the Mac App Store to get Xcode 4.3 with the iOS 5 SDK. The download is over 3.5GB in size and may take a number of hours to finish depending on the speed of our internet connection.

LAUNCHING Xcode

After successfully installing the SDK and Xcode, the following step is to activate them to develop and then construct an example iPhone application. To launch Xcode, open the Finder and look for Xcode. Because we will be using this tool frequently, utilize this chance to drag and drop it into our dock for future convenience. To start the tool, click the Xcode icon on the dock.

Once Xcode has loaded and assuming this is our first time using Xcode on this machine, we will get the Welcome page, from which we can proceed.

To enter the screen where we may choose a template, click the drop-down menu and choose Create a new Xcode project.

Select the Application item displayed under Mac OS X in the left-hand panel, then Command Line Tool in the main panel, from the template selection screen.

Click Next, then name the project sampleApp and choose Foundation from the Type selection on the accompanying settings screen.

Make sure the Use Automatic Reference Counting option is turned on.

An Objective-C programmer was responsible for keeping and releasing objects in application code until Automatic Reference Counting (ARC) functionality was added to current versions of Apple's compiler.

To monitor memory utilization, this usually meant manually inserting keep and release method calls to programs.

Failure to release an object caused memory leaks (in which the memory use of a running program rises over time) while removing an object too soon caused an application to crash.

Apple's LLVM compiler implements ARC, which reads the source code and automatically inserts necessary retain and release calls before building it, making the life of an Objective-C programmer considerably simpler.

Before clicking the Create button, go to the next page and pick an appropriate place for the project to be generated on the local file system.

Xcode will create a new project and open the main Xcode window.

WINDOWS INSTALLATION

We must first install MinGW and GNUStep Core before we can execute an Objective-C program on Windows. Both may download at <https://www.gnu.org/software/gnustep/windows/installer.html>.

First, we must install the MSYS/MinGW System package. The GNUStep Core package is then installed with a self-explanatory Windows setup.

Then, to use Objective-C with GNUStep, go to Start -> All Programs -> GNUStep -> Shell.

Navigate to the folder heyworld.m.

To compile the code, we might use.

```
$ gcc 'gnustep-config --objc-flags'
-L /GNUstep/System/Library/Libraries heyworld.m
-o hello -lgnustep-base -lobjc
```

STRUCTURE OF THE OBJECTIVE-C PROGRAM

Before delving into the core building parts of the Objective-C programming language, let's look at a basic minimum Objective-C program structure that we may use as a guide in the following chapters.

Objective-C Example of Hello Everyone

A basic Objective-C program has the following elements:

- Preprocessor Commands
- Interface
- Comments
- Method
- Implementation
- Variables
- Statements & Expressions

Consider a simple code that prints the words “Hey World.”

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

@interface SampleClass:NSObject
- (void)sampleMethod;
@end

@implementation SampleClass

- (void)sampleMethod {
    NSLog(@"Hey World \n");
}

@end

int main() {
    /* first program in the Objective-C */
    SampleClass *sampleClass = [[SampleClass alloc]
init];
    [sampleClass sampleMethod];
    return 0;
}
```

Let's look at some of the code above:

- The first line of the program, `#import <Foundation/Foundation.h>`, is a preprocessor instruction telling an Objective-C compiler to import the `Foundation.h` file before actual compilation.
- Use the following `@interface SampleClass`:
- The interface is created using `NSObject`. It inherits `NSObject`, the base class for all objects.
- `(void)sampleMethod`; shows how to create a method.
- The `@end` line indicates the end of an interface.
- The next line shows how to implement the `SampleClass` interface.
- The `sampleMethod` is defined in the line `(void)sampleMethod`.
- The next line `@end` denotes the end of an implementation.
- The next line, `int main()`, is the program's main function, from which execution begins.
- The next line `/*...*/` is ignored by compiler and was supplied to offer extra program comments. As a result, these lines are referred to as comments in the software.
- The line `NSLog(...)` is an Objective-C function that displays the message "Hello, Everyone" on the screen.
- The next line returns 0; this line ends the `main()` method and returns the number 0.

USING Xcode TO CREATE AN OBJECTIVE-C APPLICATION

Xcode will generate skeleton versions of the files required to develop an Objective-C command-line application. The `.m` filename extension is used to identify Objective-C source files. In our scenario, Xcode has already produced the main source file titled `main.m` and filled it with some basic code that we may alter. To view the code, choose `main.m` from the list of files and drag it into the editing area. The following is the skeleton code:

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

int main(int argc, const char * argv[])
```

```

{
    @autoreleasepool {
        // here insert code
        NSLog(@"Hello World");
    }
    return 0;
}

```

Change the NSLog line so that it says:

```
NSLog(@"This is the first Objective-C App");
```

After making the update, the application must be compiled and executed. Because the code is meant to display a message in the console, the first step is to display the Xcode console window by selecting View -> Debug Area -> Activate Console from the menu. Then, click the Execute option in the Xcode toolbar to run the program. When we pick this option, Xcode will compile the source code and launch the program, displaying the message in the Xcode console box.¹

USING THE COMMAND LINE TO COMPILE OBJECTIVE-C

Although Xcode offers a robust environment that will be helpful for bigger projects, it is a little overkill for developing and executing a small program like the one we've been working with in this chapter.

It's also true that some developers believe development environments like Xcode obstruct their work and prefer to create apps using a simple editor and command-line tools.

After all, this was how all apps were created until integrated development environments became popular.

Although we do not recommend that everyone switch to the vi editor and GNU compiler in place of Xcode, it is good to know that the command line option is available.

The first step in compiling code from the command line is to make sure Xcode Command Line Tools are installed.

To do so, navigate to the Xcode -> Preferences menu item and pick the Downloads category in the preferences box.

If we haven't already installed the Command Line Tools, click the Install button to get started.

Create a file named `hello.m` with the following Objective-C code in our choice text or programming editor:

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

int main(int argc, const char * argv[])
{
    @autoreleasepool {
        // here insert code
        NSLog(@"Hello World");
    }
    return 0;
}
```

Save the file and then, if we aren't already, start a Terminal session, change the directory to the folder containing the source file, and compile the program using the following command:

```
clang -framework Foundation hello.m -o hello
```

If the code compiles without errors, it may be executed as follows:

```
./hello
2012-04-18 13:27:11.621 hello[275:707] Hello World
```

When compared to utilizing Xcode, this appears to be simpler, but keep in mind that the strength of Xcode truly comes through when constructing bigger size projects.

FILENAME EXTENTION

The final two characters of the filename (known as the extension) in Objective-C source files are `.m`. Other widely used filename extensions are shown below:

Extension	Meaning
<code>.c</code>	C language source file
<code>.cc</code> , <code>.cpp</code>	C++ language source file
<code>.h</code>	Header file
<code>.m</code>	Objective-C source file
<code>.mm</code>	Objective-C++ source file
<code>.pl</code>	Perl source file
<code>.o</code>	Object (compiled) file

When we return to our Xcode project window, the right pane displays the contents of the `main.m`, which Xcode generated automatically as a template file for us.

It includes the following lines:

```
//
// main.m
// program1

#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) {
    @autoreleasepool {
        // here insert code
        NSLog (@"Hello, World");
    }
    return 0;
}
```

Within this section, we may make changes to our file. Change the program displayed in the Edit window to match Program 2.1. Comments are lines that begin with two slash characters (`//`); we'll go over them in more detail later.

Our program should now look like this in the edit window (don't worry if our comments don't match).

```
// example of First program
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog (@"Programming is interesting");
    }
    return 0;
}
```

It's now time to compile and run your first application, known as building and executing in Xcode. First, we must disclose a window pane that will display our application's results (output). We may accomplish this most quickly by selecting the center icon in the toolbar under View. Hovering over this symbol displays the text "Hide or show the Debug area." It should note that Xcode will generally reveal the Debug region anytime any data is written to it.

Xcode will now go through the two-step process of first creating and then executing your application if you tap the Run button in the upper left of the toolbar or pick Run from the Product menu.

The latter happens only if your application has no mistakes.

If we make a mistake in our program, we'll see faults marked with red stop signs and exclamation points along the route. These are known as fatal errors, and we won't be able to execute them until we fix them. Yellow triangles with exclamation points are warnings; we may still execute our program with them, but we should inspect and rectify them.

After running the program with all errors cleared, the output from our program should appear in the bottom right pane. Don't be frightened by the lengthy texts that emerge. The bolded output line is the one we're interested.

We've completed the procedural portion of the building and executed our first Xcode application.

The steps for generating a new program using Xcode are summarized below:

1. Launch the Xcode program.
2. Choose File, New, New Project ... or Create a New Xcode Project from the starting screen if this is a new project.
3. Select Program, Command Line Tool, as the kind of application, and then click Next.
4. Give your application a name and set the Type to Foundation.
Check the box for Use Automatic Reference Counting. Next should select.
5. Give your project folder a name and a location to save your project files. Select Create from the drop-down menu.
6. In the left pane, look for the file main.m (we may need to disclose it from inside the product's name folder).
That file should highlight. Type our program in the edit box in the rightmost pane.
7. Select the center icon under View in the toolbar. The Debug area will now be visible. We'll be able to view your output there.
8. Click the Launch button in the toolbar or choose Run from the Product menu to build and run our application.

USING THE TERMINAL

Some folks may prefer not to learn Xcode to begin programming using Objective-C. If we're used to using the UNIX shell and command-line tools, we might wish to utilize the Terminal application to edit, compile, and run our applications. This section looks at how to go about doing so.

The first step is to launch the Terminal software on our Mac. The Terminal program may find in the Applications folder under Utilities.

Start the Terminal program. A window will appear.

On each line, we write commands following the \$ (or %, depending on how our Terminal application is set). If we're familiar with UNIX, we'll have no trouble with this.

To begin, copy the lines from Program into a file. To begin, make a directory in which to save our software samples. Then, to input our program, launch a text editor such as vi or emacs:

```
sh-2.05a$mkdir Progs      Create directory to store
programs in
sh-2.05a$cd Progs         Change to a new directory
sh-2.05a$vi main.m        Start up aThe text editor to
enter the program
```

We may name Objective-C files whatever we like; make sure the last two characters are of .m. This tells the compiler that we're working with an Objective-C program.

After we've typed our program into a file (we're not displaying the edit instructions for entering and saving your text here), we may build and link it using the LLVM Clang Objective-C compiler, which is called Clang.

The clang command takes the following general format:

```
clang -fobjc-arc -framework Foundation files -o program
```

This option instructs us to utilize the following Foundation framework information:

```
-foundation Framework
```

Because our clang version may not understand this command-line option, if we get an error, try running Clang without this option. File is a list of files that will assemble. In our instance, we have only one such file, which we call main.m.

If the program compiles without problems, the program is the name of the file that will contain executable.

The program will be called `program1`; below is the command line to compile our first Objective-C application:

```
$ clang -fobjc-arc -framework Foundation main.m
-o program1 Compile main.m & call it program1
$
```

The command prompt returning without any alerts shows that no program flaws were found. We can now run the program by entering the name `program1` at the command prompt:

```
$ program1 Execute program1
sh: prog1: command not found
$
```

Unless we've used Terminal previously, this is the most likely outcome. The UNIX shell (the application that runs our program) doesn't know where `program1` is (we won't get into all the technicalities here), so we have two options: One method is to prefix the program name with `./so` that shell knows to look in the current directory for the code to run. The alternative option is to add the directory containing our applications (or just the current directory) to the shell's `PATH` variable. Let's start with the first approach:

```
$ ./program1 Execute prog1
2008-06-08 18:48:44.210 program1[7985:10b] Programming
is fun!
$
```

It's worth noting that developing and debugging Objective-C programs via the terminal is a viable option. However, this is not a long-term plan. There's more to creating Mac OS X or iOS programs than simply the executable file that must be "packed" into an application bundle. It's not simple to accomplish from the Terminal program, but it's one of Xcode's strengths. As a result, we recommend that begin learning to utilize Xcode to construct our projects. This will need some learning, but the work will well worth it in the end.

FIRST PROGRAM'S EXPLANATION

Let's take a closer look at this program now that we're familiar with the steps involved in compiling and running Objective-C programs.

Here it is once more:

```
//
// main.m
// program1
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog (@"Programming is interesting");
    }
    return 0;
}
```

Lowercase and uppercase letters are different in Objective-C. Furthermore, the indentation and line spacing are not paramount. We may take advantage of this by creating programs that are easy to read. The notion of the comment is introduced in the first seven lines of the program.

The notion of the remark is introduced in the first seven lines of the program.

A comment statement is used in a code to help document it and make it more readable.

Comments explain to the program's reader – whether it's the programmer or someone else tasked with maintaining the program – what the programmer had in mind while developing a certain program or series of lines.

There are two methods to add comments to an Objective-C application. One method is to use two slash characters (//).

The compiler ignores any characters after these slashes up to the end of the line.

The letters / and * may also be used to start a remark.

This is where the conversation starts. These kinds of statements must be stopped.

We use the letters * and / to finish the remark, again without any embedded spaces.

The Objective-C compiler treats any characters between the opening `/*` and the ending `*/` as part of the comment statement and ignores them.

When comments span many lines of code, such as in the following, this kind of comment is often utilized.

```
/*
This file implements a class called Fraction, which
represents fractional numbers. Methods allow the
manipulation of
fractions, such as addition, subtraction, etc.
For more information, consult the document:
/usr/docs/classes/fractions.pdf
*/
```

It's totally up to you the kind of remark you make.

It's worth noting that the `/*` style comments cannot be nested.

For three excellent reasons, get into the habit of incorporating comment statements into our software as we create it or enter it into the computer.

To begin with, documenting the software while the reasoning is still fresh in our head is significantly simpler than going back and reworking the logic after it has been done.

Second, by including comments in the program at such an early stage, we may profit from them during the debug phase, when program logic issues are identified and debugged.

Not only may a remark assist us (and others) in reading the program, but it can also direct us to the root of the logic error.

Finally, we had yet to meet a programmer who loves writing documentation for a program.

In fact, when you've done debugging your software, the prospect of returning to it to add comments is unlikely to appeal to us.

Adding comments to the software as it's being developed makes this sometimes-difficult chore a little simpler to tackle. Following line instructs the compiler to look for and execute a file called `Foundation.h`:

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

This is a system file, which means we did not generate it. `#import` tells the program to import or include the information from that file as if its contents were entered into it at that moment. Because it contains information about additional classes and functions that will utilize later in the

application, we imported the file `Foundation.h`. This line in `Program` states that the program's name is `main`:

```
int main (int argc, const char * argv[])
```

`main` is a unique name that specifies where the program should start running.

The reserved term `int`, which comes before `main`, indicates that the value returned by `main` is an integer.

For the time being, disregard what occurs between the open and closed parenthesis; these are command-line parameters. We may now explain exactly what this routine should do now that we've identified the `main` to the system.

This is accomplished by encapsulating all of the routine's program statements in a pair of curly braces. A statement is just an expression that is finished by a semicolon in the simplest instance. All program statements between the brackets are treated as part of the main procedure by the system.

The next line in the `main` reads:

```
@autoreleasepool {
```

Any program statements between and the matching closure are performed under an autorelease pool context. The autorelease pool is a feature that helps the system manage the memory our application needs as it produces new objects more effectively. "Automatic Reference Counting and Memory Management." Within our `@autoreleasepool` context, we have one sentence. This line instructs the execution of a function called `NSLog`.

The following string of characters is the parameter, or argument, to be supplied or handed to the `NSLog` routine:

```
@ "Programming is interesting"
```

The `@` symbol immediately follows a string of text contained by a pair of double quotes in this case. This is referred to as a constant `NSString` object.

The `NSLog` routine is an Objective-C library function that simply shows or logs its input. However, before doing so, it displays the date and time

the routine is performed, the program name, and some other numbers we won't get into here.

In Objective-C, all program statements must be terminated with a semicolon (;).

This is why a semicolon occurs right after the NSLog call's closing parenthesis.

The final program statement in the main is as follows:

```
return 0;
```

It states that the main should be terminated, and a status value of 0 should be returned.

By convention, 0 indicates that the program terminated normally.

Any nonzero number often indicates that a problem occurred, such as the software cannot identify a required file.

If we use Xcode and return to our output window, we'll notice the following shown after the line of output from NSLog:

```
Program ended with exit code: 0
```

We should know what that message signifies by now.

Now that we've completed talking about our first program, let's change it to include the statement "And writing in Objective-C is much more enjoyable!" This is easily accomplished by just adding another call to the NSLog function, as illustrated in Program. Remember that each Objective-C program statement must end with a semicolon. The leading remark lines have been omitted in the following program examples.

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog(@"Programming is interesting");
        NSLog(@"Programming in the Objective-C is even more interesting");
    }
    return 0;
}
```

If we enter Program and then build and execute it, we may anticipate the following result (without the text that NSLog typically appends to the output):

```
Programming is interesting
Programming in the Objective-C is even more
interesting
```

As the following software example shows, you don't need to use the NSLog function for each output line.

Let's start with a unique two-character sequence.

The newline character comprises the backslash (\) and the letter n.

A newline character instructs the system to do exactly what its name implies: go to the new line.

Any characters that follow the newline character are printed on the following line of the display. Indeed, the newline character is comparable to the carriage return key on a typewriter.

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
    @autoreleasepool {
        NSLog(@"Testing...\n...1\n...2\n...3\n...4");
    }
    return 0;
}
```

DISPLAYING THE VALUES OF VARIABLES

With NSLog, we may display not only basic sentences but also variable values and calculation results. The program uses the NSLog function to display the results of adding two integers, 55 and 35.

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
    @autoreleasepool {
        int sum;
        sum = 55 + 35;
        NSLog(@"The sum of 55 and 35 is %i", sum);
    }
    return 0;
}
```

After the autorelease pool is set up, the first program instruction within the main declares the variable `total` to be of type integer. All program variables must be defined before they can be used in a program. The definition of a variable tells the Objective-C compiler how the program should use the variable. The compiler needs this information to construct the necessary instructions for storing and retrieving values from the variable. A variable of type `int` may only contain integral values – that is, numbers with no decimal places.

Integral values include 3, 5, -20, and 0. Real numbers having decimal places, such as 2.14, 2.455, and 27.0, are known as floating-point numbers.

The result of adding the two numbers 55 and 35 is stored in the integer variable `sum`.

To visually separate the variable declarations of the routine from the program statements, we have purposefully left a blank line after the definition of this variable; this is entirely a matter of style. A single blank line in a program can sometimes make it more understandable.

The program statement is written in the same way as it would be in most other programming languages:

```
sum = 55 + 35;
```

The number 55 is added (as shown by the plus sign) to the number 35, and the result is placed in the variable `total` (as indicated by the assignment operator, the equals sign).

In Program, the `NSLog` routine call now includes two parameters in parentheses.

A comma separates these parameters. The character string shown is always the first input to the `NSLog` function. However, in addition to displaying the character string, we may also wish to display the value of specific program variables. In this scenario, you want the value of the variable `sum` to be shown after these characters:

```
The sum of 55 and 35 is
```

The `%` character in the first parameter is a special character that the `NSLog` method recognizes. The character that appears immediately after the `%` sign indicates the type of value that will be present at that moment. The `NSLog` method in the preceding software identifies the letter `i` as indicating that an integer value is to be shown.

When the NSLog procedure discovers the %i characters within a character string, it shows the value of the next parameter to the routine. Because sum is the next input to NSLog, its value is printed automatically after “The sum of 55 and 35 is.”

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
    @autoreleasepool {
        int value1, value2, sum;
        value1 = 55;
        value2 = 35;
        sum = value1 + value2;
        NSLog(@"The sum of %i and %i is %i", value1, value2,
            sum);
    }
    return 0;
}
```

Inside main, the second program instruction declares three int variables: value1, value2, and sum.

Three distinct statements may have been used to represent this statement, as follows:

- int value1;
- int value2;
- int sum;

Following the definition of the three variables, the program assigns the value 55 to value1 and the value 35 to value2.

After that, the total of these two variables is calculated, and the result is assigned to the variable sum.

There are now four parameters in the call to the NSLog function.

The first argument, often known as the format string, tells the system how the rest of the arguments should present.

Following the words “The total of,” the value of value1 should be present.

In the same way, the values of value2 and total should write at the positions indicated by the following two %i letters in the format string.

CLASSES, OBJECTS, AND METHODS

WHAT EXACTLY IS AN OBJECT?

A thing is an object. Consider object-oriented programming to be a thing and something we'd want to accomplish with it. A procedural programming language, such as C, is the polar opposite of this. In C, we usually think about what we want to accomplish first, then worry about the objects, which is nearly the inverse of object orientation.

Consider the following scenario from our daily life.

Let's pretend we have a vehicle, which is an item that we own. We don't simply have an automobile; we have a specific car built at a factory in Detroit, Japan, or elsewhere. A vehicle identification number (VIN) is a number that uniquely identifies our vehicle in the United States. Our vehicle is an instance of a car in object-oriented terminology.

Car is the name of the class from which this instance was formed to keep with the nomenclature. As a result, every time a new automobile is built, a new instance of the car class is generated, and each instance of the car is referred to as an object. It may be silver, with a black inside, a convertible or hardtop, and so on. In addition, we use your automobile to complete specific tasks.

For example, we drive our vehicle, fill it up with gas, (ideally) wash it, and get it serviced, among other things. This is seen in the following table:

Object	Actions on Objects
Our car	What You Do with It Drive it Fill it with gas Wash it Service it

The tasks indicated in the table can perform with our car and other vehicles. For example, sister drives her car, wash it, fills it with gas, and so on.

INSTANCES AND METHODS

An instance is a one-of-a-kind instance of a class, and methods are the actions done on the instance. A method may be applied to a class instance or the class itself in certain instances.

Washing our automobile, for example, is an instance. It would be a class function to determine how many different automobiles a company

produces. Assume we have two autos that seem to be identical right off the production line:

They have the same interior, paint color, and other features. They may seem identical at first, but as each automobile is driven by its owner, its distinctive qualities or properties change. For example, one automobile may have a scratch while another may have accumulated more kilometers. Each instance or object has information about its original and current attributes, which were received from the factory. These properties are subject to change on a regular basis. The petrol tank depletes, the automobile grows dirtier, and the tyres get more worn as we drive. The state of an object may be affected by applying a method to it. If our approach is to “fill up my automobile with gas,” our car’s gas tank will be full when we complete it.

The status of the car’s gas tank will have been altered as a result of the procedure. Objects are unique representations of a class, and each object has some information (data) that is usually private to that object. The techniques make it possible to access and modify the data. For applying methods to classes and instances, the Objective-C programming language offers the following syntax:

```
[ ClassOrInstance_method ] ;
```

A left bracket is followed by the name of a class or instance of that class, followed by one or more spaces, followed by the method to be performed. Finally, it is terminated with a right bracket and a semicolon. When we ask a class or an instance to do anything, we send it a message; the recipient of that message is known as the receiver. So, another way to look at the previously given generic format is as follows:

```
[ receiver_message ] ;
```

Return to the previous list and rewrite everything in this new syntax. However, we must first obtain our new car. Go to the factory for that, as follows:

```
ourCar = [Car new] ; get new car
```

We send a new message to the Automobile class (the message’s recipient), requesting it to deliver us a new car. The resultant object (representing our unique automobile) is then saved in the variable `ourCar`. From now on, `ourCar` might refer to the automobile instance we received from the factory.

Since we went to the factory to buy the automobile, the method new is known as a factory or class method.

The remainder of the activities on your new automobile will be instance methods because they are exclusive to our vehicle. Here are some examples of message phrases for our car:

```
[ourCar prep]; get it ready for the first-time use
[ourCar drive]; drive our car
[ourCar wash]; wash our car
[ourCar getGas]; put gas in our car if we need it
[ourCar service]; service our car
[ourCar topDown]; if it's a convertible
[ourCar topUp];
currentMileage = [ourCar odometer];
```

This final example demonstrates an instance method that returns data – presumably, the current mileage as displayed on the odometer. We save the information in a variable named currentMileage in our application.

Here's an instance of a method that accepts an argument that specifies a specific value that may vary from one method call to the next:

```
[ourCar setSpeed: 59]; set the speed to 59 mph
```

Kue, our sister, may use the same techniques to her own car. For example:

```
[kuesCar drive];
[kuesCar wash];
[kuesCar getGas];
```

One of the essential aspects of object-oriented programming is applying the same methods to various objects, which we'll learn more about later. Our programs are unlikely to need us to work with cars. Our items will most likely be computer-related, such as windows, rectangles, text, or perhaps a calculator or song playlist. And, like the procedures employed for our cars, our methods may resemble the following:

```
[myWindow erase]; Clear window
theArea = [myRect area]; Calculate area of the
rectangle
[userText spellCheck]; Spell-check the some text
[deskCalculator clearEntry]; Clear last entry
```

```
[favoritePlaylist showSongs]; Show songs in a playlist
of favorites
[phoneNumber dial]; Dial phone number
[myTable reloadData]; Show updated table's data
n = [aTouch tapCount]; Store number of times the
display was tapped
```

AN OBJECTIVE-C CLASS FOR THE WORKING WITH FRACTIONS

It's time to define an actual class in Objective-C and learn how to interact with class instances. Once again, the process will come first. As a result, the actual software examples may appear impractical.

We'll get into more practical details later. Assume we need to create software that works with fractions. Perhaps we must deal with addition, subtraction, multiplication, etc. If we were unfamiliar with classes, we might begin with a basic application like this.

```
// Simple program to work with the fractions
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int numerator = 2;
        int denominator = 4;
        NSLog(@"Fraction is %i/%i", numerator, denominator);
    }
    return 0;
}
```

The fraction is represented in the Program by its numerator and denominator.

The two lines in the main that follow the @autoreleasepool directive declare the variables numerator and denominator as integers and assign them starting values of 2 and 4, respectively.

This corresponds to the following lines:

```
int numerator, denominator;
numerator = 2;
denominator = 4;
```

We stored 2 in the variable `numerator` and 4 in the variable `denominator` to represent the fraction $2/4$. This may be inconvenient if we are required to save many fractions in our program. We'd have to refer to the numerator and denominator each time we wished to refer to the fraction. It would also be inconvenient to conduct operations on these fractions.

It would be preferable if a fraction could be defined as a single object with a single name for the numerator and denominator, such as `myFraction`. This is possible with Objective-C, which begins with creating a new class. The program uses a new class named `Fraction` to mimic the capabilities of the Program that is previously used. The software is now available, along with a thorough description of how it works.

```
// Program to work with the fractions - class version
#import <Foundation/Foundation.h>
//----- @interface section -----
@interface Fraction: NSObject
-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
@end
//----- @implementation section -----
@implementation Fraction
{
int numerator;
int denominator;
}
-(void) print
{
NSLog(@"%i/%i", numerator, denominator);
}
-(void) setNumerator: (int) n
{
numerator = n;
}
-(void) setDenominator: (int) d
{
denominator = d;
}
@end
//---- program-section ----
int main (int argc, char * argv[])
```

```

{
@autoreleasepool {
Fraction *myFraction;
// Create an instance of a Fraction
myFraction = [Fraction alloc];
myFraction = [myFraction init];
// Set fraction to 2/4
[myFraction setNumerator: 2];
[myFraction setDenominator: 4];
// Display the fraction using the print method
NSLog(@"Value of myFraction is:");
[myFraction print];
}
return 0;
}

```

As the comments in the Program show, the program is logically separated into three sections:

- @interface section
- @implementation section
- program section

The class and its methods are described in the @interface section, whereas the data are described in the @implementation section and the actual code that implements the methods indicated in the interface section. Finally, the program section includes the program code used to carry out the program's intended objective.

Each of these parts is included in every Objective-C application, even if we may not need to create them ourselves. As we can see, each section is usually located in its file. For the time being, we'll keep everything in one file.

@interface SECTION

When creating a new class, you must inform the Objective-C compiler where it originated. That is, we must identify its parent class. Following that, we must describe the operations or methods used when interacting with objects from this class. Furthermore, as we will see in a later chapter, we list elements known as properties in this specific portion of the

program known as the `@interface` section. This section's overall format is as follows:

```
@interface NewClass_Name: ParentClass_Name
propertyAndMethodDeclarations;
@end
```

Class names, by convention, begin with an uppercase letter; however, this is not essential. This allows someone reading our program to differentiate between class names and other sorts of variables by looking at the first character of the name. Let's take a brief detour to discuss name formation in Objective-C.

To hold integer values, we used many variables. In Program, for example, we utilized the variable `total` to hold the result of adding the two numbers 55 and 35.

The Objective-C programming language allows us to store data types other than integers in variables as long as the variable is declared properly before it is utilized in the application. Variables may use to hold floats, characters, and even objects.

The rules for creating names are straightforward: They must begin with a letter or underscore (`_`), and can include any combination of letters (upper- or lowercase), underscores, or digits 0–9. The following is a list of acceptable names:

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

The following names, on the other hand, are not acceptable for the reasons stated:

- `sum$value $`: The character `sum$value $` is not valid.
- `piece flag`: Embedded spaces are not permitted in the `piece flag`.

- 3Spencer: Names can't begin with a number.
- int: This is a reserved word.

The variable name `int` cannot use because it has a specific significance to the Objective-C compiler. This is referred to as a reserved name or a reserved term. Any name that has specific importance to the Objective-C compiler, in general, cannot be used as a variable name.

Remember that upper- and lowercase letters in Objective-C are separate.

As a result, each variable's names `sum`, `Sum`, and `SUM` refers to a separate variable.

As previously stated, when naming a class, begin with a capital letter. In contrast, instance variables, objects, and method names often begin with lowercase letters. To improve readability, capital letters are used inside names to denote the beginning of a new word, as seen in the instances below:

- `AddressBook`: This might be the name of a class.
- `currentEntry`: This may be an object.
- `current entry`: Underscores are used as word separators by certain programmers.
- `addNewEntry`: This might be the name of a method.

Keep one thing in mind when choosing a name: don't be lazy. Choose names that represent the variable or object's intended usage. The reasons are self-evident. As with the comment statement, meaningful names may significantly improve program readability and pay dividends throughout the debug and documentation phases. Because the program will be more self-explanatory, the documentation work should be more manageable.

Here's another example of the `@interface` section from Program:

```
//----- @interface section -----
@interface Fraction: NSObject
-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
@end
```

The new class is called `Fraction`, and its parent class is `NSObject`. The `NSObject` class is defined in the file `NSObject.h`, which is included in our program automatically when we import `Foundation.h`.

CLASS AND INSTANCE METHODS

We must specify methods for working with fractions. We must be able to set the value of a fraction to a specific value. Because we won't have direct access to a fraction's internal representation, we must create methods to set the numerator and denominator. We'll also create a print method to display the value of a fraction. The print method declaration in the interface file looks like this:

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

The leading minus sign (–) indicates that the method is an instance method to the Objective-C compiler. A plus symbol (+), which denotes a class method, is the only other choice.

A class method does anything with the class itself, such as generating a new instance. An instance method is a method that performs some action on the specific instance of a class, such as setting, retrieving, or displaying its value. In the case of the car, we may need to fill it with gas once it has been made. The action of filling it with gas is similar to an instance method since it is done on a specific car.

RETURN VALUES

When we declare a new method, we must inform the Objective-C compiler whether the method is public or private.

If the method returns a value, what type of value does it return?

After the leading minus or plus sign, enclose the return type in parentheses. As a result, this declaration requires that the `currentAge` instance method delivers an integer value:

```
-(int) current_Age;
```

Similarly, this line specifies a method that returns a value with double precision.

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

The Objective-C return statement returns a value from a method, similar to how we returned a value from the main in previous program examples.

If the method returns null, we indicate it using the type `void`, as given below:

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

This declares a `print` instance method that returns null. In this situation, we do not need to execute a return statement at the end of your method. Alternatively, we can perform a return with no value specified, as given below:

```
return;
```

METHOD ARGUMENTS

In the `@interface` section of Program, two further methods are declared:

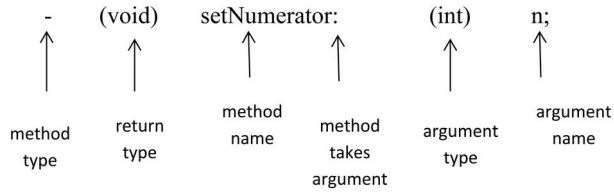
```
-(void) setNumerator: (int) nm;
-(void) setDenominator: (int) dn;
```

These are both null-returning instance methods. Each method accepts an integer parameter, denoted by the `(int)` before the argument name. The name of the argument in the instance of `setNumerator` is `nm`. This is an arbitrary name for the argument that the method uses to refer to it.

As a result, the definition of `setNumerator` states that the method will supply one integer parameter, `nm`, and that no value will return. This is identical to `setDenominator`, with the exception that the argument's name is `dn`. Take note of the declaration syntax for these methods. The colon at the end of each method name indicates to the Objective-C compiler that the method requires an argument. The type of the argument is then stated, surrounded in parentheses, in the same manner as the method's return type is supplied.

Finally, the method specifies the symbolic name used to identify that argument. A semicolon is used to finish the declaration. This syntax is shown in the diagram. When referring to a method that accepts an argument, we must attach a colon to the method name.

As a result, the right way to identify these two methods, each of which accepts a single parameter, is `setNumerator:` and `setDenominator:`. Furthermore, identifying the `print` method without a trailing colon indicates that this method does not accept any parameters.



Declaration of method.

THE @implementation SECTION

The @implementation section, as previously stated, includes the actual code for the methods specified in the @interface section. We must indicate what type of data will be stored in this class's objects. That is, we must specify the data that class members will hold. These are referred to as instance variables. To clarify terminology, we declare the methods in the @interface section and describe them (that is, provide the actual code) in the @implementation part. The following is the usual format for the @implementation section:

```
@implementation NewClass_Name
{
Member_Declarations;
}
Method_Definitions;
@end
```

The name NewClass_Name is the same as the name given to the class in the @interface section.

As in the @interface part, we may use a trailing colon followed by the parent class name:

```
@implementation Fraction: NSObject
```

This, however, is optional and is usually not done.

The memberDeclarations section describes what sorts of data are held in a Fraction and their names.

As we can see, this section is surrounded by its own set of curly braces. These declarations for our Fraction class indicate that a Fraction object contains two integer members: numerator and denominator.

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

The instance variables are the members defined in this section. When we create a new object, we generate a new and distinct collection of instance variables. As a result, if we have two `Fractions`, `fracA` and `fracB`, each will have its own set of instance variables – that is, `fracA` and `fracB` will each have their independent numerator and denominator. One of the advantages of dealing with objects is that the Objective-C system automatically maintains track of this for us. The `@implementation` section's `method-Definitions` section provides the code for each method declared in the `@interface` section. Each method's definition, like the `@interface` section, begins by specifying the kind of the method (class or instance), its return type, and its parameters and their types. Instead of a semicolon at the end of the line, the method code is surrounded by a series of curly brackets. It should be mentioned that by utilizing a specific `@synthesize` directive, we may have the compiler automatically construct methods for us.

```
//---- @implementation section ----
@implementation Fraction
{
    int numerator;
    int denominator;
}
-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}
-(void) setNumerator: (int) nm
{
    numerator = nm;
}
-(void) setDenominator: (int) dn
{
    denominator = dn;
}
@end
```

The `print` method displays the values of the instance variables `numerator` and `denominator` using `NSLog`. But whose `numerator` and `denominator` is this approach referring to? It refers to the instance variables present in the message's receiver object. That is a crucial topic, and we will return to it shortly.

The integer parameter `n` is stored in the instance variable `numerator` using the `setNumerator:` method. `SetDenominator`, on the other hand, keeps the value of its argument `d` in the instance variable `denominator`.

PROGRAM SECTION

The code to handle our problem is contained in the program section, which can be stretched across many files if necessary. As previously said, we must have a routine named `main` somewhere. That's where our program's execution always begins. :

```
//----- program section -----
int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *myFraction;
        // Create instance of a Fraction and initialize it
        my_Fraction = [Fraction alloc];
        my_Fraction = [myFraction init];
        // Set fraction to 2/4
        [my_Fraction setNumerator: 2];
        [my_Fraction setDenominator: 4];
        // Display fraction using the print method
        NSLog(@"Value of my_Fraction is:");
        [my_Fraction print];
    }
    return 0;
}
```

We define a variable named `myFraction` within the `main` with the following line:

```
Fraction *my_Fraction;
```

This line specifies that `myFraction` is a `Fraction` object; that is, `myFraction` is used to hold values from our new `Fraction` class. The asterisk that appears before the variable name is explained in further detail below.

Now that we have an object to hold a `Fraction`, we must make one, just as we would request a new automobile from a factory. This is accomplished using the following line:

```
my_Fraction = [Fraction alloc];
```

`alloc` is an abbreviation for allocation. We wish to make memory storage space available for a new fraction. This expression communicates with our newly created `Fraction` class:

```
[Fraction alloc]
```

We're instructing the `Fraction` class to use the `alloc` method, yet we never specified one, so where did it come from? The function was passed down from a parent class.

We acquire a new instance of a class when we send the `alloc` message to it.

The returned result is saved in our variable `myFraction` in `Program`. All of an object's instance variables are guaranteed to be zeroed out by the `alloc` procedure. That does not imply that the object has been correctly set up for usage. After we allocate an object, we must initialize it.

```
my_Fraction = [my_Fraction init];
```

Again, we are employing a method that we did not create. The `init` method creates a new instance of a class. Please keep in mind that we are sending the `init` message to `my_Fraction`. That is, we want to initialize a specific `Fraction` object here, so we pass it to an instance of the class rather than the class itself. Before proceeding, be certain that we grasp this point. The `init` method also returns a value, which is the object that has been initialized. The return value is saved in our `Fraction` variable `my_Fraction`. The two-line sequence of allocating a new instance of a class and then initializing it is so standard in Objective-C that the two messages are frequently merged, as shown below:

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

This inner message expression is initially evaluated:

```
[Fraction alloc]
```

As we know, the actual fraction allocated is the result of this message expression. Instead of saving the allocation result in a variable, as we did previously, we apply the `init` method to it directly. So, once again, we allocate a new `Fraction` before initializing it. The initialization result is then assigned to the variable `my_Fraction`.

As a final brevity approach, allocation and initialization are frequently included straight into the declaration line, as in:

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

Returning to Program, we can now specify the value of our fraction. These program lines accomplish exactly that:

```
// Set fraction to 2/4
[my_Fraction setNumerator: 2];
[my_Fraction setDenominator: 4];
```

The first message statement delivers the message `setNumerator:` to `myFraction`. The provided argument has the value 2. Control is then sent to your `Fraction` class's `setNumerator:` method. Because `my_Fraction` is an object of the `Fraction` class, the Objective-C system recognizes that it is the method from this class to employ.

The provided value of 2 is placed in the variable `n` within the `setNumerator:` method.

That value is stored in the instance variable `numerator` by a single program line in that method. So we've set `my_Fraction`'s `numerator` to 2.

Following that is the message that calls the `setDenominator:` method on `my_Fraction`.

Within the `setDenominator:` method, the parameter 4 is allocated to the variable `d`.

This value is subsequently put into the `denominator` instance variable, completing the assignment of 2/4 to `myFraction`. Now we may use the following lines of code from the Program to display the value of our fraction.

```
// Display fraction using the print method
NSLog(@"Value of my_Fraction is:");
[my_Fraction print];
```

The `NSLog` method simply prints the following text:

```
Value of my_Fraction is:
```

The message expression calls the `print` method:

```
[my_Fraction print];
```

The instance variables' numerator and denominator values are printed within the print method, separated by a slash character.

It appears that we have to create a lot more code in the Program to replicate what we accomplished in Program. That is correct for this simple example, but the ultimate purpose of dealing with objects is to make our programs easier to design, maintain, and expand. We'll figure it out later. Let us return for a moment to `my_Fraction`'s declaration.

```
Fraction *my_Fraction;
```

In its declaration, the asterisk (*) indicates that `my_Fraction` is a reference (or pointer) to a `Fraction` object.

The variable `my_Fraction` does not hold the data for the fraction (i.e., its numerator and denominator values). Instead, it saves a reference (which is a memory address) specifying where the object's contents are stored in memory. When we initially declare `my_Fraction`, its value is undefined since it hasn't been assigned to anything and has no default value. `my_Fraction` may be thought of as a box that holds a value. The box initially has an undefined value because it hasn't been assigned a value. The following figure illustrates this.



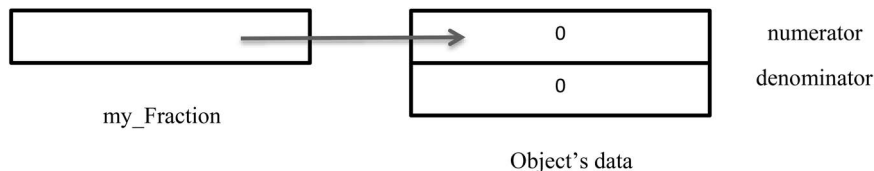
`my_Fraction`

Declaration `Fraction *my_Fraction`.

When we allocate a new object (e.g., using `alloc`), enough memory is reserved to contain the object's contents, including space for its instance variables, plus a bit extra. The `alloc` routine returns the location where that data is stored (the reference to the data) and assigns it to the variable `my_Fraction`. This all happens when the following sentence is run in Program:

```
my_Fraction = [Fraction alloc];
```

The figure given below depicts the allocation of the item and the storing of the reference to that object in `my_Fraction`.

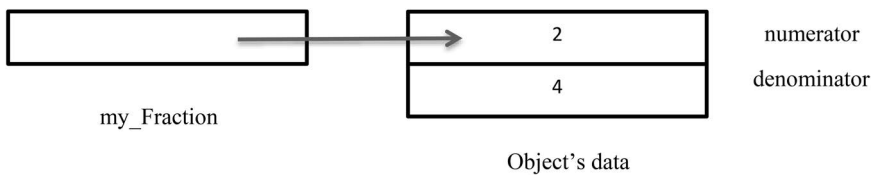


The relationship between `my_Fraction` and its data.

The figure shows a directed line. This shows that a link has been established between the variable `my_Fraction` and the allocated object. (The value held within `my_Fraction` is a memory address.) The object's data is saved at that memory location.

The fraction's numerator and denominator are then specified in Program. The figure shows a completely initialized Fraction object with a numerator of 2 and a denominator of 4.

The following example demonstrates how to work with multiple fractions in your program. Set one fraction to 3/4, another to 4/7, and display both in Program.



Setting Fraction of numerator and denominator.

```
// Program to work with the fractions - cont'd
#import <Foundation/Foundation.h>
//----- @interface section -----
@interface Fraction: NSObject
-(void) print;
-(void) setNumerator: (int) nm;
-(void) setDenominator: (int) dn;
@end
//----- @implementation section -----
@implementation Fraction
{
int numerator;
int denominator;
}
-(void) print
{
NSLog(@"%i/%i", numerator, denominator);
}
-(void) setNumerator: (int) nm
{
numerator = nm;
}
-(void) setDenominator: (int) dn
```

```

{
denominator = dn;
}
@end
//----- program section -----
int main (int argc, char * argv[])
{
@autoreleasepool {
Fraction *frac1 = [[Fraction alloc] init];
Fraction *frac2 = [[Fraction alloc] init];
// Set the 1st fraction to 3/4
[frac1 setNumerator: 3];
[frac1 setDenominator: 4];
// Set 2nd fraction to 4/7
[frac2 setNumerator: 4];
[frac2 setDenominator: 7];
// Display the fractions
NSLog(@"The First fraction is:");
[frac1 print];
NSLog(@"The Second fraction is:");
[frac2 print];
}
return 0;
}

```

ACCESSING INSTANCE VARIABLES AND DATA ENCAPSULATION

We've seen how the methods that deal with fractions may directly access the numerator and denominator instance variables by name. In reality, an instance method may be useful in various situations.

On the other hand, a class method can't access its instance variables directly since it's a class method dealing solely with the class itself, not any of its instances. What if we needed to access our instance variables from a different location, from inside our primary routine?

We won't be able to do so directly since data encapsulation is a vital idea that describes the fact that they are concealed from us. It allows someone authoring class definitions to expand and alter them without being concerned about whether programmers (i.e., class users) are tinkering with the class's internal workings. Data encapsulation adds a layer of protection between the coder and the developer of classes.

We may access our instance variables cleanly by writing custom methods to set and get their values. To set the values of our Fraction class's two instance variables, we created `setNumerator:` and `setDenominator:` methods. We'll need to create two additional methods to access the values of those instance variables. For example, we'll build two new methods called `numerator` and `denominator` to access the respective instance variables of the Fraction that is the message's recipient.

We return the matching integer value as a result. The declarations for our two new methods are as follows:

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

Here are some definitions:

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

It is worth noting that the names of the methods and the instance variables they access are identical. This is not an issue (though it may appear strange at first); it is normal practice. The program puts our two new approaches to the test.

```
// Program to access the instance variables - cont'd
#import <Foundation/Foundation.h>
//----- @interface section -----
@interface Fraction: NSObject
-(void) print;
-(void) setNumerator: (int) nm;
-(void) setDenominator: (int) dn;
-(int) numerator;
-(int) denominator;
@end
//----- @implementation section -----
@implementation Fraction
```

```

{
int numerator;
int denominator;
}
-(void) print
{
NSLog(@"%i/%i", numerator, denominator);
}
-(void) setNumerator: (int) nm
{
    numerator = nm;
}
-(void) setDenominator: (int) dn
{
denominator = dn;
}
-(int) numerator
{
return numerator;
}
-(int) denominator
{
return denominator;
}
@end
//----- program section -----
int main (int argc, char * argv[])
{
@autoreleasepool {
Fraction *my_Fraction = [[Fraction alloc] init];
// Set fraction to 2/4
[my_Fraction setNumerator: 2];
[my_Fraction setDenominator: 4];
// Display fraction using our two new methods
NSLog(@"Value of my_Fraction is: %i/%i",
[my_Fraction numerator], [my_Fraction denominator]);
}
return 0;
}

```

This NSLog statement indicates the results of sending two messages to `my_Fraction`: the first to obtain the numerator's value and the second to retrieve the denominator's value:

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

As a result, the numerator message will be delivered to the Fraction object `my_Fraction` in the initial message call. The code will return the value of the numerator instance variable for that fraction in that method. Remember that the context of a method, while running, is the object of the message's recipient. When the numerator method accesses and returns the value of the numerator instance variable, the value of `my_Fraction`'s numerator is obtained and returned. The returned integer value is subsequently provided to NSLog, where it is displayed.

The denominator method will be called for the second message call to access and return the value of `my_Fraction`'s denominator, which will subsequently be provided to NSLog to be shown.

In addition, methods that set the values of instance variables are commonly referred to as setters, whereas methods that obtain the values of instance variables are referred to as getters. Setters for the Fraction class are `setNumerator:` and `setDenominator:`, whereas getters are `numerator` and `denominator`. Setters and getters are also known as accessor techniques when used together.

Make that we grasp the distinction between setters and getters. Because its function is to take an argument and set the relevant instance variable to the value of that argument, setters do not return a value. In that instance, no value should return. Setters typically do not return values because their function is to set the value of an instance variable. On the other hand, the getter is used to “get” the value of an instance variable stored in an object and return it to the application. To do this, the getter must use the return statement to return the value of the instance variable.

Again, the concept of data encapsulation states that we cannot directly set or receive the value of an instance variable from outside of the methods provided for the class. Instead, we must construct setter and getter methods to do so. As a result, we must employ strategies to gain access to data ordinarily concealed from the “outside world.” This offers centralized access to the instance variables and indirectly prevents other code from modifying these values, making our applications more difficult to read, debug, and edit.

It's also worth noting that there's a technique called `new` that combines the acts of an `alloc` and an `init`. As a result, this line might be used to allocate and initialize a new.

```
Fraction:  
Fraction *my_Fraction = [Fraction new];
```

It is often preferable to utilize the two-step allocation and initialization strategy to comprehend that two specific activities are occurring cognitively: We create a new object and then initialize it.

This chapter covered Local Environment Setup, Installation, Hello World Example, and Compilation and Execute the Program.

NOTE

-
1. Installing Xcode and compiling Objective-C on Mac OS X.

Basic Syntax

IN THIS CHAPTER

- Tokens
- Semicolons
- Comments
- Identifiers
- Keywords
- Whitespace

The setup of the environment and the structure of the program were covered in the preceding chapter, and the fundamental syntax of Objective-C will be presented in this chapter.

OBJECTIVE-C TOKENS

In an Objective-C program, a token is a keyword, identifier, constant, literal string, or symbol. The following Objective-C statement, for example, is made up of six tokens:

```
NSLog(@"Hey, World \n");
```

Individual tokens are referred to as:

```
NSLog  
@
```

DOI: [10.1201/9781003358534-3](https://doi.org/10.1201/9781003358534-3)

```
(
    "Hey, World \n"
)
;
```

SEMICOLONS;

In an Objective-C program, a semicolon serves as a statement terminator. That is, a semicolon must follow each phrase. It represents the end of a single logical object.

Take a look at the following two statements:

```
NSLog(@"Hello, People \n");
return 0;
```

COMMENTS

Comments, like help text in our Objective-C program, are ignored by the compiler. They begin with `/*` and end with the characters `*/`, as seen below.

```
/* The first program in Objective-C */
```

There may be no comments within comments, nor can they be within strings or character literals.

IDENTIFIERS

An Objective-C identifier recognizes a variable, function, or other user-defined entity. An identifier begins with one of the letters A to Z, a to z, or an underscore and ends with zero or more letters, underscores, or digits (0 to 9).

Punctuation characters such as `@`, `$`, and `%` are not permitted in Objective-C identifiers. Objective-C is a case-sensitive programming language. As a result, in Objective-C, `Manpower` and `manpower` are two distinct IDs. Here are some examples of acceptable identifiers:

```
Nina      Dara      bac      movename  b_812
name50    _temp     j        b23b9     retVal
```

KEYWORDS

The following is a list of Objective-C reserved terminology. These reserved words may not use as constant, variable, or identifier names.

auto	return	else	switch
break	enum	register	typedef
const	extern	short	union
case	float	long	unsigned
Char	goto	signed	void
continue	for	sizeof	volatile
if	default	struct	while
do	int	Static	_Packed
double	protocol	interface	implementation
NSObject	NSInteger	retain	CGFloat
property	nonatomic;	NSNumber	weak
strong	unsafe_unretained;	readwrite	readonly

WHITESPACE IN OBJECTIVE-C

An Objective-C compiler ignores blank lines since they include only whitespace, maybe with a comment.

Whitespace in Objective-C refers to blanks, tabs, newline characters, and comments. Whitespace separates sections of a statement and helps the compiler discern where one element, such as `int`, stops, and the next element begins in a statement. As a result, in the statement that follows:

```
int age;
```

The compiler must have at least one whitespace character to distinguish between `int` and `age` (typically a space). In comparison, consider the following statement:

```
fruit = lichi + mango; // get the total fruit
```

There is no need for whitespace between `fruit` and `=`, or between `=` and `lichi`. However, we are permitted to incorporate any for the sake of readability.

DATA TYPES

Data types are a comprehensive structure used in the Objective-C programming language to declare numerous variables and functions. The type of a variable determines how much storage space it requires and how the bit pattern stored is interpreted.

The following are the Objective-C types:

Sr. No.	Types and Description
1	Basic types They are classified into two groups of arithmetic types: (x) integer types and (y) floating-point types.
2	Enumerated types They are, once more, arithmetic types, and they are used to create variables that can only be assigned discrete integer values throughout the program.
3	The type void The type void indicates that no value is accessible.
4	Derived types Pointer kinds, Array types, Structure types, Union types, and Function types are among them.

The aggregate types combine the array and structure types. The type of a function shows the kind of the function’s return value.

TYPES OF INTEGERS

The table below offers information about standard integer types, such as storage sizes and value ranges.

Type	Storage Size	Value Range
char	1 byte	–128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	–128 to 127
int	2 or 4 bytes	–32,768 to 32,767 or –2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	–32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	–2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

On a certain platform, the sizeof operator can identify the precise size of a type or variable. sizeof(type) returns the storage size of an object or type in the bytes. The example below shows how to determine the size of an int type on any machine.

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

int main() {
```

```

    NSLog(@"Storage size for the int : %d \n",
sizeof(int));
    return 0;
}

```

TYPES OF FLOATING-POINT

The below table lists the details of common float-point types, such as storage capacities, value ranges, and precision.

Type	Storage Size	Value Range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal-places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal-places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal-places

The `float.h` header file contains macros that allow us to use these values and other binary representations of real numbers information in our programs. The following example will return the amount of storage space utilized by a float type and its range values.

```

#import <Foundation/Foundation.h>

int main() {
    NSLog(@"Storage size for the float : %d \n",
sizeof(float));
    return 0;
}

```

VOID TYPE

The void type signifies that no value is available. It has three separate meanings.

Sr. No.	Types and Description
1	Function returns as void Several Objective-C functions return a void or do not return a value. The return type of a function that has no return value is void. Take the void exit as an illustration (int status).
2	Function arguments as void Some Objective-C functions take no arguments. A function that has no parameters can take a void. For example, int rand(void);

VARIABLES IN OBJECTIVE-C

Variable is the name given to a storage place that our programs may use. Each variable in Objective-C has a type that governs its memory size and organization, the range of values it may hold, and the set of operations that can be performed on it.

The name of a variable can consist of characters, numbers, and the underscore character. Begin with a letter or underscore. Upper and lower-case letters differ because Objective-C is case-sensitive.

Sr. No.	Type and Description
1	char A single octet is single (one byte). It is an integer type.
2	int The most natural integer size for the machine.
3	float A single precision floating-point value.
4	double A double precision floating-point value.
5	void Represents the lack of type.

The Objective-C programming language also provides many types of variables, which we will go over in detail in the following chapters, such as Enumeration, Pointer, Array, Structure, Union, and so on. In this part, we will look at fundamental variable kinds.

VARIABLE DEFINITION

A variable definition in Objective-C instructs the compiler where and how much storage to produce for the variable. As seen below, a variable description explains a data type and provides a list of one or more variables.

```
type variablelist;
```

A type must be a valid Objective-C data type in this example, such as char, wchar, int, float, double, bool, or any user defined object. There may be one or more identifier names in the variable list, separated by commas. Here are some instances of legitimate declarations.

```
int    e, f, g;
char   i, jk;
float  l, salary;
double m;
```

The line `int e, f, g;` declares and defines the variables `e`, `f`, and `g`, and instructs the compiler to construct `int` variables `e`, `f`, and `g`.

Variables can be initialized in their declaration (given an initial value). As shown below, the initializer consists of an equal sign followed by a constant expression.

```
type variable_name = value;
```

Example:

```
extern int d = 3, e = 5;    // declaration of d
                           and e.
int d = 3, e = 5;          // definition and
                           initializing d and e.
byte f = 22;               // definition and
                           initializes f.
char g = 'g';              // the variable g has
                           the value 'g'.
```

Variables with static storage lifetime are automatically initialized with `NULL` when defined without an initializer (all bytes have the value 0). On the other hand, all other variables have an uncertain beginning value.

VARIABLE DECLARATION

A variable declaration ensures the compiler that at least one variable of the given type and name exists, enabling the compiler to proceed with further compilation without demanding more information. A variable declaration is only relevant during compilation; the compiler needs the actual variable declaration when linking the program.

When using several files, a variable declaration is required. We define our variable in one of the files available during program linking. The `extern` keyword will use to specify a variable at any position. Variables in Objective-C applications be declared several times, but they may only be defined once per file, function, or code block.

Example: Consider the following example: variables are declared at the top but defined and initialized within the main function.

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

// Variable-declaration:
extern int d, e;
```

```

extern int f;
extern float g;

int main () {
    /* variable-definition: */
    int d, e;
    int f;
    float g;

    /* actual-initialization */
    d = 30;
    e = 40;

    f = d + e;
    NSLog(@"The value of f : %d \n", f);

    g = 70.0/3.0;
    NSLog(@"The value of g : %f \n", g);

    return 0;
}

```

A similar concept applies to function declarations, in which we supply a function name at the time of declaration and its exact definition afterward. The following example employs a C function, and as we all know, Objective-C supports C-style functions.

```

// function_declaration
int func();

int main() {
    // function_call
    int k = func();
}

// function_definition
int func() {
    return 0;
}

```

LVALUES AND RVALUES

There are two types of expressions in Objective-C.

- lvalue: “lvalue” expressions are those that relate to a memory location. An assignment can have an lvalue on either the left or right side.
- rvalue: An rvalue is a data value stored in memory at a specific address. Because an rvalue is an expression that cannot assign a value, it can appear on the right but not the left side of an assignment.

Variables are lvalues and can thus appear on the assignment’s left side. Numerical literals cannot be allocated and cannot appear on the left-hand side since they are rvalues. This is an accurate statement:

```
int g = 40;
```

CONSTANTS IN OBJECTIVE-C

Constants are values that the program cannot change during execution. These values are also referred to as literals.

Integer constants, floating constants, character constants, and string literals are all examples of fundamental data types. There are also enumeration constants.

Constants are handled the same as regular variables, except that their values are not changed after they are defined.

INTEGER LITERALS

A decimal, octal, or hexadecimal constant can be used as an integer literal. A prefix indicates the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

A literal number can also have a suffix of U or L, which stands for unsigned or long, respectively. Write the suffix in any order, using uppercase or lowercase letters.

Integer literals are used in the following examples.

```
414          /* Legal */
235u         /* Legal */
0xFeeL      /* Legal */
068          /* Illegal: 8 is not octal-digit */
032UU       /* Illegal: cannot repeat suffix */
```

Other examples of Integer literals are shown below.

```
95          /* decimal */
0215        /* octal */
0x6b        /* hexadecimal */
50          /* int */
50u         /* unsigned int */
50l         /* long */
50ul        /* unsigned long */
```

FLOATING-POINT LITERALS

A floating-point literal comprises an integer portion, a decimal point, a fractional component, and an exponent part. Floating-point literals can be expressed as decimal or exponential values.

Include the decimal point, exponent, or both when representing in decimal form; include the integer component, fractional part, or both when representing in exponential form. The signed exponent is introduced by *e* or *E*.

Here are some examples of floating-point literals:

```
3.14159     /* Legal */
314159E-5L  /* Legal */
510E        /* Illegal: incomplete exponent */
210f        /* Illegal: no-decimal or exponent */
.e55        /* Illegal: missing integer or fraction */
*/
```

CHARACTER CONSTANTS

Character literals like “x” are enclosed in single quotes and may be stored in a basic char variable.

In Objective-C, a character literal can be a basic character (such as “x”), escape sequence (such as “\t”), or universal character (such as “u02C0”).

Certain characters in C have special meanings when followed by a backslash, such as a newline (\n) or tab (\t). A list of some of these escape sequence codes is provided below.

Escape Sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character

(Continued)

Escape Sequence	Meaning
\a	Alert or bell
\b	Backspace
\f	Form-feed
\n	Newline
\r	Carriage return
\t	Horizontal-tab
\v	Vertical-tab
\ooo	Octal number of one to three digits
\xhh. . .	Hexadecimal number of one or more digits

The following are some examples of escape sequence characters.

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

int main() {
    NSLog(@"Heyyy\tWorld\n\n");
    return 0;
}
```

STRING LITERALS

Double quotations surround string literals or constants"". In that they are plain characters, escape sequences, and universal characters, characters in a string are similar to character literals. We can divide a long line into many lines using string literals and whitespace.

Here are some examples of string literals. All three variations have the same strings.

```
"heyyy, sweetheart"

"heyyy, \
sweetheart"

"heyyy, " "s" "sweetheart"
```

CONSTANT DEFINITION

There are two simple ways to define constants in C.

- Using #define preprocessor
- Using const keyword

#define Preprocessor

The following is the syntax for declaring a constant using the #define preprocessor:

```
#define identifier value
```

Example:

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

#define LENGTH 20
#define WIDTH 15
#define NEWLINE '\n'

int main() {
    int area;
    area = LENGTH * WIDTH;
    NSLog(@"Value of the area : %d", area);
    NSLog(@"%c", NEWLINE);
    return 0;
}
```

const Keyword

The const prefix can use to declare constants of a certain type, as seen below.

```
const type variable = value;
```

Example:

```
#import <Foundation/Foundation.h>
int main() {
    const int LENGTH = 30;
    const int WIDTH = 25;
    const char NEWLINE = '\n';
    int area;
    area = LENGTH * WIDTH;
    NSLog(@"Value of the area : %d", area);
    NSLog(@"%c", NEWLINE);
    return 0;
}
```

In this chapter, we covered the basic syntax of Objective-C with its relevant examples.

Data Types

IN THIS CHAPTER

- Integer Types
- Floating-Point Types
- The void Type

In the previous chapter, we covered basic syntax, and in this chapter, we will discuss Data types.

DATA TYPES AND CONSTANTS

An int variable may only use to store integral values, that is, numbers that do not contain decimal digits. Other fundamental data types supported by the Objective-C programming language include float, double, and char.

A variable of type float can use to store floating-point numbers (values containing decimal digits). The double type is similar to the float type but nearly twice the range. Finally, the char data type can hold a single character, such as the letter a, the numeric character 6, or a semicolon.

A constant in Objective-C is any literal integer, single character, or character string. The number 68, for example, indicates a constant integer value. A constant character string object is the string @ "Programming in Objective-C is interesting." Constant expressions are expressions that are entirely composed of constant values. This expression is constant because each of its terms has a constant value:

138 + 17 - 27

DOI: [10.1201/9781003358534-4](https://doi.org/10.1201/9781003358534-4)

However, if I were declared an integer variable, this statement would not be a constant expression:

```
138 + 17 - i
```

TYPE int

An integer constant comprises a series of one or more digits. The presence of a minus sign before the sequence indicates that the value is negative. Integer constants are represented by the integers 168, -10, and 0. There are no embedded spaces or commas allowed between the numerals.

Every value, whether a character, an integer, or a floating-point number, has a range of associated values. This range refers to the amount of storage available to store specific data. In general, that number is not stated in the language; it often relies on the computer we're operating on and is hence referred to as implementation or machine-dependent. An integer variable, for example, may occupy 32 bits on our machine or 64 bits. When 64 bits are utilized, substantially bigger numbers may be stored in integer variables than when 32 bits are used.

Type	Storage Size	Value Range
char	1 byte	-128 to 127 or 0 to 255
unsigned-char	1 byte	0 to 255
signed-char	1 byte	-128 to 127
int	2 or 4 byte	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned-int	2 or 4 byte	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned-short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned-long	4 bytes	0 to 4,294,967,295

TYPE float

To store values with decimal digits, use a variable defined to be of type float. The presence of a decimal point distinguishes a floating-point constant. 5., 126.8, and -.0001 are all instances of acceptable floating-point constants. The NSLog conversion characters percent f or percent g can use to show a floating-point number.

Scientific notation can also use to express floating-point constants. The value 1.8e4 is a floating-point value represented by this notation that equals 1.8 x 104.

As previously stated, the double type is the same as the float type, but with nearly twice the range.

Type	Storage Size	Value Range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

TYPE char

A char variable may use to hold a single character. Enclosing a character inside a pair of single quote marks creates a character constant. As a result, 'a', ',', and '0' are all acceptable character constants. The letter an is represented by the first constant, a semicolon by the second, and the symbol zero by the third – which is not the same as the number zero.

A single character encased in single quotes is not the same as a C-style character string, which may include any number of characters enclosed in double quotations. An NSString character string object is a string of characters wrapped in a pair of double quotes and preceded by a @ character, as explained in the previous chapter.

Even though it contradicts the preceding rule, the character constant 'n', the newline character, is a legitimate character constant. Because the backslash character is regarded as a special character, this is the case. In other words, even if the character 'n' is made up of two characters, the Objective-C compiler sees it as a single character. The backslash character is used to begin other special characters.

To show the value of a char variable, use the format characters percent c in an NSLog call. The fundamental Objective-C data types are used in Program.

```
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int integerVar = 120;
        float floatingVar = 431.79;
        double doubleVar = 8.54e+21;
        char charVar = 'W';
        NSLog(@"integerVar = %i", integerVar);
        NSLog(@"floatingVar = %f", floatingVar);
        NSLog(@"doubleVar = %e", doubleVar);
    }
}
```

```

NSLog(@"doubleVar = %g", doubleVar);
NSLog(@"charVar = %c", charVar);
}
return 0;
}

```

QUALIFIERS: long, long long, short, unsigned, and signed

On some computer systems, the defined integer variable has an increased range if the qualifier `long` is added right before the `int` declaration. A lengthy `int` declaration would look like this:

```
long int factorial;
```

The variable `factorial` is now declared to be a long integer variable. The range of a long variable, like that of floats and doubles, is determined by your computer system. The letter `l` is used as a modifier before the integer format letters in `NSLog` to represent the value of a long `int`.

This implies that the format characters `%li` may use to show a long `int`'s decimal value. To store a floating point value with a larger range, we may use a long long `int` variable or even a long double variable.

When used in front of an `int` declaration, the qualifier `short` informs the Objective-C compiler that the defined variable is intended to hold relatively tiny integer values. The primary purpose for utilizing short variables is to save memory space, which may be a problem when the software requires a lot of memory, and the quantity of memory available is restricted.

When an integer variable is used to hold only positive values, the last qualifier that may be put in front of it is utilized. The following tells the compiler that the counter variable will only hold positive values:

```
unsigned int counter;
```

Limiting the use of an integer variable to the exclusive storing of positive integers expands its range.

TYPE `id`

The `id` data type is used to hold any sort of object. It is, in some ways, a general object type. This line, for example, defines `graphic_Object` to be a variable of type `id`:

```
id graphic_Object;
```

Methods can specify to return values of type `id` in the following way:

```
-(id) new_Object: (int) type;
```

This creates a `new_Object` instance method, which accepts a single integer input named `type` and returns a result of type `id`. The `id` data type is a critical data type, and for the interest of thoroughness, we mention it briefly here. The `id` type provides the foundation for major Objective-C features like polymorphism and dynamic binding.

Type	Constant Examples	NSLog chars
char	'a', '\n'	%c
short int	—	%hi, %hx, %ho
unsigned short int	—	%hu, %hx, %ho
int	12, -97, 0xFFE0, 0177	%i, %x, %o
unsigned int	12u, 100U, 0XFFu	%u, %x, %o
long int	12L, -2001, 0xffffL	%li, %lx, %lo
unsigned long int	12UL, 100ul, 0xffeeUL	%lu, %lx, %lo
long long int	0xe5e5e5e5LL, 500ll	%lli, %llx, &llo
unsigned long long int	12ull, 0xffeeULL	%llu, %llx, %llo
float	12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1	%f, %e, %g, %a
double	12.34, 3.1e-5, 0x.1p3	%f, %e, %g, %a
long double	12.34L, 3.1e-5l	%Lf, \$Le, %Lg
id	nil	%p

THE void TYPE

The void type indicates that there is no value available. It is used in three different contexts.

Sr. No.	Types and Description
1	Function returns as void Several Objective-C functions do not return a value or return a void. The return type of a function with no return value is void. As an example, consider void exit (int status);
2	Function arguments as void Some Objective-C functions do not accept any parameters. A void can be accepted by a function that has no parameters. For instance, int rand(void);

This chapter covered data types with int, float, char, and void type.

Variables

IN THIS CHAPTER

- Variable Definition
- Variable Declaration
- Lvalues and Rvalues

In the previous chapter, we covered data types, and in this chapter, we will discuss variables.

Variable is a name assigned to a storage location that our programs may access. In Objective-C, each variable has a type that governs the amount and layout of the variable's memory, the range of values stored inside that memory, and the set of operations that can apply to the variable.

A variable's name can be letters, numbers, and an underscore character. It must start with a letter or an underscore. Because Objective-C is case-sensitive, upper and lowercase letters are different.

Sr. No.	Type and Description
1	char A single octet is typical (one byte). This is a kind of integer.
2	int The machine's most natural integer size.
3	float A floating-point value with single precision.
4	Double A floating-point value with double precision.
5	void Represents absence of type.

The Objective-C programming language also defines different variables, which we will discuss in the following parts, such as Enumeration, Pointer, Array, Structure, Union, and so on. Let us look at fundamental variable types.

OBJECTIVE-C VARIABLE DEFINITION

A variable definition in Objective-C instructs the compiler where and how much storage to produce for the variable. As seen below, a variable description explains a data type and provides a list of one or more variables.

```
type variable_list;
```

In this case, a type must be a valid Objective-C data type, such as char, w char, int, float, double, bool, or any user defined object. The variable list can have one or more identifier names separated by commas. Here are some examples of valid declarations.

```
int    d, e, f;
char   g, gh;
float  i, salary;
double j;
```

The line `int d, e, f;` declares and defines the variables `d`, `e`, and `f`; and tells the compiler to create variables of type `int` called `d`, `e`, and `f`.

In their declaration, variables can be initialized (given an initial value). The initializer is made of an equal sign followed by a constant expression, as seen below.

```
type variable_name = value;
```

Example:

```
extern int d = 3, e = 5;    // declaration of d
and e.
int d = 3, e = 5;          // definition and
initializing d and e.
byte h = 22;               // definition and
initializes h.
char z = 'z';              // the variable z has
the value 'z'.
```

Variables with static storage lifetime are automatically initialized with NULL when defined without an initializer (all bytes have the value 0). On the other hand, all other variables have an uncertain beginning value.

OBJECTIVE-C VARIABLE DECLARATION

A variable declaration ensures the compiler that at least one variable of the specified type and name exists, enabling the compiler to proceed with other compilations without demanding more information. A variable declaration is only relevant during compilation; the compiler requires the actual variable declaration when linking the program.

When working with several files, a variable declaration is required. We define your variable in one of the files available during program linking. The `extern` keyword will use to specify a variable at any position. Variables in Objective-C applications can be declared several times, but they can only be defined once per file, function, or code block.

Consider the following example. Variables are declared at the top but defined and initialized within the main function.

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

// Variable-declaration:
extern int d, e;
extern int f;
extern float g;

int main () {
    /* variable-definition: */
    int d, e;
    int f;
    float g;

    /* actual-initialization */
    d = 10;
    e = 20;

    f = d + e;
    NSLog(@"The value of f : %d \n", f);

    g = 70.0/3.0;
    NSLog(@"The value of g : %f \n", g);

    return 0;
}
```

A similar concept applies to function declarations, in which we supply a function name at the time of declaration and its exact definition afterward. The following example employs a C function, and as we all know, Objective-C supports C-style functions.

```
// functiondeclaration
int func();

int main() {
    // functioncall
    int h = func();
}

// functiondefinition
int func() {
    return 0;
}
```

OBJECTIVE-C LVALUES AND RVALUES

There are two types of expressions in Objective-C.

- lvalue: “lvalue” expressions are those that relate to a memory location. An assignment can have an lvalue on either the left or right side.
- rvalue: The term rvalue refers to a data value stored in memory at a specific address. Because an rvalue is an expression that cannot assign a value, it can appear on the right but not the left side of an assignment.

Variables are lvalues and can thus appear on the assignment’s left side. Numerical literals cannot be allocated and cannot appear on the left-hand side since they are rvalues. This is a valid statement:

```
int h = 20;
```

Constants

IN THIS CHAPTER

- Integer Literals
- Floating-point Literals
- Character Constants
- String Literals
- Defining Constants
- The `#define` Preprocessor
- The `const` Keyword

In the previous chapter, we covered variables, and in this chapter, we will discuss constants.

CONSTANTS IN OBJECTIVE-C

Constants are values that the program cannot change during execution. These values are also referred to as literals.

Integer constants, floating constants, character constants, and string literals are all examples of fundamental data types. There are also enumeration constants.

Constants are handled same as the regular variables, except that their values are not changed after they are defined.

INTEGER LITERALS

A decimal, octal, or hexadecimal constant can be used as an integer literal. For hexadecimal, a prefix provides the base or radix: 0x or 0X, 0 for octal, and nothing for decimal.

A literal number can also have a suffix of U or L, which stands for unsigned or long, respectively. Write the suffix in any order, using uppercase or lowercase letters.

The following are some examples of integer literals:

```
414          /* Legal */
235u         /* Legal */
0xFeeL       /* Legal */
068          /* Illegal: 8 is not octal-digit */
032UU        /* Illegal: can't repeat suffix */
```

Other types of Integer literals are illustrated below:

```
95           /* decimal */
0215         /* octal */
0x6b         /* hexadecimal */
50           /* int */
50u          /* unsigned int */
50l          /* long */
50ul         /* unsigned long */
```

FLOATING-POINT LITERALS

A floating-point literal comprises an integer portion, a decimal point, a fractional component, and an exponent part. Floating-point literals can be expressed as decimal or exponential values.

Include the decimal point, exponent, or both when representing in decimal form; include the integer component, fractional part, or both when representing in exponential form. Either e or E introduces the signed exponent.

Here are some instances of literals in floating-point numbers:

```
3.14159      /* Legal */
314159E-5L   /* Legal */
510E         /* Illegal: incomplete exponent */
210f         /* Illegal: no-decimal or exponent */
.e55         /* Illegal: missing integer or fraction */
```

CHARACTER CONSTANTS

Character literals like 'x' are enclosed in single quotes and may be stored in an essential char variable.

In Objective-C, a character literal can be a basic character (such as 'x'), escape sequence (such as '\t'), or universal character (such as '\u02C0').

Certain letters in C have special meanings when followed by a backslash, such as a newline (\n) or tab (\t). A list of some of these escape sequence codes is provided below:

Escape Sequence	Meaning
\\	\ character
\"	" character
\'	' character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form-feed
\n	Newline
\r	Carriage return
\v	Vertical-tab
\t	Horizontal-tab
\ooo	Octal number of one to three digits
\xhh. . .	Hexadecimal number of one or more digits

The following are some examples of escape sequence characters:

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

int main() {
    NSLog(@"Heyyy\tEveryone\n\n");
    return 0;
}
```

STRING LITERALS

Double quotations surround string literals or constants"". In that they are plain characters, escape sequences, and universal characters, characters in a string are similar to character literals. We can divide a long line into many lines using string literals and whitespace.

Here are some examples of string literals. All three variations have the same strings.

```
"heyyy, sweetie"
```

```
"heyyy, \"
```

```
sweetie"
```

```
"heyyy, \" \"s\" \"sweetie"
```

CONSTANT DEFINITION

There are two simple ways to define constants in C.

- Using the #define preprocessor
- Using the const keyword

#define Preprocessor

The following is the syntax for declaring a constant using the #define preprocessor:

```
#define identifier_value
```

Example:

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

#define LENGTH 12
#define WIDTH 6
#define NEWLINE '\n'

int main() {
    int area;
    area = LENGTH * WIDTH;
    NSLog(@"Value of the area : %d", area);
    NSLog(@"%c", NEWLINE);
    return 0;
}
```

const Keyword

The const prefix can use to declare constants of a certain type, as given below:

```
const type_variable = value;
```

Example:

```
#import <Foundation/Foundation.h>
int main() {
    const int  LENGTH = 25;
    const int  WIDTH  = 12;
    const char NEWLINE = '\\n';
    int area;
    area = LENGTH * WIDTH;
    NSLog(@"Value of the area : %d", area);
    NSLog(@"%c", NEWLINE);
    return 0;
}
```

In this chapter, we covered constant in Objective-C with relevant examples.

Operators

IN THIS CHAPTER

- Integer Literals
- Floating-point Literals
- Character Constants
- String Literals
- Defining Constants
- The `#define` Preprocessor
- The `const` Keyword

In the previous chapter, we covered constants, and in this chapter, we will discuss operators.

In Objective-C, an operator is a symbol that tells compiler to do certain mathematical or logical operations. The Objective-C programming language has several built-in operators, including the following:

- Arithmetic Operators
- Relational Operators
- Assignment Operators
- Logical Operators

- Bitwise Operators
- Misc Operators

This session will cover arithmetic, relational, logical, bitwise, assignment, and other operators.

ARITHMETIC OPERATORS IN OBJECTIVE-C

The following table covers all of the arithmetic operators provided by the Objective-C programming language. Assume variable P is equal to 40 and variable Q is equal to 20.

Operator	Description	Example
+	Adds two operands	P + Q will give 40
-	Subtracts second operand from the first	p - Q will give -22
*	Multiplies both operands	P * Q will give 200
/	Divides numerator by denominator	P / Q will give 13
%	Modulus Operator and the remainder of after integer division	P % Q will give 0
++	The increment operator increases integer value by one	P++ will give 15
--	The decrement operator decreases integer value by one	P-- will give 17

OPERATOR PRECEDENCE

We’ve seen how to conduct a simple operation like addition in Objective-C. The following software demonstrates the processes of subtraction, multiplication, and division in greater detail. The program’s final two actions introduce the concept of one operator having a higher priority, or precedence, than another. In reality, each operation in Objective-C is coupled with precedence.

This precedence determines how an expression with several operators is evaluated: The higher precedence operator is evaluated first. Expressions with the same precedence of operators are evaluated from left to right or right to left, depending on the operator. This is known as an operator’s associative property.

```
// Illustrate use of the various arithmetic operators
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
```

```

int p = 100;
int q = 2;
int r = 25;
int s = 4;
int result;
result = p - q; // subtraction
NSLog(@"p - q = %i", result);
result = q * r; // multiplication
NSLog(@"q * r = %i", result);
result = p / r; // division
NSLog(@"p / r = %i", result);
result = p + q * r; // precedence
NSLog(@"p + q * r = %i", result);
NSLog(@"p * q + r * s = %i", p * q + r * s);
}
return 0;
}

```

The program declares the integer variables `p`, `q`, `r`, `d`, and `result` then assigns the result of subtracting `q` from `p` to `result` and displays its value with an appropriate `NSLog` call.

The following statement multiplies the value of `q` by the value of `r` and stores the product in the `result`:

```
result = q * r;
```

The multiplication result is then shown using an `NSLog` function that we should be acquainted with now.

The division operator, the slash, is introduced in the following program line.

Following the dividing `p` by `r`, the `NSLog` statement displays the result of 20 achieved by dividing 300 by 15.

When attempting to divide an integer by zero, the division results in an abnormal termination or an exception. Even if the software does not crash, the results obtained by such a division are worthless.

If the divisor is found to be zero, appropriate action can take to avoid the division process.

This expression does not provide the value 350 ($10 * 35$); rather, the result given by the corresponding `NSLog` command is 420($2+10 * 35$):

```
a + b * c
```

This is because, like most other programming languages, Objective-C contains rules for the order in which several operations or words in an expression are evaluated. In general, expressions are evaluated from left to right. However, multiplication and division operations take precedence over addition and subtraction. As a result, the system evaluates the expression.

```
a + b * c
```

as follows:

```
a + (b * c)
```

We can use parenthesis to change the order of the evaluation of terms within an expression. In essence, the previously mentioned expression is a valid Objective-C expression. To get the same outcomes, the following sentence might have been replaced in Program:

```
result = a + (b * c);
```

However, if this expression were used instead, the return value would be 350:

```
result = (a + b) * c;
```

The expression is evaluated from the innermost set of parenthesis outward. Just make sure to use as many closed parenthesis as open ones. The last phrase in the Program shows that it is valid to pass an expression as a parameter to NSLog without first assigning the result of the expression evaluation to a variable. The expression

```
a * b + c * d
```

is examined following the previously mentioned rules as

```
(a * b) + (c * d)
```

UNARY MINUS OPERATOR AND INTEGER ARITHMETIC

The program repeats what we just covered while also introducing the notion of integer arithmetic.

```
// More arithmetic-expressions
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
```

```

{
@autoreleasepool {
    int p = 25;
    int q = 2;
    float r = 25.0;
    float s = 2.0;
    NSLog(@"6 + p / 5 * q = %i", 6 + p / 5 * q);
    NSLog(@"p / q * q = %i", p / q * q);
    NSLog(@"r / s * s = %f", r / s * s);
    NSLog(@"-p = %i", -p);
}
return 0;
}

```

To align the declarations of each variable, we added extra blank spaces between `int` and the declarations of `p`, `q`, and `result` in the first three sentences. This makes the software easier to read. We may have also noticed that blank space was added around each operator in each program provided thus far. This, too, is optional and is done primarily for aesthetic purposes. We can insert more blank spaces in almost any place a single blank space is permitted. A few more spacebar pushes will be rewarded if the finished program is simpler to read.

The statement in Program's initial `NSLog` call supports the concept of operator precedence. The following is how this expression is evaluated:

Because division comes before addition, the value of `p` (25) is divided by 5 first. This yields an intermediate result of 5. Because multiplication takes precedence over addition, the intermediate result of 5 is multiplied by the value of `b`, 2 to get a new intermediate result of 10. 3. Finally, the sum of 6 and 10 is computed, yielding a final value of 16.

A fresh twist is introduced in the second `NSLog` statement. We'd think that dividing `a` by `q` and then multiplying by `b` would give us the value of `p`, which is 25.

However, as seen by the output display of 24, this does not seem to be the case. Is it possible that the computer has lost a little information along the way? It's improbable. The truth is that integer arithmetic was used to evaluate this equation. We'll see that the variables `p` and `q` were declared to be of type `int` if we look back at their definitions.

The Objective-C system conducts integer arithmetic whenever a phrase evaluated in an expression consists of two integers. All decimal components of numbers are lost in this situation. Consequently, dividing the value of `p` by the value of `q`, or 25 by 2, yields an intermediate result of 12,

rather than 12.5, as we would anticipate. When we multiply this intermediate result by 2, we get 24, which explains the “missing” digit.

When we do the identical procedure using floating-point values instead of integers, we get the anticipated result, as shown in the next-to-last NSLog line in Program. The choice between a float and an int variable should be decided depending on the variable’s intended purpose.

Use an integer variable if we don’t require any decimal places. The final application will be more efficient and run faster on several machines. If we want decimal place precision, though, the answer is simple. After that, we only have to decide whether to use a float or a double. The answer to this question is contingent on the required precision and size of the numbers.

The value of the variable *p* is negated in the last NSLog statement using the unary minus operator. In contrast to a binary operator, which acts on two values, a unary operator operates on a single value. The negative symbol has two functions: It is used as a binary operator to subtract two values and as a unary operator to negate a value. Except for the unary plus operator (+), which has the same precedence as the unary minus operator, the unary minus operator takes priority over all other arithmetic operators. As a result, the following phrase multiplies $-p$ by *q*:

```
r = -p * q;
```

THE MODULUS OPERATOR

The modulus operator, represented by the percent sign (%), is the final arithmetic operator introduced in this section. Analyze the results from Program to try to figure out how this operator works.

```
//modulus operator
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int p = 25, q = 5, r = 10, s = 7;
        NSLog(@"p %% q = %i", p % q);
        NSLog(@"p %% r = %i", p % r);
        NSLog(@"p %% s = %i", p % s);
        NSLog(@"p / s * s + p %% s = %i", p / s * s + p % s);
    }
    return 0;
}
```

Take note of the statement within the main that declares and initializes the variables `p`, `q`, `r`, and `s` all in one line. As we may be aware, NSLog uses the character that immediately follows the percent sign to indicate how to print the following argument. However, if another percent sign follows, the NSLog procedure interprets this as an indicator that you truly mean to display a percent sign. The program generates a % sign and inserts it in the proper position in its output.

If we determined that the job of the modulus operator percent is to deliver the remainder of the first value divided by the second value, we are accurate. The residue of 25 divided by 5 in the first case is 0. As seen by the second line of output, dividing 25 by 10 yields a leftover of 5. As illustrated in the third output line, dividing 25 by 7 yields a leftover of 4.

Now, look at the final arithmetic expression evaluated in the previous statement. In Objective-C, every operation between two integer values is accomplished using integer arithmetic. As a result, any remainder that results from dividing two integer numbers is simply discarded. The formula `p/s` indicates that dividing 25 by 7 yields an intermediate value of 3. When this number is multiplied by the value of `s`, which is 7, the intermediate result is 21.

Finally, the final result of 25 is obtained by adding the remainder of the dividing `p/s`, as represented by the phrase `p % s`. Unsurprisingly, this value is the same as the variable `a`'s value. In general, this expression will always equal the value of `p`, provided that `p` and `q` are both integers:

```
p / q * q + p % q
```

The modulus operator `%` is only defined to deal with integers. Regarding precedence, the modulus operator takes priority over the multiplication and division operators. Of course, this means that an expression like:

```
table + value % TABLE_SIZE
```

will be assessed as:

```
table + (value % TABLE_SIZE)
```

INTEGER AND FLOATING-POINT CONVERSIONS

To write Objective-C applications efficiently, we must first grasp the rules for the implicit conversion of floating-point and integer values in Objective-C. The program shows some incredibly simple conversions between numeric data formats.

```
// Basic conversions in the Objective-C
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        float f1 = 223.125, f2;
        int i1, i2 = -160;
        i1 = f1; // floating to the integer conversion
        NSLog ("%f assigned to int produces %i", f1, i1);
        f1 = i2; // integer to the floating conversion
        NSLog ("%i assigned to float produces %f", i2, f1);
        f1 = i2 / 100; // integer divided by integer
        NSLog ("%i divided by 100 produces %f", i2, f1);
        f2 = i2 / 100.0; // integer divided by float
        NSLog ("%i divided by 100.0 produces %f", i2, f2);
        f2 = (float) i2 / 100; // type cast-operator
        NSLog ("%f assigned to float produces %f", i2, f2);
    }
    return 0;
}
```

In Objective-C, when a floating-point value is assigned to an integer variable, the decimal component of the number is truncated. So, in the preceding program, when the value of `f1` is assigned to `i1`, the number 223.125 is truncated, which implies that just its integer component, or 223, is saved in `i1`. This is confirmed by the first line of the program's output.

The value of a number does not change when an integer variable is assigned to a floating variable; the system simply translates the value and saves it in the floating variable.

The second line of output from the program confirms that the value of `i2` (-160) was appropriately translated and placed in the float variable `f1`.

The following two lines of the program's output highlight two points to keep in mind while creating arithmetic expressions. When two operands in an expression are integers (this includes short, unsigned, and long integers), the operation follows the principles of integer arithmetic. As a result, even if the result is put to a floating variable, any decimal component from a division operation is lost (as we did in the program). When the integer variable `i2` is divided by the integer constant 100, the system divides as an integer. The value stored in the float variable `f1` is the result of dividing -160 by 100, which is -1.

In the preceding program, the subsequent division involves an integer variable and a floating-point constant. In Objective-C, every operation between two values is done as a floating-point operation if either value is a floating-point variable or constant. Consequently, when `i2` is divided by `100.0`, the system considers the division as a floating-point division and returns `-1.6`, which is assigned to the float variable `f1`.

TYPE CAST OPERATOR

When declaring and defining methods, we've previously seen how enclosing a type inside a set of parentheses is used to specify the return and argument types. When used within expressions, it has a distinct purpose.

The type cast operator is introduced in the final division operation of the Program, which is as follows:

```
f2 = (float) i2 / 100; // the type cast operator
```

The type cast operator converts the value of variable `i2` to type `float` to evaluate the expression. This operator has no lasting effect on the value of the variable `i2`; it is a unary operator that works similarly to other unary operators. The expression `(float) p`, like the expression `-p`, has no permanent effect on the value of `p`.

Except for the unary minus and unary plus, the type cast operator takes priority over all arithmetic operators. Of course, we can always use parentheses in an expression to cause the terms to be evaluated in whatever order we choose.

In Objective-C, the expression is evaluated as follows as another example of using the type cast operator:

```
(int) 27.55 + (int) 23.99
```

```
27 + 23
```

Because converting a floating number to an integer has the consequence of truncating the floating-point value. The expression

```
(float) 5 / (float) 4
```

as does the following expression: yields a value of `1.25`

```
(float) 5 / 4
```

The type cast operator is frequently used to convert an item of a generic id type into an object of a specific class. The following lines, for example, typecast the value of the id variable my_Number to a Fraction object:

```
id my_Number;
Fraction *myFraction;
.....
my_Fraction = (Fraction *) my_Number;
```

The casting result is assigned to the Fraction variable my_Fraction.

RELATIONAL OPERATORS

The Objective-C programming language supports all of the relational operations listed below. Assume variable P is equal to 40 and variable Q is equal to 20.

Operator	Description	Example
==	Compares the values of two operands; if they are equal, the condition becomes true.	(P == Q) is not true.
!=	Compares the values of two operands; if the values are not equal, the condition evaluates to true.	(P != Q) is true.
>	If value of the left operand is larger than value of the right operand, the condition is determined to be true.	(P > Q) is not true.
<	The condition is true if value of the left operand is smaller than the value of the right operand.	(P < Q) is true.
>=	The condition is true if value of left operand is larger than or equal to a value of the right operand.	(P >= Q) is not true.
<=	The condition is true if value of left operand is less than or equal to value of right operand.	(P <= Q) is true.

LOGICAL OPERATORS IN OBJECTIVE-C

The following table contains all the logical operators supported by the Objective-C programming language. Assuming that variable P is 1 and variable Q is 0,

Operator	Description	Example
&&	The Logical AND operator is defined. The condition is met if both operands are non-zero.	(P && Q) is false.
	The Logical OR Operator is defined. The condition is met if any of two operands are non-zero.	(P Q) is true.

(Continued)

Operator	Description	Example
!	The Logical NOT Operator is defined. To reverse logical state of its operand in Objective-C, use. The Logical NOT operator delivers false when a condition is true.	!(P && Q) is true.

BITWISE OPERATORS

The bitwise operator acts on bits and executes actions bit by bit. The truth tables for &, |, and ^ are presented below.

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if P = 70; and Q = 23; now, in binary format, they will be as follows:

```
P = 0011 1100
Q = 0000 1101
-----
P&Q = 0000 1100
P|Q = 0011 1101
P^Q = 0011 0001
~P = 1100 0011
```

The table below lists the bitwise operations supported by Objective-C. Assume variable C equals 70, and variable D equals 23.

Operator	Description	Example
&	If a bit exists in both operands, the binary AND operator duplicates it to the result.	(P & Q) will give 12, which is 0000 1100
	The binary OR operator duplicates any bit that occurs in both operands.	(P Q) will give 61, which is 0011 1101
^	If a bit is set in one but not both operands, the binary XOR operator repeats it.	(P ^ Q) will give 49, which is 0011 0001
~	The Binary Ones Complement Operator is a unary operator that “flips” bits.	(~P) will give –61, which is 1100 0011 in 2’s complement form.

(Continued)

Operator	Description	Example
<<	Left Shift Binary Operator: The value of the left operand is shifted left by the number of bits specified by right operand.	$P \ll 2$ will give 240, which is 1111 0000
>>	Binary Operator with Right Shift The left operand's value is shifted right by the number of bits specified by right operand.	$P \gg 2$ will give 15, which is 0000 1111

ASSIGNMENT OPERATORS

The Objective-C language supports the following assignment operators:

Operator	Description	Example
=	A simple assignment operator assigns the values of the right operands to the left operand.	$Z = P + Q$ will assign value of $P + Q$ into Z
+=	The right operand is added to left operand, and the outcome is then assigned to the left operand.	$Z += P$ is equivalent to $Z = Z + P$
-=	It deducts the right operand from the left operand, assigns the result to the left operand, and then repeats.	$Z -= P$ is equivalent to $Z = Z - P$
*=	Multiplication AND assignment. The product of the right operand and the left operand is assigned to the left operand via this operation.	$Z *= P$ is equivalent to $Z = Z * P$
/=	Both division and assignment Assign the outcome to the left operand after dividing left operand by the right operand.	$Z /= P$ is equivalent to $Z = Z / P$
%=	assignment AND modulus Using two operands, the operator computes the modulus and assigns the result to the left operand.	$Z \% = P$ is equivalent to $Z = Z \% P$
<=<=	The operator assignment AND shift-left.	$Z \ll = 2$ is same as $Z = Z \ll 2$
>>=	The operator is assigned to the right shift.	$Z \gg = 2$ is same as $Z = Z \gg 2$
&=	The bitwise assignment operator AND.	$Z \& = 2$ is same as $Z = Z \& 2$
^=	OR is an assignment and assignment operator that is bitwise exclusive.	$Z \wedge = 2$ is same as $Z = Z \wedge 2$
=	OR bitwise and assignment operator inclusive	$Z = 2$ is same as $Z = Z 2$

MISC OPERATORS ⇨ SIZEOF & TERNARY

Sizeof and ?: are two more important operations given by Objective-C Language.

Operator	Description	Example
sizeof()	The variable's size is returned.	sizeof(x), where x is an integer, will return 4.
&	The address of a variable is returned.	&x; will give the actual address of the variable.
*	A variable's pointer.	*x; will pointer to a variable.
?:	Conditional Expression	If Condition is true ? Then value A : Otherwise, value B.

OPERATORS' PRECEDENCE IN THE OBJECTIVE-C

The order in which words in an expression are arranged is determined by operator precedence. This has an impact on the assessment of a phrase. Certain operators have precedence over others; for example, the multiplication operator takes precedence over the addition operator.

Certain operators have priority over others; for example, the multiplication operator takes precedence over the addition operator. For example, $z = 37 + 13 * 5$; here, z is assigned 102, not 250, since operator * has priority over +; thus, it is multiplied by $13 * 5$ before being added to 37.

Operators with the most significant precedence are at the top of the table, while those with the least significant precedence are at the bottom. Operators with greater precedence will be evaluated first inside an expression.

Category	Operator	Associativity
Postfix	() [] -> ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

CALCULATOR CLASS

It's time to create a new class. We'll create a Calculator class, a simple four-function calculator that can use to add, multiply, subtract, and divide integers. This calculator, like any other, must maintain track of the running total, often known as the accumulator. As a result, methods must allow us to set the accumulator to a specified number, clear it (or set it to zero), and get its value when you're through. The application includes the new class description and a test program to put our calculator through its paces.

```
// Implement Calculator class
#import <Foundation/Foundation.h>
@interface Calculator: NSObject
// accumulator-methods
-(void) setAccumulator: (double) value;
-(void) clear;
-(double) accumulator;
// arithmetic methods
-(void) add: (double) value;
-(void) subtract: (double) value;
-(void) multiply: (double) value;
-(void) divide: (double) value;
@end
@implementation Calculator
{
double accumulator;
}
-(void) setAccumulator: (double) value
{
accumulator = value;
}
-(void) clear
{
accumulator = 0;
}
-(double) accumulator
{
return accumulator;
}
-(void) add: (double) value
{
accumulator += value;
}
```

```

-(void) subtract: (double) value
{
    accumulator -= value;
}
-(void) multiply: (double) value
{
    accumulator *= value;
}
-(void) divide: (double) value
{
    accumulator /= value;
}
@end
int main (int argc, char * argv[])
{
    @autoreleasepool {
        Calculator *deskCalc = [[Calculator alloc] init];
        [deskCalc setAccumulator: 120.0];
        [deskCalc add: 300.];
        [deskCalc divide: 16.0];
        [deskCalc subtract: 12.0];
        [deskCalc multiply: 6];
        NSLog(@"Result is %g", [deskCalc accumulator]);
    }
    return 0;
}

```

The method definitions themselves are pretty simple. The Calculator class has only one instance variable, a double value that contains the accumulator's value.

Take note of the message that calls the multiply method:

```
[deskCalc multiply: 6];
```

The method's parameter is an integer, but it expects a double. There is no issue here since numeric parameters to methods are immediately transformed to the expected type. Because multiply: expects a double, the integer value 6 is automatically transformed to a double-precision floating number when the function is invoked. Even though this automated conversion occurs, it is better programming practice to provide the correct parameter types when invoking methods.

Remember that, unlike the Fraction class, where we may need to interact with a variety of fractions, we may only need to use a single Calculator object in our program. Nonetheless, it makes sense to construct a new class to facilitate working with this object.

We might want to add a graphical front end to our calculator at some time so the user can click buttons on the screen, similar to the calculator program that undoubtedly has loaded on our PC or phone.

This chapter covered Arithmetic Operator, Relational Operator, Logical Operator, Bitwise Operator, Assignment Operator, and Misc Operator.

Loops and Control Statement

IN THIS CHAPTER

- while Loop
- for Loop
- do-while Loop
- Nested Loop
- break Statement
- continue Statement

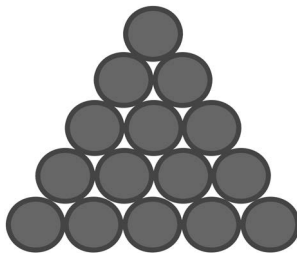
In the previous chapter, we covered operators, and in this chapter, we will discuss loops and control statements.

In Objective-C, we may execute a series of codes in various ways. This chapter focuses on these looping capabilities, which include the following:

- The for statement
- The while statement
- The do statement

Let's start with a basic example: number counting.

If we arranged 15 marbles in the shape of a triangle, we might end up with something like the following figure.



Triangle arrangement.

The triangle's first row holds one marble; the second row includes two marbles, and so on. The number of marbles needed to construct a triangle with n rows is the sum of the integers 1 through n . This is referred to as a triangular number.

Starting with 1, the fourth triangular number is the sum of the successive numbers 1–4 ($1 + 2 + 3 + 4$), or 10.

Assume we wanted to develop a program that computed and presented the eighth triangular number. We could easily compute this amount, but let's pretend we wanted to develop a program in Objective-C to accomplish this work. Program is an example of such a program.

```
#import <Foundation/Foundation.h>
// Program to calculate eighth triangular number
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int triangularNumber;
        triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;
        NSLog (@"Eighth triangular number is %i",
            triangularNumber);
    }
    return 0;
}
```

The program method works well for computing relatively small triangular numbers, but what if we needed to discover the value of the 200th triangular number, for example? It would be inconvenient to change the

Program to add all the integers from 1 to 200 explicitly. Fortunately, there is a more straightforward approach.

One of the essential qualities of a computer is its capacity to execute a set of statements repeatedly. These looping features allow programmers to create succinct programs that incorporate repeating operations that would typically need dozens or even millions of program statements to complete. The Objective-C programming language has three program statements for program looping.

THE for Statement

Let's look at a program that uses the for statement. The goal of the Program is to find the 300th triangular number. See if we can figure out how the for statement works.

```
// Program to calculate the 300th triangular number
// Introduction of for statement
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int nm, triangularNumber;
        triangularNumber = 0;
        for ( nm = 1; nm <= 300; nm = nm + 1 )
            triangularNumber += nm;
        NSLog (@"300th triangular number %i",
            triangularNumber);
    }
    return 0;
}
```

The program requires some explanation. The approach to computing the 200th triangular number is the same as that used in the preceding program to calculate the 8th triangular number: The integers from 1 to 300 are added together.

Before the for statement is reached, the variable `triangularNumber` is set to 0. We must assign a value to all variables before using them in our application. Certain sorts of variables are given default starting values, as we'll see later, but it's safer not to rely on that, and we should set them nonetheless. The for statement saves us from writing each number from 1 to 300 separately. This sentence, in a way, creates these figures for us.

The following is the general format of the for statement:

```
for ( init-expression; loop-condition; loop-expression )
program_statement
```

The three parenthesized phrases `init-expression`, `loop-condition`, and `loop-expression` provide the environment for the program loop.

The program statement that follows (which, of necessity, is concluded by a semicolon) can be any acceptable Objective-C program statement and serves as the loop’s body.

This statement is run as often as the for statement’s arguments specify.

The `init-expression` component of the for statement establishes the initial values before the loop begins. This component of the for statement is used in the Program to set the initial value of `n` to 1. As we can see, an assignment is a valid expression form.

The second part of the for statement provides the condition(s) that must be met for the loop to continue. In other words, the looping will continue as long as this condition is met. Referring again to Program, the loop-condition of the for statement is defined by the relational expression:

```
n <= 300
```

This statement is equivalent to “`n` less than or equal to 300.” The “less than or equal to” operator (the less than character [`<`] followed immediately by the equals sign [`=`]) is one of the numerous relational operators available in the Objective-C programming language. These operators are used to put certain circumstances to the test. If the condition is met, the response is yes (or “true”); otherwise, the answer is no (or “false”).

The following table includes all of the relational operators available in Objective-C.

RELATIONAL OPERATORS

Operator	Meaning	Example
<code>==</code>	Equal to count	<code>== 10</code>
<code>!=</code>	Not equal to flag	<code>!= DONE</code>
<code><</code>	Less than	<code>x < y</code>
<code><=</code>	Less than or equal to low	<code><= high</code>
<code>></code>	Greater than	<code>points > POINT_MAX</code>
<code>>=</code>	Greater than or equal to	<code>j >= 0</code>

The relational operators take priority over all arithmetic operators. This means, for instance, that an expression like:

```
x < y + z
```

is regarded as:

```
x < (y + z)
```

This is to be expected. If the value of *x* is smaller than the value of *y + z*, it is true; otherwise, it is false. Keep an eye out for the “is equal to” operator (`==`), and don’t confuse it with the assignment operator (`=`). The expression

```
x == 2
```

checks whether *x* is equal to 2, whereas the expression

```
x = 2
```

gives the variable *x* the value 2. The relational operator to employ is determined by the test being performed and, in certain cases, by our preferences. For instance, consider the relational expression

```
n <= 300
```

may be expressed similarly as:

```
n < 301
```

Returning to the previous example, the program statement `triangularNumber += n;` is performed repeatedly as long as the relational test result is true, or, in this case, as long as the value of *n* is less than or equal to 300. The result of this program statement is to add the value of *n* to the value of `triangularNumber`.

When the loop condition is no longer met, the program resumes execution with the program statement immediately after the `for` loop. After the loop has finished, execution in this program continues with the `NSLog` statement. The `for` statement’s last component contains an expression evaluated each time the body of the loop is run. This loop-expression in Program adds 1 to the value of *n*. As a result, the value of *n* is incremented

by one when its value is added to the value of `triangularNumber`, which ranges from 1 to 301.

It is worth noticing that the last value attained by `n`, 301, is not included in the value of `triangularNumber` since the loop is halted when the looping condition is no longer met or when `n = 301`.

In summary, the `for` statement is executed as follows:

1. The first expression is evaluated. This expression often sets a variable used within the loop, known as an index variable, to some starting value (such as 0 or 1).
2. The looping condition is then tested. The loop is terminated immediately if the condition is not met (the expression is false). The software statement that immediately follows the loop is executed.
3. The program statement that forms the loop's body is performed.
4. The looping expression is evaluated in step four. This expression is often used to adjust the value of the index variable, usually by adding or removing one.
5. Go back to step 2.

Remember that the looping condition is assessed immediately upon loop entrance before the loop body has been performed. Remember not to use a semicolon after the closing parenthesis after the loop because this will terminate the loop instantly. Because the above program creates all 300 triangular numbers on its way to its final destination, it could be useful to generate a database of these numbers. To save space, say we wish to print a table with only the top ten triangular numbers. This task is carried out by the program listed below.

```
// Program to generate table of the triangular numbers
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int n, triangularNumber;
        NSLog(@"TABLE OF THE TRIANGULAR NUMBERS");
        NSLog(@" n The Sum from 1 to n");
        NSLog(@"-----");
        triangularNumber = 0;
```

```

for ( n = 1; n <= 10; ++n ) {
    triangularNumber += n;
    NSLog(@" %i %i", n, triangularNumber);
}
}
return 0;
}

```

The first three NSLog statements in Program provide just to establish a basic heading and label the output columns.

After displaying the necessary headings, the application computes the first ten triangular numbers. The variable `n` counts the current number whose total from 1 to `n` we are computing, and the variable `triangularNumber` stores the value of the triangular number `n`.

The `for` statement is executed by first changing the value of the variable `n` to 1. As previously stated, the body of the program loop is the program statement immediately following the `for` statement. But what if we want to run a set of program statements rather than a single program statement? This is achieved by enclosing all such program statements in a pair of braces. This collection, or block, of statements, is then treated as a single entity by the system. In general, a block of statements can use anywhere in an Objective-C program where a single statement is permissible, as long as the block is enclosed between a pair of braces.

As a result, in Program, the expression adds `n` to the value of `triangularNumber` and the NSLog statement that follows from the body of the program loop. Take special note of how the program statements are indented. We can tell which statements are part of the `for` loop by looking at them. It's also worth noting that programmers have varied coding styles; for example, some prefer to type the loop like this:

```

for ( n = 1; n <= 10; ++n )
{
    triangularNumber += n;
    NSLog(@" %i %i", n, triangularNumber);
}

```

The opening brace is put on the line after the `for` in this case. This is strictly a matter of taste and has no bearing on the program.

Simply add the value of `n` to the previous triangular number to get the next triangular number.

Because the previous triangular number is 0 the first time through the for loop, the new value of `triangularNumber` when `n` equals 1 is just the value of `n`, or 1.

The values of `n` and `triangularNumber` are then shown, with a sufficient number of blank spaces placed into the format string to ensure that the two variables' values line up under the relevant column headers.

The looping expression is evaluated next since the loop's body has now been performed. However, the expression in this for statement seems odd. We must have made a typographical error and meant to type `n = n + 1` instead of this strange-looking expression:

```
++n
```

However, `++n` is a perfectly valid Objective-C expression. The increment operator is introduced as a new (and quite distinctive) operator in the Objective-C programming language. The double plus sign, often known as the increment operator, adds one to its operand. Because addition by one is such a general operation in programs, a specific operator was built just for it. As a result, the expression `++n` is identical to `n = n + 1`. At first look, it may appear that `n = n + 1` is more legible, but you will quickly become used to its operation and even come to enjoy its conciseness.

Of course, no programming language that has an increment operator for adding one would be complete unless it also includes a similar operator for subtracting 1. This operator is known as the decrement operator, and the double minus sign represents it. So, in Objective-C, an expression that reads

```
bean-counter = bean-counter - 1
```

may be expressed similarly using the decrement operator, as follows:

```
--bean-counter
```

Some programmers like to use the `++` or `--` symbol after the variable name, as in `n++` or `bean-counter--`. In the for statement example, this is a question of personal preference.

We may have observed that the last line of the Program's output does not line up. We may fix this little nuisance by replacing the relevant code from the Program with the following NSLog statement:

```
NSLog ("%2i %i", n, triangularNumber);
```

Here is the output from the changed program to verify that this change solves the problem.

The main modification to the NSLog statement is the addition of a field width setting. The characters %2i inform the NSLog procedure that you want to display the value of an integer at that specific moment, as well as that the size of the integer to be displayed should take up at least two columns in the display. Any integer that would normally take up fewer than two columns (i.e., the numbers 0–9) will be preceded by a space. This is referred to as proper justification.

Thus, by specifying a field width of %2i, we ensure that at least two columns are utilized to represent the value of *n*; we also ensure that the values of *triangularNumber* are aligned.

KEYBOARD INPUT

The program only computes the 200th triangular number. What if we wanted to compute the 50th or 100th triangular number? If that were the case, we would need to modify the program to ensure that the for loop was performed the right number of times. We would also need to change the NSLog statement to display the right message.

A simpler way would be to ask the application which triangular number we wish to compute. The program may calculate the appropriate triangle number when we provide our response. A scanf procedure can be used to implement such a solution. The scanf routine is conceptually identical to the NSLog routine. Unlike the NSLog function, which displays information, the scanf routine allows the programmer to input values into the program. Of course, if we're building an Objective-C program with a graphical User Interface (UI), such as a Cocoa or iOS app, we won't be utilizing NSLog or scanf at all.

The program asks the user which triangular number should be computed, then computes and shows the results.

```
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int n, number, triangularNumber;
        NSLog (@"What triangular number do we want?");
        scanf ("%i", &number);
        triangularNumber = 0;
```

```

for ( n = 1; n <= number; ++n )
    triangularNumber += n;
NSLog (@"The Triangular number %i is %i\n", number,
    triangularNumber);
}
return 0;
}

```

Nested for Loops

The application allows the user to have the program calculate any requested triangle number. But what if the customer gave us a list of five triangular numbers to calculate? In this example, the user might simply run the application five times, each time entering the next triangle number to be computed from the list.

Another approach to achieve the same aim and a considerably more fascinating technique in learning Objective-C is to let the program handle the scenario. This is best performed by putting a loop into the program that causes the complete set of computations to be repeated five times. To create such a loop, we may utilize the for statement. This approach is demonstrated through a program and its accompanying output.

```

#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int n, number, triangularNumber, counter;
        for ( counter = 1; counter <= 6; ++counter ) {
            NSLog (@"What triangular number do we want?");
            scanf ("%i", &number);
            triangularNumber = 0;
            for ( n = 1; n <= number; ++n )
                triangularNumber += n;
            NSLog (@"The Triangular number %i is %i", number,
                triangularNumber);
        }
    }
    return 0;
}

```

The program is divided into two levels of for statements. The following is the outermost for statement:

```
for ( counter = 1; counter <= 6; ++counter )
```

This instructs the program loop to be precisely exactly 6 times. The counter value is initially set to one and is increased by one until it is no longer less than or equal to 6 (in other words, until it reaches 7).

Unlike the previous instances, the variable counter is not utilized anywhere else in the program. It functions just as a loop counter in the for statement. However, it must be declared in the program because it is a variable. As stated by the braces, the program loop is made up of all the remaining program statements. We might be able to understand better how this program works if we think about it like this:

```
For 6 times
{
Get a number from the user.
Calculate the requested triangularnumber.
Display results.
}
```

The loop component referred to as Calculate the required triangular number in the preceding actually consists of setting the value of the variable triangularNumber to 0 plus the for loop that calculates the triangular number. As a result, for statement is enclosed within another for statement. This is completely legal in Objective-C, and nesting may extend to any desired level.

Appropriate indentation becomes even more important when dealing with increasingly complex program constructions, such as nested for statements. We can quickly discover which statements are present within each for statement.

for Loop Variants

Before concluding our explanation of the for loop, we should cover some of the syntactic changes that may use to create this loop. When constructing for loop, we may find that we want to initialize more than one variable before the loop starts, or that we want to evaluate more than one expression each time the loop runs. Multiple expressions can include in any of

the for loop's fields as long as they are separated by commas. For instance, in the for statement that starts with:

```
for ( x = 0, y = 0; x < 10; ++x )
....
```

Before the loop begins, the value of *x* is set to 0 and the value of *y* is set to 0. A comma separates the two expressions *x* = 0 and *y* = 0, and both expressions are considered part of the loop's init-expression field. Another example is the for loop that begins:

```
for ( x = 0, y = 100; x < 10; ++x, y -= 10 )
....
```

Before the loop begins, two index variables, *x* and *y*, are assigned to 0 and 100, respectively. The value of *x* is increased by 1 and the value of *y* is decremented by 10 each time the body of the loop is run.

We may need to include more than one expression in a certain field of the for statement, and we may also need to exclude one or more fields from the statement. Simply omit the desired field and replace it with a semicolon to do this.

When no starting expression has to be evaluated, the most typical use for omitting a field in the for statement occurs.

In this scenario, we may just leave the init expression field blank as long as we include the semicolon:

```
for ( ; y != 100; ++y )
...
```

If *y* was already set to some beginning value before the loop began, this statement might use.

A for loop with no looping-condition field creates an endless loop, that is, a loop that will theoretically execute indefinitely. Such a loop can be employed as long as another method of exit is used.

Variables can also be defined as part of the first statement inside a for loop. This is done in the same manner as we have previously defined variables. For instance, the following may use to create for loop with an integer variable counter that is both specified and initialized to the value 1:

```
for ( int counter = 1; counter <= 6; ++counter )
```

The variable counter is only known during the execution of the for loop (referred to as a local variable) and cannot be accessed outside the loop.

THE while Statement

The while statement expands the Objective-C language's looping capabilities even further. This widely used construct has the following syntax:

```
while ( expression )
program-statement
```

The expression included in parenthesis is evaluated. If the expression evaluation result is true, the following program statement is executed. After this statement (or statements, if enclosed in braces) is executed, the expression is evaluated once again. The program statement is run once again if the evaluation result is true. This operation is repeated until the expression evaluates to false when the loop is halted. The program's execution then proceeds with the statement that follows the program statement.

As an example, the following program creates a while loop that simply counts from 1 to 6.

```
// program introduces while statement
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int count = 1;
        while ( count <= 6 ) {
            NSLog ("%i", count);
            ++count;
        }
    }
    return 0;
}
```

The program first changes the value of count to 1 before starting the while loop. The following statement is performed since the count value is less than or equal to 6. The braces specify the NSLog statement and the while loop body statement that increments count. The program's output shows that this loop is repeated five times, or until the value of count reaches 6.

This program may have made you realize that we could have easily performed the same operation by utilizing `for` statement. A `for` statement may always transform into a `while` statement, and vice versa. For instance, consider the generic `for` statement.

```
for ( init-expression; loop-condition; loop-expression )
    program-statement
```

can state in the form of a `while` statement, as follows:

```
init-expression;
while ( loop-condition )
{
    Program-statement
    loop-expression;
}
```

When we get comfortable with the `while` statement, we will have a better sense of when it seems logical to use a `while` statement and when it appears more logical to use a `for` statement. In general, a loop that is repeated a set number of times is an excellent choice for implementation as `for` statement. Furthermore, if the beginning expression, looping expression, and looping condition all use the same variable, the `for` statement is most likely the correct choice.

The following program shows another application of the `while` statement. The program finds the greatest common divisor between two integer numbers. The greatest common divisor of two integers is the biggest integer value that divides the two numbers evenly. The greatest common divisor of 10 and 15 is 5, for example, since 5 is the greatest integer that equally divides both 10 and 15.

A process, or algorithm, that may use to find the greatest common divisor of two random numbers is based on one developed by Euclid (Greek mathematician) in approximately 300 B.C. It's as simple as this:

Problem: Identify the greatest common divisor of two nonnegative integers, `d` and `e`.

- Step 1: If `d = 0`, we are finished and the greatest common divisor equals `u`.
- Step 2: Return to step 1 and calculate `temp = d % e`, `d = e`, `e = temp`.

Don't be concerned about how the preceding algorithm works; simply accept it on trust. We are more interested in designing a program to identify the greatest common divisor than analyzing how the method works.

Developing the computer program becomes considerably easier after presenting the answer to the problem of finding the greatest common divisor in terms of an algorithm. Examining the algorithm's phases reveals that step 2 is repeated as long as the value of *v* is not equal to 0. This realization results in the obvious implementation of this algorithm in Objective-C using a while statement. The program calculates the greatest common divisor of two nonnegative integer values entered by the user.

```
// Find greatest common divisor of two nonnegative
integers
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        unsigned int c, d, temp;
        NSLog (@"Please type in the two nonnegative
integers.");
        scanf ("%u%u", &c, &d);
        while ( d != 0 ) {
            temp = c % d;
            c = d;
            d = temp;
        }
        NSLog (@"Their greatest common divisor %u", u);
    }
    return 0;
}
```

After entering and storing the two integer values in variables *c* and *d* (using the %u format characters to read in an unsigned integer value), the program starts a while loop to compute their greatest common divisor. After exiting the while loop, the value of *c*, which represents the greatest common divisor of *d* and the initial value of *c*, is shown with an appropriate message.

Consider the job of reversing the digits of an integer input from the terminal for the following program that demonstrates the usage of the while statement. For example, if a user enters the number 6543, the application should reverse the digits and display the value 3456.

To create such a program, we must first develop an algorithm that does the specified task. Analyzing your own approach to tackling the problem frequently leads to the development of an algorithm. The solution to the challenge of reversing the digits of a number is as simple as “successively read digits of the number from right to left.” We may have a computer program read the digits of the number sequentially by developing a technique to isolate or extract each digit of the number, beginning with the rightmost digit. The extracted digit can then be shown as the next digit of the inverted number at the terminal.

We can get the rightmost digit from an integer number by taking the remaining after dividing it by 10. For instance, $3456\%10$ yields the value 4, which is the rightmost digit of 3456 and the first digit of the reversed number. (Recall that the modulus operator returns the remainder of dividing one integer by another.) Using the same approach, we can acquire the next digit of the number if we first divide the number by 10, keeping in mind how integer division works. Thus, $3456/10$ is 345, and $345\%10$ equals 5, the following digit of the inverted number.

We can continue this technique until we get to the last digit. When the result of last integer division by 10 is 0, we know that the number’s last digit has been retrieved.

The program invites the user to input a number and then displays the digits from that number from right to left.

```
// Program to reverse digits of a number
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number, right_digit;
        NSLog(@"Enter number.");
        scanf ("%i", &number);
        while ( number != 0 ) {
            right_digit = number % 10;
            NSLog ("%i", right_digit);
            number /= 10;
        }
    }
    return 0;
}
```

THE do Statement

Both of the looping constructions examined thus far in this section test the criteria before executing the loop. As a result, if the requirements are not met, the loop body may never be performed. When designing programs, it is sometimes preferable to have the test run toward the conclusion of the loop rather than at the beginning. Naturally, the Objective-C language has a unique language construct called the do statement to handle such a situation. This statement's syntax is as follows:

```
do
program-statement
while ( expression );
```

The do statement is run in the following order: the program statement comes first. The expression within the parenthesis is then evaluated. If the result of evaluating the expression is true, the loop is continued and the program statement is run once more. The program statement is repeated as long as the expression evaluation remains true. When the expression is evaluated as false, the loop is ended, and the following statement in the program is executed normally.

The do statement is just a transposition of the while statement, with the looping conditions placed at the end rather than beginning of the loop.

To reverse digits of a number, the program employed a while statement. Return to that software and try to figure out what would happen if the user had put in 0 instead of 45678. The while statement's loop would never have been run, and no output would have been displayed. If we used a do statement instead of a while statement, you would ensure that the program loop was completed at least once, ensuring that at least one digit was displayed in all circumstances.

```
// Program to reverse digits of a number
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number, right_digit;
        NSLog(@"Enter number.");
        scanf ("%i", &number);
        do {
            right_digit = number % 10;
```

```

NSLog ("%i", right_digit);
number /= 10;
}
while ( number != 0 );
}
return 0;
}

```

do-while Loop

In contrast to for and while loops, which test the loop condition at the top of the loop, the do-while loop in Objective-C examines its condition at the bottom of the loop.

A do-while loop is identical to a while loop, but it is guaranteed to run at least once.

```

do {
    statements;

} while( condition );

```

Because the conditional expression comes at the conclusion of the loop, the loop's statements are only executed once before the condition is tested. If the condition is true, the flow of control returns to do, and the statement in the loop is executed once again. This step is repeated until the specified condition is no longer true.

```

#import <Foundation/Foundation.h>

int main () {

    /* local-variable definition */
    int x = 8;

    /* do loop execution */
    do {
        NSLog(@"The value of x: %d\n", x);
        x = x + 1;
    } while( x < 10 );

    return 0;
}

```

THE break Statement

When running a loop, we may wish to exit the loop as soon as a specific condition happens, such as detecting an error condition or finding the data we're seeking in a list of data. We may accomplish this with the break statement. When the break statement is executed, the program exits the current loop, whether for, while, or do loop. Subsequent statements in the loop are skipped, and the loop's execution is ended. The statement that comes after the loop is executed.

Only the innermost loop in which break is executed is ended when a break is done from inside a series of nested loops.

The break statement is just the keyword break followed by a semicolon, as shown below:

```
break;
```

continue Statement

The continue statement is similar to break statement because it does not terminate the loop. Any statements that follow after the continue statement up to the conclusion of the loop are skipped when the continue statement is performed. Otherwise, the loop's execution continues as usual.

The continue statement is most commonly used to skip a number of statements within a loop based on some condition but then to continue running the loop otherwise. The continue statement is written in the following format:

```
continue;
```

Use the break and continue statements only when you've mastered the art of constructing program loops and elegantly leaving them. These assertions are far too simple to misuse, resulting in difficult-to-follow plans.

In this chapter, we covered loops and control statements where we discussed for loop, while, do-while, nested loop and break, and continue statement.

Decision Making

IN THIS CHAPTER

- if Statement
- if-else
- if-else-if
- Nested if
- switch

In the previous chapter, we covered loops and control statements, and in this chapter, we will discuss decision-making statements.

The capacity to make judgments is a key aspect of every programming language. When executing the looping statements, decisions were made on when to end a loop. Other decision-making features provided by the Objective-C programming language are explored in this chapter:

- The if statement
- The switch statement
- The conditional operator

THE if Statement

The if statement is a language construct in the Objective-C programming language that allows for flexible decision-making. The following is the typical format of this statement:

```
if ( expression )
program-statement
```

Consider the possibility of translating a sentence like “If it is not raining, We will go Walking” into Objective-C. Using the prior if statement structure, this may be “written” in Objective-C as follows:

```
if ( it is not raining )
I will go walking
```

The if statement is used to specify whether a program statement (or statements, if surrounded in braces) should execute based on certain criteria. If it’s not raining, I’ll go walking. Likewise, in the program statement:

```
if ( count > MAXIMUMSONGS )
[playlist maxExceeded];
```

the max Only if the value of count is more than the value of MAXIMUMSONGS is an exceeded message sent to the playlist; otherwise, it is ignored.

An actual program sample will assist in emphasizing the point. Assume we wish to develop a program that receives an integer from the keyboard and then shows its absolute value. An integer’s absolute value may be calculated simply by subtracting the number if it is less than zero. In the above statement, the term “if it is less than zero” indicates that the program must make a choice. This choice may influence by using an if statement, as seen in the following program.

```
// Calculate absolute value of an integer
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number;
        NSLog (@"Type in number: ");
```

```
scanf ("%i", &number);
if ( number < 0 )
number = -number;
NSLog (@ "Absolute value is %i", number);
}
return 0;
}
```

The program runs twice to ensure that it was working properly. Of course, it would be preferable to run the program several more times to have a higher level of confidence that it is truly running properly, but at the very least, we will know that we have tested both possible results of the program's choice.

After displaying a message to the user and storing the integer value input in a number, the program checks the value of the number to determine if it is less than zero. If it is, the following program instruction is run, negating a number's value. If the value of a number is greater than zero, this program statement is skipped. (We don't want to negate anything that is already positive.) The program then displays the absolute value of the integer, and the program execution concludes.

Consider another program that uses the if statement. To the Fraction class, we'll add a new method named `convertToNum`. This method returns the value of a fraction as a real number. In other words, it will divide the denominator by the numerator and return the result as a number with double precision. So, if we have a fraction of 1/2, we want the procedure to return 0.5.

Such a method's declaration may look like this:

```
-(double) convertToNum;
```

We may write its definition like this:

```
-(double) convertToNum
{
return numerator / denominator;
}
```

No, not exactly. This approach, as outlined, has two significant flaws. Can we find them? The first is about mathematical conversions. Remember that both the numerator and denominator are integer instance variables.

So, what happens when two integers are divided? That's right, it's done as an integer division! If we tried to convert 1/2, the above code would return 0! This is readily remedied by using the type cast operator to convert one or both operands to floating-point values before division:

```
(double) numerator / denominator
```

Because of the high priority of this operator, the numerator value is first transformed to double before being divided. Furthermore, we do not need to convert the denominator because the laws of arithmetic conversion do so for us.

The second issue with this technique is that you must check for division by zero (which we should always do!). We don't want our program to terminate unexpectedly because the invoker of this method neglected to set the denominator of the fraction or set the denominator of the fraction to zero.

The improved `convertToNum` method is available here:

```
-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return NAN;
}
```

If the fraction's denominator is 0, we opted to return the special value NAN (which stands for Not A Number). This symbol is defined in `math.h` system header file, which is automatically imported into our application.

Let's utilize this new approach in Program.

```
#import <Foundation/Foundation.h>
@interface Fraction: NSObject
-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;
-(double) convertToNum;
@end
@implementation Fraction
```

```

{
int numerator;
int denominator;
}
-(void) print
{
NSLog (@" %i/%i ", numerator, denominator);
}
-(void) setNumerator: (int) nm
{
numerator = nm;
}
-(void) setDenominator: (int) dm
{
denominator = dm;
}
-(int) numerator
{
return numerator;
}
-(int) denominator
{
return denominator;
}
-(double) convertToNum
{
if (denominator != 0)
return (double) numerator / denominator;
else
return NAN;
}
@end
int main (int argc, char * argv[])
{
@autoreleasepool {
Fraction *aFraction = [[Fraction alloc] init];
Fraction *bFraction = [[Fraction alloc] init];
[aFraction setNumerator: 1];
[aFraction setDenominator: 4];
[aFraction print];
NSLog (@" =");
NSLog (@"%g", [aFraction convertToNum]);
}
}

```

```

[bFraction print]; // never assigned value
NSLog (@" =");
NSLog ("%g", [bFraction convertToNum]);
}
return 0;
}

```

After setting `aFraction` to $1/4$, the program converts the fraction to a decimal value using the `convertToNum` function. This number is then shown as 0.25.

Because the value of `bFraction` is not explicitly set in the second example, its numerator and denominator are initialized to zero, as is the default for instance variables.

This describes the `print` method's outcome. It also causes the `if` statement within the `convertToNum` function to return the value `NAN`, which is shown as `nan` by `NSLog`.

THE if-else CONSTRUCT

If we're asked if a number is even or odd, we'll most likely determine it by looking at the last digit. If this digit is 0, 2, 4, 6, or 8, it is obvious that the number is even. Otherwise, we'll say the number is odd.

A computer can more easily identify whether a number is even or odd by checking whether it can be evenly divided by two than by checking the integer's last digit to see if it is 0, 2, 4, 6, or 8. The number is even if it is, and odd if it is not.

Previously, we discussed how to calculate the remainder `%`. It is, therefore, the best operator for determining if an integer is evenly divisible by two. It is even if the residue after division by two is zero; otherwise, it is odd.

Now, develop a program that checks if an integer number entered by the user is even or odd and then displays an appropriate message at the terminal.

```

// Program to determine if number is even or odd
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number_to_test, remainder;
        NSLog (@"Enter number to be tested: ");
    }
}

```

```
scanf ("%i", &number_to_test);
remainder = number_to_test % 2;
if ( remainder == 0 )
    NSLog (@\"Number is even.\");
if ( remainder != 0 )
    NSLog (@\"Number is odd.\");
}
return 0;
}
```

After entering the number, the remaining after division by two is computed. The first if statement checks to determine if the value of this remainder is equal to zero. If it is, it shows the message “Number is even.”

The second if statement checks the residual to determine if it is not equal to zero and shows a message indicating that the number is odd if it is not.

If the first if statement is successful, the second must fail, and vice versa. If we recall from our earlier discussions about even/odd numbers, we stated that if the number is evenly divided by 2, it is even; otherwise, it is odd.

This “else” idea is so often used in programming that a separate construct is supplied to handle it. This is known as the if-else in Objective-C, and the usual format is as follows:

```
if ( expression )
    program-statement 1
else
    program-statement 2
```

The if-else statement is simply an expansion of the if statement’s basic style. If the outcome of the expression’s evaluation is true, program statement 1 is run immediately; otherwise, program statement 2 is executed. Any program statement 1 or program statement 2 will be executed in either scenario, but not both.

The if-else statement may include in the preceding program by replacing the two if statements with the single if-else statement.

We’ll see how this new program design helps minimize the program’s complexity while improving its readability.

```
// Determine if number is even or odd
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
```

```

{
@autoreleasepool {
int number_to_test, remainder;
NSLog(@"Enter number to be tested:");
scanf ("%i", &number_to_test);
remainder = number_to_test % 2;
if ( remainder == 0 )
NSLog(@"Number is even.");
else
NSLog(@"Number is odd.");
}
return 0;
}

```

Remember that the double equals symbol (==) represents the equality test, and the single equals sign (=) represents the assignment operator. Forgetting this and accidentally using the assignment operator within the if statement can cause a lot of problems.

COMPOUND RELATIONAL TESTS

So far in this section, we've used if statements to put very simple relational checks between two integers. The above program compared the value of the integer to zero, but the below program compared the fraction's denominator to zero. Setting up more advanced testing becomes sometimes desirable, if not required. Assume we wish to count the number of grades from a test between 60 and 69, inclusive. In this example, we would want to test a grade's value against one restriction and against limitations 60 and 69 to guarantee that it was within the desired range.

The Objective-C programming language has the capabilities required to conduct these sorts of compound relational tests. A compound relational test is just one or more basic relational tests connected by the logical AND or OR operator. The letter pairs && and || (two vertical bar characters) represent these operators, respectively. The following Objective-C statement, for example, increases the value of grades_60_to_69 only if the value of the grade is larger than or equal to 60 and less than or equal to 69:

```

if ( grade >= 60 && grade <= 69 )
++grades_60_to_69;

```

Similarly, the following statement causes the NSLog statement to be executed if index is less than 0 or larger than 88:

```
if ( index < 0 || index > 88 )
NSLog (@"Error - index out of range");
```

In Objective-C, the compound operators may utilize to create exceedingly complicated expressions. The Objective-C programming language provides the programmer with complete expression flexibility, yet this flexibility is frequently abused. Simplicity is nearly always easier to read and debug.

When creating compound relational expressions, use parenthesis extensively to improve readability and prevent getting into difficulties due to an incorrect assumption about the precedence of operators in the or expression. (The && operator takes priority over any arithmetic or relational operator, but not over the || operator.) Blank spaces can also improve the readability of an expression. An extra blank space surrounding the && and || operators visually distinguish them from the expressions they connect.

To illustrate the usage of a compound relational test in a real-world programming example, let's create software that determines if a given year is a leap year. Everybody is aware that a leap year is one that divides evenly by four. We might not be aware, though, that a leap year is only possible if the year is also divisible by 400.

Consider how we would go about creating a test for such a circumstance. To begin, we may compute the year's remainders after dividing it by 4, 100, and 400 and assign these values to properly named variables such as rem4, rem100, and rem400, respectively. The remainder might then be tested to see if the intended requirements for a leap year were satisfied.

If a year is evenly divisible by 4 and not by 100, or if it is evenly divisible by 400, it is a leap year, according to our prior definition. Pause for a minute to consider the last sentence and confirm if it is comparable to the previously given definition. Now that we've rephrased our description in these words, it's a reasonably simple operation to convert it into a program statement, as follows:

```
if ( (rem4 == 0 && rem100 != 0) || rem400 == 0 )
NSLog (@"It is a leap year.");
```

Parentheses are not required around the following subexpression:

```
rem4 == 0 && rem100 != 0
```

This is because the expression will be assessed in this manner regardless: Remember that `&&` takes precedence than `||`.

In reality, the following test would work just as well in this case:

```
if ( rem4 == 0 && ( rem100 != 0 || rem400 == 0 ) )
```

If we add a few sentences before the test to define the variables and allow the user to enter the year from the terminal, we have a code that determines if a year is the leap year, as shown in the program.

```
// This program determines if year is a leap year
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int year, rem4, rem100, rem400;
        NSLog(@"Enter the year to be tested: ");
        scanf ("%i", &year);
        rem4 = year % 4;
        rem100 = year % 100;
        rem400 = year % 400;
        if ( (rem4 == 0 && rem100 != 0) || rem400 == 0 )
            NSLog(@"It's a leap year.");
        else
            NSLog(@"No, it is not a leap year.");
    }
    return 0;
}
```

The previous examples include the years 1955, which is not a leap year because it is not evenly divisible by 4, 2000, which is a leap year because it is evenly divisible by 400, and 2013, which is not a leap year because it is evenly divisible by 100 but not by 400. Use a year that is equally divisible by 4 and not by 100 to complete the collection of test cases.

We noted that Objective-C provides the programmer with a great deal of freedom when creating expressions. In the prior program, for example, you didn't have to compute the intermediate results `rem4`, `rem100`, and `rem400` we could have done it immediately inside the `if` statement, as follows:

```
if ( ( year % 4 == 0 && year % 100 != 0 ) || year %
400 == 0 )
```

Using blank spaces to separate the various operators keeps the preceding expression readable. If we were to disregard this and delete the unneeded set of parenthesis, we may get something like this:

```
if (year%4==0&&year%100!=0 | year%400==0)
```

This expression is totally acceptable and, believe it or not, executes just like the one preceding it. Obviously, the extra blanks contribute significantly to our understanding of complicated statements.

Nested if Statements

In previous talks on the basic style of the if statement, we stated that if the result of evaluating the expression inside the parenthesis is true, the statement that follows is performed. It is absolutely acceptable for this program statement to be another if statement, such as the following:

```
if ( [chess_Game isOver] == NO )
if ( [chess_Game whoseTurn] == YOU )
[chess_Game yourMove];
```

If the isOver message sent to chess_Game returns NO, the next statement is performed; this phrase, in turn, is another if statement. This if statement compares the result of the whoseTurn function to YOU. The chess_Game object receives the yourMove message if the two values are equivalent. As a result, the yourMove message is sent only if the game isn't over and it's your turn. In truth, this assertion might have been written using compound relational, as follows:

```
if ( [chess_Game isOver] == NO && [chess_Game
whoseTurn] == YOU )
[chess_Game yourMove];
```

A more real implementation of nested if statements would be to add an otherwise clause to the preceding example, as illustrated here:

```
if ( [chess_Game isOver] == NO )
if ( [chess_Game whoseTurn] == YOU )
[chess_Game yourMove];
else
[chess_Game myMove];
```

The statement is executed in the manner indicated above. The otherwise clause kicks in if the game isn't ended and it's not your turn. This communicates myMove to chess_Game. If the game is finished, the whole if statement that follows is skipped, including its accompanying else clause.

Take note of how the else clause is related to the if statement that checks the value returned by the whoseTurn function, rather than the if statement that tests if the game is finished. In general, an else clause is usually connected with the final if statement that does not contain an else clause. We may extend the above example by adding an else clause to the outermost if statement. If the game is over, this else clause is executed:

```
if ( [chess_Game isOver] == NO )
if ( [chess_Game whoseTurn] == YOU )
[chess_Game yourMove];
else
[chess_Game myMove];
else
[chess_Game finish];
```

Using indentation to indicate how we believe a statement will be interpreted in the Objective-C language may not necessarily correspond with how the system perceives the statement. For instance, in the above example, eliminating the first otherwise clause does not result in the statement being interpreted as its format indicates:

```
if ( [chess_Game isOver] == NO )
if ( [chess_Game whoseTurn] == YOU )
[chess_Game yourMove];
else
[chess_Game finish];
```

This statement will instead be read as follows:

```
if ( [chess_Game isOver] == NO )
if ( [chess_Game whoseTurn] == YOU )
[chess_Game yourMove];
else
[chess_Game finish];
```

THE else if Construct

We've seen how the else statement comes into play when a test is run against two alternative outcomes: either the number is even or odd, or the year is a leap year. However, the programming decisions we must make are not always so simple. Consider the challenge of building a program that shows -1 if the number entered by the user is less than zero, 0 if it is equal to zero, and 1 if it is larger than zero. (This is an implementation of what is widely known as the sign function.)

In this scenario, we must do three tests to determine whether the number entered is negative, zero, or positive. The straightforward if-else construct will not function. Of course, we can always use three separate if statements in this scenario, but this technique does not always work, especially if the tests are not mutually exclusive.

We may deal with the problem stated above by adding an if statement to our else sentence. We previously said that the statement after an else may be any legitimate Objective-C program statement, so why not another if? As an example, in the general case, we could write:

```
if ( expression 1 )
program-statement 1
else
if ( expression 2 )
program-statement 2
else
program-statement 3
```

This essentially converts the if statement from a two-valued to a three-valued logic decision. We may keep adding if statements to the else clause as demonstrated above to expand the decision to an n-valued logic decision essentially.

The preceding construct is so common that it is known as an else-if construct and is normally structured differently than that seen previously:

```
if ( expression 1 )
program-statement 1
else if ( expression 2 )
program-statement 2
else
program-statement 3
```

This latter style technique improves the statement's readability and makes it apparent that a three-way decision is being taken.

```
// Program to implement sign function
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number, sign;
        NSLog (@"Type in number: ");
        scanf ("%i", &number);
        if ( number < 0 )
            sign = -1;
        else if ( number == 0 )
            sign = 0;
        else // Must be positive
            sign = 1;
        NSLog (@"Sign = %i", sign);
    }
    return 0;
}
```

If the input number is less than zero, the sign is assigned the value -1; if the number is equal to zero, the sign is assigned the value 0; if the number is larger than zero, the sign is assigned the value 1.

The following program examines a character entered from the terminal and identifies it as an alphabetic character (a-z or A-Z), a number (0-9), or a special character (anything else).

The format characters %c is used in the scanf function to read a single character from the terminal.

```
// This program categorizes single character
// that is entered from the keyboard
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        char c;
        NSLog (@"Please enter single character:");
        scanf (" %c", &c);
        if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
            NSLog (@"It is an alphabetic character.");
    }
}
```

```

else if ( c >= '0' && c <= '9' )
    NSLog(@"It is a digit.");
else
    NSLog(@"It is a special character.");
}
return 0;
}

```

After the character is read in, the first test evaluates whether the char variable *c* is an alphabetic character. This is accomplished by determining if the character is a lowercase or uppercase letter. The following phrase performs the former test:

```
( c >= 'a' && c <= 'z' )
```

This expression is true if *c* is a lowercase letter and fits inside the range of characters ‘a’ through ‘z’. This expression performs the latter test:

```
( c >= 'A' && c <= 'Z' )
```

This expression holds true if *c* is an uppercase letter and falls inside the range of characters ‘A’ through ‘Z’. These tests are performed on computer systems that store characters in an ASCII format.

If the variable *c* is an alphabetic character, the first if test succeeds and displays the message “It’s an alphabetic character.” The else if clause is invoked if the test fails. This clause specifies whether the character is a number or not. It should note that this test compares the letter *c* to the characters ‘0’ and ‘9’ rather than the numerals 0 and 9. This is due to a character being read in from the terminal, and the characters ‘0’ through ‘9’ do not correspond to the integers 0–9. In actuality, the character ‘0’ is internally represented as the number 48 in ASCII, the character ‘1’ as the number 49, and so on.

If the character *c* is a digit, the statement “It’s a digit.” is shown. Otherwise, if *c* is neither alphabetic or a digit, the last else clause is executed, and the terminal displays the statement “It’s a special character.” The program’s execution is then complete.

Even if `scanf` is used here to read a single character, you must still hit the Return key after typing the character to transmit the input to the program. In general, while reading data from the terminal, the application does not see any of the data entered on the line until we push the Return key.

Assume we wish to write a program that allows the user to enter simple expressions of the following kind for the next example:

number operator number

The code will evaluate the expression, and the results will be shown at the terminal. The operators you wish to be recognized are the standard addition, subtraction, multiplication, and division operators. We'll utilize the Calculator class from the Program "Data Types and Expressions." The calculator will be given each expression to compute.

To identify which operation is to be executed, the following program employs a large if statement containing several else if clauses.

```
// Program to evaluate simple expressions of form
number operator number
// Implement Calculator class
#import <Foundation/Foundation.h>
@interface Calculator: NSObject
// accumulator-methods
-(void) setAccumulator: (double) value;
-(void) clear;
-(double) accumulator;
// arithmetic-methods
-(void) add: (double) value;
-(void) subtract: (double) value;
-(void) multiply: (double) value;
-(void) divide: (double) value;
@end
@implementation Calculator
{
double accumulator;
}
-(void) setAccumulator: (double) value
{
accumulator = value;
}
-(void) clear
{
accumulator = 0;
}
-(double) accumulator
```

```

{
return accumulator;
}
-(void) add: (double) value
{
accumulator += value;
}
-(void) subtract: (double) value
{
accumulator -= value;
}
-(void) multiply: (double) value
{
accumulator *= value;
}
-(void) divide: (double) value
{
accumulator /= value;
}
@end
int main (int argc, char * argv[])
{
@autoreleasepool {
double value1, value2;
char operator;
Calculator *deskCalc = [[Calculator alloc] init];
NSLog(@"Type in expression.");
scanf ("%lf %c %lf", &value1, &operator, &value2);
[deskCalc setAccumulator: value1];
if ( operator == '+' )
[deskCalc add: value2];
else if ( operator == '-' )
[deskCalc subtract: value2];
else if ( operator == '*' )
[deskCalc multiply: value2];
else if ( operator == '/' )
[deskCalc divide: value2];
NSLog(@"%.2f", [deskCalc accumulator]);
}
return 0;
}

```

The `scanf` function call reads three values into the variables `value1`, `operator`, and `value2`. The `%lf` format characters can use to read in a double value. This is the syntax used to read in the value of the variable `value1`, which is the expression's first operand.

The operator is then entered. You read it into the character variable `operator` because the operator is a character ('+', '-', '*', or '/') rather than a number. The `%c` format characters instruct the system to read the next character from the terminal. The blank spaces within the format string indicate that an arbitrary amount of blank spaces on the input are permissible. When entering these numbers, we can use blank spaces to separate the operands from the operator.

After reading the two numbers and the operator, the program saves the first value in the calculator's accumulator. The value of the operator is then tested against the four permissible operators. When a match is found, the matching message is delivered to the calculator, which then operates. The accumulator value is obtained and shown in the last `NSLog`. The program's execution is then complete.

At this time, a few words on program completeness are in order. Although the previous program completes the work that we set out to achieve, it is not truly complete since it does not account for user errors. For example, what would happen if the user typed in a? for the operator by mistake? The program would just exit the `if` statement and no notifications would ever show at the terminal to notify the user that he had entered his expression improperly.

Another example is when the user enters a division operation with zero as the divisor. We're probably aware that you should never try to divide an integer by zero in Objective-C. The program should check this case.

Attempting to predict the ways in which a program can fail or yield unexpected effects and then taking preventive measures to account for such eventualities are essential components of creating excellent, dependable programs. Running a sufficient number of test cases against a program may frequently point to areas of the program that fail to account for certain circumstances. But it goes even further. While coding a program, it must become a matter of self-discipline always to question, "What would happen if...?" and enter the essential program statements to handle the issue effectively.

A modified version of the program accommodates for division by zero and the entry of an unknown operator.

```

//program to evaluate simple expressions of form value
operator value
#import <Foundation/Foundation.h>
// here, insert interface and implementation sections
for the Calculator class
int main (int argc, char * argv[])
{
    @autoreleasepool {
        double value1, value2;
        char operator;
        Calculator *deskCalc = [[Calculator alloc] init];
        NSLog(@"Type in expression.");
        scanf ("%lf %c %lf", &value1, &operator, &value2);
        [deskCalc setAccumulator: value1];
        if ( operator == '+' )
            [deskCalc add: value2];
        else if ( operator == '-' )
            [deskCalc subtract: value2];
        else if ( operator == '*' )
            [deskCalc multiply: value2];
        else if ( operator == '/' )
            if ( value2 == 0 )
                NSLog(@"Division by zero.");
            else
                [deskCalc divide: value2];
        else
            NSLog(@"Unknown-operator.");
        NSLog(@"%.2f", [deskCalc accumulator]);
    }
    return 0;
}

```

When the slash operator is used for division, an additional test is performed to check whether value2 is 0. If it is, a notice is printed on the terminal; otherwise, the division operation is performed, and the results are displayed. In this situation, pay close attention to the nesting of the if statements and the accompanying else clauses.

The else clause catches any fall-throughs at the end of the program. As a result, any value of the operator that does not match any of the four characters examined causes this else clause to be invoked, resulting in the terminal displaying “Unknown operator.”

A better technique to deal with the division-by-zero problem is to run the test within the division function.

We may alter our divide: technique as shown below:

```
- (void) divide: (double) value
{
    if (value != 0.0)
        accumulator /= value;
    else {
        NSLog(@"Division by zero");
        accumulator = NAN;
    }
}
```

If the value is nonzero, we divide; otherwise, we display the message and set the accumulator to NAN. In general, having the method handle special scenarios is preferable to relying on the programmer's inventiveness.

THE switch Statement

The sort of if-else statement chain you saw in the last program example, with the value of a variable being sequentially compared against multiple values, is so prevalent in programming that an Objective-C program statement exists to accomplish exactly this purpose. The statement is known as a switch statement, and its typical format is as follows:

```
switch ( expression )
{
    case value1:
        program-statement
        program-statement
        ...
        break;
    case value2:
        program-statement
        program-statement
        ...
        break;
    ...
    case valuen:
        program-statement
        program-statement
        ...
}
```

```

break;
default:
program-statement
program-statement
.....
break;
}

```

The expression enclosed in parentheses is compared to the values `value1`, `value2`, . . . , `value n`, which must be simple constants or constant expressions. If a case with a value equal to the value of the expression is identified, the program statements that follow the case are performed. It should note that when more than one of these program statements is included, they do not need to be enclosed between braces.

The `break` statement signifies a certain case's conclusion and terminates the switch statement's execution. Always add a `break` statement after each instance. If this is not done for a certain instance, program execution will proceed into the following case whenever that case is run. This is often done on purpose; if we want to do so, make sure to include remarks to warn others of our goal.

If value of the expression does not match any of the case values, the special optional case named `default` is executed. This is basically analogous to the last example's `catchall` otherwise. In reality, the switch statement's general form may be stated as if statement, as follows:

```

if ( expression == value1 )
{
Program-statement
Program-statement
...
}
else if ( expression == value2 )
{
Program-statement
Program-statement
...
}
...
else if ( expression == valuen )
{
Program-statement

```

```

Program-statement
...
}
else
{
Program-statement
Program-statement
.....
}

```

Example:

```

// Program to evaluate simple expressions of form
// the value operator value
#import <Foundation/Foundation.h>
// Insert interface and implementation sections
for the
// Calculator class
int main (int argc, char * argv[])
{
@autoreleasepool {
double value1, value2;
char operator;
Calculator *deskCalc = [[Calculator alloc] init];
NSLog(@"Type in expression.");
scanf ("%lf %c %lf", &value1, &operator, &value2);
[deskCalc setAccumulator: value1];
switch ( operator ) {
case '+':
[deskCalc add: value2];
break;
case '-':
[deskCalc subtract: value2];
break;
case '*':
[deskCalc multiply: value2];
break;
case '/':
[deskCalc divide: value2];
break;
default:
NSLog(@"Unknown-operator.");

```

```

break;
}
NSLog (@("%.2f", [deskCalc accumulator]));
}
return 0;
}

```

Following the reading of the expression, the value of the operator is sequentially compared to the values supplied by each case. The statements provided within the case are performed when a match is discovered. The break statement then exits the switch statement, where the program's execution is completed. If none of the cases matches the value of the operator, the default case is run, which shows "Unknown operator."

In the preceding program, the break statement in default case is actually unneeded because no statements follow this case inside the switch. Nonetheless, remembering to add the break at the conclusion of each instance is a good programming practice.

Remember that no two case values can be the same when writing a switch statement. We can, however, correlate more than one case value with a certain sequence of program instructions. This is accomplished simply by stating the various case values before the common statements are performed. In the following switch statement, for example, the multiply: method is run if the operator equals an asterisk or the lowercase letter x:

```

switch ( operator )
{
.....
case '*':
case 'x':
[deskCalc multiply: value2];
break;
.....
}

```

BOOLEAN VARIABLES

Almost everybody learning to program will eventually be required to develop a program that generates a table of prime numbers. To refresh our recall, a prime number is one that is not evenly divided by any other numbers other than 1 and itself. The first prime integer is 2 by definition. The following prime is 3 since it is not equally divisible by any integer other than 1 and 3, while 4 is not prime because it is evenly divisible by 2.

We might use numerous techniques to construct a table of prime numbers. If we were asked to create all prime numbers up to 55, the most obvious (and simplest) technique would simply test each integer, p , for the divisibility by all integers from 2 to $p-1$. If any such integer evenly divides p , then p is not a prime number; otherwise, it is.

Program creates a list of prime numbers ranging from 2 to 55.

```
// Program to generate table of the prime numbers
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int p, d, isPrime;
        for ( p = 2; p <= 55; ++p ) {
            isPrime = 1;
            for ( d = 2; d < p; ++d )
                if ( p % d == 0 )
                    isPrime = 0;
            if ( isPrime != 0 )
                NSLog (@"%i ", p);
        }
    }
    return 0;
}
```

Several points should make concerning the program. The outermost for statement creates a loop that iterates over the numbers 2–55. The loop variable p indicates the value presently being tested to check if it is prime. The variable $isPrime$ is given the value 1 in the loop's first line. The use of this variable will soon be made obvious.

The digits 2 to $p-1$ are divided by p using a second loop. A test is performed inside the loop to determine if the result of p divided by d is 0. If it is, then p cannot be a prime number since it may be divided equally by any integer other than 1. The variable $isPrime$ is set to zero to indicate that p is no longer a prime number option.

When the innermost loop completes, the value of $isPrime$ is checked. If its value is unequal to zero, no integer that evenly divides p was discovered; hence, p must be a prime number, and its value is printed.

We may have observed that the variable $isPrime$ only accepts 0 or 1, with no other values. It has the value 1 as long as p is still a prime number. However, once a single even divisor is discovered, its value is set to

0 to indicate that the *p* no longer meets the conditions for being prime. Variables used in this manner are commonly known as Boolean variables. A flag has typically just one of two possible values.

Furthermore, the value of a flag is often tested at least once throughout the program to determine if it is on (true or YES) or off (false or NO), and some action is made based on the test findings. The concept of a flag as true or false is most naturally translated into the numbers 1 and 0, respectively, in Objective-C. When we set the value of *isPrime* to 1 inside the program's loop, we are setting it to true to signify that *p* "is prime." If an even divisor is found during the execution of the inner for loop, the value of *isPrime* is set false to signal that *p* is no longer "is prime."

It is no wonder that the number 1 is commonly used to represent the true or on state, and the value 0 is generally used to represent the false or off state. This representation is analogous to the concept of a single bit in a computer. When the bit is turned on, its value is one; when it is turned off, its value is 0. However, there is an even stronger case for these logic values in Objective-C. It has to do with how the Objective-C programming language handles the concepts of true and false.

When we started talking about this chapter, we mentioned that if the criteria provided within the if statement is met, the following program statement is performed. But what does it mean to be satisfied? In Objective-C, satisfied implies nonzero and nothing else.

```
if ( 200 )
    NSLog (@"This will always print.");
```

Because the condition in the if statement (in this example, merely the value 200) is nonzero and therefore met, the NSLog statement is executed.

We employed the concepts of "nonzero means satisfied" and "zero signifies not satisfied" in this chapter's programs.

This is because when a relational expression is evaluated in Objective-C, it is assigned the value 1 if it is fulfilled and 0 if it is not satisfied. As a result, the statement's assessment:

```
if ( number < 0 )
    number = -number;
```

really goes like this: The number 0 of the relational expression is evaluated. If the condition is met, if the number is smaller than 0, the expression has a value of 1; otherwise, it has a value of 0.

The if statement validates the expression evaluation result. If the result is nonzero, the next statement is executed; otherwise, the statement is skipped.

The previous concept also applies to condition assessment within for, while, and do statements. The evaluation of compound relational expressions, such as the following statement, follows the same procedure as described previously:

```
while ( char != 'e' && count != 70 )
```

If both given requirements are met, the result is 1, but if either condition is not met, the result is 0. The evaluation findings are then verified. If the result is zero, the while loop ends; otherwise, it continues.

Returning to the program and the concept of flags, it is entirely legal in Objective-C to use an expression like this one to test if the value of a flag is true:

```
if ( isPrime )
```

This expression is usually associated with the following:

```
if ( isPrime != 0 )
```

The logical negation operator, `!`, is used to verify whether the value of a flag is false simply. The logical negation operator is used in the following expression to test if the value of `isPrime` is false (read this statement as “if not `isPrime`”):

```
if ( ! isPrime )
```

In general, an expression like this one undermines the logical value of the expression:

```
! expression
```

As a result, if the expression is 0, the logical negation operator returns a 1. If the outcome of the expression evaluation is nonzero, the negation operator returns a 0.

As seen in the following formula, the logical negation operator may use to flip the value of a flag simply:

```
mymove = ! mymove;
```

This operator has higher precedence than other relational operators and all binary arithmetic operators since it has the same precedence as the unary minus operator. To determine whether the value of a variable *x* exceeds the value of a variable *y*, as in:

```
! ( c < d )
```

The parenthesis is essential to guarantee that the expression is properly evaluated. Of course, we might have worded the same remark similarly:

```
c >= d
```

A few built-in Objective-C features make working with Boolean variables a bit easier. One example is the special type `BOOL`, which may use to declare variables with either a true or false value. The other is the predefined values `YES` and `NO`. Using preset values in our programs might help make them easier to develop and read. Examine program, which has been rebuilt to make use of these functionalities.

```
// Program to generate table of the prime numbers
// second version using the BOOL type and predefined
values
#import <Foundation/Foundation.h>
int main (int argc, char * argv[])
{
    @autoreleasepool {
        int p, d;
        BOOL isPrime;
        for ( p = 2; p <= 55; ++p ) {
            isPrime = YES;
            for ( d = 2; d < p; ++d )
                if ( p % d == 0 )
                    isPrime = NO;
            if ( isPrime == YES )
                NSLog ("%i ", p);
        }
    }
    return 0;
}
```

THE CONDITIONAL OPERATOR

The conditional operator is one of the most unusual operators in the Objective-C language. In contrast to all other operators in Objective-C, which are either unary or binary, the conditional operator is a ternary operator, which means it accepts three operands. The question mark (?) and the colon (:) are the two symbols used to represent this operator (:). The first operand comes before the:, second between the: and the third after the:.

The following is the general format of the conditional expression:

```
condition ? expression1 : expression2
```

In this syntax, condition is an expression, generally a relational expression, that the Objective-C system evaluates first whenever the conditional operator is encountered. Expression 1 is assessed, and the outcome of the evaluation becomes the operation result if the result of the condition evaluation is true (that is, nonzero). If condition evaluates false (zero), expression2 is evaluated, and its result becomes the operation's outcome.

When deciding which of two values to assign to a variable depending on a condition, conditional expressions are widely utilized. Let's say we have an integer variable *c* and another integer variable *s*. We might write the following line if we wished to assign -1 to *s* if *c* was less than 0, and *c*2 to *s* otherwise:

```
s = ( c < 0 ) ? -1 : c * c;
```

The condition *x* < 0 is the first tested when the prior statement is executed. To improve the readability of the statement, parentheses are usually inserted around the condition expression. This is frequently not necessary since the conditional operator has very low precedence lower than all other operators except the assignment operators and the comma operator.

If *c* is less than zero, the expression immediately after the? is evaluated. This expression is essentially the constant integer value -1, assigned to the variable *s* if *c* is less than zero.

If *x* is greater than zero, the expression immediately after the: is evaluated and assigned to *s*. If *c* is larger than or equal to zero, value of *c* * *c*, or *c*2, is assigned to *s*.

As another example of the conditional operator, the following sentence assigns the maximum of *x* and *y* to the variable *max_value*:

```
max_value = ( x > y ) ? x : y;
```

If the statement after the: (the “else” section) contains another conditional operator, the consequences of an else if clause can be achieved. For example, the sign function in Program may be implemented in a single program line using two conditional operators, as seen below:

```
sign = ( number < 0 )?  -1 : (( number == 0 )?  0 : 1);
```

If the number is less than zero, sign is assigned -1; if number is equal to zero, sign is assigned 0; otherwise, sign is assigned 1. The parenthesis surrounding the “else” element of the preceding expression is unneeded. Because conditional operator associates from right to left, multiple uses of this operator in the single statement, such as in, are possible.

```
e1 ? e2 : e3 ? e4 : e5
```

As a result, they are evaluated in the following order:

```
e1 ? e2 : ( e3 ? e4 : e5 )
```

Conditional expressions are not restricted to the right side of an assignment; they may use in any context where expressions can utilize. This implies that we may display the variable number’s sign without first assigning it to a variable using an NSLog command, as illustrated here:

```
NSLog(@"Sign = %i", ( number < 0 )?  -1
: ( number == 0 )?  0 : 1);
```

In this chapter, we covered decision-making statements with relevant examples.

Functions

IN THIS CHAPTER

- Defining Method
- Function Call by Value
- Function Call by Reference

In the previous chapter, we covered decision-making statements, and in this chapter, we will discuss functions.

FUNCTIONS IN OBJECTIVE-C

In Objective-C, a function is a set of statements that carry out a specific task. Every Objective-C program has a single C function named `main()`; even the most straightforward programs may declare additional functions. We may divide your code into many functions.

We must divide our code into discrete functions, but each function should logically perform a specific duty. A function declaration specifies the function's name, return type, and parameters to the compiler. The definition of a function specifies its body.

In Objective-C, the term for the function is a method. The Objective-C basic framework has many methods that our application may invoke.

The `appendString()` method, for example, appends one string to another. A method is known by several names, including function, subroutine, and process.

CREATING A METHOD

In the Objective-C programming language, a method definition takes the following general form:

```
- (return-type) method-name:( argument-Type1 )
argument-Name1
Joining-Argument2:( argument-Type2 )argument-Name2
.....
joining-Argumentn:( argument-Typen )argument-NameN {
    body of function
}
```

A method specification comprises a method header and a method body in the Objective-C programming language.

The following are the components of a method:

- **Return type:** A method may return a value as its returntype in Objective-C. The return type specifies the data type of the function's return value. Some methods perform the required operations but do not return a value. In this instance, the void is utilized as the returntype.
- **Method Name:** The actual name of the method. The method signature consists of the method's name and the list of its arguments.
- **Arguments:** An argument serves a similar purpose as a placeholder. When calling a function, we send a value-containing parameter. This value is referred to as the parameter or argument itself. The parameter list of a method describes the type, order, and the number of its arguments. Arguments are optional, and a method may not have any.
- **Joining Argument:** A joining argument makes it simpler to interpret and more clear when it is used.
- **Method Body:** The method body consists of statements that explain the method's functionality.

Example: The max() method's source code is shown below. This function takes two arguments, numbr1 and numbr2, and returns the greater of the two:

```

/* function returning max between the two numbers */
- (int) max:(int) numbr1 secondNumber:(int) numbr2 {

    /* local variable-declaration */
    int result;

    if (numbr1 > numbr2) {
        result = numbr1;
    } else {
        result = numbr2;
    }

    return result;
}

```

DECLARATIONS OF METHOD

A method declaration tells the compiler the function's name and how to call it. In Objective-C, the function's actual body can provide individually.

In Objective-C, a method declaration consists of the following elements:

```

- (return-type) function-name:( argument-Type1 )
argument-Name1
Joining-Argument2:( argument-Type2 )argument-Name2
....
Joining-Argumentn:( argument-Typen )argument-Namen;

```

The method declaration for the above-described function `max()` is as follows:

```

- (int) max:(int)numbr1 andNum2:(int)numbr2;

```

Method declaration is required when declaring a method in one source file and invoking it in another. In this case, the function should define at the beginning of the file that calls it.

CALLING A METHOD

When writing an Objective-C method, we specify what the function must do. We must call that function and accomplish the stated task using a method.

When a program calls a function, control is given to the called method. Program control is returned to the main program when a called method's return statement or function-ending closing brace is reached.

To invoke a method, supply the necessary arguments along with the method name; if the method returns a result, we may store it. As an illustration:

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

@interface SampleClass:NSObject
/* methoddeclaration */
- (int)max:(int)numbr1 andNumb2:(int)numbr2;
@end

@implementation SampleClass

/* method returning max between the two numbers */
- (int)max:(int)numbr1 andNumb2:(int)numbr2 {

    /* local variable-declaration */
    int result;

    if (numbr1 > numbr2) {
        result = numbr1;
    } else {
        result = numbr2;
    }

    return result;
}

@end

int main () {

    /* local variable-definition */
    int x = 300;
    int y = 200;
    int ret;

    SampleClass *sampleClass = [[SampleClass alloc]
init];

    /* calling the method to get max value */
    ret = [sampleClass max:x andNumb2:y];
```

```

    NSLog(@"Max valu : %ed\n", ret );
    return 0;
}

```

FUNCTION ARGUMENTS

If a function receives arguments, it must specify variables in Objective-C that accept the values of the parameters. These variables are referred to as the function's formal parameters in Objective-C.

The formal parameters work the same way as other local variables within the function, being formed upon entry and erased upon exit.

When calling a function, there are two ways to give parameters to it.

Sr. No.	Call Type and Description
1	Call by value This method converts an argument's true value into the function's formal parameter. In this scenario, changes to the parameter within the function do not affect the argument.
2	Call by reference This method puts the address of an argument into the formal parameter. Within the function, the address is used to retrieve the actual argument used in the call. This indicates that changes to the parameter affect the argument.

By default, Objective-C uses call-by-value to pass arguments. This generally means that code within a function cannot modify the arguments used to call the function, and the last example uses the `max()` method similarly.

In this chapter, we covered functions with relevant examples.

Appraisal

A general-purpose, object-oriented computer language called Objective-C enhances the C language with Smalltalk-style messaging.

Before the debut of Swift, Cocoa and Cocoa Touch were Apple's primary programming languages for the OS X and iOS operating systems and their corresponding interfaces for application development (APIs). Early in the 1980s, the program code Objective-C was created. The programming language Objective-C was invented in the early 1980s. It was selected as the core programming language for NeXT's NeXTSTEP operating system, the ancestor of both OS X and iOS.

Portable Objective-C programs that do not require the Cocoa or Cocoa Touch libraries, or employ components that can be ported or re-implemented for other platforms, may be created for any system that supports GNU Compiler Collection (GCC) or Clang.³

Typically, Objective-C "implementation" software files have .m file-name extensions, but Objective-C "header/interface" files have .h filename extensions, the same as C header files. The file extension for papers written in Objective-C++ is .mm. In the early 1980s, Objective-C was created primarily by Brad Cox and Tom Love at their company Stepstone. Both were introduced to Smalltalk at ITT's Programming Technology Center in 1981. Approximately about that time, the first Objective-C code appeared. In 1986, Cox explained Objective-C for the first time in his book *Object-Oriented Programming, An Evolutionary View*.

Although he was careful to stress that the problem of reusability includes more than just the language, Objective-C was often compared to other languages in terms of the features and differences. The year 1988 must have seen the licensing of the language and the development of the NeXTSTEP code library.

After acquiring NeXT in 1996, Apple Computer employed OpenStep in its new operating system, Mac OS X. This comprises Objective-C, NeXT's Objective-C-based developer tool Project Builder, and its interface design tool Interface Builder (both now merged into one Xcode application).

OpenStep interface objects comprise the bulk of Apple's current Cocoa API, and it is the most significant Objective-C development environment in use today. Swift, introduced by Apple at the 2014 WWDC, was billed as "Objective-C without the C." The history of the programming language Objective-C is extensive. Objective-C was voted the winner of the TIOBE Programming Language Award for 2011 in January 2012. Despite languishing as a niche language for most of that period, the advent of the iPhone has catapulted its popularity.

This award is presented to the language with the largest use increase over the last year; Objective-C rose from eighth to fifth place on the index in 2011. Objective-C may be traced back to International Telephone and Telegraph (ITT) division, where Tom Love and Brad Cox attempted to boost programmer efficiency. Alan Kay and his colleagues at Xerox PARC introduced Smalltalk, a revolutionary new programming language, in a special issue of *Byte* magazine in 1981.

Kay coined the word "object-oriented" to describe Smalltalk's modern approach to program design. Smalltalk envisioned programs as collections of objects that exchanged messages, thereby calling "methods" (perform actions). The invoked methods were selected "dynamically," that is, during program execution. A Smalltalk application is capable of real-time self-editing in response to user input.

Objective-C has existed for 35 years, making it older than most contemporary macOS and iOS programmers. Swift, launched in 2014, is quickly overtaking Objective-C and increasing its market share. Swift is apparently used in almost 50% of the top downloaded apps from the App Store, a significant increase in just five years. The million-dollar question is whether Objective-C or Swift should use from the outset for new iOS applications. Let's study Objective-C current's situation. First, no one has ever received an official statement from Apple on the deprecation of Objective-C support or a likely date when it would be terminated.

A substantial amount of Objective-C code is still being maintained, and several popular programs are written in this language. Consider: who would commit to converting these apps to a new language for no other reason? Consider how much time and money such a project would need with little visible return. We may thus assume that Objective-C will exist for the foreseeable future. Objective-C programmers may be certain that their skills will remain in demand. The situation has changed as new applications, and young developers choose to learn the language.

A COMPARISON OF OBJECTIVE-C AND SWIFT

Let's look at how Objective-C compares to Swift and what can tip the scales, in this case, to see if it's a good idea to use it for our new iOS app project.

Let's examine how Swift and Objective-C vary from one another.

- Because of runtime code compilation, Objective-C performance is slow. According to an official Apple release, Swift is 2.6 times more efficient than Objective-C.
- Objective-C inherits the code structure of C and, as a result, has a very challenging syntax with various special symbols. On the other hand, Swift is close to plain English, making it easier to learn than Objective-C.
- The biggest difference between Objective-C and Swift regarding code safety is how they handle null references. In Objective-C, a null pointer does not cause an application crash, making it difficult to detect and rectify. On the other hand, Null pointers cause detectable crashes in Swift. In general, Objective-C programs may have more hidden issues than Swift ones.
- Objective-C necessitates the upkeep of two code files: time- and effort-consuming. With Swift, we need to maintain one code file, saving time and human work.
- Objective-C frequently necessitates more lines of code than Swift.
- Objective-C does not allow dynamic libraries and instead relies on static ones. They are larger and have a negative impact on application performance. Swift uses dynamic libraries, which reduce the overall size of the program while improving performance and responsiveness.

Swift

Swift is a general-purpose programming language with a focus on safety and speed.

Chris Lattner and other Apple programmers designed it.

It was influenced by Objective-C, Rust, Ruby, and Python.

Objective-C

Objective-C is a general-purpose computer language that is considered a superset of C. It was designed to include object-oriented characteristics.

Brad Cox and Tom Love founded Stepstone.

C and Smalltalk served as inspiration for the language.

(Continued)

Swift

Swift made her debut appearance in 2014.

Swift type is fixed.

Swift is an Apache-licensed open-source project.

It just contains courses.

It was designed for the creation of iOS, Mac, Apple TV, and Apple Watch apps.

Swift polymorphism is not found directly.

It makes use of true and false values.

Objective-C has fewer template types than Swift.

Objective-C

In 1984, Objective-C was first introduced.

The Objective-C type is dynamic.

The Objective-C license is a General Public License.

Structs and classes are both present.

Smalltalk communications facilities are built into Objective-C.

Polymorphism exists instantly at compile time in Objective-C.

It makes use of YES and NO values, as well as BOOL.

Objective-C has fewer templates than Swift.

C AND OBJECTIVE-C

C Language

It is a procedure-based language.
Problems are handled in order.

The C programming language is a subset of Objective-C.

The pointers in the C programming language are vulnerable to security risks.

It is essentially a low-level language that is too similar to assembly language.

Classes are not available in the C programming language. Bjarne Stroustrup designed the C++ programming language to add object-oriented features to the C programming language, such as classes.

It employs a top-down programming approach.

Large amounts of program code are divided into smaller portions known as functions.

Objective-C

Objective-C is an object-oriented programming language. It defines the syntax and semantics of an object-oriented programming language. However, it does not support multiple inheritance properties.

The Objective-C programming language is a subset of the C programming language. In addition to the C programming language, it features classes and objects.

In comparison to C, Objective-C uses null pointers and is hence type-safe.

Objective-C is a high-level programming language that mixes C with a communications style similar to small talk.

Objective-C is a class-based object-oriented programming language with a dynamic runtime.

It employs a bottom-up programming approach.

This programming language divides large program code into smaller pieces known as Objects and Classes.

(Continued)

C Language

It only works with pointers.

Variables must declare at the start of a program.

Exception handling is not available in the C programming language.

The creation of functions with default arguments is forbidden in C.

Objective-C code cannot execute in C.

For dynamic memory allocation and deallocation, C has `malloc()`, `calloc()`, and `free()` functions.

The C programming language does not secure data.

There is no inline function available.

Overloading of functions and operators is not permitted in C.

It is also known as a function-driven language.

There is no support for encapsulation, data hiding, inheritance, polymorphism, or abstraction.

Templates are not supported in the C programming language.

It is advantageous for embedded services.

Objective-C

Pointers and references are both supported.

In this programming language, variables may declare anywhere.

Exception handling in Objective-C may be accomplished through catch and try sections.

It allows for the construction of functions with default arguments.

C code may be executed using Objective-C.

For memory allocation and deallocation, Objective-C has new and delete operators.

Objective-C security is achieved using the concept of encapsulation.

It supports the inline function.

Overloading of functions and operators is possible in Objective-C.

It is referred to as object-oriented programming.

The essential features of Objective-C are encapsulation, data hiding, inheritance, polymorphism, and abstraction.

Objective-C supports templates.

It is useful for gaming, networking, and so on.

C++

C++ is a high-level, procedural, object-oriented programming language. It was known as “C with Classes.” It is an extension of the C language.

Bjarne Stroustrup invented it at Bell Laboratories in 1980.

It supports multiple inheritances.

In C++, structures and classes are treated the same way.

Methods are used in C++ to call the implemented functionality.

It is a static, typed language.

The terms true, false, and bool are used.

Objective-C

The Objective-C programming language is a general-purpose object-oriented programming language that offers syntax and semantics for object-oriented programming. It is a superset of the C programming language.

Productivity Products International's Brad Cox and Tom Love created the device in 1980. (PPI).

It does not work with multiple inheritances.

Objective-C does not handle structures and classes in the same way.

Messaging is used in Objective-C to call implemented functionality.

It is a dynamically typed language.

It makes use of the YES, NO, and BOOL.

(Continued)

C++	Objective-C
There is a Standard Template Library.	It is missing template libraries.
C++ is used by organizations such as Evernote, LinkedIn, Opera, Microsoft, NASA, and Facebook.	Objective-C is used by Uber, Pinterest, Instagram, Slack, Instacart, and other companies.

MERITS

- The language was developed as a GCC preprocessor package for modern C compilers.
- Objective-C is better compatible with C++.
- Objective-C is more stable.
- Objective-C makes it easier to use private APIs.

DEMERITS

- Unlike C++, Objective-C does not support operator overloading.
- The first version of Objective-C did not support garbage collection, but it still uses a C-based runtime, increasing the program's size.
- It requires a header file to work correctly because it is dependent on C Compilers.
- Objective-C is a pretty difficult language, which is to be expected considering its age.

SHOULD WE CHOOSE OBJECTIVE-C?

Even a casual examination of the main features of both languages demonstrates that Swift outperforms Objective-C. Furthermore, one of Objective-C's advantages over Swift has recently been removed.

Swift 5, the most recent version of the programming language, now features ABI stability, which is an essential step toward achieving exceptional performance and security. Swift's lack of ABI stability was a significant concern until recently, although it has now been resolved.

With the release of the latest Swift version, Apple is encouraging developers to use it for future iOS applications. It makes sense because the younger language has several advantages over the older and more challenging Objective-C.

WHAT IS THE POINT OF LEARNING OBJECTIVE-C IN 2022?

Most macOS, XCode, and iOS kernel codes are written in C and C++. Because Objective-C is not a programming language but rather a massive preprocessor for pure C, it “fits in” with them rather well.

Apple’s whole ecosystem is built on Objective-C. Swift is mainly based on it. To understand how the system works from the inside and why Swift appears and acts the way it does, it is required to comprehend the “base.” Objective-C facilitates understanding of concepts such as the fact that not all NSObject objects are NSObject and the distinction between NSInteger, NSInteger, and NSInteger. Aside from how Swizzling works, what a Selector is, how the responder chain works, and so on. Swift has firmly contained and abstracted this, so it is not immediately evident. Furthermore, Objective-C, like any C language, aids knowledge of the underlying behavior of links, pointers, and memory.

IN DEVELOPMENT, OBJECTIVE-C IS STILL UTILIZED

To begin, it appears that any project that has lasted greater than two to three years has Objective-C code. It may be a hidden layer that has been made dependent and does not fit into the common code base, but it exists and requires regular maintenance: bug repairs and feature enhancements.

While specific programs cannot be rebuilt in Swift, doing so would require time and money that neither the business nor the customer is prepared to invest. As a result, the Objective-C database is growing.

Third, languages with a lower degree of abstraction than Swift are employed for complex operations with network, memory, and device resources. It might be Objective-C, Objective-C++, or merely C and C++, depending on the level of abstraction necessary.

BLOCKS IN OBJECTIVE-C

A class in Objective-C is a sort of object that combines data with related functionality. It is sometimes preferable to specify a single task or unit of action rather than a set of methods.

Blocks are a language-level feature that has been added to C, Objective-C, and C++, allowing us to create distinct blocks of code that are passed around to methods and functions like values. Because blocks are Objective-C objects, they should add to collections like NSArray or NSDictionary. They may also catch values from other scopes, making them similar to closures or lambdas in other programming languages.¹

Simple Block Declaration Syntax

```
returntype (^blockName)(argumentType);
```

Implementation of a Simple Block

```
returntype (^blockName)(argumentType) = ^{  
};
```

Example:

```
void (^simpleBlock)(void) = ^{  
    NSLog(@"This is block");  
};
```

Blocks Take Arguments and Return Values

Like methods and functions, blocks can accept arguments and return values.

Here's an example of creating and running a block with arguments and return values.

```
double (^multiplyTwoValues)(double, double) =  
    ^(double firstValue, double secondValue) {  
        return firstValue * secondValue;  
    };
```

```
double result = multiplyTwoValues(22,46);  
NSLog(@"The Result is %f", result);
```

Blocks Using the Type Definitions

Here's an example of how to use typedef in a block. Please remember that this example is not yet compatible with the online compiler. Use XCode to run the code.

```
#import <Foundation/Foundation.h>  
  
typedef void (^CompletionBlock)();  
@interface SampleClass:NSObject  
- (void)performActionWithCompletion:(CompletionBlock)  
completionBlock;  
@end  
  
@implementation SampleClass
```

```

- (void)performActionWithCompletion:(CompletionBlock)
completionBlock {

    NSLog(@"ActionPerformed");
    completionBlock();
}

@end

int main() {

    /* first program in the Objective-C */
    SampleClass *sampleClass = [[SampleClass alloc]
init];
    [sampleClass performActionWithCompletion:^(
        NSLog(@"Completion is call to intimate action is
performed");
    )];

    return 0;
}

```

NUMBERS IN OBJECTIVE-C

The following table covers the essential Objective-C methods for dealing with NSNumber to keep fundamental data types such as int, float, and bool in object form in the Objective-C programming language.

Sr. No.	Method and Description
1	+ (NSNumber *)numberWithBool:(BOOL)value Creates and returns NSNumber object with the provided value as a BOOL.
2	+ (NSNumber *)numberWithChar:(char)value Creates and returns NSNumber object with the specified value as a signed char.
3	+ (NSNumber *)numberWithDouble:(double)value Creates and returns NSNumber object with the specified value as a double.
4	+ (NSNumber *)numberWithFloat:(float)value Creates and returns NSNumber object with the specified value as a float.
5	+ (NSNumber *)numberWithInt:(int)value Creates and returns NSNumber object with the specified value as a signed int.
6	+ (NSNumber *)numberWithInteger:(NSInteger)value Creates and returns NSNumber object with the supplied value, treated as an NSInteger.
7	- (BOOL)boolValue Returns receiver's value as BOOL.

(Continued)

Sr. No.	Method and Description
8	- (char)charValue Returns receiver's value as char.
9	- (double)doubleValue Returns receiver's value as double.
10	- (float)floatValue Returns receiver's value as float.
11	- (NSInteger)integerValue Returns receiver's value as NSInteger.
12	- (int)intValue Returns receiver's value as an int.
13	- (NSString *)stringValue Returns receiver's value as a human-readable string.

Here's a simple example of NSNumber being used to multiply two integers and return the result.

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

@interface SampleClass:NSObject
- (NSNumber *)multiplyC:(NSNumber *)c withD:(NSNumber *)d;
@end

@implementation SampleClass

- (NSNumber *)multiplyC:(NSNumber *)c withD:(NSNumber *)d {
    float numb1 = [c floatValue];
    float numb2 = [d floatValue];
    float product = numb1 * numb2;
    NSNumber *result = [NSNumber numberWithFloat:product];
    return result;
}

@end

int main() {
    NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];

    SampleClass *sampleClass = [[SampleClass alloc] init];
    NSNumber *c = [NSNumber numberWithFloat:10.5];
    NSNumber *d = [NSNumber numberWithFloat:10.0];
    NSNumber *result = [sampleClass multiplyC:c withD:d];
```

```

NSString *resultString = [result stringValue];
NSLog(@"Product: %@", resultString);

[pool drain];
return 0;
}

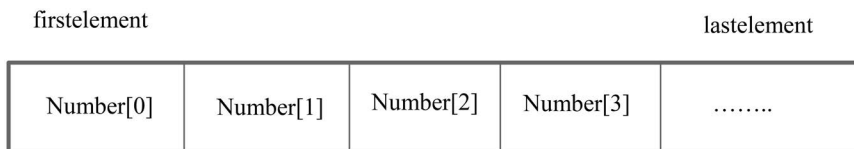
```

ARRAYS IN OBJECTIVE-C

The array data structure in the Objective-C programming language may carry a sequence of elements of the same type with a defined size. An array stores a collection of data; however, it is typically more practical to conceive an array as a collection of variables of same type.

We create a single array variable called `numbers` and use `numbers[0]`, `numbers[1,...]`, and `numbers[99]` to show the different variables instead of defining them individually as `number0`, `number1,...`, and `number99`.

An index is used to access a specific array element. All arrays consist of consecutive memory regions. The first element relates to the address with the lowest value, while the final element corresponds to the address with the most significant value.



Array in Objective-C.

DECLARING ARRAYS

In Objective-C, a programmer declares an array by providing the type of elements and number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is known as a single-dimensional array. The `arraySize` constant must be greater than zero, and the type can be any Objective-C data type that is allowed. To define a 10-element double array named `balance`, for example, use the following statement:

```
double balance[20];
```

`Balance` is now a variable array with a capacity of 10 double numbers.

ARRAYS INITIALIZATION

In Objective-C, we may initialize an array one by one or all at once, as seen below.

```
double balance[5] = {3000.0, 4.0, 5.4, 17.0, 64.0};
```

The number of values between the braces { } cannot be more than the number of array objects specified between the square brackets []. Here's an example of allocating a single array element.

If no array size is given, an array only large enough to hold the initialization is created. As a result, writing:

```
double balance[] = {3000.0, 4.0, 5.4, 17.0, 64.0};
```

We'll build the same array we did in the last example.

```
balance[4] = 64.0;
```

The previous statement gives the array's fifth item the value 64.0. Because all arrays have 0 as the first element's index, also known as the base index, the array with the fourth index will be the fifth, that is, the last element.

ACCESSING ARRAY ELEMENTS

To find an element, the array name is indexed. This is performed by enclosing the element's index in square brackets after the array name. As an illustration,

```
double salary = balance[7];
```

The code above selects the eighth element of the array and assigns its value to the salary variable. The following example uses all three of the previously described concepts: declaration, assignment, and array access.

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

int main () {
    int nmb[ 10 ];    /* nmb is an array of 10 integers
    */
    int c,d;

    /* initialize elements of array nmb to 0 */
```

```

    for ( c = 0; c < 10; c++ ) {
        nm[ c ] = c + 100;    /* set element at location
x to x + 100 */
    }

    /* output each array element's value */
    for (d = 0; d < 10; d++ ) {
        NSLog(@"The Element [%d] = %d\n", d, nmb[d] );
    }

    return 0;
}

```

ARRAYS IN OBJECTIVE-C IN DEPTH

Arrays are essential in Objective-C and require significantly more information. Any Objective-C developer should be familiar with the following array of concepts:

Sr. No.	Concept and Description
1	Multi-dimensional arrays Objective-C supports multidimensional arrays. The most fundamental type of multidimensional array is the two-dimensional array.
2	Passing arrays to functions By providing the array's name without an index, we may send a pointer to the procedure.
3	Return array from a function An Objective-C function can return an array.
4	Pointer to an array We may generate a reference to the first element of an array by just specifying the array name without any index.

POINTERS IN OBJECTIVE-C

Objective-C pointers are easy to understand and entertaining to use. Some Objective-C programming jobs are made simpler with pointers, while others, such as dynamic memory allocation, are impossible to do without them. As a result, learning tips are essential to becoming an experienced Objective-C programmer. Let us begin by taking simple and easy measures to learn them.

Each variable, as you may know, is a memory location. In Objective-C, each memory location has an address that may access by using the

ampersand (&) operator, which represents a memory address. Consider the following example, which outputs the variables' addresses.

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

int main () {
    int var1;
    char var2[10];

    NSLog(@"Address of the var1 variable is: %x\n",
&var1 );
    NSLog(@"Address of the var2 variable is: %x\n",
&var2 );

    return 0;
}
```

WHAT EXACTLY ARE POINTERS IN OBJECTIVE-C?

A pointer in Objective-C is a variable whose value is the direct address of a memory area, or the address of another variable. A pointer, like any variable or constant, must be defined before it may use to contain any variable address. A pointer variable declaration has the following general form:

```
type *varname;
```

The base type of a pointer in Objective-C is type, which must be a valid Objective-C data type, and the name of the pointer variable is varname. The asterisk * used to declare a pointer in Objective-C is the same asterisk used for multiplication. However, the asterisk denotes a variable as a pointer in this situation. The following are valid pointer declarations:

```
int      *ip;      /* pointer to integer */
double  *dp;      /* pointer to double */
float    *fp;      /* pointer to float */
char     *ch       /* pointer to character */
```

All pointer values, whether integer, float, character, or otherwise, have the same real data type: a lengthy hexadecimal number denoting a memory location. The only difference between pointers of the different data types is the variable or constant pointed to by the pointer.

HOW DO POINTERS WORK?

We will only use pointers for a few key activities regularly. (a) We construct a pointer variable, (b) assign a variable's address to a pointer variable, and (c) eventually access the value at the address accessible via the pointer variable. This is done with the unary operator `*`, which returns the variable's value at the place specified by its argument. These processes are used in the following illustration:

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

int main () {
    int var = 30;      /* actual variable declaration */
    int *ip;           /* pointer variable declaration */
    ip = &var;         /* store address of var in
pointer variable */

    NSLog(@"Address of var variable is : %x\n", &var
);

    /* address stored in the pointer variable */
    NSLog(@"Address stored in ip variable is : %x\n",
ip );

    /* access value using the pointer */
    NSLog(@"The Value of *ip variable is : %d\n", *ip
);

    return 0;
}
```

OBJECTIVE-C NULL POINTERS

If the allocated address is unknown, assigning a NULL value to a pointer variable in Objective-C is typically a good idea. This is done during the variable declaration process. In Objective-C, a null pointer is a pointer that has the value NULL allocated to it.

The NULL pointer is a zero-valued constant specified in several standard libraries in Objective-C. Examine the following program:

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

int main () {
```

```

int  *ptr = NULL;
NSLog(@"Value of ptr : %x\n", ptr );
return 0;
}

```

When the given code in Objective-C is compiled and executed, the following result is obtained:

```
Value of ptr: 0
```

The majority of operating systems prevent applications from accessing memory that is reserved by the operating system at address 0. Memory address 0 is significant because it suggests that the pointer is not meant to point to a memory location that is easily accessible. If a pointer has the value null (zero), it is assumed to point to nothing.

Use the following if statement to check for a null pointer:

```

if(ptr)      /* succeeds if c is not null */
if(!ptr)     /* succeeds if c is null */

```

DETAILS ABOUT OBJECTIVE-C POINTERS

Pointers are essential in Objective-C programming and consist of numerous simple concepts. Any Objective-C developer should understand the following essential pointer concepts:

Sr. No.	Concept and Description
1	Objective-C – Pointer arithmetic On pointers, available 4 arithmetic operators are: ++, --, +, –
2	Objective-C – Array of pointers Arrays may use to contain a collection of pointers.
3	Objective-C – Pointer to pointer Objective-C supports the pointer on the pointer and so on.
4	Passing pointers to functions in Objective-C The called function has the ability to modify the given parameter in the calling code when an argument is supplied by reference or address.
5	Return pointer from functions in Objective-C A function in Objective-C may return a reference to a local variable, a static variable, or dynamically allocated memory.

STRINGS IN OBJECTIVE-C

In the Objective-C programming language, the string is represented by NSString, and its subclass NSMutableString provides several methods for

generating string objects. The @"..." syntax in Objective-C is the simplest way to create a string object.

```
NSString *greeting = @"Hey";
```

Here's a simple example of creating and printing a string.

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

int main () {
    NSString *greeting = @"Hey";
    NSLog(@"Greeting message: %@\n", greet );

    return 0;
}
```

Objective-C has a multitude of string manipulation capabilities.

Sr. No.	Method and Purpose
1	- (NSString *)capitalizedString; Returns receiver's capitalized representation.
2	- (unichar)characterAtIndex:(NSUInteger)index; The character at the given array position is returned.
3	- (double)doubleValue; As a double, this method returns the floating-point value of the receiver's text.
4	- (float)floatValue; As a float, it returns the floating-point value of the receiver's text.
5	- (BOOL)hasPrefix:(NSString *)aString; A Boolean value indicates if a provided string matches the receiver's beginning characters.
6	- (BOOL)hasSuffix:(NSString *)aString; A Boolean value indicates if a provided string matches the receiver's terminating characters.
7	- (id)initWithFormat:(NSString *)format ...; Returns an NSString object produced by using a format string as a template and then inserting the remaining input values.
8	- (NSInteger)integerValue; NSInteger value of the receiver's text is returned.
9	- (BOOL)isEqualToString:(NSString *)aString; Using a literal Unicode-based comparison returns a Boolean result showing if a string is given and the receiver is equal.
10	- (NSUInteger)length; The receiver's Unicode character number is returned.

(Continued)

Sr. No.	Method and Purpose
11	- (NSString *)lowercaseString; The receiver's lowercase representation is returned.
12	- (NSRange)rangeOfString:(NSString *)aString; The range of the receiver's first occurrence of a given string is returned.
13	- (NSString *)stringByAppendingFormat:(NSString *)format ...; Returns a string constructed by attaching to the receiver a string made from a provided format string and the following arguments.
14	- (NSString *)stringByTrimmingCharactersInSet:(NSCharacterSet *)set; Returns a new string formed by removing characters from both ends of the receiver section of a given character set.
15	- (NSString *)substringFromIndex:(NSUInteger)anIndex; Returns a new string containing the receiver's characters from the supplied index to the end.

The following example makes use of some of the functions outlined above:

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

int main () {
    NSString *str1 = @"Hey";
    NSString *str2 = @"Everyone";
    NSString *str3;
    int len ;

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

    /* uppercase-string */
    str3 = [str2 uppercaseString];
    NSLog(@"Uppercase-String : %@\n", str3 );

    /* concatenates the str1 and str2 */
    str3 = [str1 stringByAppendingFormat:@"Everyone"];
    NSLog(@"Concatenated-string:  %@\n", str3 );

    /* total length of str3 after the concatenation */
    len = [str3 length];
    NSLog(@"Length of the Str3 : %d\n", len );

    /* initWithFormat */
```

```

    str3 = [[NSString alloc] initWithFormat:
@"%@ %@", str1, str2];
    NSLog(@"Using initWithFormat:  %@\n", str3 );
    [pool drain];

    return 0;
}

```

STRUCTURES IN OBJECTIVE-C

Arrays in Objective-C allow us to create variable types that can hold many data objects of the same kind. Nonetheless, the structure is another user-defined data type available in Objective-C programming that allows us to blend data items of different types.

Data is represented using structures. Let's say we want to maintain track of your library books. We might want to keep the following aspects of each book in mind:

- Title
- Author
- Book ID
- Subject

Creating a Structure

To define a structure, use the `struct` statement. The `struct` command generates a new data type with several members for our code. The `struct` statement format is given below:

```

struct [structure tag] {
    memberdefinition;
    memberdefinition;
    .....
    Memberdefinition;
} [one or more structure variables];

```

Each member definition in Objective-C is a standard variable definition, such as `int`, `float`, or any other acceptable variable definition, and the structure tag is optional. Although it is not required, we can declare one or more structural variables after the structure's definition but before the final semicolon. This is how we would describe the `Book` structure.

```
struct Books {
    NSString *title;
    NSString *authors;
    NSString *subjects;
    int    bookid;
} book;
```

Access to Structure Members

The member access operator is used to acquire access to any structure member (.). In Objective-C, the member access operator is represented by a period between the name of the structure variable and the name of the structure member we want to access. The struct keyword would use to define variables of the structure type. The following example shows how to use structure.

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

struct Books {
    NSString *title;
    NSString *authors;
    NSString *subjects;
    int    bookid;
};

int main() {
    struct Books Book1;          /* Declare Book1 of the
type Book */
    struct Books Book2;          /* Declare Book2 of the
type Book */

    /* book1-specification */
    Book1.title = @"Objective-C Programming";
    Book1.authors = @"Luka Mli";
    Book1.subjects = @" Tutorial of Objective-C
Programming";
    Book1.bookid = 4387307;

    /* book2-specification */
    Book2.title = @"Telecom Billing";
    Book2.authors = @"Sara Ali";
    Book2.subjects = @" Tutorial of Telecom Billing";
    Book2.bookid = 9875701;
```

```

/* info of print Book1 */
NSLog(@"Book1 title : %@\n", Book1.title);
NSLog(@"Book1 authors : %@\n", Book1.authors);
NSLog(@"Book1 subjects : %@\n", Book1.subjects);
NSLog(@"Book1 bookid : %d\n", Book1.bookid);

/* print Book2 info */
NSLog(@"Book2 title : %@\n", Book2.title);
NSLog(@"Book2 authors : %@\n", Book2.authors);
NSLog(@"Book2 subjects : %@\n", Book2.subjects);
NSLog(@"Book2 bookid : %d\n", Book2.bookid);

return 0;
}

```

Function Arguments as Structures

Like any other variable or pointer, a structure can be sent as a function argument. We'd access structural variables the way we did in the previous example.

```

#import <Foundation/Foundation.h>

struct Books {
    NSString *title;
    NSString *authors;
    NSString *subjects;
    int bookid;
};

@interface SampleClass:NSObject
/* functiondeclaration */
- (void) printBook:( struct Books) book ;
@end

@implementation SampleClass

- (void) printBook:( struct Books) book {
    NSLog(@"Book title is: %@\n", book.title);
    NSLog(@"Book authors is: %@\n", book.authors);
    NSLog(@"Book subjects is: %@\n", book.subjects);
    NSLog(@"Book bookid is: %d\n", book.bookid);
}

@end

```

```

int main() {
    struct Books Book1;           /* Declare Book1 of the
type Book */
    struct Books Book2;           /* Declare Book2 of the
type Book */

    /* book1 specification */
    Book1.title = @"Objective-C Programming";
    Book1.authors = @"Pluka Emli";
    Book1.subjects = @" Sessions of Objective-C
Programming";
    Book1.bookid = 5677307;

    /* book2 specification */
    Book2.title = @"Telecom Billing";
    Book2.author = @"Shaara Ali";
    Book2.subject = @" Sessions of Telecom Billing";
    Book2.book_id = 7895701;

    SampleClass *sampleClass = [[SampleClass alloc]
init];
    /* print Book1 info */
    [sampleClass printBook: Book1];

    /* Print Book2 info */
    [sampleClass printBook: Book2];

    return 0;
}

```

Pointers to Structures

We may define pointers to structures in the same manner that we define pointers to any other variable, as seen below:

```
struct Books *structpointer;
```

In Objective-C, a structural variable's address can now place in the previously stated pointer variable. To find address of a structure variable in Objective-C, use the & operator before the structure's name, as illustrated below:

```
Structpointer = &Book1;
```

Use the `->` operator to access structure members from a pointer to that structure, as demonstrated below:

```
Structpointer->title;
```

Let us recreate the above example using a structure pointer; perhaps this will help us understand the concept.

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

struct Books {
    NSString *title;
    NSString *authors;
    NSString *subjects;
    int    book-id;
};

@interface SampleClass:NSObject
/* functiondeclaration */
- (void) printBook:( struct Books *) book ;
@end

@implementation SampleClass
- (void) printBook:( struct Books *) book {
    NSLog(@"Book title : %@\n", book->title);
    NSLog(@"Book authors : %@\n", book->authors);
    NSLog(@"Book subjects : %@\n", book->subjects);
    NSLog(@"Book bookid : %d\n", book->bookid);
}

@end

int main() {
    struct Books Book1;          /* Declare Book1 of the
    type Book */
    struct Books Book2;          /* Declare Book2 of the
    type Book */

    /* book1 specification */
    Book1.title = @"Objective-C Programming";
    Book1.authors = @"Luka Mli";
```

```

    Book1.subjects = @" Session of Objective-C
Programming";
    Book1.bookid = 4387307;

    /* book2 specification */
    Book2.title = @"Telecom Billing";
    Book2.authors = @"Sara Ali";
    Book2.subjects = @" Session of Telecom Billing";
    Book2.bookid = 9875701;

    SampleClass *sampleClass = [[SampleClass alloc]
init];
    /* print Book1 info by passing address of the Book1
*/
    [sampleClass printBook:&Book1];

    /* print Book2 info by passing address of the Book2
*/
    [sampleClass printBook:&Book2];

    return 0;
}

```

Bit Fields

Data may be packed into a structure using Bit Fields. This is especially useful when memory or the data storage is at a limit.

Packing many things into a machine word, for example. Flags of one bit can compress.

If the file format is not standard, it can be read in. Consider 9-bit integers as an example.

By adding :bit length after the variable in an Objective-C structure declaration, we may do this. An example would be as follows:

```

struct packed_struct {
    unsigned int f1:1;
    unsigned int f2:1;
    unsigned int f3:1;
    unsigned int f4:1;
    unsigned int type:4;
    unsigned int my_int:9;
} pack;

```

In this scenario, the `packed_struct` contains six members: A4 bit type, four 1-bit flags `f1..f3`, and a 9-bit `my_int`

If the maximum length of the bit field is less than or equal to the integer word length of the machine, Objective-C automatically packs the bit fields indicated above as tightly as it can. If this is not the case, specific compilers may allow for field memory overlap, while others may store the next field in the word after it.

PREPROCESSORS IN OBJECTIVE-C

The Objective-C Preprocessor is a separate step in the compilation process that is not included in the compiler. An Objective-C Preprocessor is just a text replacement tool that instructs the compiler to conduct the required pre-processing before compilation. The Objective-C Preprocessor will be termed OCPP.

All preprocessor instructions are preceded by a pound symbol (#). It must be the first non-blank character for readability, and a preprocessor directive should begin in the first column. The following section contains all of the essential preprocessor directives.

Sr. No.	Directive and Description
1	<code>#define</code> Preprocessor macro is replaced.
2	<code>#include</code> Inserts a specific header from another file.
3	<code>#undef</code> Undefines preprocessor macro
4	<code>#ifdef</code> If this macro is defined, it will return true
5	<code>#ifndef</code> If this macro is not defined, it will return true.
6	<code>#if</code> Checks if the compile-time condition is true.
7	<code>#else</code> Alternative for <code>#if</code>
8	<code>#elif</code> <code>#else</code> an <code>#if</code> in one statement
9	<code>#endif</code> Ends preprocessor conditional
10	<code>#error</code> Prints error message on the <code>stderr</code>
11	<code>#pragma</code> Issues special commands to the compiler using the standardized method

Examples of preprocessors: Examine the examples below to understand the various directives better.

```
#define MAXARRAY_LENGTH 22
```

The OCPP is instructed to replace MAXARRAY_LENGTH instances with 20. For constants, use #define to increase readability.

```
#import <Foundation/Foundation.h>
#include "myheader.h"
```

These directives instruct the OCPP to get the foundation.h from the Foundation Framework and insert it into the current source file. The next line tells OCPP to retrieve myheader.h from the local directory and put it into the current source file.

```
#undef FILE_SIZE
#define FILE_SIZE 41
```

This informs the OCPP to undefine the current FILE_SIZE to 41.

```
#ifndef MESSAGE
    #define MESSAGE "We want"
#endif
```

This informs the OCPP to define MESSAGE only if it has not been defined previously.

```
#ifdef DEBUG
    /* here are our debugging statements */
#endif
```

If DEBUG is used, the OCPP is instructed to process the statements provided. This is required if the - DDEBUG option is passed to the gcc compiler during compilation. This specifies DEBUG, allowing us to turn to debug on and off during compilation.

PREDEFINED MACROS

ANSI C defines a number of macros. Although each is available for use in programming, the predefined macros are not directly updated.

Sr. No.	Macro and Description
1	<code>__DATE__</code> Current date as the character literal in “MMM DD YYYY” format
2	<code>__TIME__</code> Current time as a character literal in the “HH:MM:SS” format
3	<code>__FILE__</code> This contains current filename as a string literal.
4	<code>__LINE__</code> This contains current line number as a decimal constant.
5	<code>__STDC__</code> Defined as 1 when compiler complies with the ANSI standard.

Consider the following scenario:

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

```
int main() {
    NSLog(@"File is :%s\n", __FILE__ );
    NSLog(@"Date is :%s\n", __DATE__ );
    NSLog(@"Time is :%s\n", __TIME__ );
    NSLog(@"Line is :%d\n", __LINE__ );
    NSLog(@"ANSI is :%d\n", __STDC__ );

    return 0;
}
```

OPERATORS OF PREPROCESSORS

The Objective-C preprocessor includes the following operators to help us build macros.

Macro Continuation (\)

A macro is often limited to a single line. In Objective-C, use the macro continuation operator to continue a macro that is too lengthy for a single line. As an illustration,

```
#define message_for(c, d) \
    NSLog(@"#c " and " #d ": We will miss you\n")
```

Stringize (#)

When used within a macro definition, the stringize or number-sign operator (`#`) converts a macro parameter into a string constant. Use this

operator with a predetermined argument or parameter list in a macro. As an illustration:

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

#define message_for(c, d) \
    NSLog(@"#c " and " #d ": We will miss you\n")

int main(void) {
    message_for(Carole, Debra);
    return 0;
}
```

Token Pasting (##)

The token-pasting operator (##) connects two arguments within a macro declaration. It allows us to combine two separate tokens in the macro definition into a single token. As an illustration:

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

#define tokenpaster(n) NSLog(@"token" #n " = %d",
token##n)

int main(void) {
    int token24 = 80;

    tokenpaster(29);
    return 0;
}
```

defined() Operator

The preprocessor defined operator is used in constant expressions to check whether an identifier has been defined using #define. The value is true (non-zero) if the specified identifier is defined. If no symbol is specified, the value is false (zero). The following is the defined operator's definition:

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

#if !defined (MESSAGE)
    #define MESSAGE "We Walk"
#endif
```

```
int main(void) {
    NSLog(@"Message is: %s\n", MESSAGE);
    return 0;
}
```

PARAMETERIZED MACROS

The capability of OCPP to mimic functions using parameterized macros is one of its most important features. For instance, we might use the following code to square a number:

```
int square(int d) {
    return d * d;
}
```

We may alter the above code using the following macro:

```
#define square(d) ((d) * (d))
```

Macros with arguments must be defined with the `#define` directive before they may use. The parenthesized argument list must occur directly after the macro name. There may be no spaces between the macro name and the open parenthesis. As an illustration:

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

#define MAX(c,d) ((c) > (d)? (c) : (d))

int main(void) {
    NSLog(@"Max between 60 and 40 is %d\n", MAX(60,
40));
    return 0;
}
```

Typedef IN OBJECTIVE-C

The `typedef` keyword in the Objective-C programming language is used to rename a type. An example of how to define the word `BYTE` for one-byte numbers is given below:

```
typedef unsigned char BYTE;
```

Following this type definition, the identifier `BYTE` can be used to abbreviate the type `unsigned char`, such as:

```
BYTE  by1, by2;
```

The convention uses uppercase letters in these definitions to remind the user that the type name is simply a symbolic abbreviation. However, as seen below, lowercase letters can also utilize.

```
typedef unsigned char byte;
```

We may also use typedef to name a user-defined data type. For example, we might use typedef with the structure to create a new data type and then use that data type to define structure variables explicitly, as seen below:

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

typedef struct Books {
    NSString *title;
    NSString *authors;
    NSString *subjects;
    int bookid;
} Book;

int main() {
    Book book;
    book.title = @"Objective-C Programming";
    book.authors = @"PeeksPoint";
    book.subjects = @"Programming-Lecture";
    book.bookid = 400;

    NSLog( @"Book title : %@\n", book.title);
    NSLog( @"Book authors : %@\n", book.authors);
    NSLog( @"Book subjects : %@\n", book.subjects);
    NSLog( @"Book Id : %d\n", book.bookid);

    return 0;
}
```

typedef vs #define

The #define directive in Objective-C is used to declare aliases for multiple data types, similar to typedef, but with the following differences:

- Typedef can only provide types symbolic names; however, #define may also use to define aliases for values, such as 1 as ONE, and so on.
- Typedef declarations are interpreted by the compiler, whereas #define statements are processed by the pre-processor.

The simplest basic application of #define is as follows:

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

```
#define TRUE 1
#define FALSE 0

int main( ) {
    NSLog( @"Value of TRUE is: %d\n", TRUE);
    NSLog( @" Value of FALSE is: %d\n", FALSE);

    return 0;
}
```

TYPE CASTING IN OBJECTIVE-C

Type casting is a technique for transforming one data type to another. To convert a long number to a basic integer, we may use the type cast long to int. As seen below, we may use the cast operator explicitly to convert values from one type to another.

```
(typename) expression
```

For floating-point operations in Objective-C, we frequently utilize CGFloat, derived from the basic float type in 32-bit situations and double in 64-bit ones. Consider the following example, in which the cast operator in a floating-point operation divides one integer variable by another:

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

int main() {
    int sum = 47, count = 35;
    CGFloat mean;

    mean = (CGFloat) sum / count;
    NSLog(@"Value of mean is : %f\n", mean );

    return 0;
}
```

It is important to note that the cast operator takes precedence over the division operator. As a result, the total value is converted to double before being divided by count, yielding a double value.

Type conversions can be implicit (performed automatically by the compiler) or explicit (performed by the user) (expressed explicitly using the cast operator). It is considered excellent programming practice to use the cast operator when type conversions are necessary.

INTEGER PROMOTION

When values of the integer type “smaller” than int or unsigned int are transformed to int or unsigned int, this is known as integer promotion. Consider adding a character to an int.

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

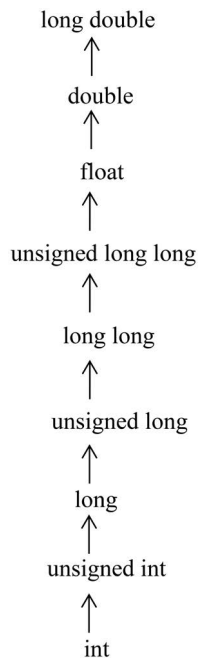
int main() {
    int y = 57;
    char c = 'c';
    int sum;

    sum = y + c;
    NSLog(@"Value of sum is: %d\n", sum );

    return 0;
}
```

USUAL ARITHMETIC CONVERSION

The standard arithmetic conversions are performed implicitly to cast their values in a common type. If the operands continue to be of different types, the compiler changes them to the highest type in the following hierarchy.



Usual arithmetic conversions.

The assignment and logical operators `&&` and `||` do not go through standard arithmetic conversions. Consider the following example to understand the idea better.

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

int main() {
    int y = 47;
    char c = 'c';
    CGFloat sum;

    sum = y + c;
    NSLog(@"Value of sum is: %f\n", sum );
    return 0;
}
```

LOG HANDLING IN OBJECTIVE-C

In this part, we will discuss log handling and provide examples.

NSLog METHOD

We use the Objective-C programming language's `NSLog` function to print logs, which we first used in the Hey World example.

Consider the following basic code: “Hey Everyone.”

DISABLING LOGS IN THE LIVE APPS

Because we use `NSLogs` in our application, it will be written in device logs, which is undesirable in a live build. As a result, we use a type definition for printing logs, as shown below:

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

#if DEBUG == 0
#define DebugLog(...)
#elif DEBUG == 1
#define DebugLog(...) NSLog(__VA_ARGS__)
#endif

int main() {
    DebugLog(@"Debug log, custom addition gets \
printed during debug " );
    NSLog(@"NSLog gets print" );
    return 0;
}
```

ERROR HANDLING IN OBJECTIVE-C

The Foundation framework's `NSError` class handles error handling in Objective-C programming.

An `NSError` object has more extensive and expandable error information than an error code or error text. An `NSError` object has three basic properties: an error domain (represented as a text), a domain-specific error code, and a user info dictionary that contains application-specific information.

`NSError`

In Objective-C programs, `NSError` objects convey information about runtime errors that users should be aware of. A program will often display this error information in a dialog or sheet. It may, however, analyze data and either request that the user attempt to recover from the error or attempt to resolve the problem on its own.

The `NSError` Object is made up of the following components:

- **Domain:** The error domain cannot be null and must be one of the predefined `NSError` domains or an arbitrary string indicating a new domain.
- **Error code:** This is the error code.
- **User Info:** The `userInfo` dictionary of the error may be null.

The example shows how to create a custom error.

```
NSString *domain = @"com.MyCompany.MyApplication.
ErrorDomain";
NSString *desc = NSLocalizedString(@"Process is unable
to be completed", @"");
NSDictionary *userInfo = @{ NSLocalizedDescriptionKey
: desc };
NSError *error = [NSError errorWithDomain:domain
code:-101 userInfo:userInfo];
```

Here is the complete code for the error as mentioned above example, which was passed as a pointer reference:

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

```

@interface SampleClass:NSObject
- (NSString *) getEmployeeNameForID:(int) id
withError:(NSError **)errorPtr;
@end

@implementation SampleClass

- (NSString *) getEmployeeNameForID:(int) id
withError:(NSError **)errorPtr {
    if(id == 1) {
        return @"Employee TestName";
    } else {
        NSString *domain = @"com.MyCompany.
MyApplication.ErrorDomain";
        NSString *desc =@"Process is unable to be
completed";
        NSDictionary *userInfo = [[NSDictionary alloc]
initWithObjectsAndKeys:desc,
@"NSLocalizedDescriptionKey",NULL];
        *errorPtr = [NSError errorWithDomain:domain
code:-101
userInfo:userInfo];
        return @"";
    }
}

@end

int main() {
    NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
    SampleClass *sampleClass = [[SampleClass alloc]
init];
    NSError *error = nil;
    NSString *name1 = [sampleClass
getEmployeeNameForID:1 withError:&error];

    if(error) {
        NSLog(@"Errorfinding Name1: %@",error);
    } else {
        NSLog(@"Name1: %@",name1);
    }
}

```

```

    error = nil;
    NSString *name2 = [sampleClass
getEmployeeNameForID:2 withError:&error];

    if(error) {
        NSLog(@"Errorfinding Name2: %@",error);
    } else {
        NSLog(@"Name2: %@",name2);
    }

    [pool drain];
    return 0;
}

```

COMMAND-LINE ARGUMENTS

When they run, we may provide particular values from the command line to our Objective-C applications. These variables are referred to as command-line arguments and are frequently useful for our program, especially when we wish to control our program from outside rather than hard-coding such values inside the code.

The `main()` function parameters handle command-line arguments, where `argc` is the number of arguments given and `argv[]` is a pointer array referring to each argument passed to the program. The example below shows how to check for command-line parameters and take necessary action.

```

#import <Foundation/Foundation.h>

int main( int argc, char *argv[] ) {
    if( argc == 3 ) {
        NSLog(@"Argument supplied is %s\n", argv[1]);
    } else if( argc > 3 ) {
        NSLog(@"supplied many arguments.\n");
    } else {
        NSLog(@"One argument expect.\n");
    }
}

```

The output of running the aforementioned code with only one parameter, let's say "testing," is as follows.

The argument supplied is testing.

When the code below is created and executed with two arguments, say `testing1` and `testing2`, the following result is obtained and passed too many arguments.

The following result is achieved when the code is compiled and executed without any parameters.

Argument is expected.

It should notice that `argv[0]` includes the program name, `argv[1]` is the first command-line parameter supplied, and `*argv[n]` is the last argument. If no arguments are supplied, `argc` is set to one; otherwise, if one argument is specified, `argc` is set to 3.

A space separates all command-line arguments. If the argument contains a space, it can be passed by surrounding it in double quotes "" or single quotes ". Rewrite the above example, but this time include the program name and a command-line parameter inside double quotes:

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

int main( int argc, char *argv[] ) {
    NSLog(@"Program name is %s\n", argv[0]);

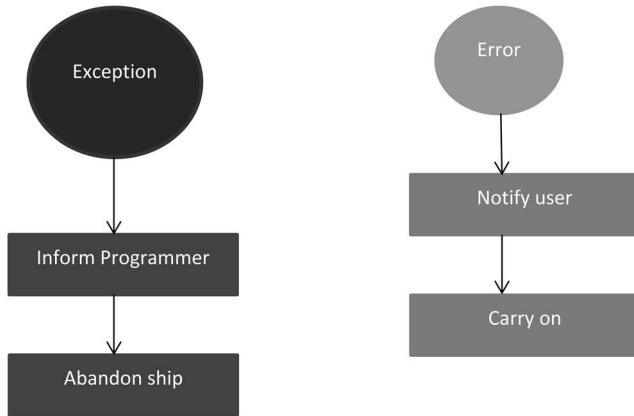
    if( argc == 3 ) {
        NSLog(@"Argument supplied is %s\n", argv[1]);
    } else if( argc > 3 ) {
        NSLog(@"supplied many arguments.\n");
    } else {
        NSLog(@"Argument expected.\n");
    }

    return 0;
}
```

EXCEPTIONS AND ERRORS

Two types of errors can arise when a program is written in Objective-C. Unexpected errors are “severe” programming mistakes that cause our application to stop. These are referred to as exceptions because they represent an unusual event in our software. Expected mistakes, on the other hand, occur naturally during the execution of a program and may be used to evaluate the success of an activity. These are referred to as errors.

We might also consider the contrast between exceptions and errors as distinguishing their target audiences. In general, exceptions are used to notify the programmer of a problem, whereas errors notify the user that a requested action is not being accomplished.



Control flow of exception and error.

Attempting to access a non-existent array index, for example, is an exception (a programming fault), but failing to open a file is an error (a user error). In the first case, something went horribly wrong with our program's flow, and it should have stopped immediately following the exception. In the latter scenario, we'd want to notify the user that the file couldn't be opened and maybe request that they try again, but there's no reason our software couldn't keep running after the mistake.

EXCEPTION HANDLING

The main benefit of Objective-C exception handling capabilities is that they separate error detection from error management. When a portion of code detects an exception, it may “throw” it to the nearest error handling block, which may “catch” and handle specific issues appropriately. The ability to throw exceptions from anywhere eliminates the need to constantly check for success or failure messages from each function involved in a particular job.

`@throw` is used to discover exceptions, whereas `@try`, `@catch()`, and `@finally` are used to catch and manage them. We should be familiar with these exception handling constructs if we have worked with exceptions in C#.

It is critical to understand that exceptions in Objective-C are somewhat slow. As a result, its use is limited to identifying serious programming faults rather than basic control flow. Please refer to the section on Error Handling if we are attempting to determine what to do in the case of an expected problem (e.g., inability to load a file).

THE CLASS `NSException`

Exceptions are represented by instances of the `NSException` class or a subclass thereof. This is a straightforward approach for encapsulating all relevant information about an exception. The following are the three qualities of an exception:

- **name** – An instance of `NSString` that uniquely identifies the exception.
- **reason** – An `NSString` object describing the exception in a human-readable format.
- **userInfo** – An instance of `NSDictionary` holding exception-specific information.

Several variables in the Foundation framework specify “standard” exception names. These strings are used to determine the type of exception that was caught.

Using the `initWithName:reason:userInfo:` initialization function, we can easily create new exception objects with specific data. It is possible to catch and throw custom exception objects by following the same steps outlined in the following sections.

MAKING EXCEPTIONS

Let’s start with the default exception-handling behavior of a program. When we attempt to access an index that does not exist in Objective-C, the `objectAtIndex:` function of `NSArray` returns an `NSRangeException` (a subclass of `NSException`). So, if we request the tenth item in a three-component array, we’ll have an exception to play with:

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

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

```

    NSArray *crew = [NSArray arrayWithObjects:
                      @"Drave",
                      @"Pgwood",
                      @"Daank", kin];

    // throw-exception.
    NSLog(@"%@", [crew objectAtIndex:10]);

}
return 0;
}

```

When an uncaught exception occurs, Xcode pauses the program and displays the line that produced the error.

```

NSLog(@"%@", [crew objectAtIndex:10]); thread1: signal
SIGABRT

```

CATCHING EXCEPTIONS

To manage exceptions, any code that may result in one should be put in a `@try` block. The `@catch()` directive is then applied to catch specific exceptions. We may execute any housekeeping code in a `@finally` block if necessary. All three of these exception-handling instructions are demonstrated in the example below:

```

@try {
    NSLog(@"%@", [crew objectAtIndex:10]);
}
@catch (NSEException *exception) {
    NSLog(@"Caught an exception");
    // We will silently ignore exception.
}
@finally {
    NSLog(@"Cleaningup");
}

```

The message `[crew objectAtIndex:10]` causes the software to produce an `NSRangeException`, which is caught by the `@catch()` instruction. The actual exception handling takes place within the `@catch()` clause. In this instance, we just display a helpful error message, but in other circumstances, we'll probably need to create code to address the problem.

When an error is detected in the `@try` block, the program moves to the appropriate `@catch()` block, preventing any code after the exception from being executed. This is an issue if the `@try` block needs to be cleaned up (e.g., if it opened a file, it needs to be closed). This issue is resolved by the `@finally` block since its execution is assured regardless of whether an exception was thrown or not. This makes it an excellent spot for resolving any outstanding issues with the `@try` block.

The parentheses after the `@catch()` directive let us define the type of exception to catch. In this case, the exception is an `NSException`, which is the most fundamental exception type. An exception, on the other hand, can be of any type, not only `NSException`. For instance, the following `@catch()` directive will handle a generic object:

```
@catch (id genericException)
```

EXCEPTION THROWING

In our code, whenever we come into unusual situations, we create an instance of `NSException` and fill it with the pertinent data. The closest `@try/@catch` block will take care of it when we throw it using the appropriately called `@throw` directive.

The following example generates random numbers between a specified interval. The method throws a custom error if the caller provides an incorrect interval.

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

int generateRandomInteger(int minimum, int maximum) {
    if (minimum >= maximum) {
        // Create-exception.
        NSException *exception = [NSException
            exceptionWithName:@"RandomNumberIntervalEx
ception"
            reason:@"*** generateRandomInteger(): "
                "maximum parameter not greater
than the minimum parameter"
            userInfo:nil];

        // Throw-exception.
        @throw exception;
    }
}
```

```

        // Return random-integer.
        return arc4random_uniform((maximum - minimum) + 1)
+ minimum;
    }

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

        int result = 0;
        @try {
            result = generateRandomInteger(0, 10);
        }
        @catch (NSException *exception) {
            NSLog(@"Problem! Caught exception: %@",
[exception name]);
        }

        NSLog(@"Random Number: %i", result);

    }
    return 0;
}

```

This code will not throw an error since it provides a valid interval (0, 10) to `generateRandomInteger()`. If we change the interval to (0, -10), we'll see the `@catch()` section in operation. This is what occurs behind the scenes when the foundation classes get exceptions (e.g., the `NSRangeException` raised by `NSArray`).

We may also re-throw previously caught exceptions. This is useful when we want to be alerted of an exception but do not want to handle it ourselves. As a convenience, we may omit the argument to the `@throw` directive:

```

@try {
    result = generateRandomInteger(0, -10);
}
@catch (NSException *exception) {
    NSLog(@"Problem! Caught exception: %@", [exception
name]);

    // Rethrow current-exception.
    @throw
}

```

The captured exception is subsequently forwarded to the next-highest handler, in this instance, the top-level exception handler. This should display the results of our `@catch()` block and the normal Terminating program owing to an uncaught exception... message, followed by an abrupt shutdown.

The `@throw` directive does not have to throw `NSException` objects; it can throw anything. The following example throws an `NSNumber` object instead of a normal exception. Also, notice how we may target certain objects by following the `@try` block with a series of `@catch()` statements:

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

int generateRandomInteger(int minimum, int maximum) {
    if (minimum >= maximum) {
        // Generate the number using "default"
interval.
        NSNumber *guess = [NSNumber
                           numberWithInt:generateRandomInteger(0, 10)];

        // Throw-number.
        @throw guess;
    }
    // Return random-integer.
    return arc4random_uniform((maximum - minimum) + 1)
+ minimum;
}

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

        int result = 0;
        @try {
            result = generateRandomInteger(30, 10);
        }
        @catch (NSNumber *guess) {
            NSLog(@"Warning: Used the default
interval");
            result = [guess intValue];
        }
        @catch (NSException *exception) {
            NSLog(@"Problem! Caught exception: %@",
[exception name]);
        }
    }
}
```

```

        NSLog(@"Random Number: %i", result);
    }
    return 0;
}

```

Rather than returning an `NSException` object, `generateRandomInteger()` attempts to generate a new integer between the specified “default” bounds. The example shows how to utilize `@throw` with different objects; however, it isn’t the best application design or the most effective use of Objective-C exception-handling tools. If we wish to utilize the thrown value, as the previous code does, a simple conditional check using `NSError`, as discussed in the next section, would be preferable.

Some of Apple’s core frameworks also expect an `NSException` object to be thrown, so use caution when integrating with the standard libraries.

MANAGING ERRORS IN OBJECTIVE-C

Errors are intended to be a quick and easy way to identify whether or not an action was successful, whereas exceptions are intended to warn programmers when anything goes wrong. Errors, unlike exceptions, are intended to be used in ordinary control flow statements.

THE `NSError` CLASS

The only thing that mistakes and exceptions have in common is that they are both implemented as objects. The `NSError` class contains all of the information needed to signal errors:

- `code` – An `NSInteger` that identifies the mistake uniquely.
- `Domain` – An `NSString` object that describes the errors.
- `userInfo` – An instance of `NSDictionary` holding problem-specific information. This is significantly more commonly used than the `userInfo` dictionary of `NSException`.

Aside from these critical properties, `NSError` contains many variables that aid in the display and processing of errors.

- `localizedDescription` – An `NSString` containing the whole error explanation, frequently including the failure’s reason. This number is often displayed to the user via an alert panel.

- `localizedFailureReason` – An `NSString` with a different explanation for the error's cause. Customers that want to remove the error's cause from its comprehensive explanation should utilize this option.
- `recoverySuggestion` – An `NSString` gives the user information on how to recover from a mistake.
- `localizedRecoveryOptions` – An `NSArray` containing titles for the buttons in the error dialog. A single OK button is displayed to dismiss the alert if this array is empty.
- `helpAnchor` – When the Help anchor button in an alert panel is pressed, an `NSString` is shown.
- The `initWithDomain:code:userInfo` method, like `NSException`, may be used to initialize `NSError` objects with detailed data.

ERROR DOMAINS

A namespace for error codes is what an error domain is. Codes within a single domain must be unique, though they may overlap with codes from other domains. Domains provide information about an issue's source and avoid code clashes. The four principal built-in error domains are `NSMachErrorDomain`, `NSPOSIXErrorDomain`, `NSOSStatusErrorDomain`, and `NSCocoaErrorDomain`. Most of Apple's standard Objective-C frameworks include error codes in the `NSCocoaErrorDomain`; however, some frameworks define their error domains (e.g., `NSXMLParserErrorDomain`).

We should always add new error codes to our error domain if we need to produce new error codes for your libraries and apps; we should never enlarge any established error domains. It is a straightforward task to create our domain. Because domains are simply strings, we must define a string constant that does not conflict with any of the application's other error domains. Apple suggests that domains be in the form `com.<company>.project>.ErrorDomain`.

DETECTING ERRORS

There are no language constructs for dealing with `NSError` objects (though several built-in classes are designed to handle them). They are designed to be used in conjunction with functions that return an object when successful and `nil` when failing. The typical method for capturing mistakes is as follows:

Make a variable called `NSError`. It requires no allocation or setup.

This variable should be sent as a double pointer to a possibly error-prone function. If something goes wrong, the function will use this reference to log any errors.

Check the function's return value to see if it was successful or not. If the action fails, we might use `NSError` to handle or display the error.

A function usually does not return an `NSError` object; instead, it returns the intended result if it succeeds and `nil` if it fails. To find issues, always use the function's return value; never use the presence or absence of an `NSError` object to establish if an action was successful. Error objects are designed to represent a possible error, not to tell whether or not one occurred.

We begin by creating a file path to `~/Desktop/SomeContent.txt`. Then, we create an `NSError` reference and provide it to the `stringWithContentsOfFile:encoding:error:` function to record any problems encountered when loading the file. We've supplied a pointer to the `*error` pointer, indicating that the function asks a pointer to a pointer (i.e., a double pointer). This enables the procedure to supply the variable's content. Finally, we look at the return result (rather than the lack of the error variable) to see if `stringWithContentsOfFile:encoding:error:` succeeded or failed. If it does, we may work with the value of the content variable; otherwise, we use the error variable to display information about what went wrong.

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

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

        // Generate the desired file path.
        NSString *filename = @"SomeContent.txt";
        NSArray *paths =
        NSSearchPathForDirectoriesInDomains(
                                NSDesktopDirectory,
        NSUserDomainMask, YES
                                );
        NSString *desktopDir = [paths
        objectAtIndex:0];
        NSString *path = [desktopDir
                                stringByAppendingPathComponent:filename];
```

```

        // Trying to load file.
        NSError *error;
        NSString *content = [NSString stringWith
ContentsOfFile:path
                                encoding:NSUTF8String
Encoding
                                error:&error];

        // Check if its worked.
        if (content == nil) {
            // Some type of error occurred.
            NSLog(@"The Error loading file %@!",
path);
            NSLog(@"Description: %@", [error
localizedDescription]);
            NSLog(@"Reason: %@", [error
localizedFailureReason]);
        } else {
            // Content load successfully.
            NSLog(@"Content-loaded!");
            NSLog(@"%@", content);
        }
    }
    return 0;
}

```

CUSTOM ERRORS

We can establish custom errors by accepting a double reference to an `NSError` object and populating it with our data. Remember that your function or method should return either an object or null based on its success (do not return the `NSError` reference). The following example handles incorrect inputs sent to the `generateRandomInteger()` function using an error instead of an exception. Noting that the `**error` is a double-pointer, we may initialize the underlying value inside the method.

Using `if (error!= NULL)` to validate that the user has provided a valid `**error` parameter is important. This is done in error-generating functions. Because the `**error` parameter is a double-pointer, the `*error` parameter may assign a value to the referenced variable. Again, we examine the return value for errors (`if (result == nil)`) instead of the error variable.

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

```

NSNumber *generateRandomInteger(int minimum, int
maximum, NSError **error) {

```

```

    if (minimum >= maximum) {
        if (error != NULL) {

            // Create error.
            NSString *domain = @"com.MyCompany.
RandomProject.ErrorDomain";
            int errorCode = 4;
            NSMutableDictionary *userInfo =
[NSMutableDictionary dictionary];
            [userInfo setObject:@"The Maximum
parameter is not greater than minimum parameter"
                        forKey:NSLocalizedString
Key];

            // Populate error-reference.
            *error = [[NSError alloc] initWith
Domain:domain
                                code:errorCode
                                userInfo:userInfo];
        }
        return nil;
    }
    // Return random-integer.
    return [NSNumber
            numberWithInt:arc4random_uniform((maximum
- minimum) + 1) + minimum];
}

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

        NSError *error;
        NSNumber *result = generateRandomInteger
(0, -10, &error);

        if (result == nil) {
            // Check to see whats went wrong.
            NSLog(@"Error occurred!");
            NSLog(@"Domain: %@ Code: %li", [error
domain], [error code]);
            NSLog(@"Description: %@", [error
localizedDescription]);

```

```

        } else {
            // Safe to use returned-value.
            NSLog(@"The Random Number: %i", [result
intValue]);
        }

    }
    return 0;
}

```

The `localizedDescription`, `localizedFailureReason`, and associated characteristics of `NSError` are stored in its `userInfo` dictionary using special keys such as `NSLocalizedDescriptionKey`, `NSLocalizedFailureReasonErrorKey`, and so on. So, to define the problem, we just add some strings to the appropriate keys, as seen in the preceding sample.

We should offer constants for certain error domains and codes to guarantee consistency between classes.

OBJECTIVE-C LANGUAGE'S FEATURES

Here is a list of elements that, in my opinion, contribute to Objective-C being a good and interesting programming language.

Classes are objects

Each Objective-C class is an instance of a meta-class generated and managed automatically by the runtime. Class methods may create, classes can be sent as arguments, classes can store in collections, and so on. To produce an instance, we simply send a message to the Objective-C class we wish to instantiate. It is not necessary to redesign a “factory” system. At the language level, no specific constructor mechanism is necessary. This contributes to the language remaining basic and effective. Furthermore, meta-classes are objects.

DYNAMIC TYPING

In the same vein as Ruby, Python, Smalltalk, and Groovy... This is quite useful because we don't always know what our runtime objects will be. It is simple to use dynamic typing in Objective-C. For example, this specifies a variable that may hold a reference to an object:

```
id myObject;
```

OPTIONAL STATIC TYPING

Nonetheless, Objective-C provides static typing as well. It's the best of both worlds. This defines a variable that can hold a reference to an `NSView` or a subclass object:

```
NSView *myObject;
```

CATEGORIES

We may use categories to create new methods and add them to classes whose source code we do not know (such as the standard Cocoa classes provided by Apple). This makes class expansion possible without the requirement for subclassing. Very useful for customizing existing classes to the requirements of frameworks we wish to use or design.

MESSAGE SENDING

Messages are how we communicate with objects. Typically, the message's receiver has a method that directly matches the message (i.e., that has the same name or, in Objective-C terms, the same selector). The method will be invoked in this case. However, this is not only possible conclusion. An object can also handle a message by forwarding it to another object, broadcasting it to multiple objects, introspecting it, applying custom logic, and so on.

THE SYNTAX FOR EXPRESSING MESSAGES

The message patterns of Objective-C are comparable to natural language sentences with gaps (prefixed with colons). When we write code to send a message to an object, we fill in the blanks with real data to construct a logical statement. This expressing message is borrowed from Smalltalk and enhances the code's expressiveness.

Sending a message to an ordered collection, for example, requesting that an item at index 12 be inserted:

```
[myCollection insert:myObject atIndex:12]
```

A message sending expression can be regarded as a sentence, with the receiver as the subject and the message as the remainder of the sentence (e.g., an action that we want the recipient to perform): "at index 12 of myCollection, insert myObject."

INTROSPECTION

Object introspection is easy. For example, we might use the following syntax to ask an object for its class:

```
[myObject class]
```

Check to see whether an object has the method “foo”:

```
[myObject respondsToSelector:@selector(foo)]
```

Request the signature of an object’s “foo” method:

```
[myObject methodSignatureForSelector:@selector(foo)]
```

Determine whether one class is a subclass of another:

```
[class1 isKindOfClass:class2]
```

DYNAMIC OPERATING TIME

The runtime of Objective-C is dynamic. It enables the dynamic production of messages, the dynamic addition of methods to existing classes, the changing of method implementations, and so on.

AUTOMATIC GARBAGE COLLECTION

On its thread, the automatic garbage collector collaborates with the application code. It uses a generational method to boost performance by focusing on memory zones more likely to hold garbage. It is compatible with objects and raw C memory blocks allocated with the `NSAllocateCollactable()` function and related approaches. The `malloc()` function works as predicted, allowing access to memory that the collector does not keep.

The garbage collector is an optional service; if we do not want to use it, our application can rely on a reference counting mechanism. This system employs a novel technique for delayed release that considerably minimizes the work required for human reference counting.

When writing, the iPhone did not support the automatic garbage collector.

C INSIDE

The Objective-C programming language is an object-oriented extension and superset of the C programming language. This implies that the raw power of C is available and that C libraries may utilize it directly.

Furthermore, because Mac OS X, a UNIX system, is written mainly in C and Objective-C for higher-level components, this offers a symbiotic relationship between the language and the operating system.

C++ FLUENT

Objective-C is more than just a C superset; it can also comprehend and execute C++ code. In this configuration, the language is known as Objective-C++ and allows for combining Objective-C and C++ code statements. It also enables the usage of C++ libraries directly.

SIMPLICITY

Objective-C Smalltalk-inspired's object structure favors simplicity. Many of the traits that make languages complex (templates, overloading, multiple inheritances, etc.) are absent from Objective-C, which leverages its dynamic nature to enable easier programming paradigms.

ACCESS TO APPLE'S PRODUCTS

Each new version of Mac OS X through Objective-C, and now the iPhone OS, is packed with intriguing new features that can access directly from Objective-C. This significantly improves the usability of Objective-C.

NOTE

1. Objective-C blocks.

Bibliography

1. Objective-C Tutorial: https://www.tutorialspoint.com/objective_c/index.htm, accessed on June 1, 2022.
2. Programming with Objective-C: <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html>, accessed on June 1, 2022.
3. Difference between C and Objective-C: <https://www.geeksforgeeks.org/difference-between-c-and-objective-c/>, accessed on June 1, 2022.
4. Objective-C Programming: <https://educationecosystem.com/guide/objective-c>, accessed on June 1, 2023.
5. Introduction to Objective-C Programming: <https://medium.com/ios-objective-creation/lesson-1-introduction-to-objective-c-programming-22f5fe71172>, accessed on June 1, 2022.
6. An Overview of Objective-C: https://www.techotopia.com/index.php/An_Overview_of_Objective-C_Object_Oriented_Programming, accessed on June 1, 2022.
7. Objective-C – Overview: https://www.tutorialspoint.com/objective_c/objective_c_overview.htm, accessed on June 1, 2022.
8. Designveloper. (2021, February 19). What Is Objective-C? Retrieved from <https://www.designveloper.com/blog/what-is-objective-c/>, accessed on June 1, 2022.
9. The Objective-C Compiler: <https://www.objc.io/issues/6-build-tools/compiler/>, accessed on June 1, 2022.
10. Primer for Objective-C Developers: <https://docs.microsoft.com/en-us/xamarin/ios/get-started/objective-c-developers/primer>, accessed on June 1, 2022.
11. Objective-C – Foundation Framework: https://www.tutorialspoint.com/objective_c/objective_c_foundation_framework.htm, accessed on June 2, 2022.
12. An Overview of the Objective-C Foundation Framework: https://www.techotopia.com/index.php/An_Overview_of_the_Objective-C_Foundation_Framework, accessed on June 2, 2022.
13. Swift vs Objective-C: [https://www.javatpoint.com/swift-vs-objective-c#:~:text=Advantages%20\(Pros\)%20of%20Objective%20C&text=There%20is%20a%20lot%20of,C%2B%2B%20runs%20smoothly%20on%20this](https://www.javatpoint.com/swift-vs-objective-c#:~:text=Advantages%20(Pros)%20of%20Objective%20C&text=There%20is%20a%20lot%20of,C%2B%2B%20runs%20smoothly%20on%20this), accessed on June 2, 2022.
14. Objective-C Environment Setup: https://www.tutorialspoint.com/objective_c/objective_c_environment_setup.htm, accessed on June 2, 2022.

15. Objective-C Tutorial – Learn Objective-C Programming Language: http://referencedesigner.com/tutorials/objectivec/objectivec_02.php, accessed on June 2, 2022.
16. Installing and Using GNUstep and Objective-C on Windows: https://www.techotopia.com/index.php/Installing_and_using_GNUstep_and_Objective-C_on_Windows, accessed on June 2, 2022.
17. Objective-C Editor Features: https://editrocket.com/features/objective-c_editor.html, accessed on June 2, 2022.
18. Objective-C Program Structure: https://www.tutorialspoint.com/objective_c/objective_c_program_structure.htm, accessed on June 2, 2022.
19. Objective-C Program Structure: https://www.tutorialspoint.com/objective_c/objective_c_program_structure.htm, accessed on June 3, 2022.
20. What are Structures in Objective-C?: <https://www.educative.io/answers/what-are-structures-in-objective-c>, accessed on June 3, 2022.
21. Hello World in Objective-C and Compiling with GCC: <https://www.mahesh-subramaniya.com/article/hello-world-in-objective-c-and-compiling-with-gcc.html>, accessed on June 3, 2022.
22. Objective-C and Objective-C++ Dialect Options: https://gcc.gnu.org/onlinedocs/gcc/Objective-C-and-Objective-C_002b_002b-Dialect-Options.html, accessed on June 3, 2022.
23. Write Objective-C Code: <https://developer.apple.com/library/archive/referencelibrary/GettingStarted/RoadMapiOS-Legacy/chapters/WriteObjective-CCode/WriteObjective-CCode/WriteObjective-CCode.html>, accessed on June 3, 2022.
24. Objective-C – Basic Syntax: https://www.tutorialspoint.com/objective_c/objective_c_basic_syntax.htm, accessed on June 3, 2022.
25. Objective-C – Structures: https://www.tutorialspoint.com/objective_c/objective_c_structures.htm, accessed on June 3, 2022.
26. What is Objective C? A Beginner's Guide to Objective C: <https://www.edureka.co/blog/what-is-objective-c/>, accessed on June 3, 2022.
27. Pointers to Structures in Objective-C: <https://www.geeksforgeeks.org/pointers-to-structures-in-objective-c/>, accessed on June 3, 2022.
28. Objective-C – Basic Syntax: https://www.tutorialspoint.com/objective_c/objective_c_basic_syntax.htm, accessed on June 3, 2022.
29. Why does objective-c still require semicolons? Retrieved from <https://developer.apple.com/forums/thread/651319>, accessed on June 3, 2022.
30. Role of Semicolon in Various Programming Languages: <https://www.geeksforgeeks.org/role-of-semicolon-in-various-programming-languages/>, accessed on June 3, 2022.
31. Objective-C Documentation: <https://nshipster.com/objective-c-documentation/>, accessed on June 4, 2022.
32. Objective-C Basic Syntax: http://www.java2s.com/Tutorials/Objective_C/Language_Tutorial/0020__Objective_C_Basic_Syntax.htm, accessed on June 4, 2022.
33. Objective-C Reserved Keywords: <https://www.binpress.com/objective-c-reserved-keywords/>, accessed on June 4, 2022.

34. Removing leading and trailing whitespace: <https://riptutorial.com/objective-c/example/10335/removing-leading-and-trailing-whitespace>, accessed on June 4, 2022.
35. Objective-C – Basic Syntax: https://www.tutorialspoint.com/objective_c/objective_c_basic_syntax.htm, accessed on June 4, 2022.
36. Pointers to Structures in Objective-C: <https://www.geeksforgeeks.org/pointers-to-structures-in-objective-c/>, accessed on June 4, 2022.
37. Objective-C – Basic Syntax: https://www.tutorialspoint.com/objective_c/objective_c_basic_syntax.htm, accessed on June 4, 2022.
38. Why does objective-c still require semicolons? Retrieved from <https://developer.apple.com/forums/thread/651319>, accessed on June 4, 2022.
39. Commenting Objective-C Code: https://www.techotopia.com/index.php/Commenting_Objective-C_Code#:~:text=Comments%20in%20Objective%2DC%20enable,considered%20to%20be%20good%20practice, accessed on June 5, 2022.
40. Objective-C Documentation: <https://nshipster.com/objective-c-documentation/>, accessed on June 5, 2022.
41. Objective-C Basic Syntax: http://www.java2s.com/Tutorials/Objective_C/Language_Tutorial/0020__Objective_C_Basic_Syntax.htm, accessed on June 5, 2022.
42. Objective C Introduction: <https://www.roseindia.net/iphone/objectivec/objective.shtml>, accessed on June 5, 2022.
43. Objective-C Reserved Keywords: <https://www.binpress.com/objective-c-reserved-keywords/>, accessed on June 5, 2022.
44. Removing leading and trailing whitespace: <https://riptutorial.com/objective-c/example/10335/removing-leading-and-trailing-whitespace>, accessed on June 5, 2022.
45. Objective-C Data Types: https://www.tutorialspoint.com/objective_c/objective_c_data_types.htm#:~:text=In%20the%20Objective%2DC%20programming,bit%20pattern%20stored%20is%20interpreted, accessed on June 6, 2022.
46. Types: <https://code.tutsplus.com/objective-c-succinctly-data-types--mobile-21986t>, accessed on June 6, 2022.
47. Primitive Data Types in Objective-C: <https://andybargh.com/primitive-data-types-objective-c/>, accessed on June 6, 2022.
48. Objective-C Data Types: https://developer.apple.com/documentation/objectivec/objective-c_data_types, accessed on June 6, 2022.
49. Data Types: https://www.techotopia.com/index.php/Objective-C_2.0_Data_Types, accessed on June 6, 2022.
50. How to use variables in Objective-C?: <https://www.educative.io/answers/how-to-use-variables-in-objective-c>, accessed on June 7, 2022.
51. Objective-C – Variables. [Online]: https://www.tutorialspoint.com/objective_c/objective_c_variables.htm, accessed on June 7, 2022.
52. iOS 6 Programming Cookbook: Solutions for iOS Developers. O'Reilly Media, Inc: <https://www.oreilly.com/library/view/ios-6-programming/9781449342746/ch01s08.html>, accessed on June 7, 2022.

53. Learn Swift from Objective-C: <https://codewithchris.com/learn-swift-from-objective-c/>, accessed on June 7, 2022.
54. Learning Objective-C by Developing iPhone Games: <https://www.informat.com/articles/article.aspx?p=2161675>, accessed on June 7, 2022.
55. Working with Variables and Constants in Objective-C: https://www.techotopia.com/index.php/Working_with_Variables_and_Constants_in_Objective-C, accessed on June 7, 2022.
56. Objective-C Constants: https://www.tutorialspoint.com/objective_c/objective_c_constants.htm, accessed on June 8, 2022.
57. Working with Variables and Constants in Objective-C. Techotopia: https://www.techotopia.com/index.php/Working_with_Variables_and_Constants_in_Objective-C, accessed on June 8, 2022.
58. Objective-C Constants: https://developer.apple.com/documentation/objectivec/objective-c_constants, accessed on June 8, 2022.
59. Objective-C Operators and Expressions: https://www.techotopia.com/index.php/Objective-C_Operators_and_Expressions, accessed on June 9, 2022.
60. Objective-C Logical Operators: https://www.tutorialspoint.com/objective_c/objective_c_logical_operators.htm, accessed on June 9, 2022.
61. Objective-C Operators: https://www.tutorialspoint.com/objective_c/objective_c_operators.htm#:~:text=Objective%2DC%20language%20is%20rich,Bitwise%20Operators, accessed on June 9, 2022.
62. What is the '?' operator?: <https://stackoverflow.com/questions/17966700/what-is-the-operator>, accessed on June 9, 2022.
63. Objective-C Operators: http://www.java2s.com/Tutorials/Objective_C/Language_Tutorial/0050__Objective_C_Operator.htm, accessed on June 9, 2022.
64. Objective-C Operators: https://www.tutorialspoint.com/objective_c/objective_c_operators.htm, accessed on June 9, 2022.
65. Objective-C Operators: <https://mycodetips.com/objective-c/objective-c-operators-3711.html>, accessed on June 9, 2022.
66. Objective-C Loops: https://www.tutorialspoint.com/objective_c/objective_c_loops.htm#:~:text=Repeats%20a%20statement%20or%20group,that%20manages%20the%20loop%20variable, accessed on June 10, 2022.
67. Control Statements and Loops in Objective-C. dummies: <https://www.dummies.com/article/technology/programming-web-design/c/control-statements-and-loops-in-objective-c-172896/>, accessed on June 10, 2022.
68. ObjectiveC Control Statements and Loops: <https://mycodetips.com/objective-c/objective-c-control-statements-and-loops-3713.html>, accessed on June 10, 2022.
69. Objective-C for loop: https://www.tutorialspoint.com/objective_c/objective_c_for_loop.htm, accessed on June 10, 2022.
70. The Loop Control Structure – C Programming: <https://developerinsider.co/the-loop-control-structure-c-programming/>, accessed on June 10, 2022.
71. Objective-C for Loop: http://www.java2s.com/Tutorials/Objective_C/Language_Tutorial/0060__Objective_C_for_Loop.htm, accessed on June 10, 2022.

72. C Loop Statements: For, While, Do While: <https://www.guru99.com/c-loop-statement.html>, accessed on June 10, 2022.
73. Objective-C – Decision Making: https://www.tutorialspoint.com/objective_c/objective_c_decision_making.htm, accessed on June 11, 2022.
74. What is decision-making in Objective-C? Retrieved from <https://www.educative.io/answers/what-is-decision-making-in-objective-c>, accessed on June 11, 2022.
75. Programming in Objective-C (4th ed.). Addison-Wesley Professional: <https://www.oreilly.com/library/view/programming-in-objective-c/9780321712172/ch06.html>, accessed on June 11, 2022.
76. Objective-C – Decision Making: https://www.tutorialspoint.com/objective_c/objective_c_decision_making.htm, accessed on June 11, 2022.
77. What is Decision Making? Definition, Styles, and Techniques: <https://www.managementstudyguide.com/what-is-decision-making.htm>, accessed on June 11, 2022.
78. Objective-C – Functions: https://www.tutorialspoint.com/objective_c/objective_c_functions.htm#:~:text=A%20function%20is%20a%20group,your%20code%20into%20separate%20functions, accessed on June 12, 2022.
79. What are functions in Objective-C? Retrieved from <https://www.educative.io/answers/what-are-functions-in-objective-c>, accessed on June 12, 2022.
80. An Overview of Objective-C Functions: https://www.techotopia.com/index.php/An_Overview_of_Objective-C_Functions, accessed on June 12, 2022.
81. How to declare and define functions in Objective-C like in C? Retrieved from <https://stackoverflow.com/questions/35092468/how-to-declare-and-define-functions-in-objective-c-like-in-c>, accessed on June 12, 2022.
82. Understanding the difference between functions and methods in Objective-C. Medium: <https://medium.com/@jsogarro/understanding-the-difference-between-functions-and-methods-in-objective-c-152ed1629db9>, accessed on June 12, 2022.
83. Objective-C – Functions: https://www.tutorialspoint.com/objective_c/objective_c_functions.htm, accessed on June 12, 2022.
84. Classes & Functions: <https://oddinstitute.com/classes-functions-objective-c/>, accessed on June 12, 2022.

Index

A

Advantages of Objective-C, [3](#)
appendString() method, [161](#)
Apple, [171](#), [217](#)
Arithmetic operators in Objective-C, [98](#)
 integer and floating-point conversions,
 [103–105](#)
 modulus operator, [102–103](#)
 operator precedence, [98–100](#)
 relational operators, [106](#)
 type cast operator, [105–106](#)
 unary minus operator and integer
 arithmetic, [100–102](#)
Array elements, accessing, [177–178](#)
Arrays initialization, [177](#)
Arrays in Objective-C, [176](#), [178](#), [184](#)
 accessing array elements, [177–178](#)
 arrays initialization, [177](#)
 declaring arrays, [176](#)
 in depth, [178](#)
Assemblies vs. frameworks, [26–27](#)
Assignment operators, [108](#)
Automatic garbage collection, [216](#)

B

Bit fields, [189–190](#)
Bitwise operators, [107–108](#)
Blank spaces, [140](#)
Blocks in Objective-C, [172](#)
 blocks take arguments and return
 values, [173](#)
 blocks using type definitions,
 [173–174](#)
 lambda expressions vs, [28](#)
 simple block declaration syntax, [173](#)

Blocks take arguments and return values,
 [173](#)
Blocks using type definitions, [173–174](#)
Boolean variables, [154–158](#)
Break statement, [131](#)

C

C# primer, [25](#)
C++ and Objective-C, [4](#), [170–171](#)
C++ fluent, [217](#)
Calculator class, [110–112](#)
Calling a method, [163–165](#)
C and Objective-C, [169–170](#)
Catching exceptions, [205–206](#)
Categories, [215](#)
 access to Apple's products, [217](#)
 automatic garbage collection, [216](#)
 C++ fluent, [217](#)
 C inside, [216–217](#)
 dynamic operating time, [216](#)
 extension methods vs, [26](#)
 introspection, [216](#)
 message sending, [215](#)
 simplicity, [217](#)
 syntax for expressing messages, [215](#)
Character constants, [80–81](#), [94](#)
Char variable, [85](#)
Choosing Objective-C, [2–3](#), [171](#)
C inside, [216–217](#)
Clang, [14](#), [20](#), [21](#)
Class and instance methods, [57](#)
Classes, [6](#), [214](#)
Class NSException, [204](#)
Cocoa, [1](#), [166](#)
Cocoa Touch, [1](#), [166](#)
Code, creating, [21–23](#)

Command-line arguments, 201–202
 Command line to compile Objective-C,
 36–37
 Comments, 72
 Compiler, 13–14
 Compound relational tests, 139–142
 Conditional operator, 159–160
 Configuring Objective-C environment
 editor of text, 30
 local environment, configuration of, 29
 Constant definition, 81, 95
 const keyword, 82, 95–96
 #define preprocessor, 82, 95
 Constant expressions, 83
 Constants in Objective-C, 79, 92
 character constants, 80–81, 94
 floating-point literals, 80, 93
 integer literals, 79–80, 93
 string literals, 81, 94–95
 Const keyword, 82, 95–96
 Continue statement, 131
 convertToNum method, 135
 Creating a method, 162–163
 Custom errors, 212–214
 “Customize Table Cell” option, 6–8
 Custom macros, 15–17

D

Data encapsulation, 66–70
 Data types, 73–74
 and constants, 83
 long integer variable, 86
 long long int variable, 86
 qualifiers, 86
 short variables, 86
 signed variable, 86
 type char, 85–86
 type float, 84–85
 type id, 86–87
 type int, 84
 unsigned variable, 86
 void type, 87
 Decision making, 132
 Boolean variables, 154–158
 compound relational tests, 139–142
 conditional operator, 159–160
 else if construct, 144–151

 if-else construct, 137–139
 if statement, 133–137
 nested if statements, 142–143
 switch statement, 151–154
 Declarations of method, 163
 Declaring arrays, 176
 Defined() operator, 193–194
 #define directive in Objective-C, 195–196
 #define preprocessor, 82, 95
 Demerits of Objective-C, 171
 Detecting errors, 210–212
 Developer folder, 32
 Development, Objective-C in, 172
 Disabling logs in live apps, 198
 Disadvantages of Objective-C, 3
 Do statement, 129–130
 Do-while loop, 130
 Dynamic operating time, 216
 Dynamic typing, 214
 Dynamism, 3

E

Else if construct, 144–151
 Environment setup and program
 structure, 29
 class and instance methods, 57
 configuring Objective-C environment
 configuration of the local
 environment, 29
 editor of text, 30
 fractions, Objective-C class for the
 working with, 52–54
 GCC compiler, 30
 command line to compile
 Objective-C, 36–37
 filename extension, 37–39
 first program’s explanation, 42–46
 Hello Everyone, Objective-C
 example of, 34–35
 Mac OS installation, 32
 structure of the Objective-C
 program, 34–35
 terminal, using, 40–41
 UNIX/LINUX installation, 30–31
 variables, displaying the values of,
 46–48
 Windows installation, 33

- Xcode, launching, 32–33
- Xcode to create Objective-C application, 35–36
- @implementation section, 59–61
- instances and methods, 49–52
- instance variables and data encapsulation, 66–70
- @interface section, 54–56
- method arguments, 58–59
- object, 49
- program section, 61–66
- return values, 57–58
- Error domains, 210
- Error handling in Objective-C, 199, 203, 209
 - class NSError, 204
 - command-line arguments, 201–202
 - custom errors, 212–214
 - error domains, 210
 - errors, detecting, 210–212
 - exceptions, catching, 205–206
 - exceptions, making, 204–205
 - exceptions and errors, 202–203
 - exception throwing, 206–209
 - NSError, 199–201, 209–210
- Exceptions and errors, 202–203
- Exception throwing, 206–209
- Extension methods vs categories, 26

F

- Features of Objective-C language, 214
 - classes, 214
 - dynamic typing, 214
 - optional static typing, 215
- Filename extention, 37–39
- First program's explanation, 42–46
- Floating-point, types of, 75
- Floating-point literals, 80, 93
- Floating-point value, 103–105
- For loop variants, 123–125
- For statement, 115
 - break statement, 131
 - continue statement, 131
 - do statement, 129–130
 - do-while loop, 130
 - for loop variants, 123–125
 - keyboard input, 121–122

- nested for loops, 122–123
- relational operators, 116–121
- while statement, 125–128
- Foundation Framework, 5
- Fraction class, 59, 61–62
- Fractions, Objective-C class for the working with, 52–54
- Frameworks, assemblies vs., 26–27
- Function arguments, 165
 - as structures, 186–187
- Functions, 161
 - calling a method, 163–165
 - creating a method, 162–163
 - declarations of method, 163
 - function arguments, 165

G

- GCC compiler, 29, 30, 166
 - displaying the values of variables, 46–48
- filename extention, 37–39
- first program's explanation, 42–46
- Mac OS installation, 32
- structure of the Objective-C program, 34
 - Objective-C example of Hello Everyone, 34–35
- UNIX/LINUX installation, 30–31
- using the command line to compile Objective-C, 36–37
- using the terminal, 40–41
- using Xcode to create an Objective-C application, 35–36
- Windows installation, 33
- Xcode, launching, 32–33

H

- Headers, namespaces and, 27
- Hello Everyone, Objective-C example of, 34–35

I

- Id data type, 86–87
- Identifiers, 72
- If-else construct, 137–139

If statement, 133–137
 Boolean variables, 154–158
 compound relational tests, 139–142
 conditional operator, 159–160
 else if construct, 144–151
 if-else construct, 137–139
 nested if statements, 142–143
 switch statement, 151–154
 @implementation section, 59–61
 Index, 176
 Instance, 6
 and methods, 49–52
 Instance variables, 66–70
 Integer and floating-point conversions, 103–105
 Integer arithmetic, 100–102
 Integer constant, 84
 Integer literals, 79–80, 93
 Integer promotion, 197
 Integers, types of, 74–75
 @interface section, 54–56
 Interfaces vs. protocols, 26
 International Telephone and Telegraph (ITT) division, 167
 Interop Objective-C, 25–26
 Introspection, 216
 Int variable, 83
 ITT division, *see* International Telephone and Telegraph division

K

Keywords, 72–73

L

Lambda expressions vs blocks, 28
 Languages, comparison of, 26
 Learning Objective-C, 5, 172
 Lexing, 17–18
 List vs NSArray initialization, 28
 Live apps, disabling logs in, 198
 LLVM Intermediate Representation (LLVM IR), 14
 Log handling in Objective-C, 198
 disabling logs in live apps, 198
 NSLog method, 198
 Logical operators in Objective-C, 106

assignment operators, 108
 bitwise operators, 107–108
 sizeof and ternary, 109
 Long integer variable, 86
 Long long int variable, 86
 Loops and control statement, 113
 for statement, 115
 break statement, 131
 continue statement, 131
 do statement, 129–130
 do-while loop, 130
 for loop variants, 123–125
 keyboard input, 121–122
 nested for loops, 122–123
 relational operators, 116–121
 while statement, 125–128
 lvalues, 79, 91

M

Mac OS installation, 32
 Macro continuation, 192
 Macros, 194
 Making exceptions, 204–205
 Malloc() function, 216
 max() method, 162
 Member access operator, 185
 memberDeclarations section, 59
 Merits of Objective-C, 171
 Messages
 sending, 215
 syntax for expressing, 215
 Method arguments, 58–59
 Modulus operator, 102–103

N

Named parameters vs. selectors, 27
 Namespaces and headers, 27
 Nested for loops, 122–123
 Nested if statements, 142–143
 NeXTSTEP code library, 166
 NSArray, 28
 NSError, 199–201
 NSError class, 209–210
 NSError Object, 199
 NSLog function, 44, 46, 198
 NSRangeException, 205

NSString character string object, 85
 Null pointers, 180–181
 Numbers in Objective-C, 174–176

O

Object, 6, 49
 Objective-C Preprocessor (OCPP), *see*
 Preprocessors in Objective-C
 Object-Oriented Programming (OOP),
 2, 5–6
 OCPP, *see* Preprocessors in Objective-C
 OOP, *see* Object-Oriented Programming
 OpenStep interface objects, 167
 Operators, 97
 arithmetic operators in Objective-C,
 98
 integer and floating-point
 conversions, 103–105
 modulus operator, 102–103
 operator precedence, 98–100
 relational operators, 106
 type cast operator, 105–106
 unary minus operator and integer
 arithmetic, 100–102
 associative property, 98
 logical operators in Objective-C,
 106
 assignment operators, 108
 bitwise operators, 107–108
 sizeof and ternary, 109
 operators' precedence in Objective-C,
 109
 Calculator class, 110–112
 precedence, 98–100
 precedence in Objective-C, 109
 Calculator class, 110–112
 Optimizations, 23–25
 Optional static typing, 215

P

Parameterized macros, 194
 Parsing, 18–20
 Pointers in Objective-C, 178, 179
 details about, 181
 null pointers, 180–181
 strings in Objective-C, 181–184

structures in Objective-C, 184
 access to structure members,
 185–186
 bit fields, 189–190
 creating a structure, 184–185
 function arguments as structures,
 186–187
 pointers to structures, 187–189
 working of, 180
 Predefined macros, 191–192
 Preprocessors in Objective-C, 14–15, 190
 operators of, 192
 defined() operator, 193–194
 macro continuation, 192
 stringize, 192–193
 token pasting, 193
 parameterized macros, 194
 predefined macros, 191–192
 typedef in Objective-C, 194–195
 typedef vs #define, 195–196
 Program section, 61–66
 Properties, 27
 Protocols, interfaces vs., 26
 Punctuation characters, 72

Q

Qualifiers, 86

R

Recipe class, 9
 Recipe class, making, 8
 recipes array replacing tabledata, 12–13
 setting up the array of recipe objects,
 9–12
 Relational operators, 106, 116–121
 Return values, 57–58
 rvalues, 79, 91

S

Selectors, named parameters vs., 27
 Semicolons, 72
 setDenominator, 58, 67
 setNumerator, 58, 67
 Short variables, 86
 Signed variable, 86

Simple block declaration syntax, 173
 Single-dimensional array, 176
 Sizeof and ternary, 109
 Static analysis, 20
 Static keyword, 27
 Stringize, 192–193
 String literals, 81, 94–95
 Strings in Objective-C, 181–184
 Structures of Objective-C, 34–35, 184
 access to structure members, 185–186
 bit fields, 189–190
 creating a structure, 184–185
 function arguments as structures, 186–187
 pointers to structures, 187–189
 Swift, 2, 167
 Objective-C vs, 4, 168–169
 Swift 5, 171
 Switch statement, 151–154
 Syntax, 71
 constant definition, 81
 const keyword, 82
 #define preprocessor, 82
 constants in Objective-C, 79
 character constants, 80–81
 floating-point literals, 80
 integer literals, 79–80
 string literals, 81
 tokens, 71
 comments, 72
 data types, 73–74
 floating-point, types of, 75
 identifiers, 72
 integers, types of, 74–75
 keywords, 72–73
 lvalues and rvalues, 79
 semicolons, 72
 variable declaration, 77–78
 variable definition, 76–77
 variables in Objective-C, 76
 void type, 75
 whitespace in Objective-C, 73

T

Tabladata, recipes array replacing, 12–13
 Terminal, using, 40–41
 Text Editor, 29

@throw directive, 203, 208
 TIOBE Programming Language Award, 167
 Tokenization (lexing), 17–18
 Token-pasting operator, 193
 Tokens, 71
 comments, 72
 data types, 73–74
 floating-point, types of, 75
 identifiers, 72
 integers, types of, 74–75
 keywords, 72–73
 lvalues and rvalues, 79
 semicolons, 72
 variable declaration, 77–78
 variable definition, 76–77
 variables in Objective-C, 76
 void type, 75
 whitespace in Objective-C, 73
 Type casting in Objective-C, 196
 integer promotion, 197
 usual arithmetic conversion, 197–198
 Type cast operator, 105–106
 Type char, 85–86
 Type checking, 20–21
 Typedef
 vs #define, 195–196
 in Objective-C, 194–195
 Type definitions, blocks using, 173–174
 Type float, 84–85
 Type id, 86–87
 Type int, 84

U

UI, *see* [User Interface](#)
 Unary minus operator and integer arithmetic, 100–102
 UNIX/LINUX installation, 30–31
 Unsigned variable, 86
 User Interface (UI), 121
 Using Objective-C, 5
 Usual arithmetic conversion, 197–198

V

Variable declaration, 77–78, 90–91
 Variable definition, 76–77, 89–90

Variable factorial, 86
Variables, 88
 displaying the values of, 46–48
 lvalues and rvalues, 91
 in Objective-C, 76
viewDidLoad function, 9
Void type, 75, 87

W

While statement, 125–128
Whitespace in Objective-C, 73
Windows installation, 33

X

Xcode, 13
 to create an Objective-C application,
 35–36
 launching, 32–33

Y

yourMove message, 142